BASICS OF MECHATRONICS

Go through a detailed explanation about the various electronic components such as DC Motors IMU Sensors, Servo motors PWM signals, Li-Po batteries and key mechatronic concepts such as motor characteristization and designing a control system.

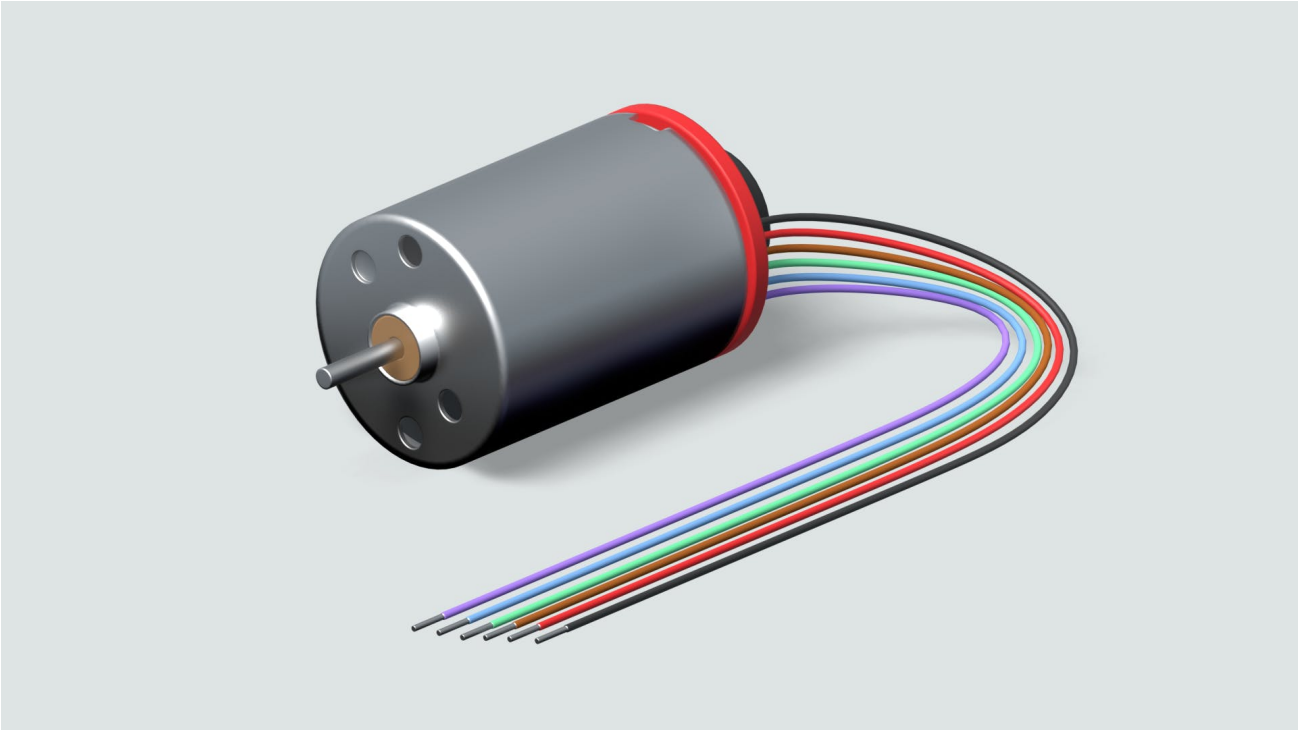
3.1 DC Motors

DC motors are an integral component in all the projects included in this Kit and therefore its good to know the technical details of the subcomponents and the underlying electronic concepts used to run the DC motor.

This section covers

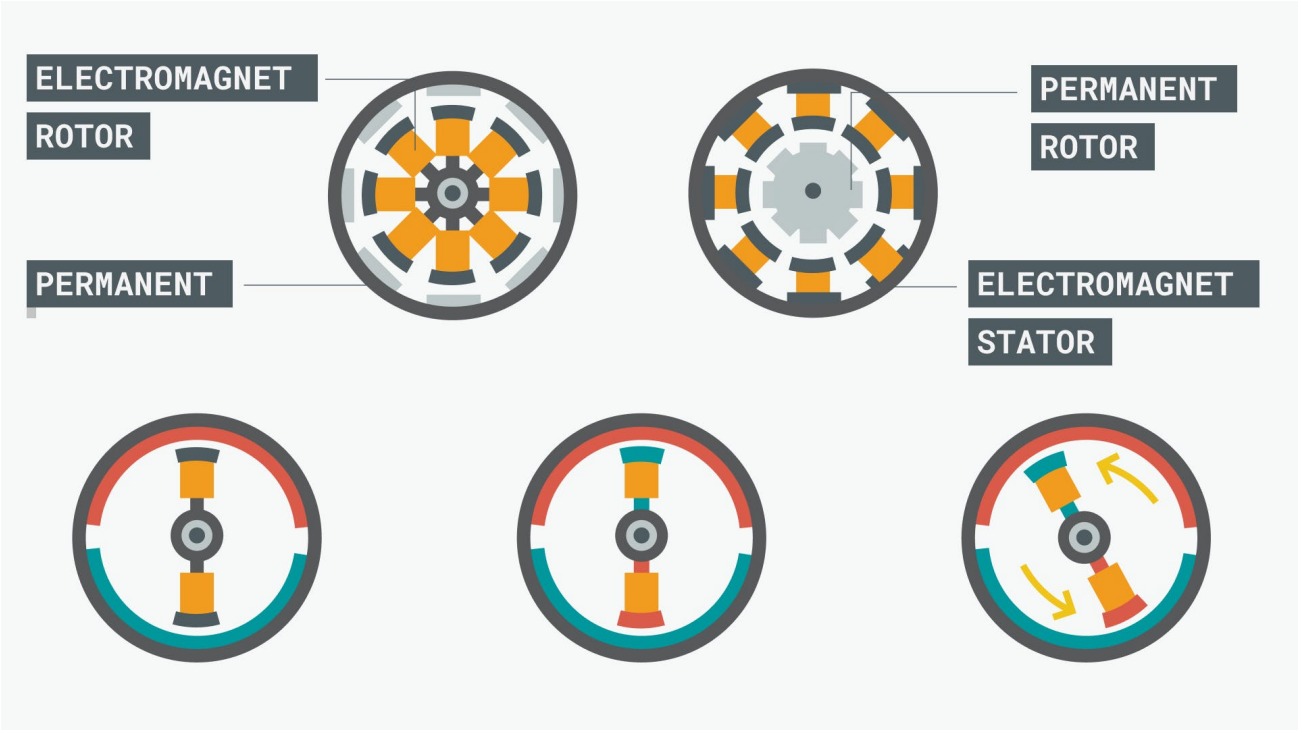
- What a DC motor is,
- The different parts of the DC Motor,
- About the general characteristics of DC Motors,
- Driving DC Motors using H bridges,
- Controlling the Speed of DC Motors with PWM,
- Reading the speed of the motors through encoders.

Direct Current Motors



A DC (Direct Current) motor is a type of motor that will cause the motor shaft to rotate around its longitudinal axis when applying an electric current between its terminal pins. Thus, the DC motor is a type of actuator that transforms electrical current into rotational motion.

Parts of the DC Motor



There are two parts inside a motor: the rotor (the shaft is part of this) and the stator. Looking at the cross-section of a motor, you can see that the rotor is the moving part and the stator is the static part. The stator and the rotor use both permanent magnets and electromagnets. Depending on the type of motor, the stator can be a permanent magnet while the rotor is an electromagnet, or vice-versa. Turning on the electromagnet creates attraction and repulsion forces that make the motor spin.

How Does it Work

The DC motor spins when we apply DC voltage through its two terminal pins. We can vary the speed of the motor by changing the voltage level. Also, motors can run freely in both directions just by reversing the direction of the current. DC motors alone are not very good for precise movement, but they can provide very high rotational speeds.

Characterstics of Motors

When searching for a DC motor, there are some parameters that you should look for in the motor's datasheet:

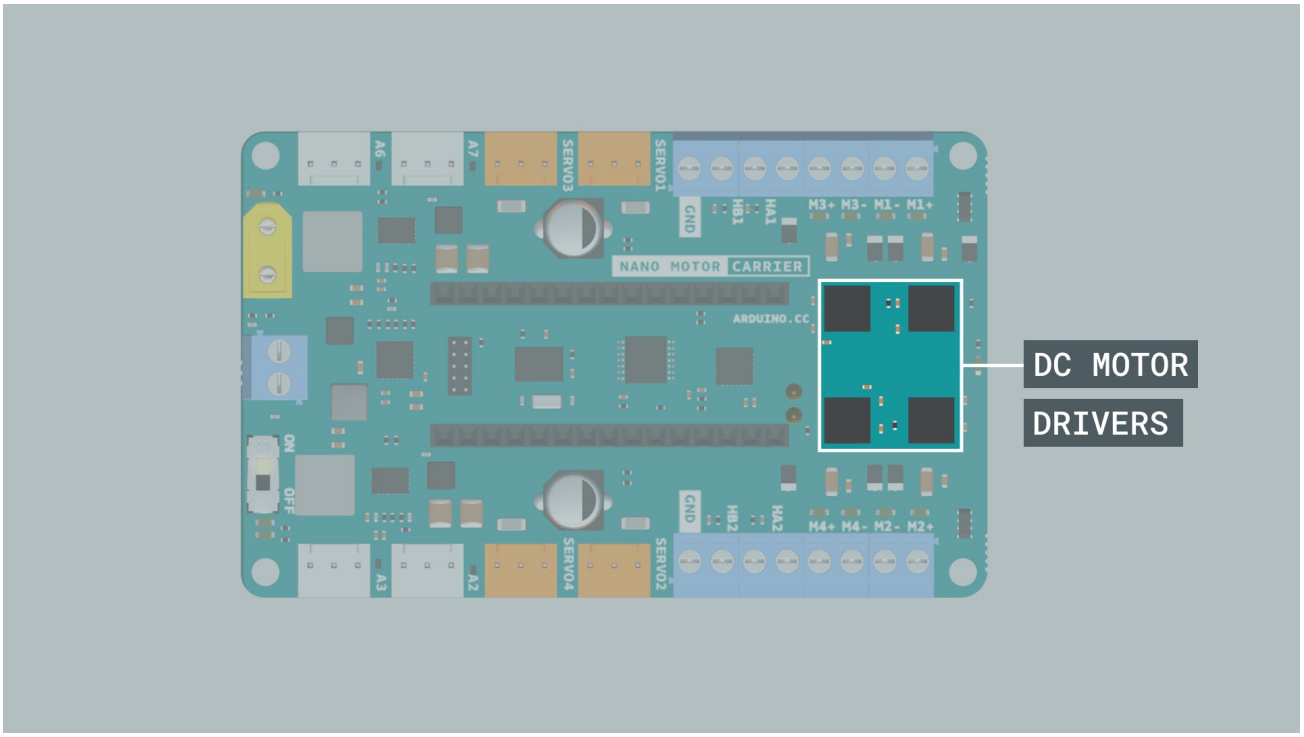
- **Speed**: This is commonly presented in RPM (Revolutions per minute), ar
  you a reference of how fast the motor can spin. Keep in mind that som

parameters are measured under specific conditions. If you add a load to the motor, the motor will go slower, so you might have to customize your motor speed to your particular project. In the datasheets, this parameter is commonly measured as **no load**. This indicates the maximum speed the motor can reach.

- **Stall Torque**: This is commonly presented in kg-cm and sometimes in Newton-meters, and it gives you a frame of reference for the maximum strength of the motor (i.e., when the motor can no longer rotate because of overload).
- **Stall current**: The amount of electric current in Amperes that is consumed by the motor under a maximum load (stall).
- **Operating voltage**: The range of voltages that the motor is designed to work within.

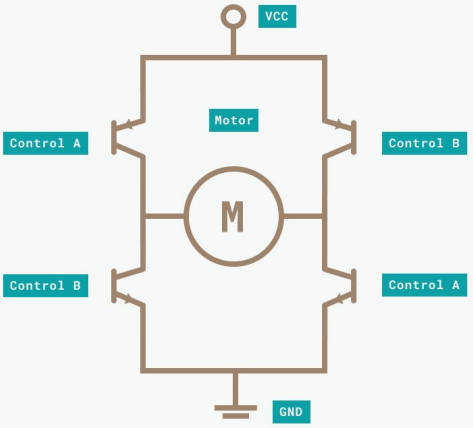
Driving DC Motors

When controlling motors from a microcontroller, it is not recommended to connect the output pins of the microcontroller directly to the motor terminals as the motor's current demand can damage the chip. In addition, most electric motors need a higher voltage than the one that can be provided by a microcontroller. To control a DC motor from a microcontroller without damaging the microcontroller, you'll need to use a circuit in between called **driver**. This can be a transistor, a relay or an H-Bridge. The **Arduino Nano Motor Carrier** uses H-Bridges to drive the DC motors.

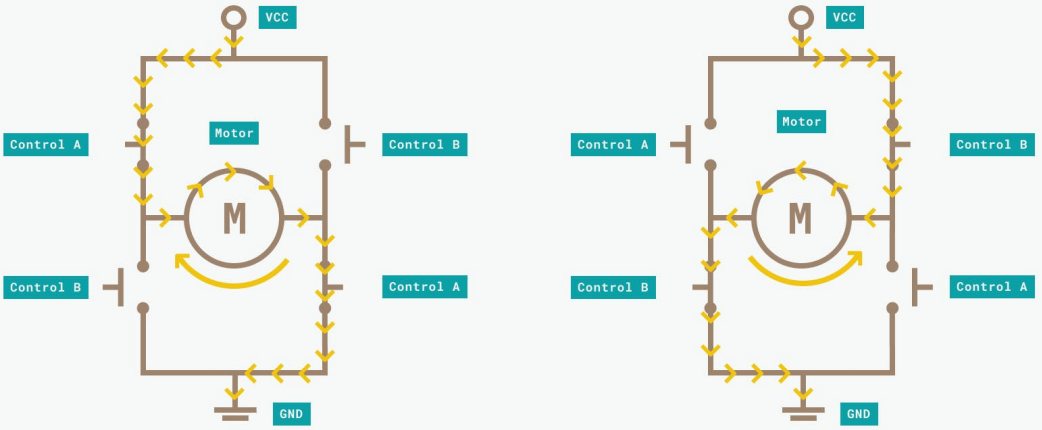


The H-Bridge

An H-Bridge is an electronic circuit that allows us to change the current direction applied to a load. They are commonly used as motor drivers to change the direction and the speed of a DC motor and to manage the higher power since microcontrollers usually don’t have enough current to power an electric motor. For example, In this kit we will use the H bridges in the Arduino Nano Motor Carrier to provide 12V to our motors while safely controlling them with 3.3V signals from the microcontroller.



The H-Bridge contains four transistors arranged so that the current can be driven to control the direction of the spin and the angular speed. These transistors behave as four switches that are controlled in pairs. The current flows in a different direction depending on which switches are activated. This allows the direction of the motor itself to be controlled.



Help

The image above shows a simplified version of the internal structure of an H-bridge and how the switches can be controlled to change the motor's direction. Switches in the above diagram are transistors that are controlled by the microcontroller on the Motor Carrier. If we consider a logical HIGH level when the switch is closed and LOW level when the switch is opened, we have the following behavior.

Signal Control A	Signal Control B	Movement
HIGH	LOW	Clockwise
LOW	HIGH	Counter-Clockwise
LOW	LOW	Slow Stop
HIGH	HIGH	Fast Stop

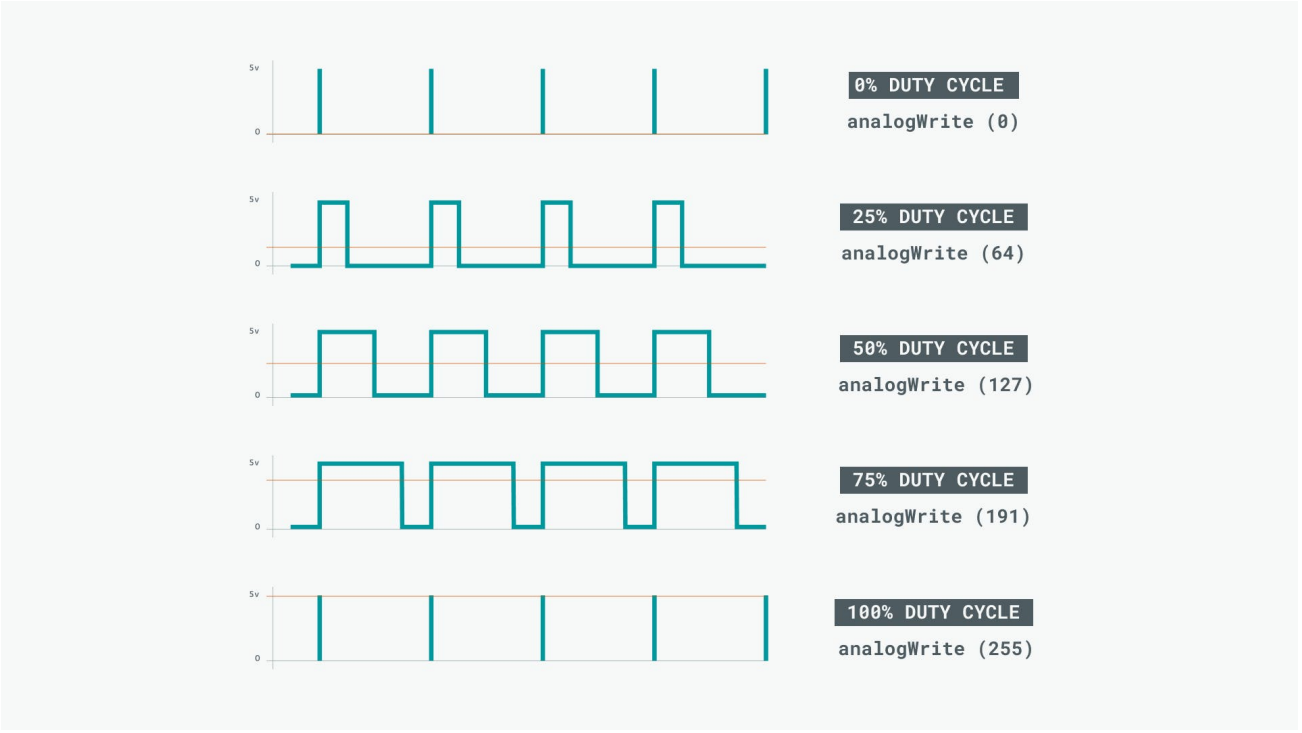
H-Bridges in the Kit

The current that a motor consumes is proportional to its torque. Usually, the motors indicate maximum consumption for maximum torque (e.g., 1.5A for 19.61Nm). When choosing an H-Bridge, it is important to select one with a current rate that is higher than your motor consumption. If the motor consumes more current than the H-Bridge can supply, it may lead to overcurrent damage. In this kit, we are going to use 4 H bridges chips located in the top side of the Motor Carrier: The chips are controlled through I2C via the SAMD11 microcontroller present in the Motor Carrier. This driver has a current sense resistor that limits the amount of current the chip can drive to prevent it from damage due to current spikes. In the Motor Carrier, the current is limited to 1.5A.

Controlling Speed : Pulse Width Modulation

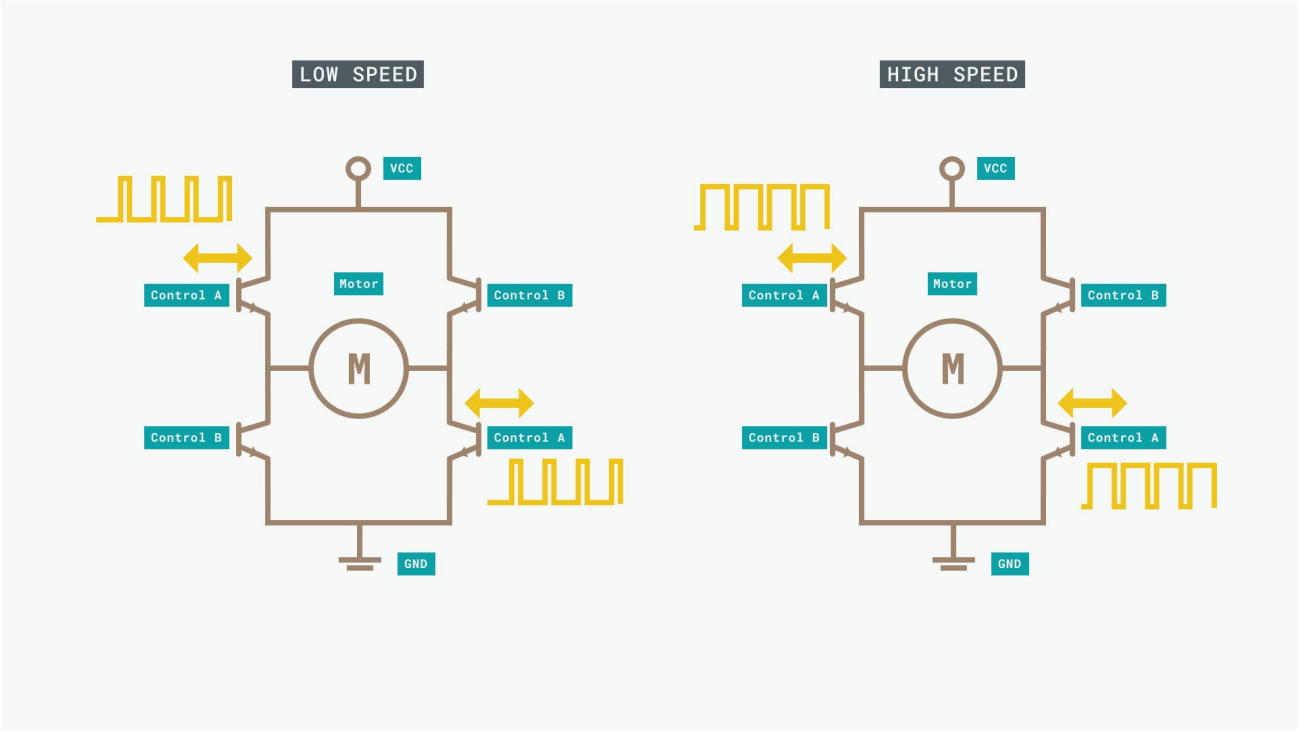
We can change the speed of a motor by controlling the voltage level in its terminals, The higher the voltage applied, the faster the motor spins. It is common practice to use a digital signal called PWM (Pulse Width Modulation) to control the speed of a motor instead of providing analog voltages. This is possible because since the motor cannot change speed so fast, it acts as a low pass filter and it behaves as if it is receiving the average (analog) value of the voltage and current being applied.

Pulse Width Modulation, or PWM, is a digital modulation technique that consists of changing the width of a signal's pulse at a fixed frequency. The width of the pulse is referred to as the duty-cycle and goes from 0% (minimum width) to 100% (maximum width).



To visualize how PWM functions, let's look at an LED. When an LED turns on, it doesn't immediately go from OFF to fully ON. Instead, it starts as OFF and glows brighter. It's almost instantaneous and not visible to the naked eye. But if you could turn off the LED before it reached 100% brightness, and then keep switching it on and off without ever letting the LED reach its maximum output, then the LED will glow less. Doing this repeatedly and quickly enough will make the LED look like it is only glowing at 25% of its full brightness. This switching is the "pulse" part of the PWM—pulsing on and off. It should be also noted that the human eye is also of a low-pass nature, and that contributes to the enhancement of this effect.

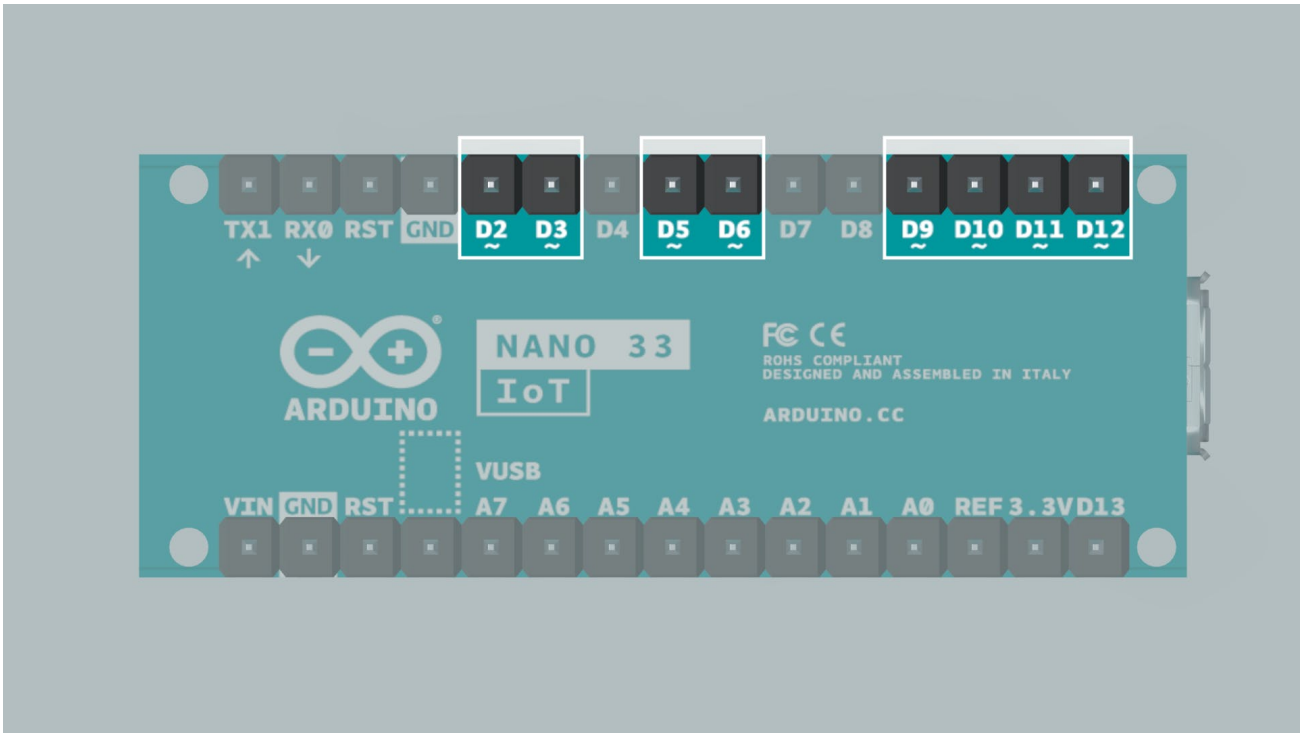
The same premise applies to a DC motor. In order to reduce the speed of the motor, you can lower the duty cycle of the PWM signal that goes to the motor driver. The motor's inertia (its inability to immediately stop when no more energy is being provided) combined with the properties of coils (that don't allow for sudden changes of current within the coils), is what contributes to PWM being able to control the speed. Many H-bridges allows you to control the speed (and direction) of the motors using a PWM signal:



PWM in Microcontrollers

Most microcontrollers have the capability to generate dedicated PWM signals. Typically, not all the digital pins have this secondary option, so you must check the datasheet if you need a PWM signal to see which pins have this capability. There are also libraries that you can use to generate “software” PWM signals in any GPIO (General Purpose Input/Output) pin.

In the Arduino boards, PWM pins are denoted with a “tilde” **~** symbol next to the pin number. On an Arduino Nano 33 IoT, the pins are 2, 3, 5, 6, 9, 10, 11 and 12.



In this kit we will be using PWM signals to control the speed of all the DC motors in the three projects.

To Learn More

Read more about PWM and how to generate PWM signals from an Arduino board below.

- Arduino's reference: **analogWrite**
- Arduino's reference: **description of PWM**
- Arduino's reference: **the secrets of Arduino's PWM**

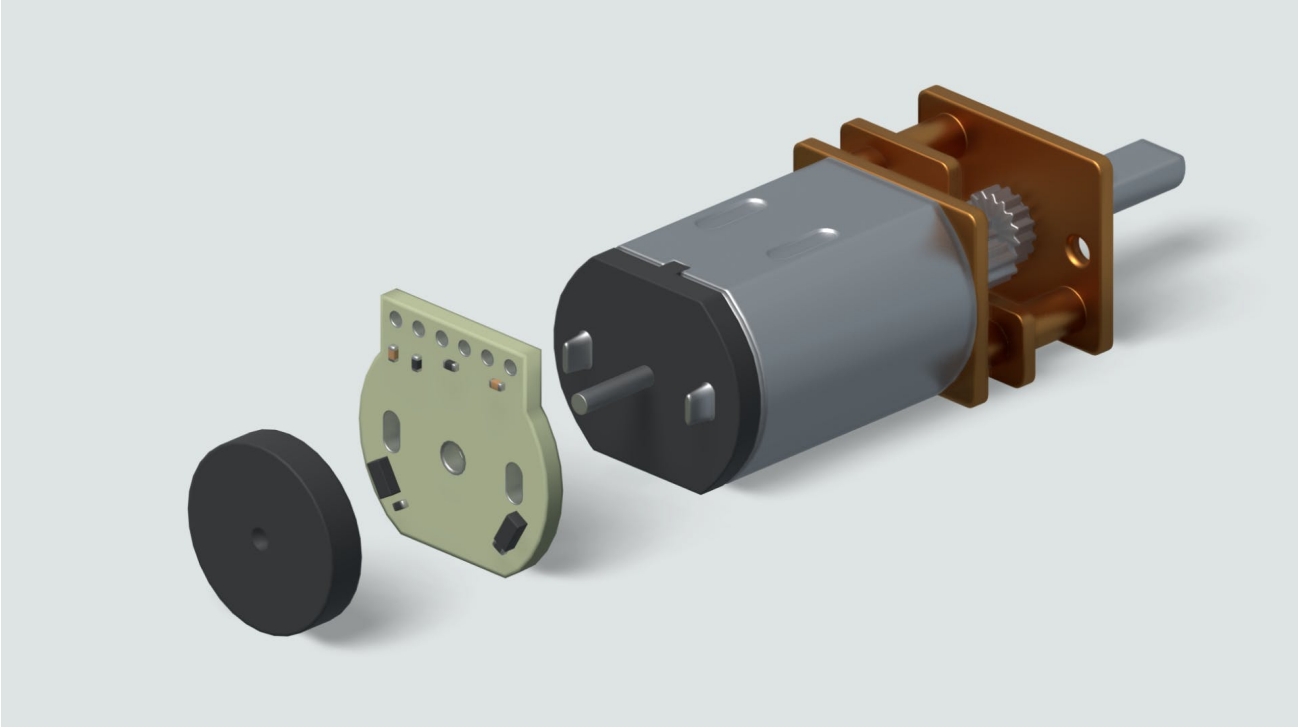
Reading Motor Speed

Motor speed is calculated by using a special device called as Magnetic encoders or simply as encoders. Encoders sensors that can report information about the rotation speed and the spinning direction of the motor when mounted on a motor. A common way to use magnetic encoders is to attach them to the motors driving the wheels of a robot. In doing so, the sensor will be able to detect the speed of the robot,

direction (angular rotation direction), and the distance traveled (by knowing the robot's wheel radius). The DC motors in this kit comes with Hall Sensor based encoders which are nothing but a module with two Hall-effect sensors and magnetic discs.

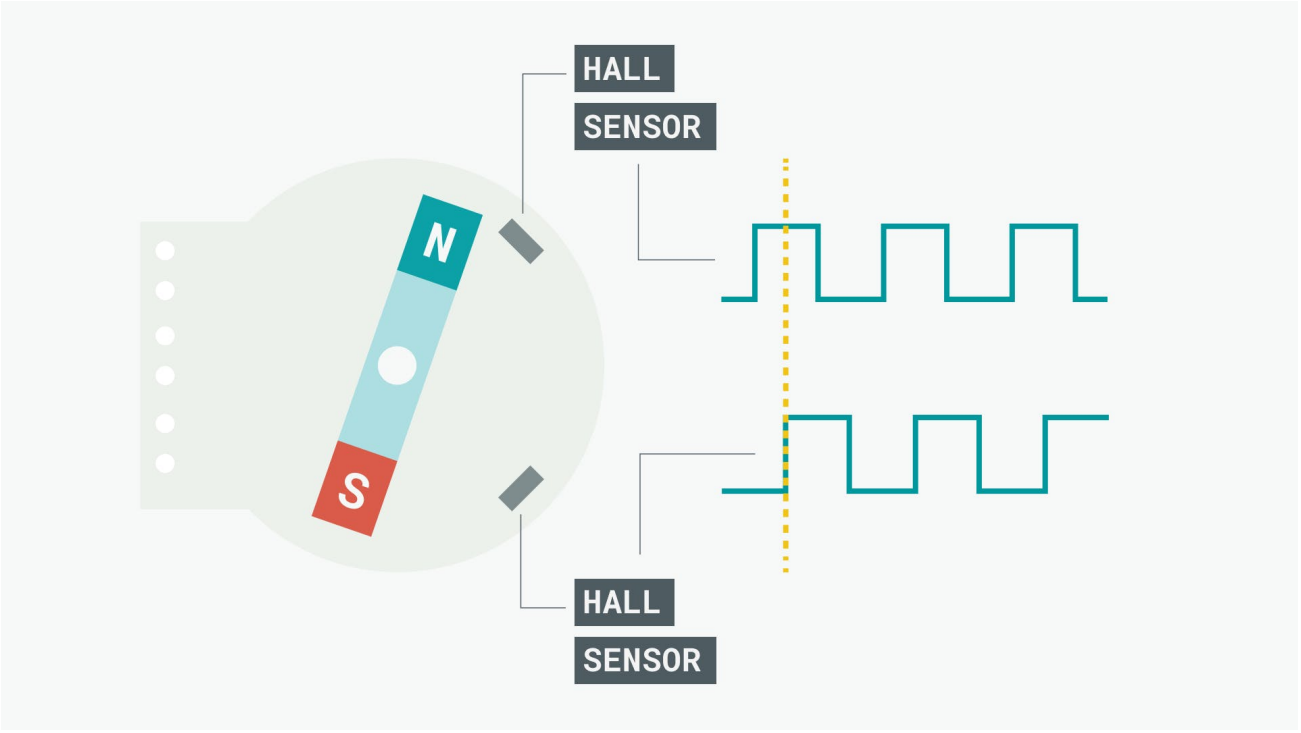
Hall Sensor Based Encoders

A **Hall effect sensor** is capable of detecting the **Hall effect**. This consists of the production of a voltage difference across an electrical conductor when a magnetic field is applied. As the motor turns, the disc rotates past the sensors. Each time a magnetic pole passes a sensor, the encoder outputs a digital pulse, also called a "tick". By counting the frequency of those ticks, the speed of the motor can be determined.



Encoder Output Signals

The encoder has two outputs, one for each Hall effect sensor. The sensors are positioned so that there is a phase of 90 degrees between them. This means that the square wave outputs of the two Hall effect sensors on one encoder are 90 degrees out of phase. This is called a quadrature output.

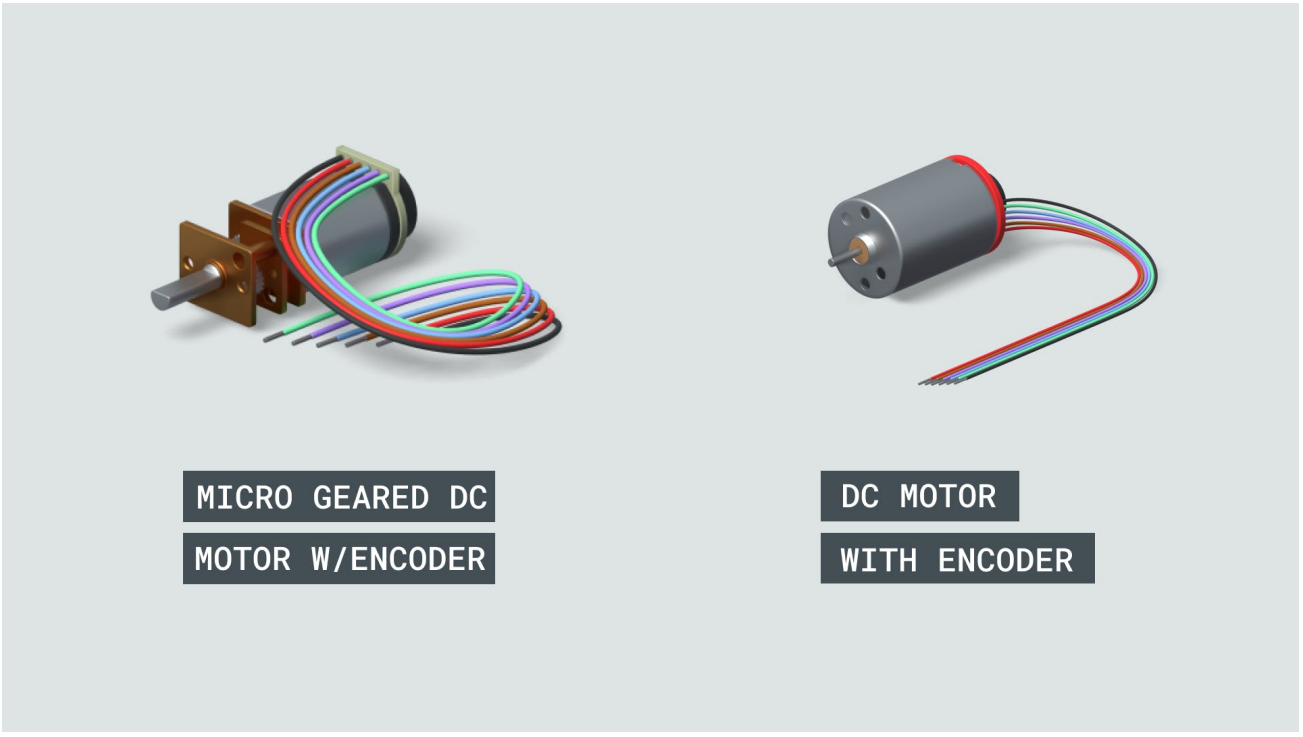


The picture above shows the typical output of an encoder. Having the output pulsing 90 degrees out of phase allows the direction of the motor's rotation to be determined. If output A is ahead of output B, the motor is turning forward. If output A is behind output B, the motor is turning backward.

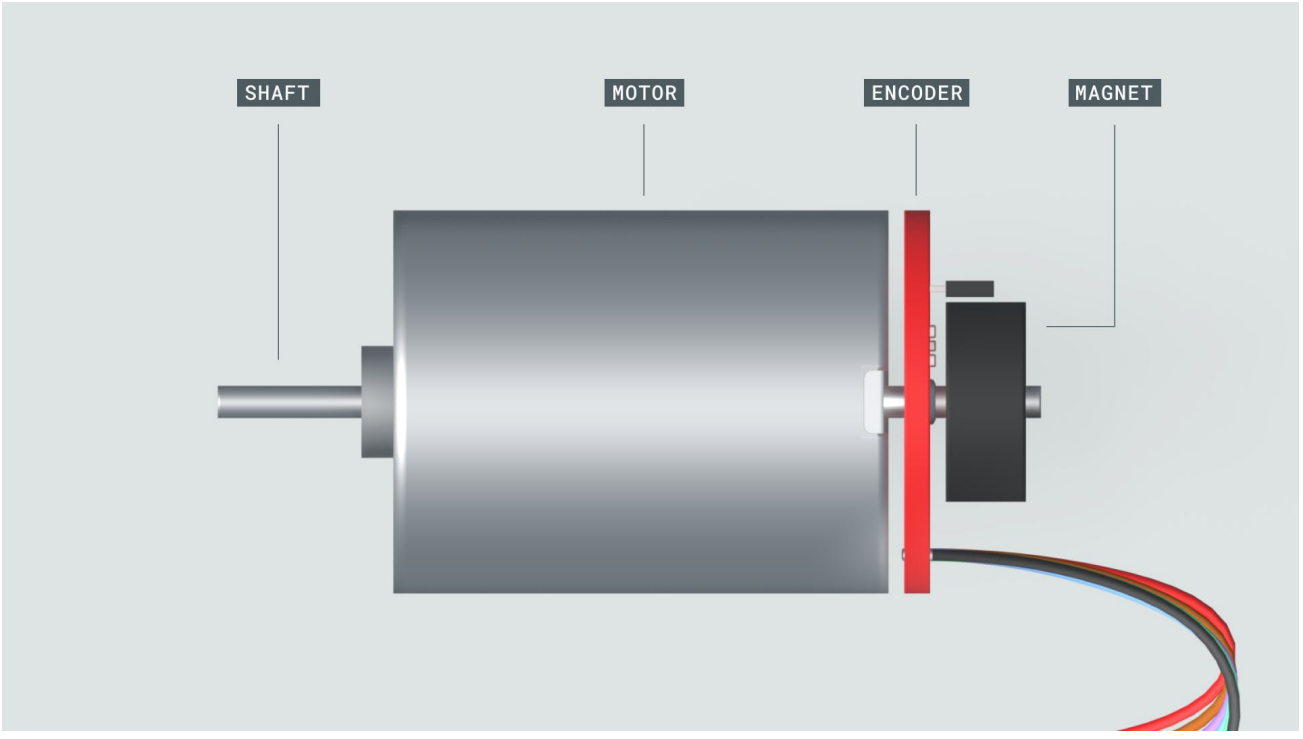
As explained earlier, by measuring the frequency of the pulse signal in A or B (in this case it's not important from which one), we will obtain the speed at which the motor is turning. This information can then be used to obtain linear speed (e.g., the speed of a vehicle).

DC Motors in the Kit

In this kit we will use two micro DC motors with a gearbox (100:1), an encoder and a bigger DC motor. The small geared motors will be used to move the wheels of the rover, to lift the drawing robot, and to drive the motorcycle forwards and backwards. The bigger DC motor (without gearbox) will be used to move the inertia wheel in the motorcycle project.



DC Motor with Encoder



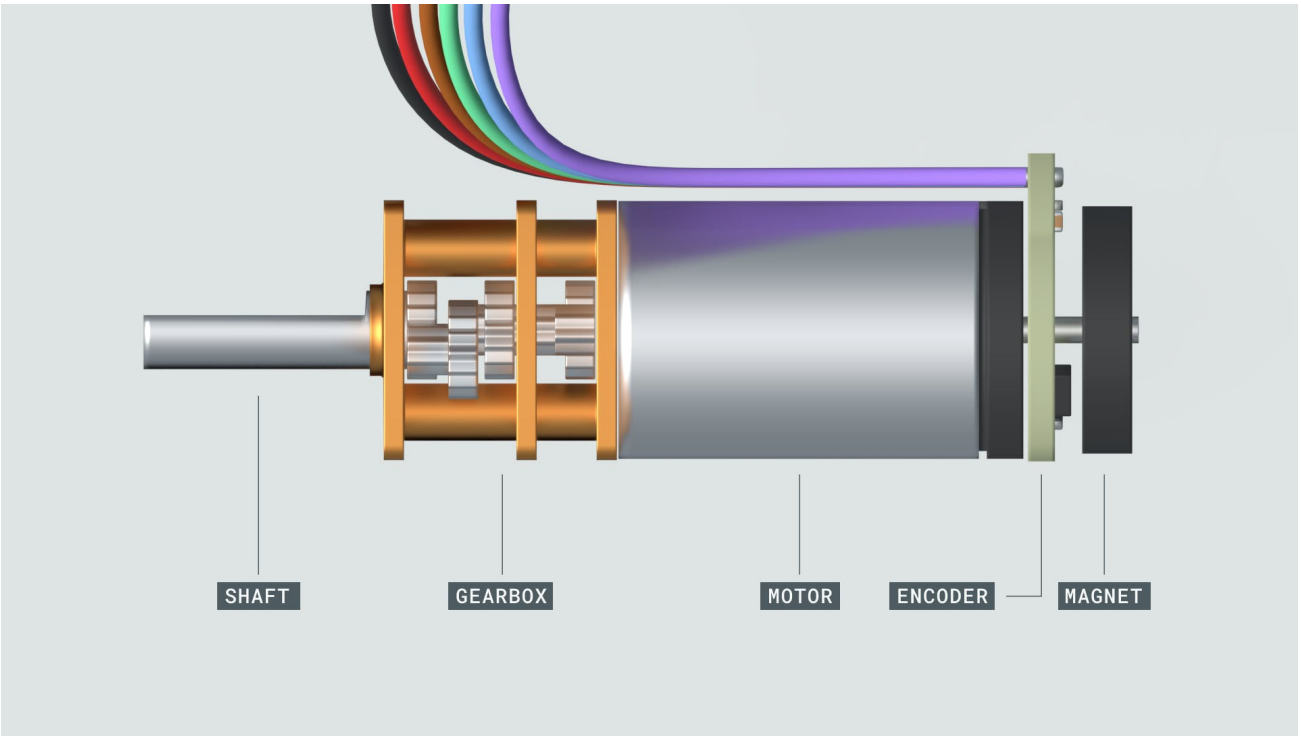
This simple yet powerful DC motor is used to rotate heavier objects like the inertia wheel of the Self-Balancing Motorcycle. It also contains an encoder and a m attached to the shaft of the motor.

Help

The specs of the bigger motor are:

- Speed (No load): 7500 RPM
- Stall torque: 150g-cm

Micro Geared DC Motors with Encoders



For some projects, you might need more torque or the ability to slow down your motor's speed. To achieve this, this micro geared motor comes with a gearbox. Gearboxes come in different reduction rates (e.g., 1:100, 1:1000, etc.). The rule of thumb is the higher the reduction rate the more torque you get, and the slower the maximum resulting speed.

The specs of the micro motors with the encoder are:

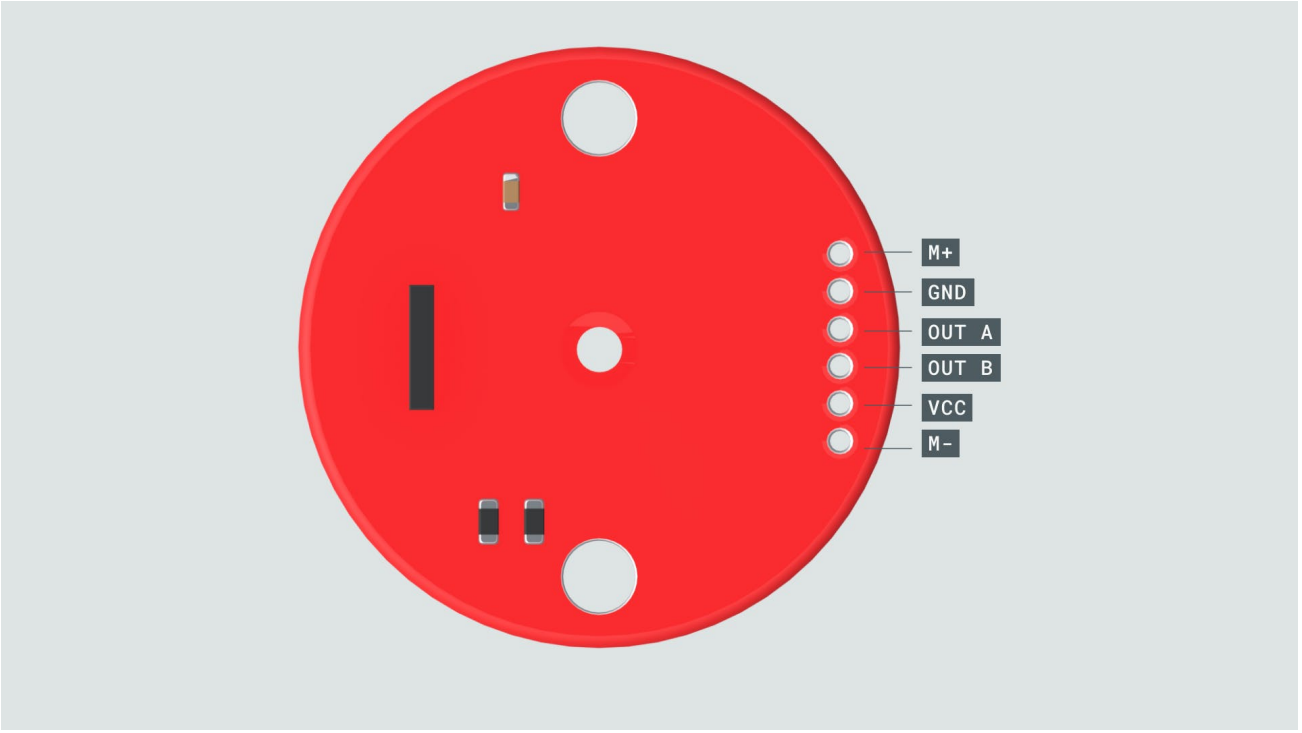
- Speed (No load): 320 RPM
- Stall Torque: 2.2 Kg-cm
- Gear ratio: 100:1
- Ticks per revolution (without gearbox): 12

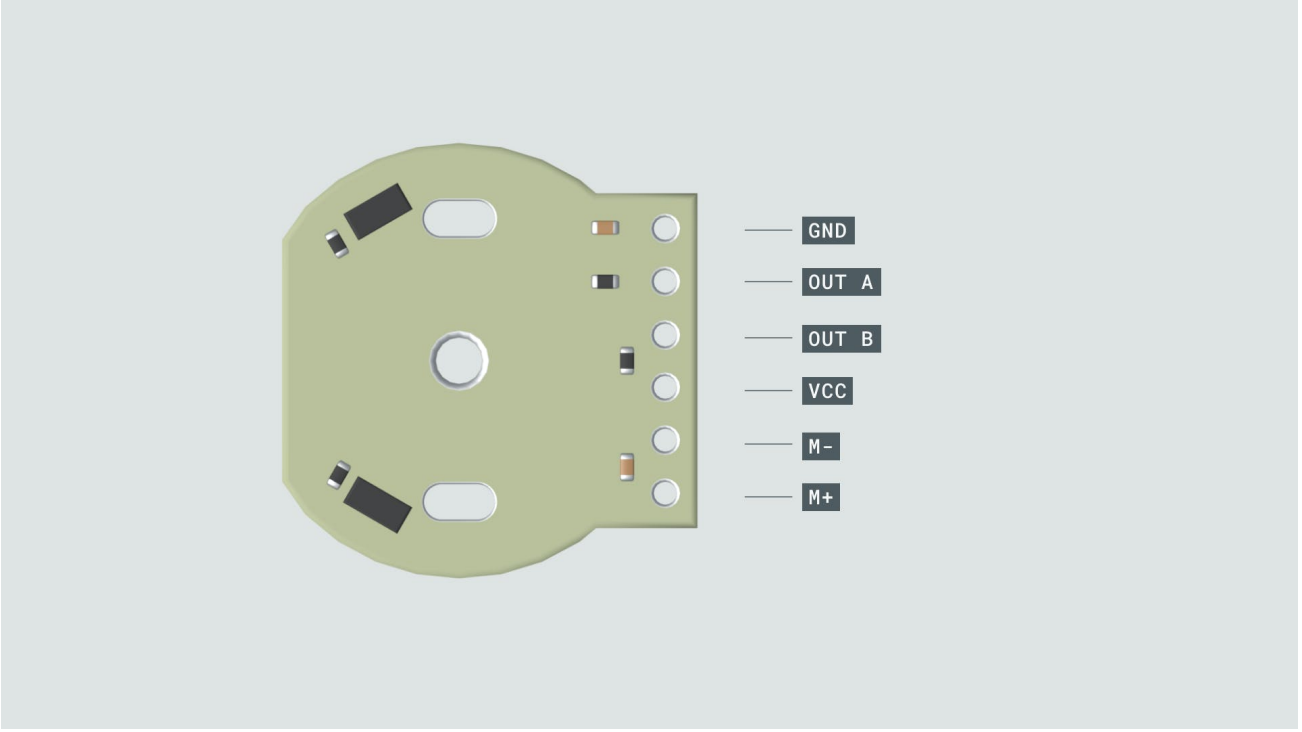
- Ticks per revolution (when accounting for gearbox): 1200

Note: According to the datasheet, the micromotor encoder has 3 PPR (Pulses per revolution) provided by the two hall sensors which are 90 degrees out of phase. Since each pulse has 4 ticks, then each rotation has 12 ticks. However, since the final shaft requires 100 revolutions of the motor shaft to rotate a complete 360°, we need to multiple this by 100 (the gearbox ratrio). Thus, the ticks per revolution for the rotation provided by the final shaft is 1200. This will be the effective ticks per revolution that will be used in the projects.

Pinouts of the DC Motors

There are two pieces of hardware involved in measuring the rotational angle of the motor. First, the rotary encoder hardware, attached to the motor shaft, consists of an integrated circuit, which remains stationary, and a magnetic disk, which rotates along with the motor shaft. As the magnetic poles rotate relative to the chip, they act on two electromagnetics within the chip such that they generate two digital signals A and B with the quadrature form discussed later in this chapter. These signals correspond to the wires labelled OUT A and OUT B on the rotary encoder chip, as shown in the image below. The wires GND and VCC are for ground and voltage input, respectively. They will be connected to a supply voltage source so that current can be provided to generate the A and B signals.





Second, the Arduino Nano Motor Carrier contains a data buffer for each of the two encoder ports. Each time the A or B signal changes from the encoder chip, the value in the data buffer is incremented or decremented by one. The resulting integer value is the encoder count, which we can read any time we need it.

To Learn More

More information about the drivers can be found below.

- **Datasheet for Drivers**

You can see more detailed specs of the motors here:

- **Datasheet DC motor TFK-280SA-22125**
- **Datasheet geared motor with encoder**

3.2 Servo Motors

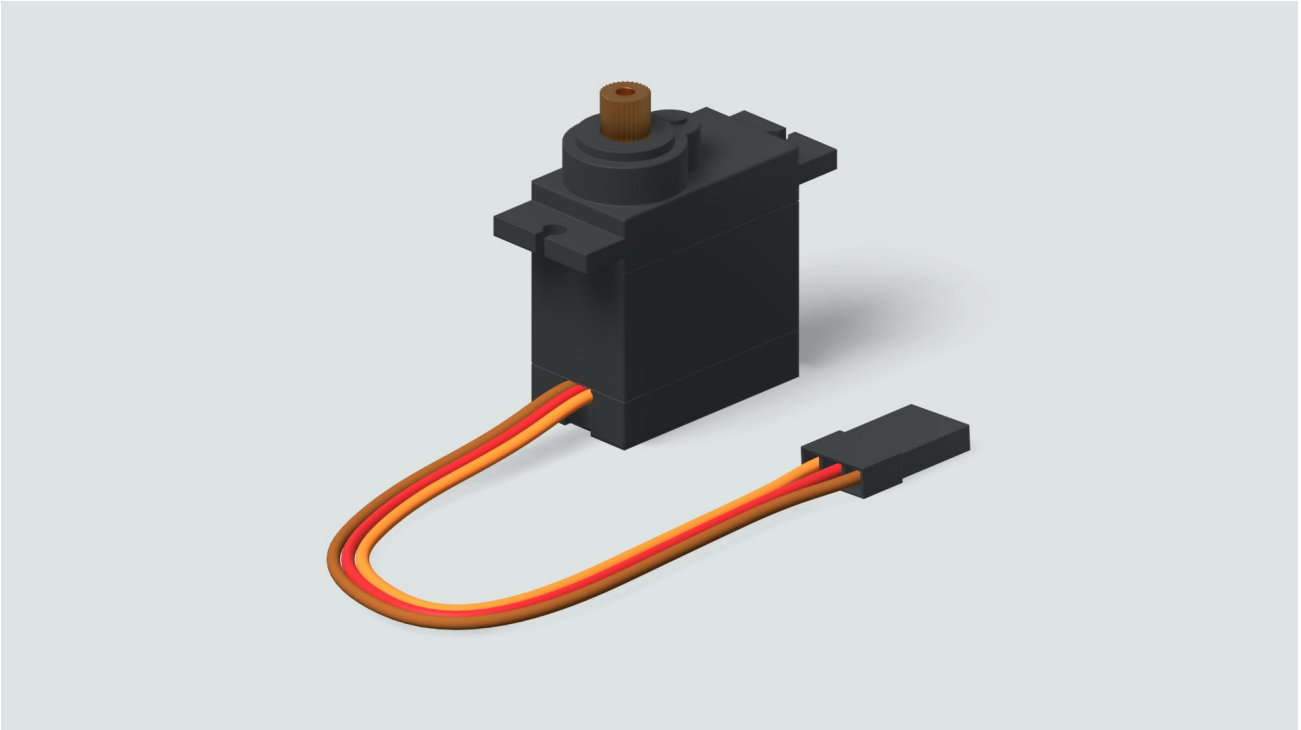
Servo motors are used to steer remote control airplanes by adjusting the wing flaps, flight position for drones, controlling valves used in flow control or continuous drive of wheels for robots. They can be used to position or adjust almost anything you can think of.

This section covers

- What servo motors are
- The different parts of the Servo Motor
- How servo motors work using pulse width modulation (PWM)

The Standard Servo

Servo motors are actuators that allow for precise control of position (angle) or angular velocity from a microcontroller. They have an embedded control circuit inside the housing. This circuit can be analog or digital, and this is determined by the kind of functions the motor is designed to perform. From a functional perspective, there are two types of servo motors: the standard servo and continuous rotation servo. In this kit we will only be using a standard Servo motor to turn within the range of 0 - 180.



A standard servo cannot continuously rotate like a DC motor, so it can't be used to drive a wheel. However, it can be used to move things back and forth at specific angles with high accuracy. For instance, if it is used on the lid of a box, it can open the lid by turning to the 90° position. It can then close the lid by returning to the 0° position.

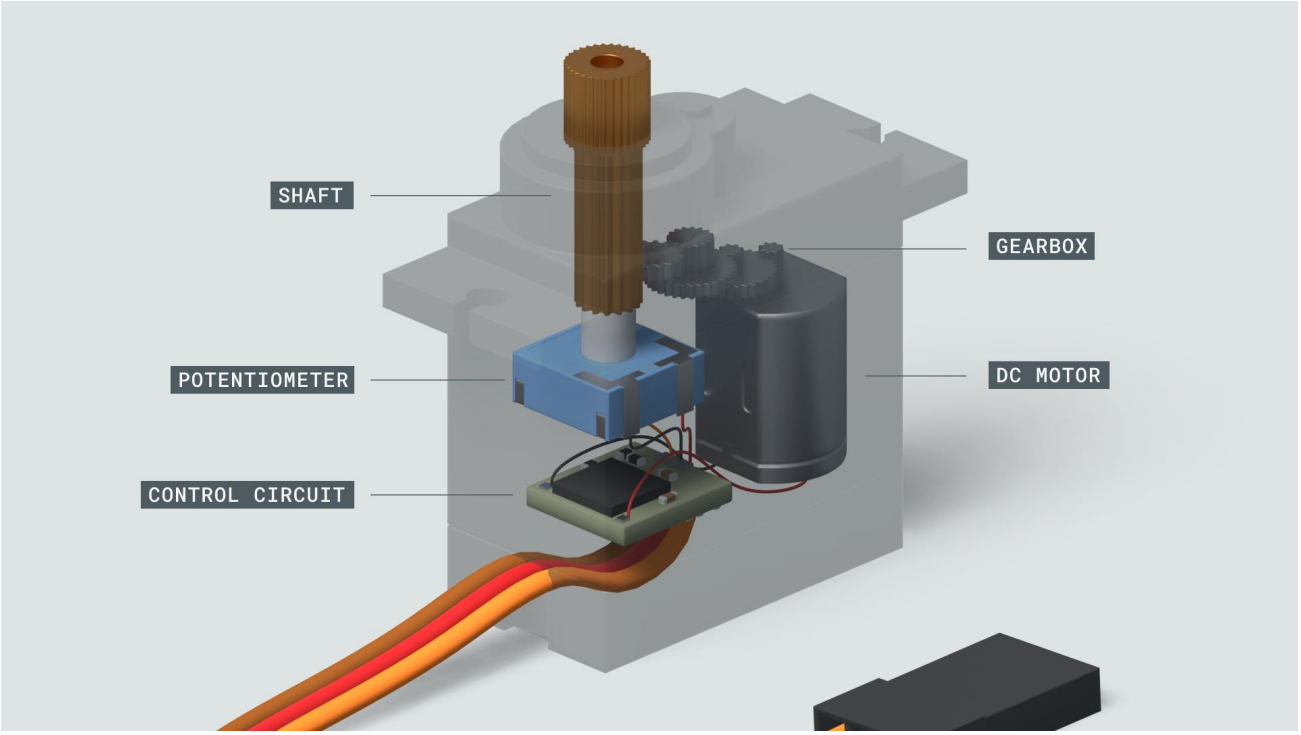
The Arduino Engineering Kit Rev2 comes with one servo motor that will be used in all three of the following projects.

- Self-balancing Motorcycle: the servo motor will be used in the front to steer the motorcycle.
- Webcam controlled Rover: the servo motor will be used for the lifting mechanism in the front of the rover.
- Drawing robot: the servo motor will be used in the mechanism to change the pen color.

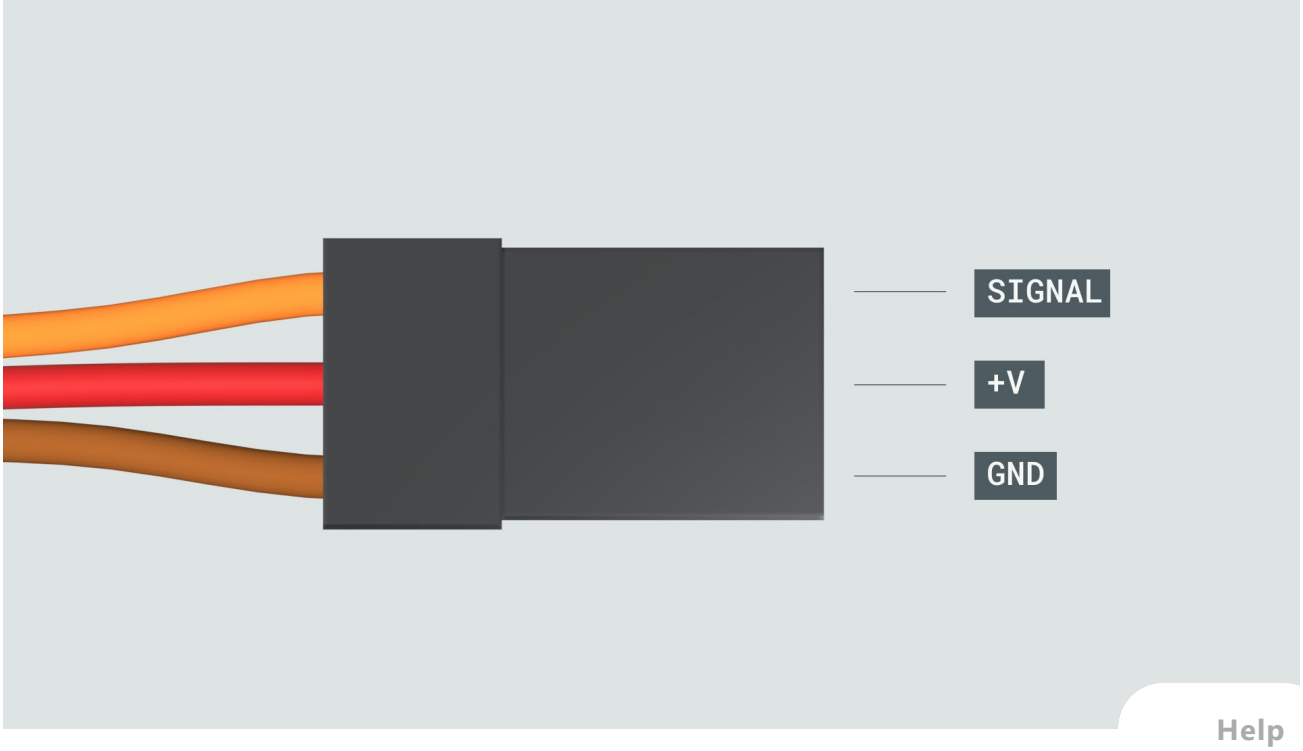
Parts of the Servo

A standard servo actually uses a DC motor, but it also has a gearbox, a posi
sensor, and a control circuit. These components work together to allow the

precisely move from one position to the other.



Servo motors have three terminals, one for ground (GND), one for power (5V) and one for the control signal. It is important to double-check the pinout of your servo motor as it might vary from manufacturer to manufacturer.



Pulese Width Modulation

To control servo motors, we also use PWM signals. However, even if it is the same signal modulation we use for the DC motors, in the case of servo motors it is used in a completely different way. In the case of servos the PWM duty cycle is translated to a specific turning angle for standard servos, and direction and speed for continuous servos.

The width of the pulse of the PWM signal for servo control has to be limited between 1ms and 2 ms as represented in the image below, and every command is repeated every 20 ms. Note that as opposed to normal use of PWM signals, we never use a full duty cycle as it can damage the motor. For this reason In the Arduino IDE, there is a servo library available for your use. This library allows you to send pulses into the servo at the right intervals in the background, thus allowing that Arduino to continue running the remaining code.

To Learn More

More information about servos can be found below.

- **Datasheet of the servo motor**

- The Arduino servo library can be found **here** and **here**

3.3 Inertial Measurement Unit (IMU)

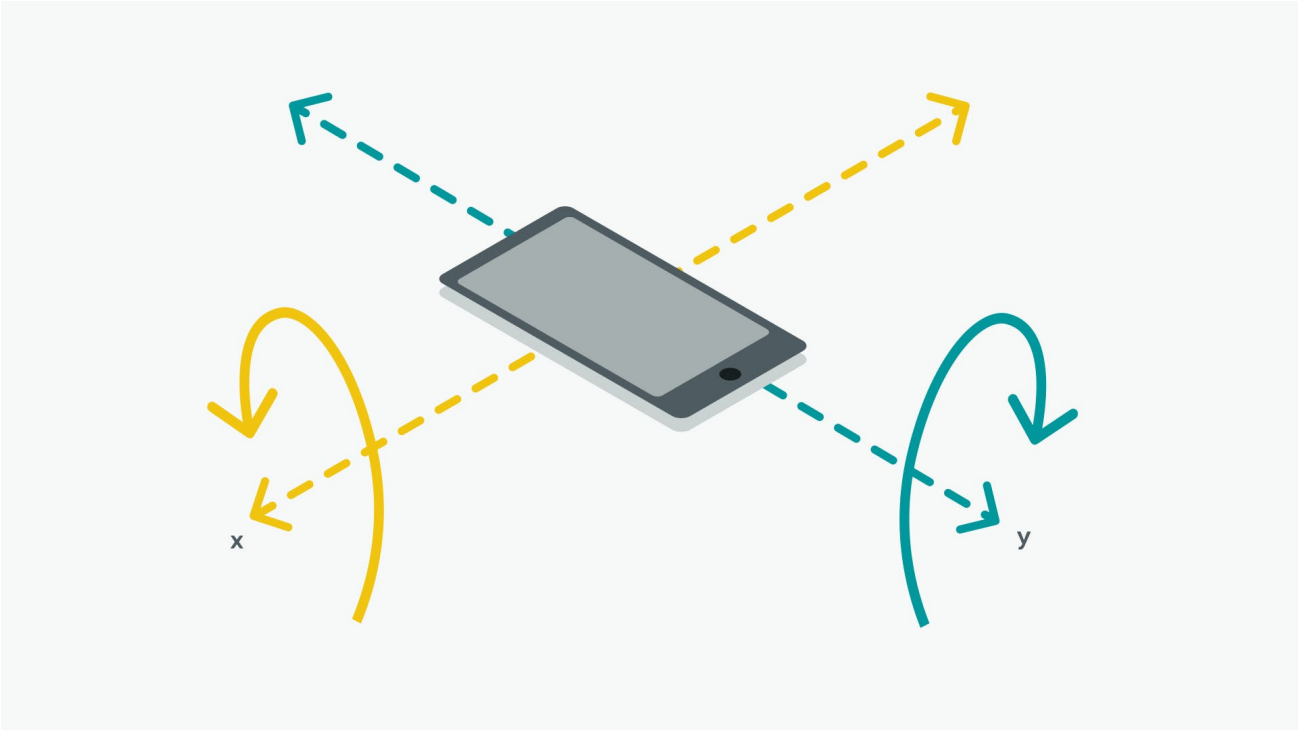
The Inertial Measurement Units are devices used to measure a body's orientation and acceleration using a combination of accelerometers, gyroscope and magnetometers. They are commonly used in mobile phones, wearables, autonomous robots and aircrafts.

This section covers

- Explanation of the IMU and its parts
- Brief description of accelerometers, gyroscope and magnetometer
- Overview of the IMU on the carrier board

The Imu

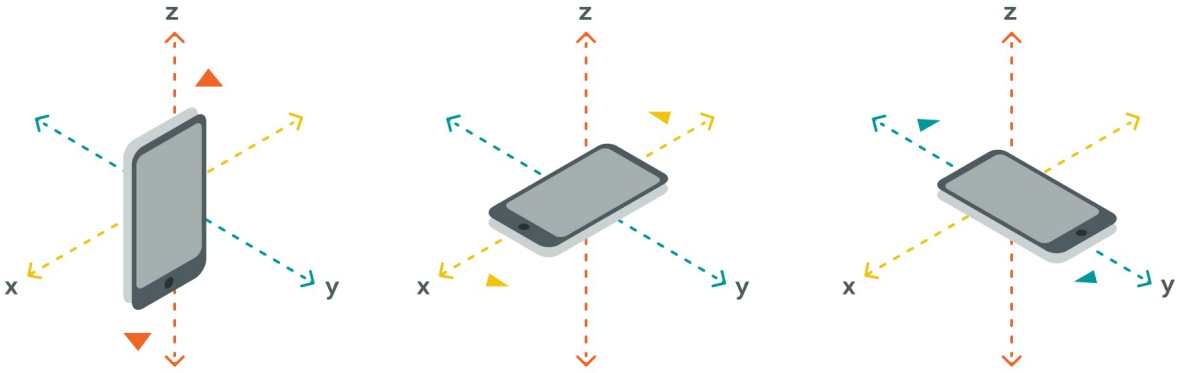
An IMU (Inertial measurement unit) is an electronic module that measures changes in linear acceleration, angular rotation, and, in some cases, the magnetic field around the module. Thus, IMU chips include different sensors, accelerometers, gyroscopes, and magnetometers, in a single housing.



The raw data coming from the sensors on the IMU is then processed and combined into other information that is easier to use in our projects. The information we extract is referred to as: pitch, roll, and yaw. Each one of these measurements correlates to one of the X, Y, or Z axes.

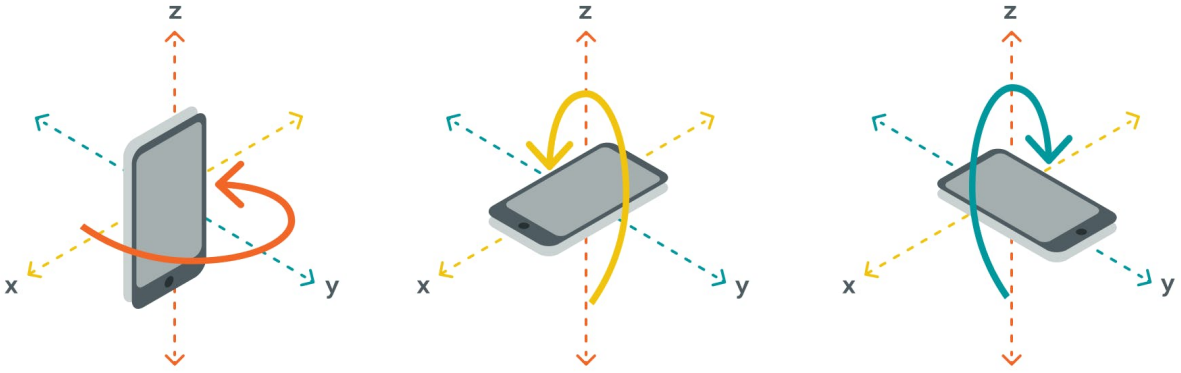
Accelorometer

The accelerometer can measure the speed (linear velocity) and the changes in the object to which it is attached. During stress or through intense vibrations caused by movement, the sensing element generates different amounts of voltage. The movement data is then provided as an x, y, and z value set.



Gyroscope

The gyroscope detects orientation. Using gravity as a basis for measurement, it uses x, y, and z values to provide rotation data. Gyroscopes can be used to measure rotation rate (angular velocity).

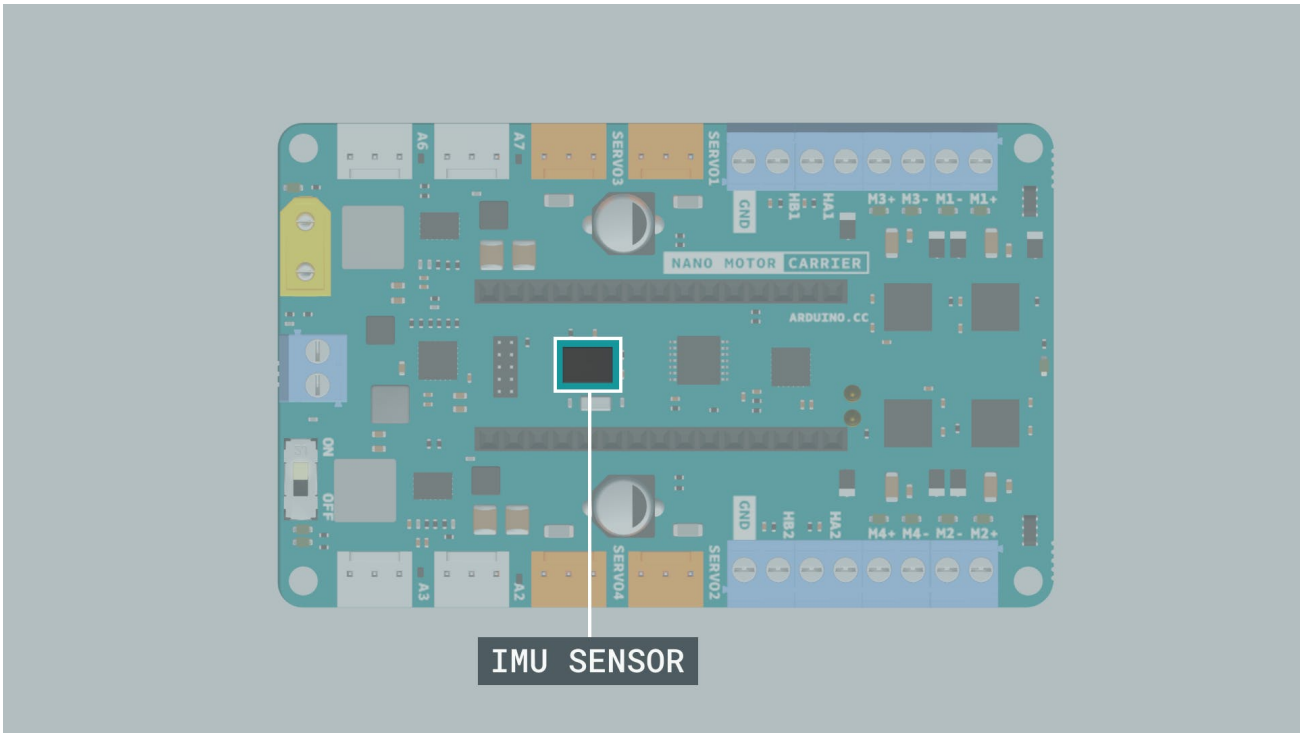


Magnetometer

A magnetometer measures magnetic fields. In IMU modules, they are generally used to detect the earth magnetic field as a way to orient our device like a digital compass. Despite its usefulness, not all IMU modules have a magnetometer. Using one improves the accuracy of the measurements, and the addition of a magnetometer to the module will give a North Magnetic Pole reference point.

The IMU on the Carrier

In this kit, we are going to use an IMU to measure the vertical position of the self-balancing motorcycle and detect when it is falling. More specifically, we will use the IMU on the carrier board, to send the information via I2C. This carrier features the BN0055, a 9-axis (acc+gyro+magnetometer) orientation sensor.



To Learn More

More information about the IMU we used can be found here:

- The filtering processes required to make sense of the raw sensor data c
  found at: **IMU Data Fusing: Complementary, Kalman, and Mahony F**

- To compare the datasheet to the sensor on the IMU Shield: **BNO055 Orientation Sensor Datasheet**.

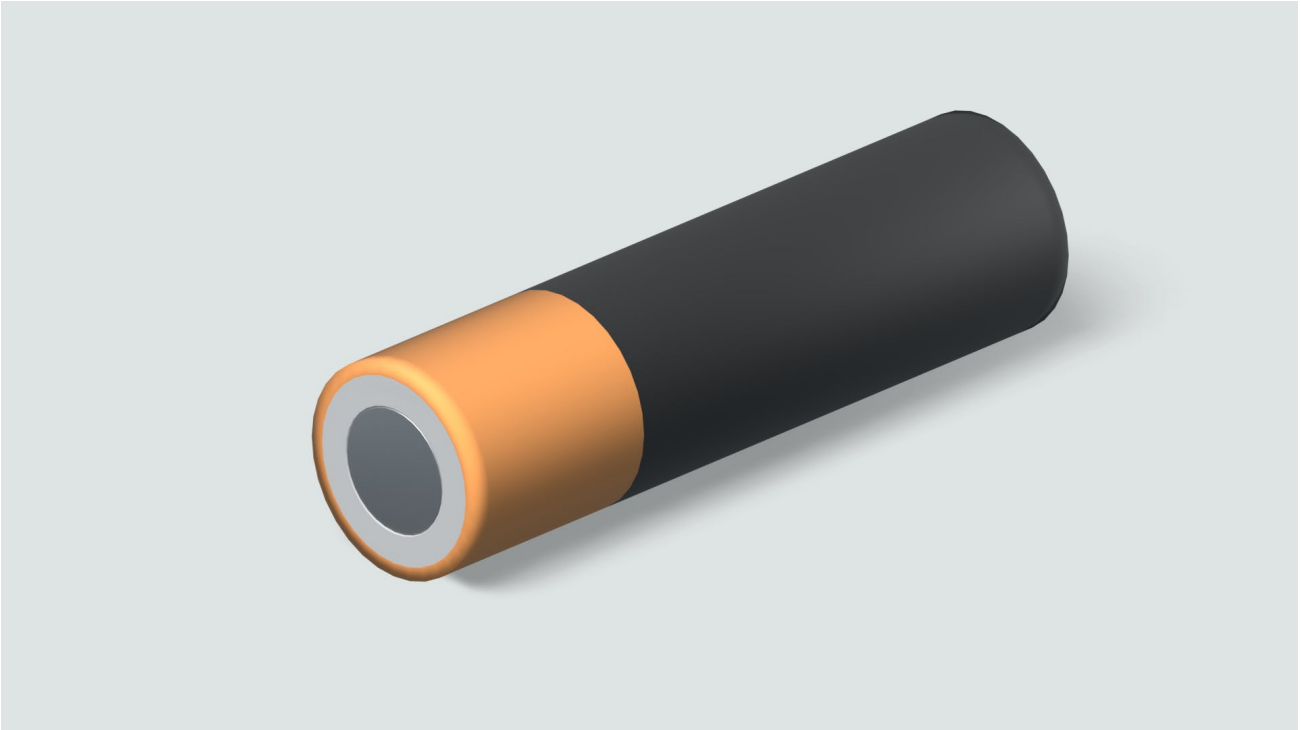
3.4 Battery & Holder

This section covers

- An overview of the Li-ion batteries
- The electrical properties of the battery
- Handling the Li-ion battery
- Tech specs of the batteries included in the kit

Li-ion Batteries

Lithium ion (Li-ion) batteries are a newer type of batteries that are popular for applications where weight and shape are critical, like in tablets and phones. These batteries are based on lithium-ion technology, which uses a semisolid polymer electrolyte instead of a liquid one. This allows these batteries to be made in any size or shape and to be much lighter than other lithium batteries. However, the sensitive chemistry inside the Li-ion batteries makes them more fragile and can lead to fire if the battery gets punctured.



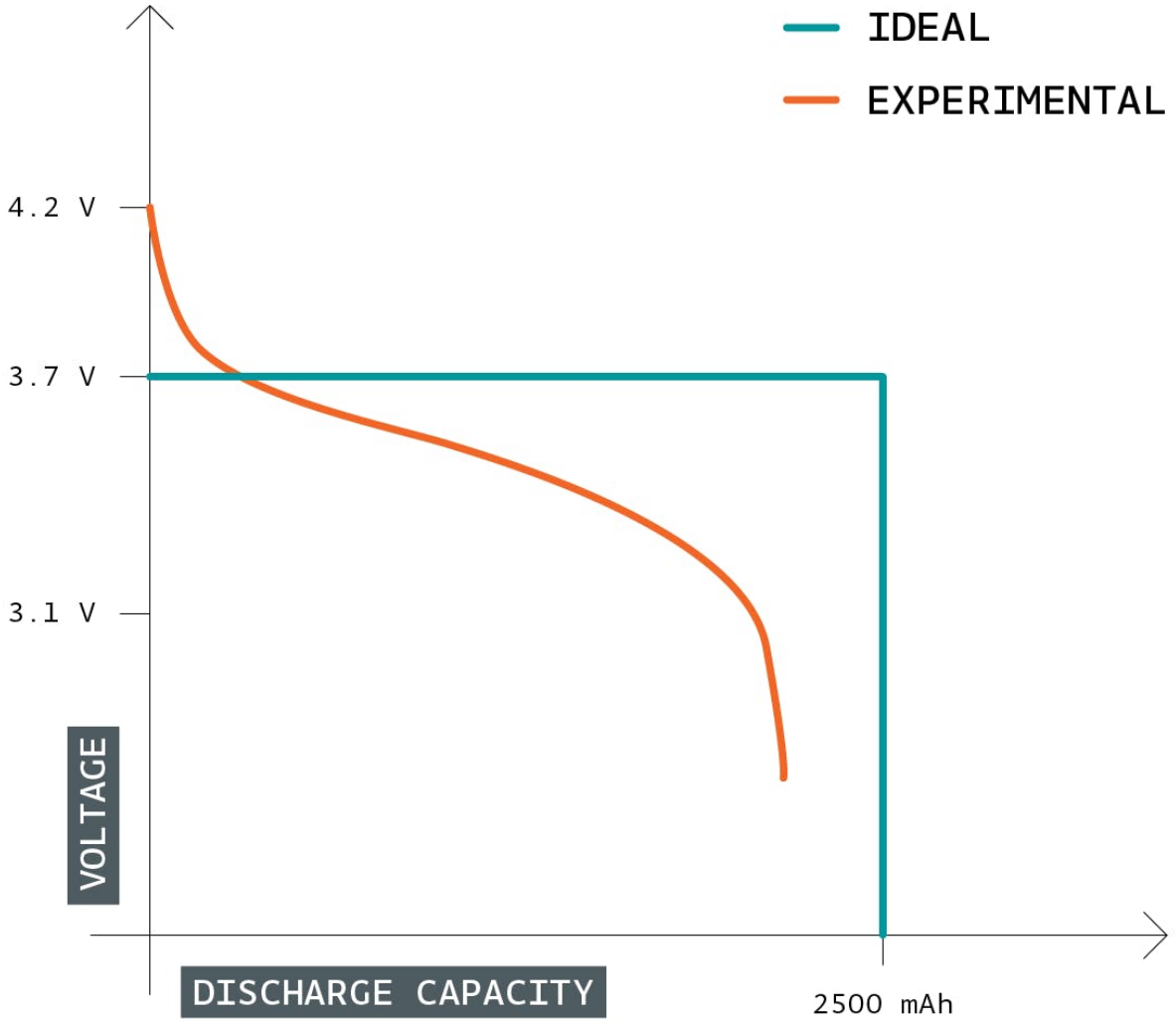
Electrical Properties

We define batteries with a rating system. There are three main ratings on a LiPo battery: voltage, capacity, and discharge rate.

Voltage

Each LiPo cell has a nominal voltage of 3.7 V. In many batteries, multiple cells are added in series. This means that the voltage gets added together. Therefore, batteries with 2 cells (2S) are 7.4 V, batteries with 3 cells (3S) are 11.1 V, and so forth.

As a matter of note, the nominal voltage is the default (or ideal), resting voltage of a battery pack. This is how the battery industry standardizes and compares batteries. However, this does not represent the full charge voltage of the cell. LiPo batteries are fully charged when they reach 4.2V/cell, and their minimum safe charge is 3.0V/cell. 3.7V is a median value, and that is the nominal charge of the cell. A representation of both the ideal (blue) and experimental values (orange) for voltage as a function of discharge capacity is shown in the figure below.



Note: Many battery controllers have protection for over- and under-charging conditions.

Capacity

The capacity of a battery is the measurement of how much power the battery can hold, much like the size of the fuel tank in a car. The unit of measure for the capacity is in milliamp hours (mAh), which represents how much current can be extracted from the battery to discharge it in one hour.

Consequently, the capacity determines how long a battery can run before it needs to be recharged. Extreme temperatures and high current draw can reduce the effective capacity.

Discharge rating

In many 2S/3S battery datasheets you also see a C which represents the discharge capabilities of the battery within the linear range of operation. The maximum current can be calculated as follows:

$$\text{maximum current draw} = \text{battery capacity} \times \text{discharge rating}$$

For example if the discharge rating is 5C and the capacity of the battery 2500mAh then we can obtain the maximum peak current of the battery as follows:

$$12,500\text{mA} = 2500\ \text{mAh} \times 5\ \text{C}$$

In some cases, the datasheet may not include the discharge rate C, but instead present the maximum continuous current directly. In practise, the discharge rating is a function of the operating conditions and falls over the lifetime of the battery.

> **Note:** The discharge rating is a function of the operating conditions and falls over the lifetime of the battery.

Handling the Batteries

As its name indicates, lithium-ion batteries contain lithium. Lithium is an alkali metal, which means that it reacts with water and combusts. Lithium also combusts when it reacts with oxygen, but only when heat is applied. The process of using the battery causes excess oxygen and lithium atoms to accumulate on either end (cathode or anode) of the battery. This can cause lithium oxide Li2O to build up on the anode or cathode. The Li2O causes the internal resistance of the battery to increase. The result of higher internal resistance is that the battery will heat up more during use.

As a note, internal resistance is best described as the measure of opposition a circuit presents to the passage of current.

Battery Life

The lithium oxide build-up usually takes around 300-400 charge/discharge cycles before a tipping point is reached. Therefore, that is the typical life of a Li-ion battery. However, if we heat the batteries up during a run, discharge them at less than 3.0 volts per cell, physically damage them in any way, or allow water to enter the batteries, the battery life is reduced. This hastens the buildup of Li2O.

Safety Tips

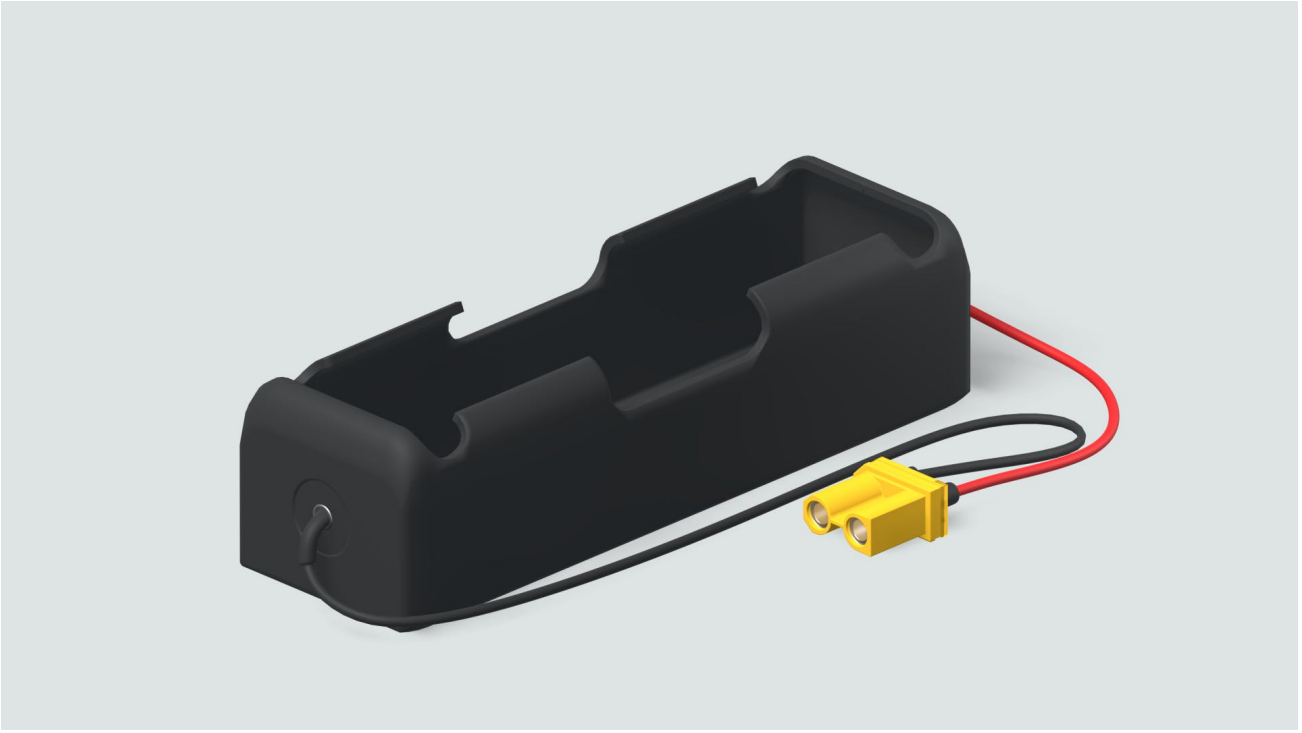
Li-Ion batteries must be handled with care. Discharging the battery under the minimum safe charge can cause irreparable damage to the battery. Therefore make sure to keep track of the battery voltage level in your projects (e.g. adding safety measures in your code or circuit) to avoid harming your battery. In addition, overcharging it can cause the battery to ignite. For this reason, Li-ion batteries should always be charged with a specialized charger, and you may never leave your Lipos unattended during charging.

Battery in the Kit

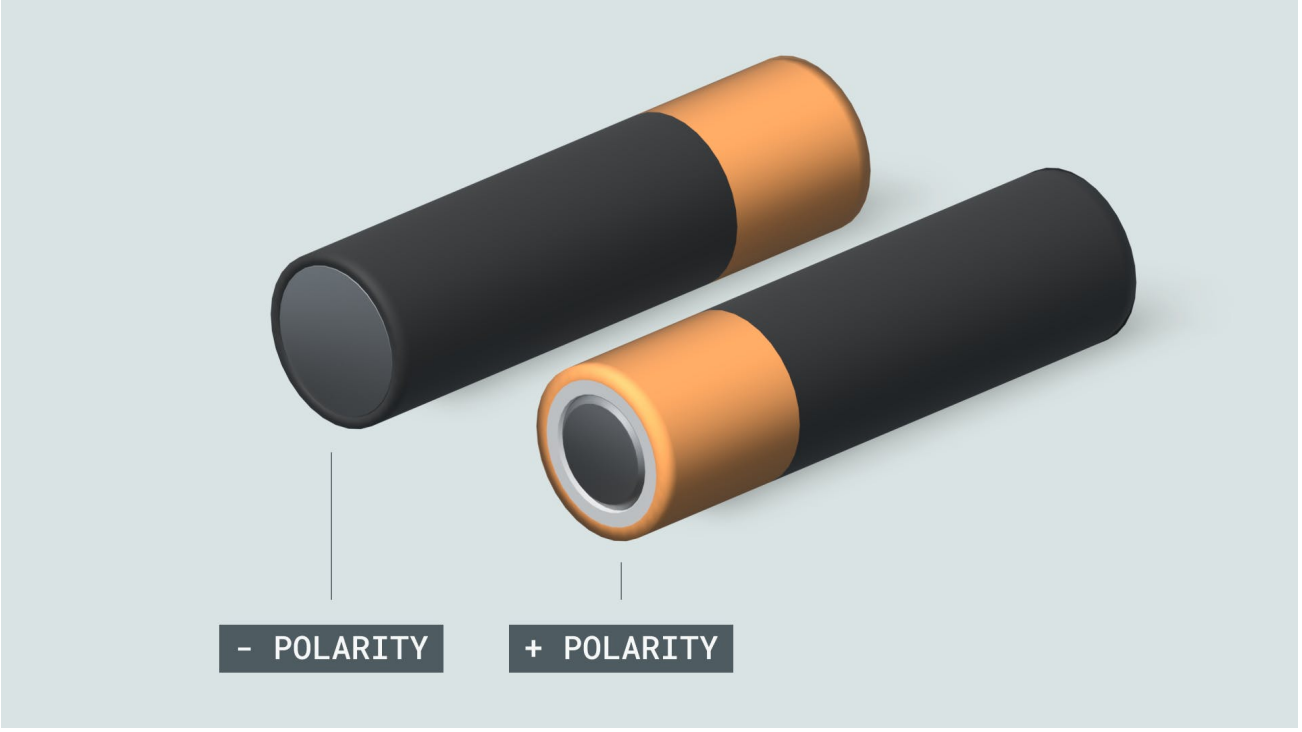
- **2500 mAh** is the capacity of the battery. It represents how much current a fully charged battery can provide during one hour before it discharges.
- **3,6 V** is the nominal voltage of the battery. The Arduino Nano Motor Carrier has a boost converter circuit that allows us to power the motors that requires a higher voltage than the battery has.
- **20A** is the maximum current at which the battery can be discharged at a specific moment without damaging the battery or reducing its capacity.
- The battery weighs approximately **43.8 g** so as to lessen its impact on the project weight, and it is **rechargeable** through the Arduino Nano Motor Carrier.

The datasheet can be found **here**.

Attaching the Battery to the Holder



When attaching the battery to the holder, the most important thing to remember is to look at which orientation the battery should have. You can see the correct orientation by looking at the shape of the battery and the connections in the holder.



3.5 Running a DC Motor

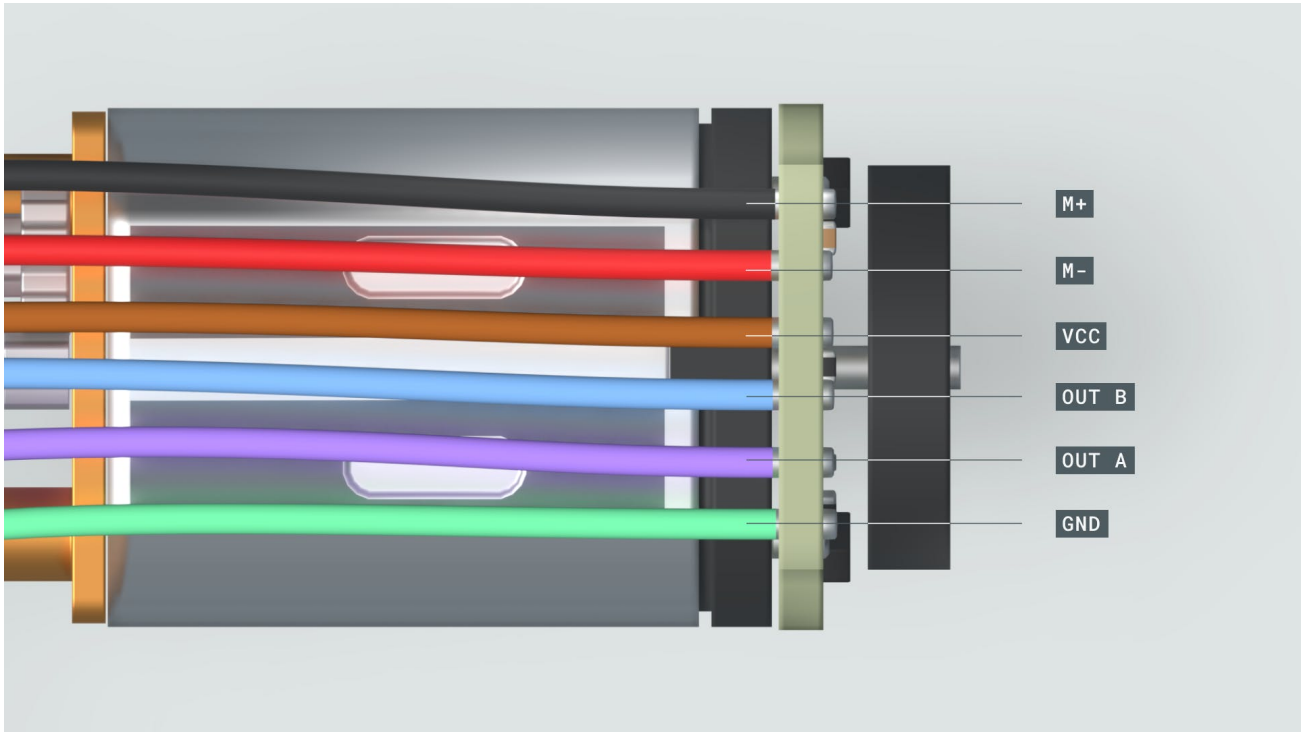
In this section, you will use the **MATLAB Support Package for Arduino Hardware** to access the Arduino Nano 33 IoT board and two of the external devices that come in the Arduino Engineering Kit Rev2: the 100:1 Gearhead DC Motor and the Magnetic Encoder, which comes already connected to the motor's drive shaft. From the MATLAB environment, you will learn how to drive the motor at specific voltages and sense the rotational motion of the motor through the rotary encoder.

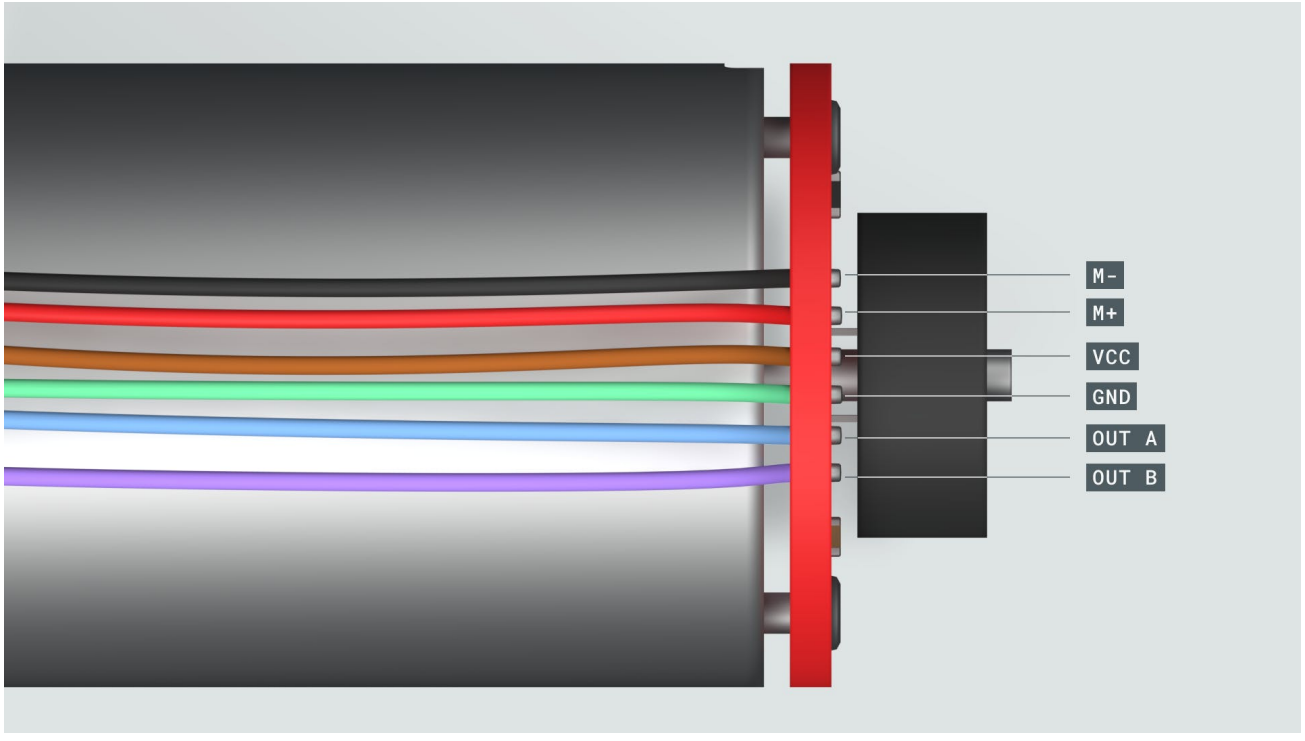
In this section, you will learn:

- About the pinouts of the motor cables,
- How to interface the 100:1 gearhead DC motor to the Arduino Nano Motor Carrier,
- How to control the motor through MATLAB and Arduino,
- About pulse width modulation (PWM) signal structure, and how it's used to drive a DC motor and other devices,
- About the mechanism by which a rotary encoder measures rotational angle,
- How to configure and manipulate digital pins on Arduino Nano 33 IoT,
- How to configure and drive a DC motor from MATLAB,
- How to configure, read, and process data from a rotary encoder in MATLAB.

The Motor Cables

Before you learn to run the motor, it's important to know what the different cables on the motor header are. The provided motors have a pre-attached rotary encoder integrated circuit that we will discuss later. The rotary encoder chip has a 6-wire header:





Note: In this case we are not following the naming convention for the cable colors. In the coming chapters, just follow the illustrations to know where to connect the cables.

Help

The four wires called **GND**, **OUT A**, **OUT B** and **VCC** are related to the rotary encoder itself. The two wires labelled **M+** and **M-** connect directly to the motor drive leads, which are hidden under the rotary encoder chip. Locate the ends of the two motor drive wires.

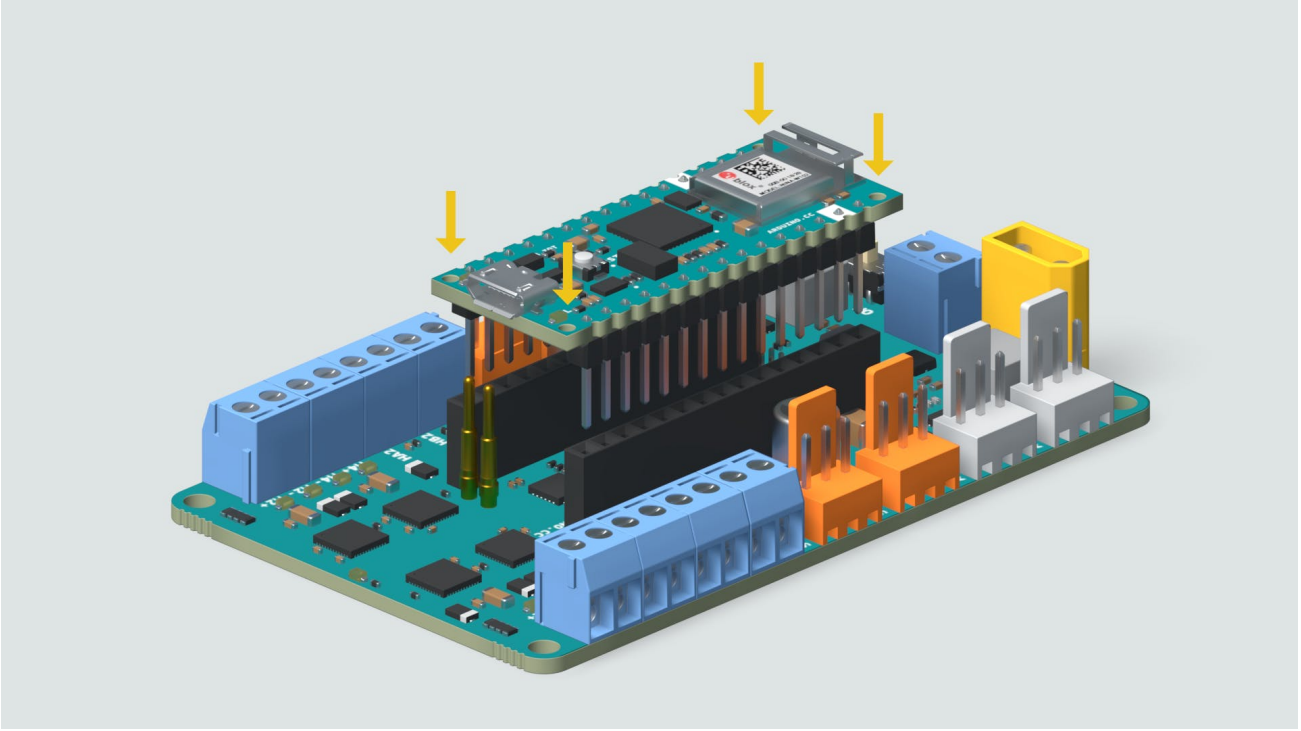
Torque is delivered to the motor by applying a voltage across these two wires. The magnitude of the voltage corresponds to the amount of torque applied, and the sign of the voltage is analogous to the direction of the applied torque.

The Basic Setup

Make sure you disconnect the USB cable between Arduino Nano 33 IoT and your computer, ensure that the Motor Carrier's switch is in OFF position and the battery is placed correctly on the battery holder.

Attach the Arduino Nano 33 IoT

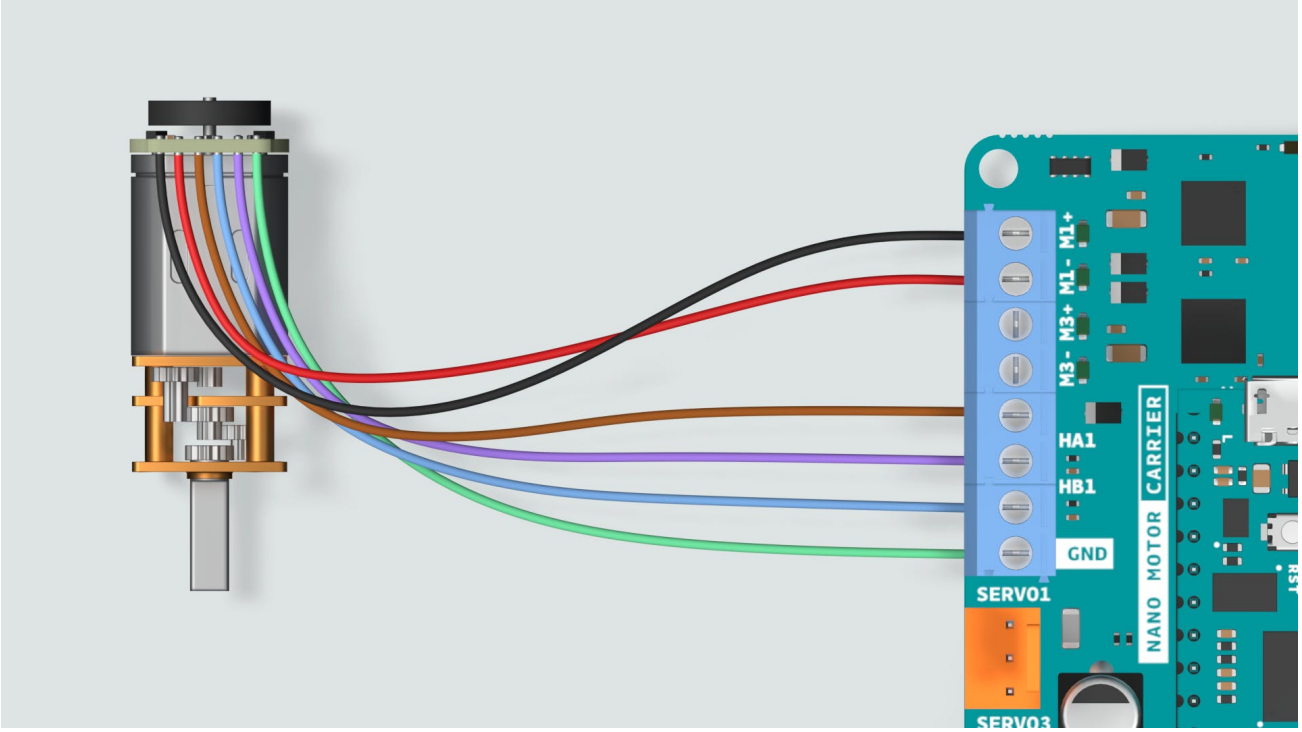
Then attach the Motor Carrier to the Arduino Nano 33 IoT board. You can do this by aligning the corresponding pin labels and pressing down on the board firmly.



Note: It is strongly recommended that to perform any operation of mounting or removing parts from a circuit, you always disconnect them from both power sources (battery and the USB cable).

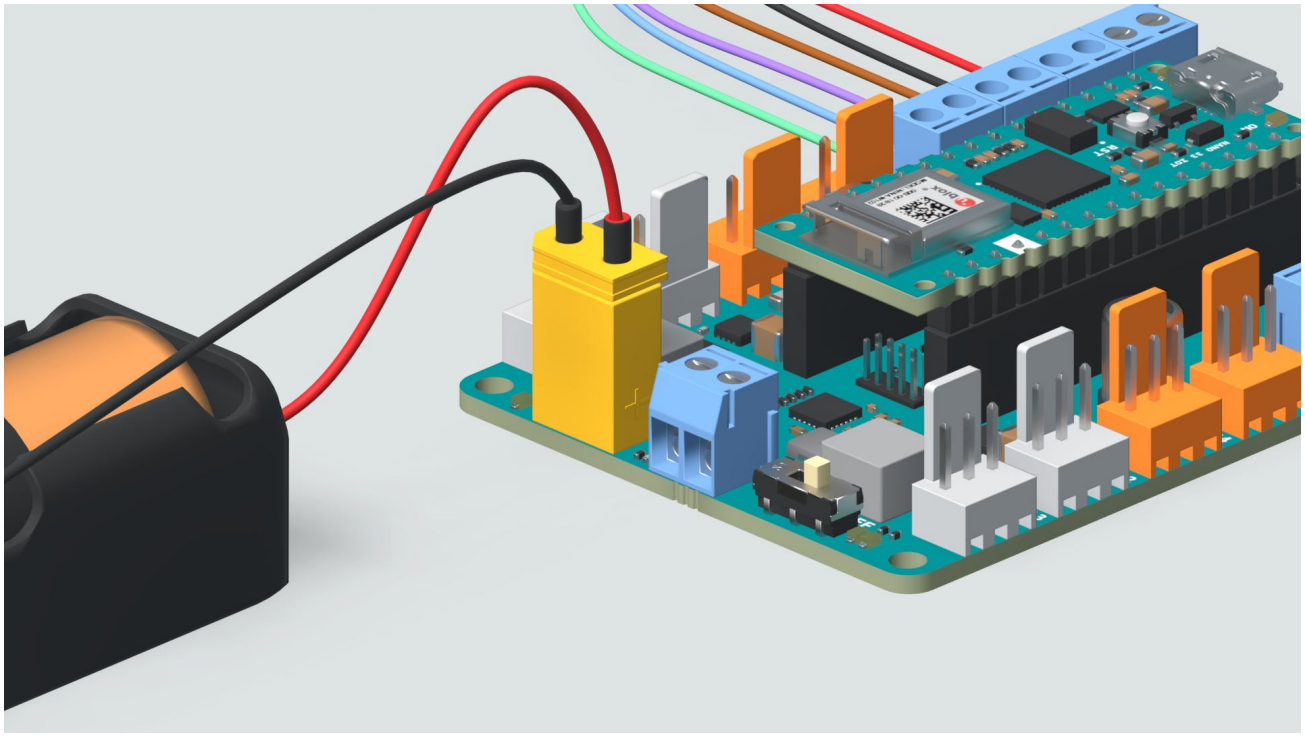
Attach the Motor

On the Motor Carrier, locate the headers for DC motor **M1**. You should see the labels **M1+** and **M1-** printed on the Motor Carrier board next to the corresponding screw terminals. Connect the remaining cables to the corresponding terminals as shown in the figure below.

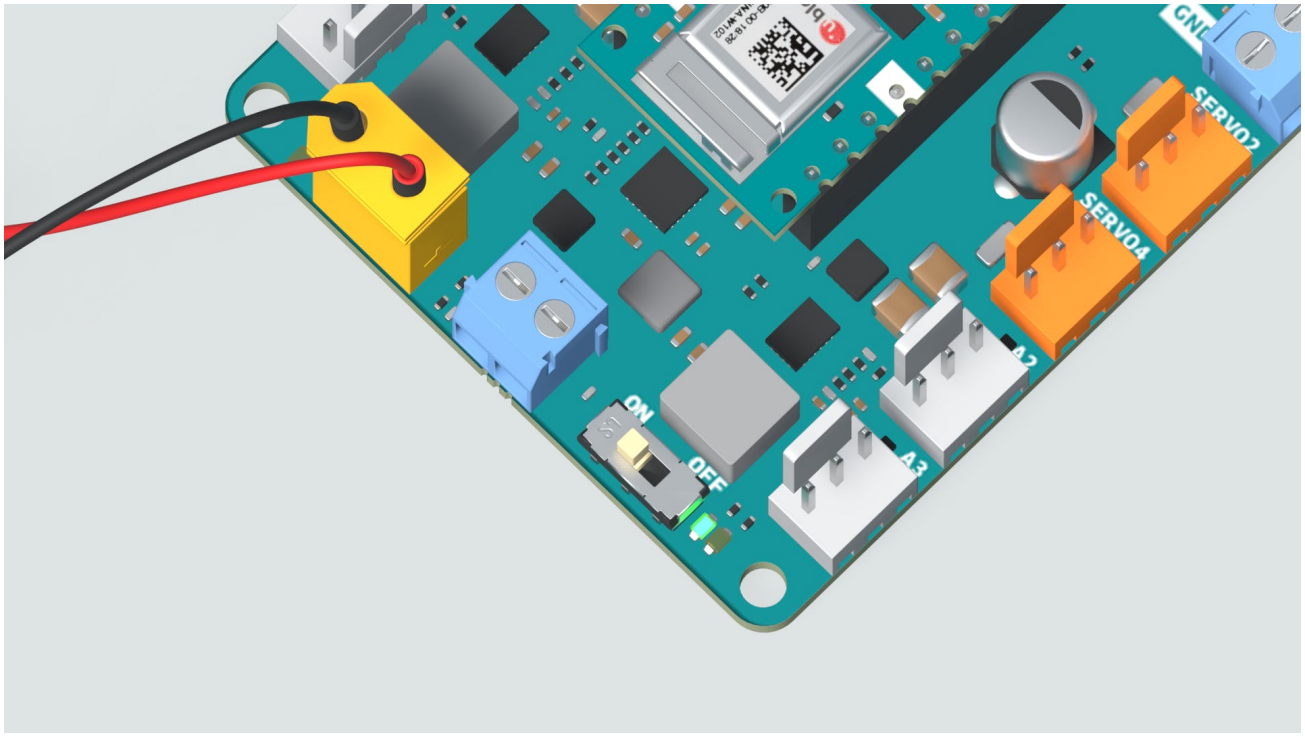


Connect the Battery

Now locate your LiPo (Lithium Polymer) battery and attach it to the battery header on the Motor Carrier as shown. Ensure that the black wire is aligned with **GND** and the red one is aligned with VIN to ensure the correct polarity:



Plug in the USB cable again, switch the Motor Carrier power switch to ON, and ensure that the nearby LED illuminates.



Note: it is strongly recommended that to perform any operation of mounting or removing parts from a circuit, you always disconnect them from both power sources (battery and the USB cable).

Running the Motor

Next, you will access the Arduino Nano Motor Carrier through MATLAB so that you can use the DC motor and rotary encoder. You will need access to the Arduino Nano Motor Carrier, which is a board that interfaces the Arduino Nano 33 IoT with up to four DC motors, up to two rotary encoders, and up to four servo motors, as explained earlier. In this example we will use one DC motor and one rotary encoder.

Note: If you haven't done the basic setup, we recommend you to complete the Basic Setup step before you continue with the MATLAB setup.

Initialise the Arduino object

Open MATLAB and ensure that the Arduino Nano Motor Carrier board's power source is switched ON. You must re-establish your MATLAB connection to Arduino. Execute the following commands to do so:

```
>> clear a
>> a = arduino
```

Initialize the Motor Carrier Object

Now let's create a second MATLAB object to provide access to the Arduino Nano Motor Carrier. Use the following command to create a Carrier object in the MATLAB Workspace that is associated with the Arduino object a :

```
>> carrier = motorCarrier(a)
```

Just as the Arduino Nano Motor Carrier provides a physical interface between the motor drive wiring and the Arduino Nano 33 IoT, the **carrier** object is an intermediary between the Arduino object and the DC and servo motors we may connect.

Controlling the DC Motor

Now let's create a third object to give you control of the motor connected to **M1** in MATLAB. Use the following command to create a DC Motor object that is associated with the Carrier object, and examine the displayed object properties in the **Command Window**:

```
>> dcm = dcmotor(carrier,'M1')
```

Now you can drive the motor. You can see that the **dcmotor()** object has three properties: `MotorNumber`, `Speed` and `Running`. You can change the `Speed` property directly by assigning values between -1 and 1. You can control the `IsRunning` property using the methods **start** and **stop**. Try the following commands to control the motor speed:

```
>> start(dcm)
>> dcm.Speed = 0.5;
>> dcm.Speed = 0.3;
>> dcm.Speed = -0.3;
>> dcm.Speed = -0.1;
>> dcm.Speed = -0.05;
>> dcm.Speed = -0.01;
>> dcm.Speed = -0.3;
>> stop(dcm)
>> start(dcm)
>> stop(dcm)
>> dcm.Speed = -0.5;
>> start(dcm)
>> stop(dcm)
```

The voltage applied to the DC motor is controlled by a PWM signal. The magnitude of `dcm.Speed` indicates the duty cycle of the PWM signal. When `dcm.Speed` is positive, the PWM signal multiplies with the battery's potential difference to produce some positive fraction of the battery's voltage rating. When `dcm.Speed` is negative, the same multiplication occurs, but the "reference" and "ground" voltages are reversed in the circuitry. As a result, a negative fraction of the battery's volt
is applied to the motor and hence a negative torque is obtained.

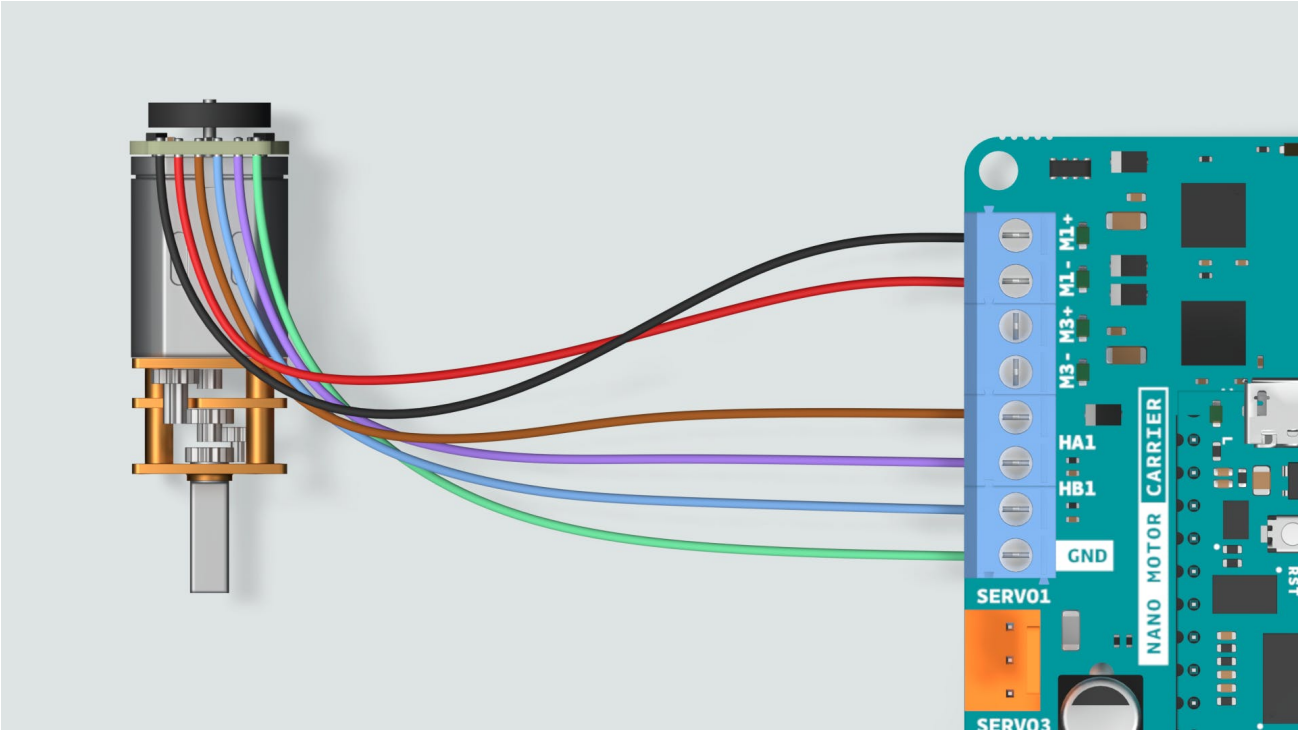
You may have observed that there is a **dead band** when `dcm.Speed` is very close to 0. This is because the applied torque is not large enough to overcome the static friction in the various axles in the gearbox. This is something that should be considered when designing a motor control system.

Controlling the Position and Speed

The position and the speed of the DC motor can be controlled by using values read through the Rotary encoder attached to the motor. These values are generated by the signals produced by the HAL sensor. These signals correspond to the wires labelled **OUT A** and **OUT B** on the rotary encoder chip. The wires **GND** and **VCC** are for ground and voltage input, respectively. They will be connected to a supply voltage source so that current can be provided to generate the A and B signals.

> **Note:** If you haven't connected your DC motor to the Arduino Nano Motor Carrier, go through the steps in The Basic Setup section before you proceed.

Switch the Arduino Nano Motor Carrier power to OFF and disconnect the USB cable. Attach the encoder wires to the corresponding screw terminals for encoder port 1 (labeled 5V, HB1, HA1, and GND).



Create a MATLAB Object

Connect the USB cable and power on the Arduino Nano Motor Carrier and then create a MATLAB object to access the rotary encoder count buffer at encoder port **1**, using the following commands:

```
>> clear a carrier
>> a = arduino
>> carrier = motorCarrier(a)
>> dcm = dcmotor(carrier,'M1')
>> enc = rotaryEncoder(carrier,1)
```

To read the encoder count buffer, use the following command, and note the result:

```
>> readCount(enc)
```

The geometry of this encoder is such that there are three full cycles of quadrature when the motor shaft turns one full revolution. Recall that the quadrature signals undergo four total changes in a full cycle. This means that there are 12 quadrature signal changes per revolution for the motor shaft. Thus, we can measure th position of the motor shaft with a resolution of 30 degrees, if we know the

count. Manually rotate the magnetic disk one full clockwise revolution (when looking down on the magnetic disk), and read the encoder count again:

```
>> readCount(enc)
```

What is the change in encoder count? If your encoder was wired as instructed, you should have seen the count increase between readings. If the **OUT A** and **OUT B** pins on the encoder were wired the opposite way, the count would have decreased between readings. From a purely electrical point of view, there is no right or wrong way to wire the encoder; you just need to be aware of this fact and take it into account when you calibrate the sensor.

Just to confirm that everything works as expected, rotate the magnetic disk one full counterclockwise revolution. What is the change in encoder count? Once more, if the encoder was wired as instructed, the values would decrease between readings; if the wiring was the opposite, the values would increase. If your application uses only changes in rotation over a span of time, then it does not matter what the initial value is in the encoder count buffer. However, if your application requires knowledge of absolute rotation from the start of execution, it is useful to reset the count to zero. Enter the following commands to read the current encoder count, reset the buffer, and reread the count:

```
>> readCount(enc)
>> resetCount(enc)
>> readCount(enc)
```

The chosen encoder hardware provides a resolution of 12 counts per motor shaft revolution (see technical specifications at this **link**). Thus, you can convert the motor shaft's encoder count to the physical angle of the motor shaft in degrees as follows:

```
>> shaftAngle = (readCount(enc)/12)*360
```

This DC motor has a gear ratio of 100:1 (see technical specifications at **this link**) between the motor shaft and the output shaft that attaches to the device you're driving, such as a wheel. Use the following commands to get the angle of t
shaft in degrees, and then normalize it to the range of 0 to 360 degrees:

```
>> axleAngle = (readCount(enc)/12)*360/100
>> axleAngleNorm = mod(axleAngle,360)
```

Now let's displace the output shaft angle using the DC motor rather than manual rotation:

```
>> dcm.Speed = 0.5;
>> start(dcm)
>> readCount(enc)
>> readCount(enc)
>> readCount(enc)
>> stop(dcm)
```

Now, you're able to determine the angular position of the motor shaft and output shaft. To get the angular speed in rpm (revolutions per minute), you can use the method **readSpeed** as follows:

```
>> dcm.Speed = 0.5;
>> start(dcm)
>> readSpeed(enc)
>> stop(dcm)
```

Now, let's determine the speed of the output shaft in rpm and then in degrees per second. Use the following command:

```
>> dcm.Speed = 0.5;
>> start(dcm)
>> rpm = readSpeed(enc)/100
>> degPersec = rpm/60*360
```

Troubleshooting

If you don’t have the expected behavior after you have run the first sketch:

- Check that the Arduino Nano Motor Carrier is ON
- Check that the motors are connected in the motor connector ports

- Check the back part of the motors, be sure that the black disk is not touching the PCB components
- Try to move the motor axis by hand

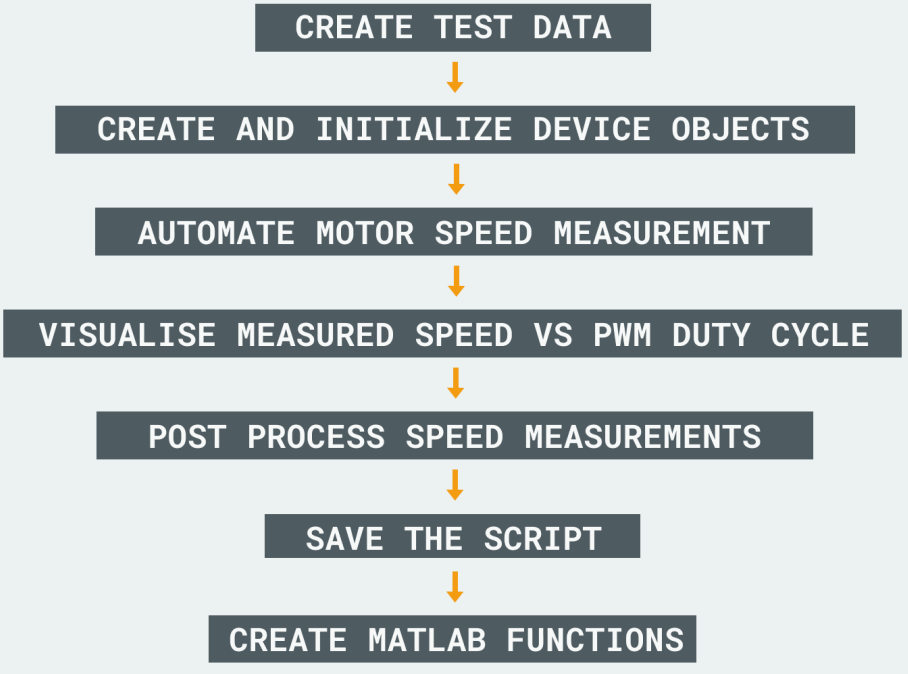
If you don’t have the expected behavior after you have run the second sketch:

- Check that the Arduino Nano Motor Carrier is ON
- Check the encoder connections
- Be sure that the battery is charged
- Update the Arduino Nano Motor Carrier firmware uploading the example: **File > Examples > ArduinoMotorCarrier > Flasher**.

3.6 Characterizing a DC Gear Motor in MATLAB

In this section, you will learn:

- How to automate processes in MATLAB using a script that measures the motor speed,
- To issue different PWM commands to characterize the response of the 100:1 DC gearhead motor,
- How to use code sections to partition scripts into smaller parts,
- How to use for-loops to repeat blocks of code,
- How to use text labels and comments to organise and document MATLAB scripts,
- How to create and call a MATLAB function.



Create test Data

Let's begin writing our motor characterisation program. Open the existing code in your live script, and type the following lines to define your test PWM command values:

```
%% 1. Create test data

maxPWM = 1.00;                              % Maximum duty cycle
incrPWM = 0.05;                             % PWM increment
PWMcmdRaw = (-maxPWM:incrPWM:maxPWM);   % Column vector of duty cycles from
-1 to 1
```

As seen in chapter 2, you can break down your program into logical sections by clicking on **Section Break** or by typing "%%" at the beginning of each section. You can also add comments to individual lines of code by using the "%" character.

Run your live script and examine your **Workspace**. You have created a vector containing a series of numbers ranging from -1 to 1 with 0.05 increments. That will be used to generate the PWM signal we will apply to the DC motor to characterise its behavior.

Create and Initialize Device Objects

Now let's add some code to create and/or initialize the four device objects that we will need for this experiment: Arduino (`a`), the DC motor (`dcm`), the Carrier (`carrier`), and the encoder (`enc`). Add the following lines of code to your live script:

```
%% 2. Create and initialize device objects

clear a dcm carrier enc              % Delete existing device objects

a = arduino;
carrier = motorCarrier(a);
dcm = dcmotor(carrier,'M1');         % Connect a DC motor at 'M1' port on
the Arduino Nano Motor Carrier board

enc = rotaryEncoder(carrier,1);      % Connect the encoder of 'M1' at the
encoder port 1 on the Arduino Nano Motor Carrier board
```

Rerun your live script to ensure correct behavior; you can check the results by looking at the **Workspace**. If everything went as expected, your live script now incl

objects needed to perform the test.

Automate Motor Speed Measurement for each PWM Command

Previously, you prepared the code by creating the objects that connect to Arduino, the Motor Carrier, and the encoder. Next, we will see how to automate multiple measurements.

Now you will add a new section of code that starts the motor with a PWM value, reads the motor speed, and then stops the motor. Add the following section to your live script to measure the speed for the first PWM value:

```
%% 3. Measure raw motor speed for each PWM command

dcm.Speed = 0;
gearRatio = 100;                              % As per the motor spec sheet,
gear ratio equals 100:1
start(dcm)                                    % turn on motor
dcm.Speed = PWMcmdRaw(1);
pause(1)                                      % wait for steady state
speedRaw(1) = readSpeed(enc)/gearRatio;       % read motor speed in rpm of
the output shaft
stop(dcm);                                    % turn off motor
dcm.Speed = 0;
```

Try running this section with different index values of `PWMcmdRaw` to obtain different `speedRaw` values. In the example code, the index is 1; we take the first index of PWMcmdRaw, send it to the motor, measure the speed and store it the first element of speedRaw. Examine the results in the MATLAB **Workspace**.

You now have a live script that you can use to get speed measurements for any PWM command value. Let's generalize this section of the script so that it measures the speed for all the values in the vector `PWMcmdRaw` . You can repeat parts of your code using a **For-Loop**. A for-loop is a block of code that executes a known number of times, called iterations. To create a for-loop, you need a header (which defines the number of iterations), and the end keyword (which defines the end of the repeating code). A for-loop checks the value of an index variable to decide whether the condition to leave the loop has been met. The index indicates the current iteration. Here is a simple example of a for-loop:

```
for idx = 1:10
y(idx) = (2 + idx) / idx;
end
```

This for-loop executes for 10 iterations, and the index variable `idx` takes a new scalar value from 1 to 10 during each iteration.

In our case, you want to iterate through all the elements of `PWMcmdRaw`, and take a new speed measurement during each iteration. The result should be a vector that is the same size as `PWMcmdRaw`. Add a for-loop header and an end keyword to your code, as shown below. Remember to update the indices in `PWMcmdRaw` and `speedRaw` to the index variable, `ii`.

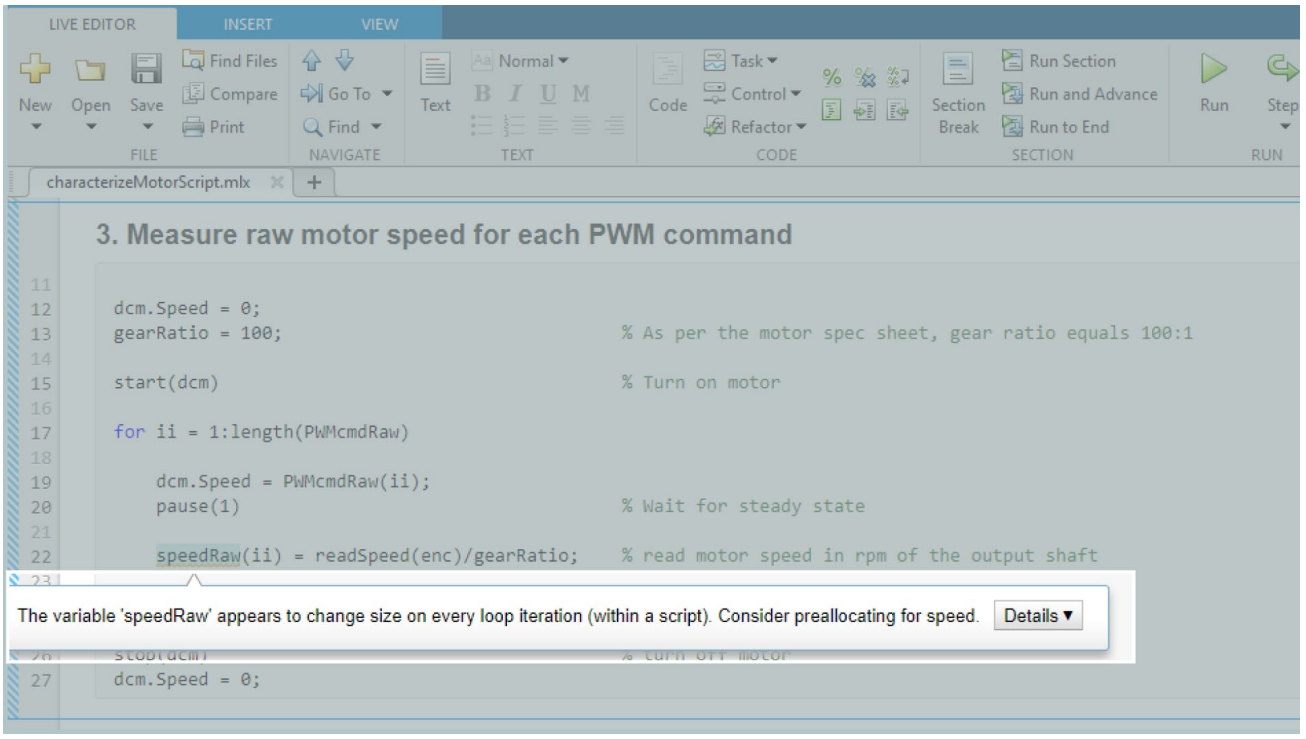
```
%% 3. Measure raw motor speed for each PWM command

dcm.Speed = 0;
gearRatio = 100;                                % As per the motor spec
sheet, gear ratio equals 100:1
start(dcm)                                      % turn on motor

for ii = 1:length(PWMcmdRaw)
    dcm.Speed = PWMcmdRaw(ii);
    pause(1)                                    % wait for steady state
    speedRaw(ii) = readSpeed(enc)/gearRatio;    % read motor speed in rpm
of the output shaft
end

stop(dcm)                                       % turn off motor
dcm.Speed = 0;
```

Execute the section and ensure that all possible PWM values in the `PWMcmdRaw` vector are tested. At this point, the **Live Editor** window should be showing a warning on the assignment to `speedRaw(ii)`. Hover over the speedRaw variable in the Live Editor window to see details about the warning:



The warning appears because the script is assigning values to increasing indices of `speedRaw`, and as a result `speedRaw` needs to increase its size on every iteration to accommodate the new element. In some cases, this can lead to performance issues because the variable's memory may need to be reallocated many times. You can solve this problem if you know how large the array is going to get by defining the vector in advance.

Before assigning values to an array inside a for-loop, you should allocate memory for the array, so that it will not get resized inside the loop. The `zeros` function is often used for this purpose. Add the following line of code before the for-loop to allocate space for speedRaw before populating it with values:

```
speedRaw = zeros(size(PWMcmdRaw));              % Preallocate vector for
speed measurements
```

Final code snippet for automating the motor speed measurements:

```
%% 3. Measure raw motor speed for each PWM command

speedRaw = zeros(size(PWMcmdRaw));              % Preallocate vector for
speed measurements

dcm.Speed = 0;
```

Help

```
gearRatio = 100;                                    % As per the motor spec
sheet, gear ratio equals 100:1

start(dcm)                                          % Turn on motor

for ii = 1:length(PWMcmdRaw)

    dcm.Speed = PWMcmdRaw(ii);
    pause(1)                                        % Wait for steady state

    speedRaw(ii) = readSpeed(enc)/gearRatio;    % read motor speed in rpm
of the output shaft

end
```

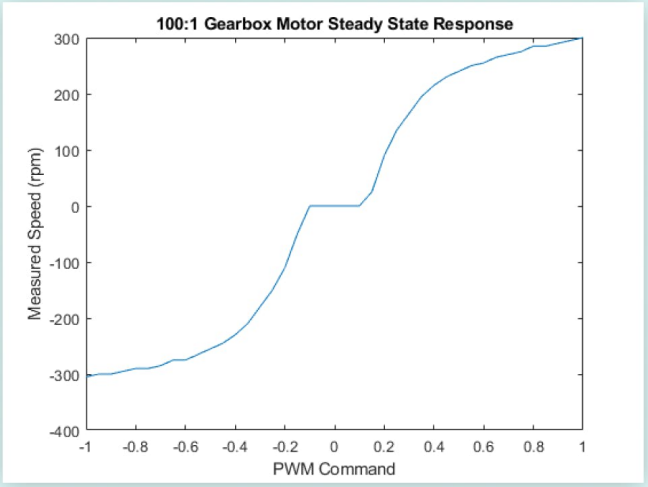
Visualising Measured Speed VS. PWM Duty Cycle

Now you have some real speed measurements in your **Workspace**. Let's visualize them against their corresponding PWM command values using the `plot` command we learned about earlier. Add the following section to your live script to plot and annotate the raw data:

```
%% 4. Graph raw data

plot(PWMcmdRaw,speedRaw)      % raw speed measurements
title('100:1 Gearbox Motor Steady State Response')
xlabel('PWM Command')
ylabel('Measured Speed (rpm)')
```

Execute the section and examine the figure in the output column of the **Live Editor**:



Notice some interesting features of the speed-PWM relationship that you can deduct from studying the graph. There is a "dead zone" around PWM=0, where there is zero rotational speed for non-zero PWM commands. Furthermore, there may be some non-monotonic portions of the curve, where the measured speed does not increase with increasing PWM command. This typically happens around PWM = +/-1, but could also result from experimental error.

Post-Processing Speed Measurements

Collecting this data will be relevant later when you want to model how a more complex machine that uses several motors behaves. Later you will create a motor control system in Simulink, in which a user will request a motor speed in rpm (revolutions per minute). The system will calculate the required PWM command to run the motor at that speed and send it to the Arduino board. To perform this operation correctly, the controller needs a one-to-one mapping between motor speeds and PWM commands. This means that there can only be one PWM command per speed, so you must remove repeated values and nonincreasing values from the data. First you must identify which values of `speedRaw` are not in increasing order. Examine speedRaw in the **Command Window**:

Help

```
>> speedRaw
```

Locate the multiple zeros in the speed measurements and look for nonincreasing values.

It is not easy to see where the nonincreasing values are, so let's make a first-order difference. Enter the following command:

```
>> diff(speedRaw)
```

This will output another vector showing the differences between pairs of values in the array. To remove non-increasing values in `speedRaw`, you need to know the indices where `diff(speedRaw)` is non-positive. To do this, you can use relational operators to create a logical condition. Enter the following logical expressions, and interpret the result:

```
>> pi > 3
>> pi >= 3
>> pi == 3
>> pi < 3
>> x = 1:5
>> x == 3
>> x < 3
>> y = x < 3
```

For the first half of the previous commands (all the ones related to pi), the operations will produce a different result based on the level of truth of the statement.

For the second set of operations (the ones related to the variables `x` and `y`), the results are different, since `x` is an array containing the numbers 1 to 5. Examine `x` and `y` in your **Workspace**. The numeric vector `x` is the same size as the logical index vector `y`. You can use a logical index expression or variable to index into a variable of the same size. Try the following commands:

```
>> x(y)
>> x(~y)
>> z = rand(1,5)
>> z(y)
```

Help

Now let's get the logical indices and values of `speedRaw` where it is strictly increasing. Enter the following commands:

```
>> idx = diff(speedRaw) > 0
>> speedMono = speedRaw(idx);
```

How does the size of `speedMono` compare to that of `speedRaw`? They are different; think about why this is the case. Note, to get help with built-in MATLAB functions like `diff`, `rand`, and `tic`, use the doc command to access the documentation. For example:

```
>> doc diff
```

The documentation for each function will show you the various ways the function can be called, and examples for each syntax.

Now you have made a monotonic version of the speed measurements, but there are some problems. First, how do you now align the PWM command values to the new speed vector if they are different sizes? You can use the same logical index to isolate the corresponding PWM values. Use the following command to filter the PWM values, and plot both the raw and filtered values:

```
>> PWMcmdMono = PWMcmdRaw(idx);
>> plot(PWMcmdRaw,speedRaw,PWMcmdMono,speedMono)
```

To illustrate a second issue that arises when post-processing the information obtained from the motor, enter the following commands:

```
>> speedMono == 0

 >> PWMcmdMono(speedMono == 0)
```

Due to the way we filtered out the non-monotonic points, a speed of zero is to be driven by a non-zero PWM. Although commanding a small enough PWM value will result in zero speed due to friction, it would waste power, especially if the

commands zero speed for a long time. Therefore, you should change that value of `PWMcmdMono` to `0`, which will have the same result. Enter the following command:

```
>> PWMcmdMono(speedMono == 0) = 0;
```

Let's incorporate the post-processing into the live script. Add a new section before the section labeled **Graph raw data**, and type the following code:

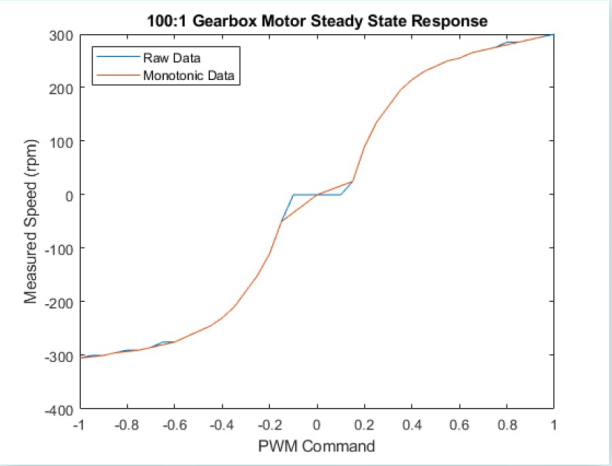
```
%% 4. Post-process and save data

idx = (diff(speedRaw) > 0);             % find indices where vector is
increasing
speedMono = speedRaw(idx);              % Keep only increasing values of
speed
PWMcmdMono = PWMcmdRaw(idx);            % Keep only corresponding PWM
values
PWMcmdMono(speedMono == 0) = 0;         % enforce zero power for zero speed
save motorResponse PWMcmdMono speedMono % save post-processed measurements
```

Now update the **Graph raw data** section to include both the raw data and post-processed data. Add the following lines of code as follows:

```
%% 5. Graph raw and post-processed data

plot(PWMcmdRaw,speedRaw)                                % raw speed
measurements
hold on
plot(PWMcmdMono,speedMono)                              % non-monotonic
measurements filtered out
title('100:1 Gearbox Motor Steady State Response')
xlabel('PWM Command')
ylabel('Measured Speed (rpm)')
legend('Raw Data','Monotonic Data','Location','northwest')
```



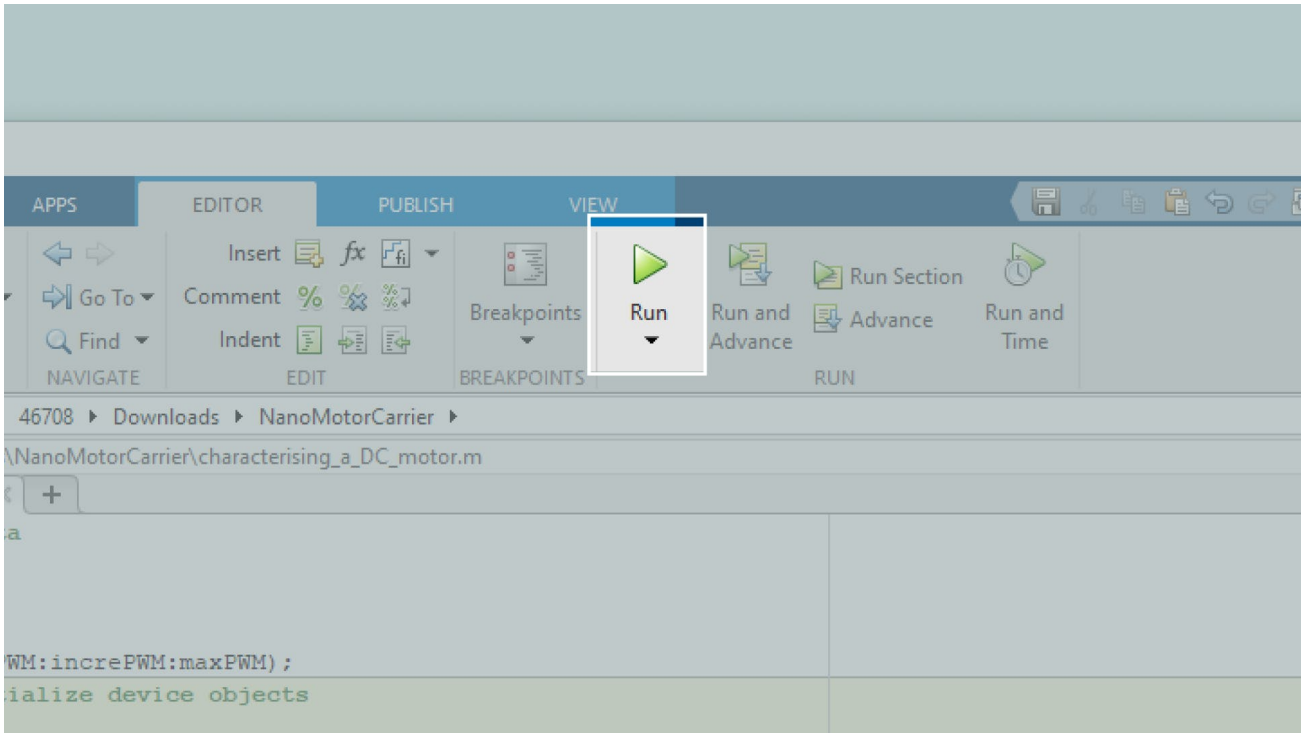
Finally, delete all device objects to release the Arduino Nano 33 IoT board to be used in other processes. Add the following section at the end of your live script:

```
%% 6. Delete device objects

clear a dcm carrier enc
```

Saving and Rerunning Scripts

You now have a fully functional live script that automatically performs a series of measurements, post-processes those measurements, saves the experimental data to disk, and plots the results. You may want to repeat this analysis multiple times, for the same motor or perhaps other DC motors. You can save your live script as a `.mlx` file so that you can run it again later or share it with other MATLAB users. Save your live script as `myMotorCharacterization.mlx`. An `.mlx` file contains not only your code and text annotations, but also any numeric or graphical results that were displayed in the output column. Thus, your live script is an active report of your analysis and results. Recall that you can run your live script in its entirety using the **Run** button in the **Live Editor** window:



You can also run the live script directly in the **Command Window** using its name as the command. Try running your live script:

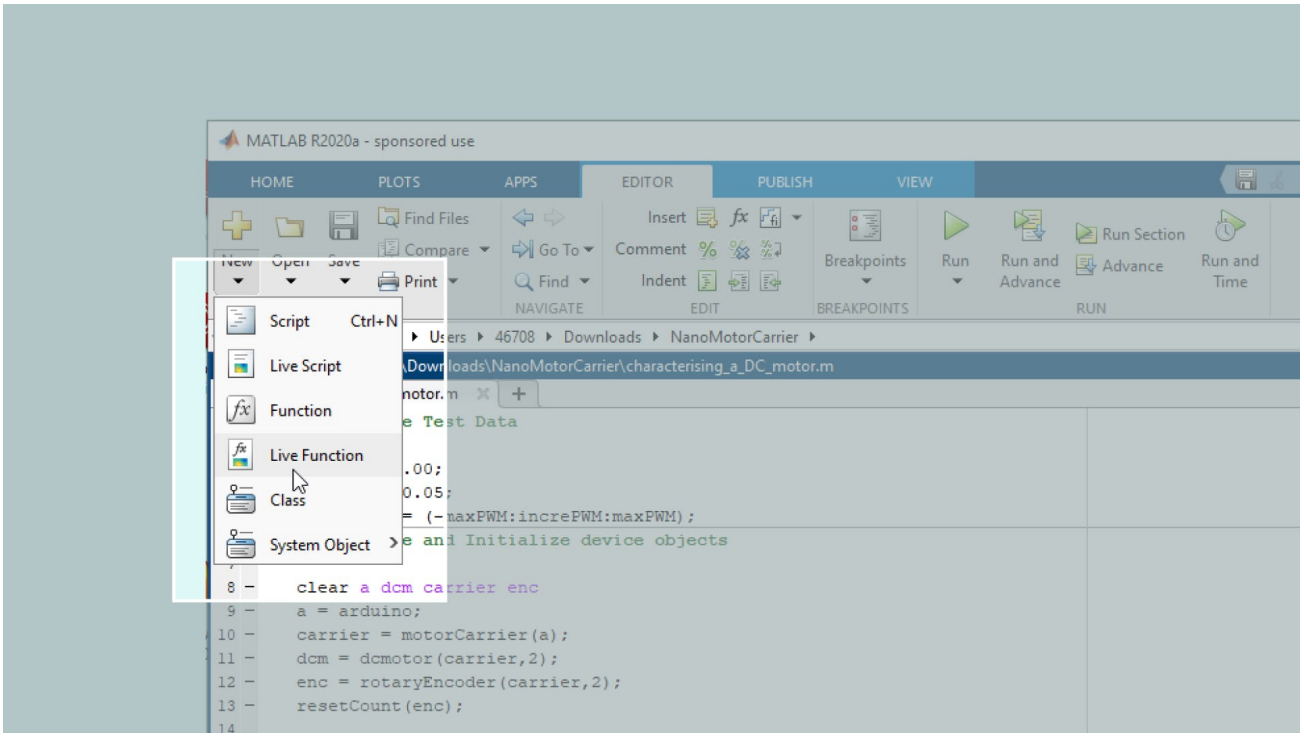
```
>> myMotorCharacterization
```

Creating MATLAB Functions

Look at your **Workspace**. Which of the existing variables are useful to you? Which variables are intermediate values from your live script that you no longer need?

You can compartmentalize parts of your MATLAB program into functions, so that the intermediate calculations are hidden and you only need to manage inputs and outputs.

Looking at your live script, the first two sections are specific to this DC motor, encoder, and test case (`PWMcmdRaw`). The following three sections (measure raw motor speed for each PWM command, post-process and save data, and graph raw and post-processed data) can be generalized for any DC motor, encoder, and test case. Let's create a MATLAB function for that code. Start in the **Live Editor** window by clicking **New > Live Function**.



Next, you need to determine what the inputs and outputs to your function are. Section 3 of your live script requires the test vector `PWMcmdRaw`, the DC motor object `dcm`, and the encoder object `enc`.

The next line of code is an example of a possible function header. Place the inputs in parentheses (`(  )`) with the three variables listed above:

```
function z = Untitled(PWMcmdRaw,dcm,enc)
```

Now consider what variables you want to access from the motor characterization algorithm. The only meaningful variables created in these sections are `PWMcmdMono` and `speedMono`. In the function header, list these two variables as outputs in square brackets (`[  ]`).

```
function [PWMcmdMono,speedMono] = Untitled(PWMcmdRaw,dcm,enc)
```

Finally, let's name the function so that we know how to call it in MATLAB code. Replace the function name with myMotorFunction:

Help

```
function [PWMcmdMono,speedMono] = myMotorFunction(PWMcmdRaw,dcm,enc)
```

Now let's fill in the algorithm. Cut and paste sections 3, 4, and 5 from your live script to your live function, between the function header and the end keyword.

```
function [PWMcmdMono,speedMono] = myMotorFunction(PWMcmdRaw,dcm,enc)

3. Measure raw motor speed for each PWM command
speedRaw = zeros(size(PWMcmdRaw));               % Preallocate vector for
speed measurements

dcm.Speed = 0;
gearRatio = 100;                                 % As per the motor spec
sheet, gear ratio equals 100:1

start(dcm)                                       % Turn on motor

for ii = 1:length(PWMcmdRaw)

    dcm.Speed = PWMcmdRaw(ii);
    pause(1)                                    % Wait for steady state

    speedRaw(ii) = readSpeed(enc)/gearRatio;    % read motor speed in rpm
of the output shaft
end
```

```
4. Post-process and save data
idx = diff(speedRaw) > 0;                % find indices where vector is
increasing
speedMono = speedRaw(idx);               % Keep only increasing values of
speed
PWMcmdMono = PWMcmdRaw(idx);             % Keep only corresponding PWM
values

PWMcmdMono(speedMono == 0) = 0;          % enforce zero power for zero speed

save motorResponse,'PWMcmdMono','speedMono' % save post-processed
measurements
```

```
5. Graph raw and post-processed data
plot(PWMcmdRaw,speedRaw)   % raw speed measurements
hold on
plot(PWMcmdMono,speedMono) % non-monotonic measurements filtered out

title('100:1 Gearbox Motor Steady State Response')
xlabel('PWM Command')
ylabel('Measured Speed (rpm)')
legend('Raw Data','Monotonic Data')
```

Help

Save the file as myMotorFunction.mlx. Now you have a working MATLAB function that you can call from any MATLAB code environment. Let's try calling it from your live script. In the empty section of your live script, enter the call to your function:

```
1. Create test data
maxPWM = 1.00;                          % maximum duty cycle
incrPWM = 0.05;                         % PWM increment
PWMcmdRaw = (-maxPWM:incrPWM:maxPWM)'; % column vector of duty cycles from
-1 to 1
```

```
2. Create and initialize device objects
clear a carrier dcm enc                 % Delete existing device objects

a = arduino;
carrier = motorCarrier(a);
dcm = dcmotor(carrier,'M1');            % Connect a DC motor at 'M1' port on
the Arduino Nano Motor Carrier board

enc = rotaryEncoder(carrier,1);         % Connect the encoder of 'M1' at the
encoder port 1 on the Arduino Nano Motor Carrier board
```

```
3-5. Call motor characterization function
[PWMcmdMono,speedMono] = characterizeMotorFcn(PWMcmdRaw,dcm,enc);
```

```
6. Delete device objects
clear a carrier dcm enc
```

Now run your live script and ensure it still works as before.

To further understand the benefits of using MATLAB functions, clear your **Workspace** in the **Command Window**, and run your live script again.

```
>> clear

>> myMotorCharacterization
```

Examine your **Workspace**. The list of variables should be more concise now. There is one more thing you may want to generalize in this function, and that is the r[illegible] the MAT-File that we use in the **save** command. As of now, the analysis dat[illegible]

saved to the same MAT-File, `motorResponse.mat` , every time you call the function. This is not very useful if you want to characterize multiple DC motors. Let's add a new input to the function for the file name. In the live function, add the new input as shown below:

```
function[PWMcmdMono,speedMono] =
myMotorFunction(PWMcmdRaw,dcm,enc,filename)
```

Now let's use the new input in the **save** command. Rewrite the **save** command as follows:

```
save(filename,'PWMcmdMono','speedMono')% save post-processed measurements
```

Finally, we need to modify the function call, since the function header has changed. In the live script, change the function call as follows:

```
[PWMcmdMono,speedMono] =
myMotorFunction(PWMcmdRaw,dcm,enc,'motorResponse');
```

Now you can save the analysis data to any MAT-file you want each time you call `myMotorFunction` . Try calling your live script once more:

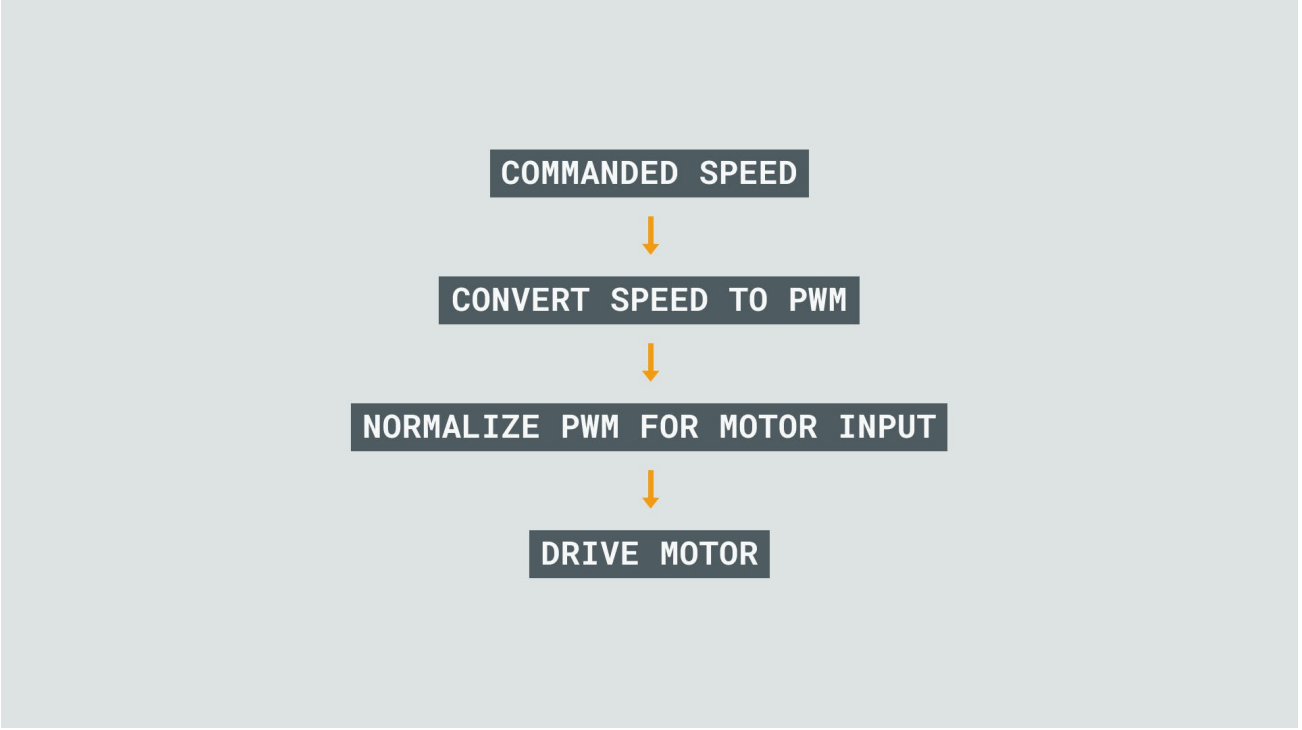
```
>> myMotorCharacterization
```

3.7 Designing a Motor Control System

In this section, you will learn:

- Design a motor control system that takes the users desired speed as input
- How to determine the necessary PWM command to achieve that speed
- Use simulink to model and simulate this system
- Running the simulink model directly on the Arduino Nano 33 IoT

CONTROL SYSTEM FLOW CHART

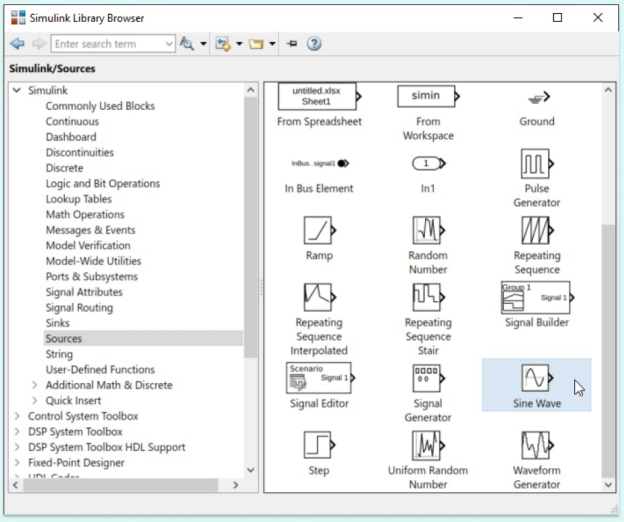


In this diagram, the blocks represent some functionality or process and the arrows represent data flow. The directionality of the arrows indicate which process generates the data and where the data is used. Simulink enables you to create your algorithm as a block diagram and execute it over an interval of time.

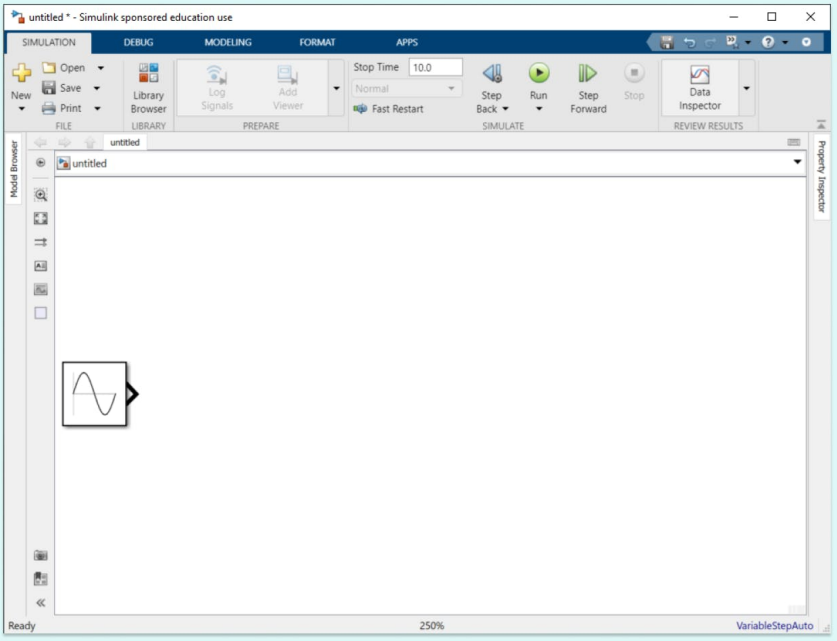
Create the Blocks

Add the Sin block

Let's add some blocks to the model **canvas**. Open the **Simulink Library Browser** by clicking the **Library Browser button** in the toolstrip. You will use a sine wave to simulate the speed input.

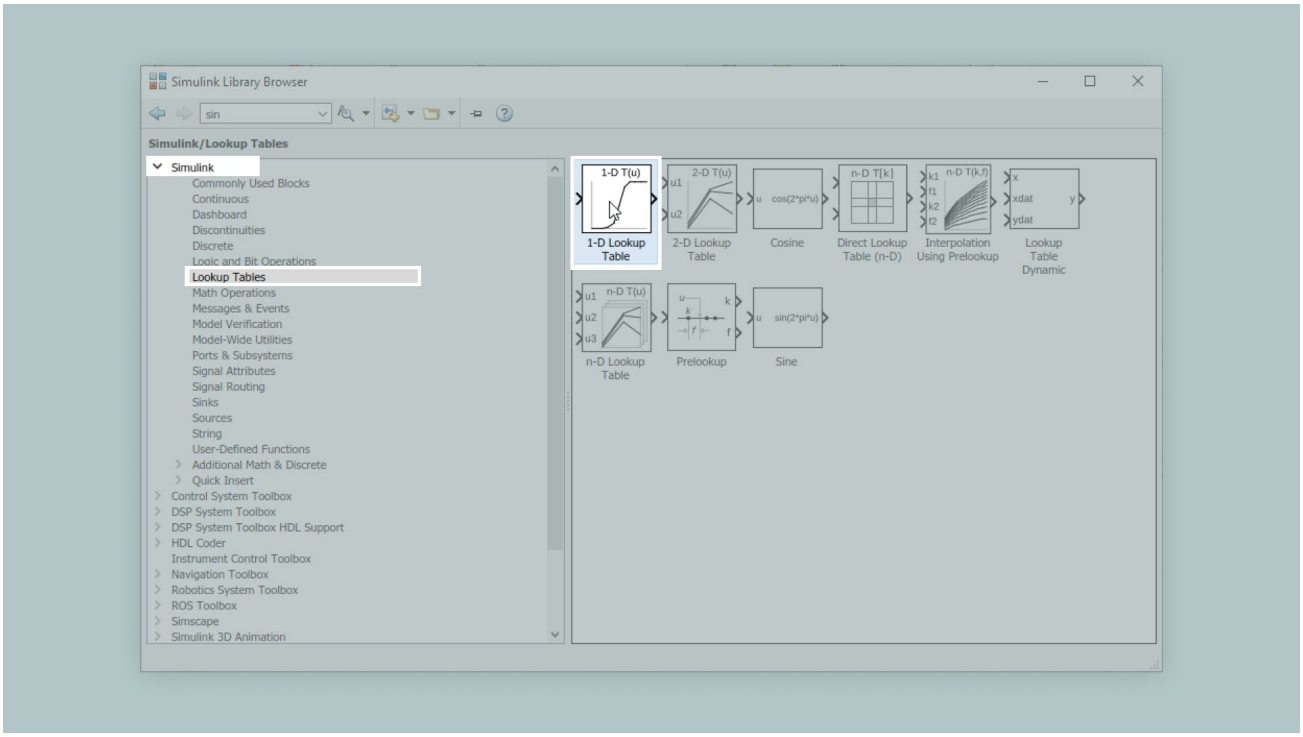


Locate the **Sine Wave** block and drag it from the **Simulink Library Browser** window to the **Simulink Editor** window.

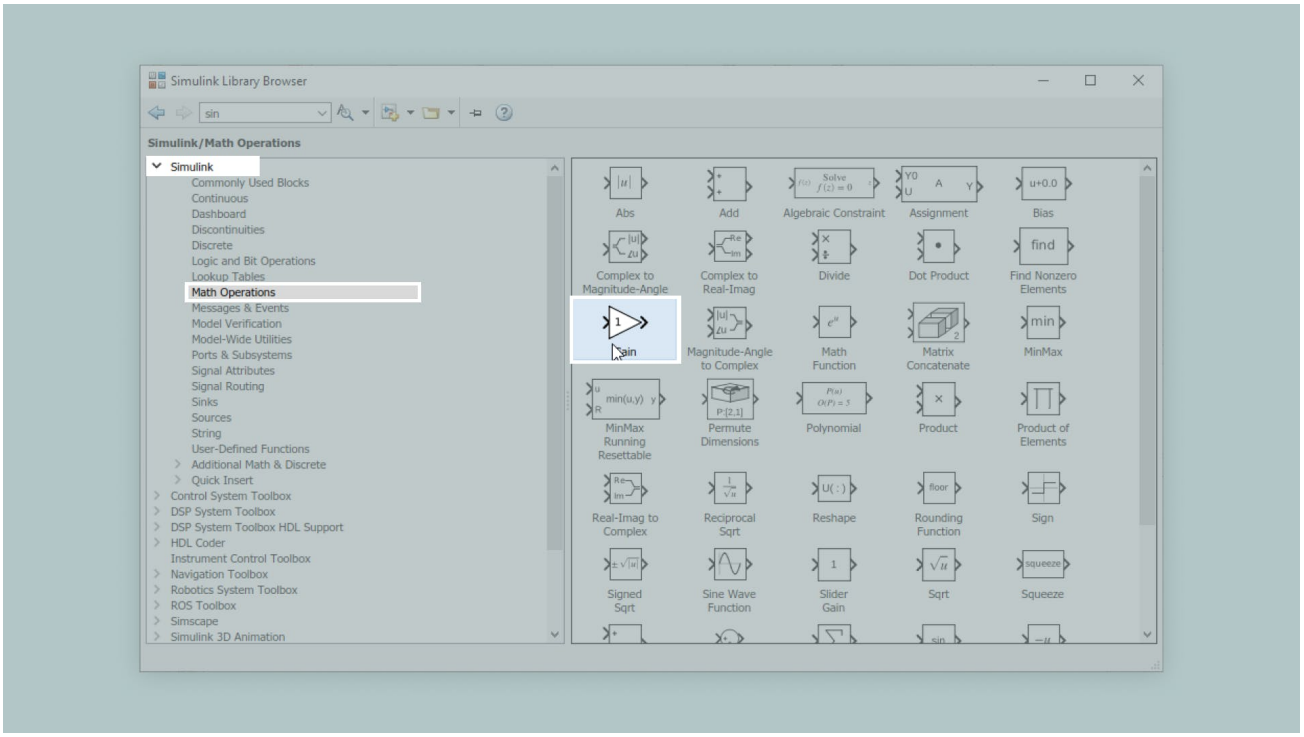


Add a Lookup Table and Gain

Next you need to convert the user's speed signal coming from the **Sine Wave** block into a PWM signal between -1 and 1. To do this, use a lookup table, which uses **interpolation** among known data points to determine an output value for an arbitrary input value. Locate the **1-D Lookup Table** block in **Simulink > Lookup Tables** and drag it into the **Simulink Editor window**.

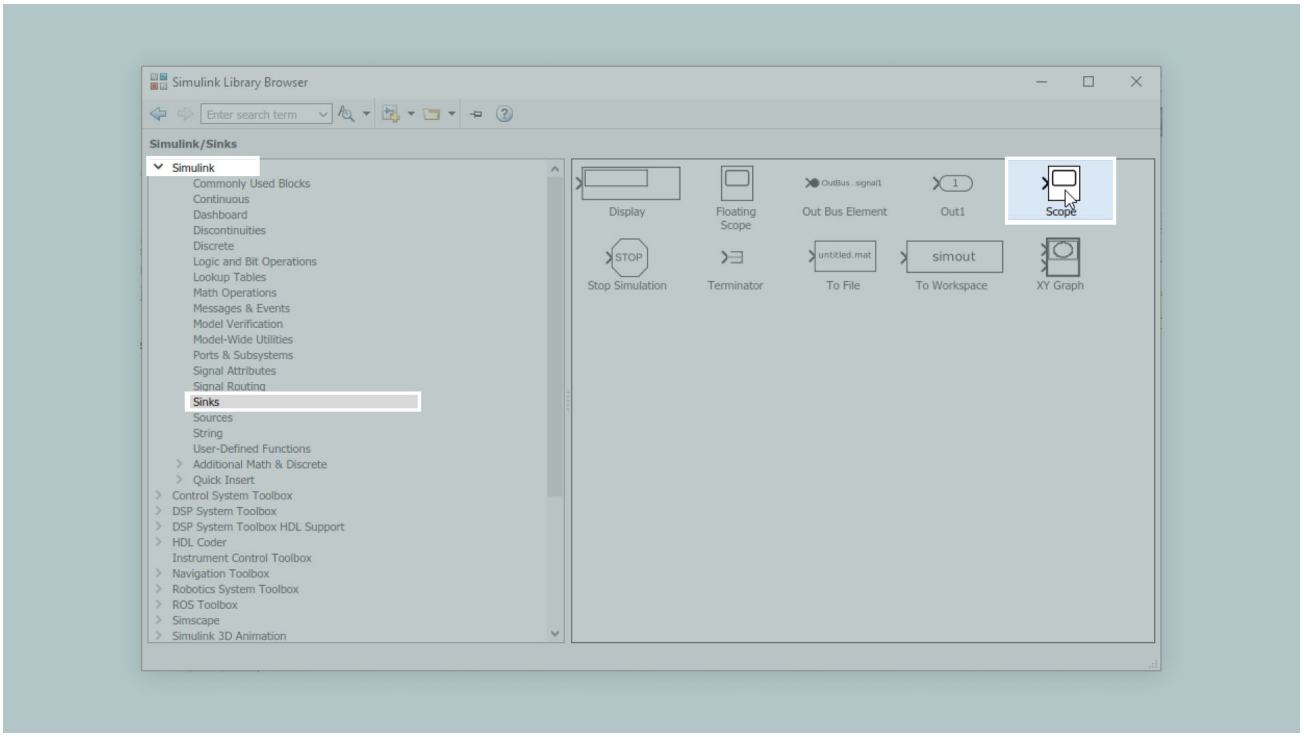


The **Lookup Table** will output PWM command values between -1 and 1. Ultimately, you will communicate the magnitude of the duty cycle to the DC motor hardware using integers that range from -100 to 100 for the **M1 M2 DC Motors** block or -255 to 255 for the **M3 M4 DC Motors** block. To prepare for this scaling, you need a **Gain** block. A **Gain** block multiplies a Simulink signal by a constant value. Add a **Gain** block from **Simulink > Math Operations**:



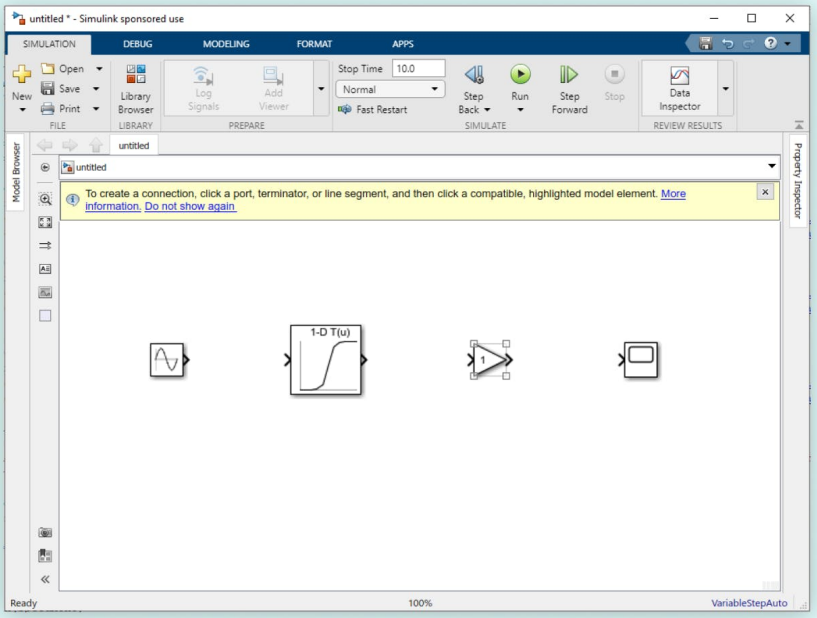
Add a Sink block

Now let's add a block to visualize the output of the lookup table. Examine the **Simulink > Sinks** library.

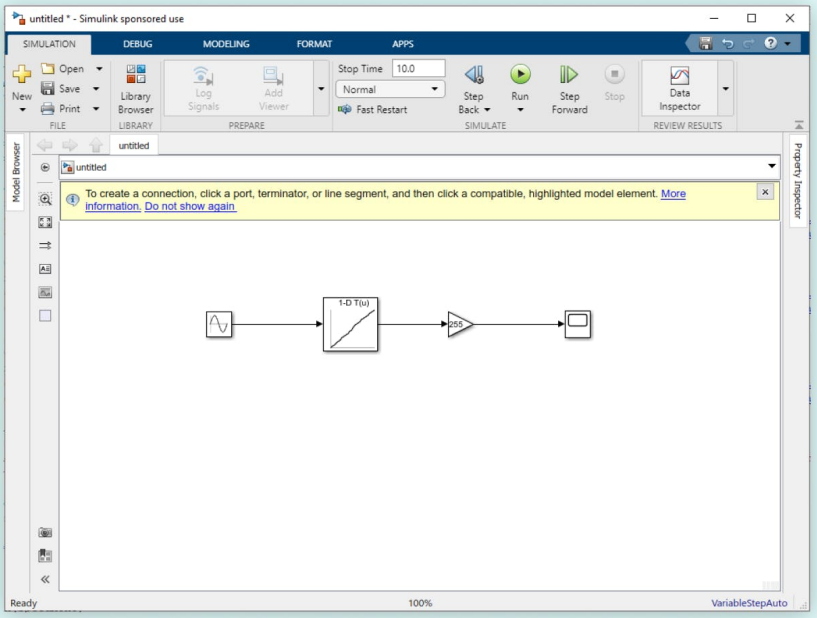


Connect the blocks

Once you've added the blocks to the simulink window, arrange them linearly as shown in the image below.



Now you need to connect the blocks using signals. Left click on the outport and drag it to the import and when the signal line turns solid, you have connected the signal and you can stop the click and drag.



Configure the Blocks

Now let's configure the blocks to perform your specific algorithm. The **Sine Wave** block needs to generate speed values in counts per second. Examine the range of speeds that you measured for the motor; to do this, lets go to the MATLAB command line and run the command

```
>> arduinosetup
```

Then, configure the Arduino Nano 33 IoT as in "Configuring Arduino Libraries" and run the command

```
>> characterizeMotorScript
```

This characterizes the motor response.

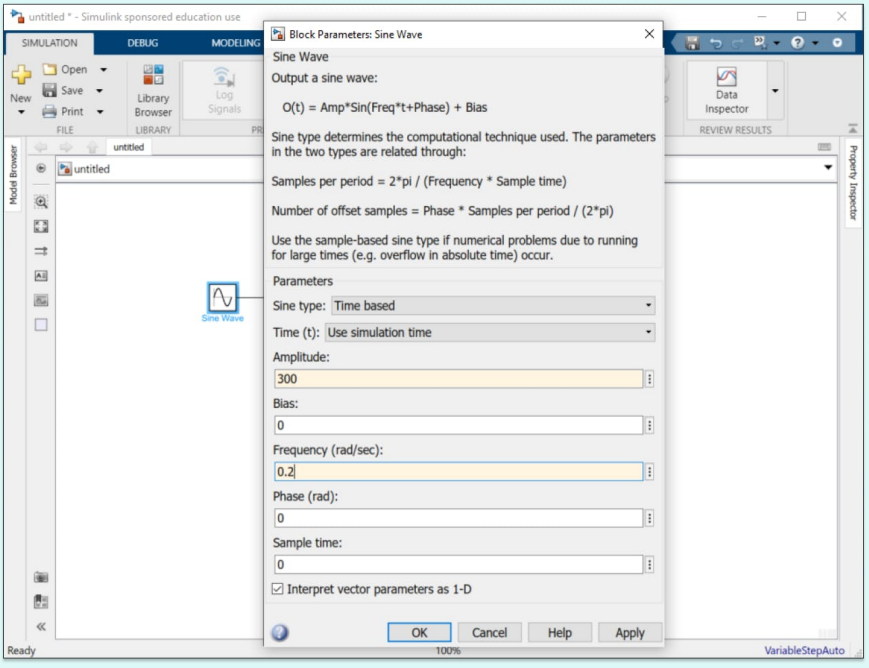
Now let's configure the blocks to perform your specific algorithm. The **Sine Wave** block needs to generate speed values in counts per second. Examine the range of speeds that you measured for the motor; to do this, look into the data files

generated in the MATLAB section of this chapter by issuing the following commands in the command window:

```
>> load motorResponse
>> min(speedMono)
>> max(speedMono)
```

Configure the Sin Block

Double-click the **Sine Wave** block to open its **block parameter dialog**. Configure the sine wave to have an amplitude of 300 and a frequency of 0.2, to cover the range of measured speeds. Then click **OK**:

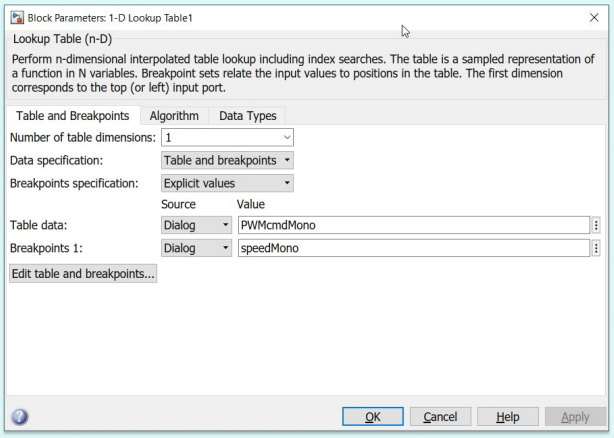


Note: We obtained the value of 300 empirically by looking at the max speed obtained from the motor characterisation process. You can modify this value based on your results.

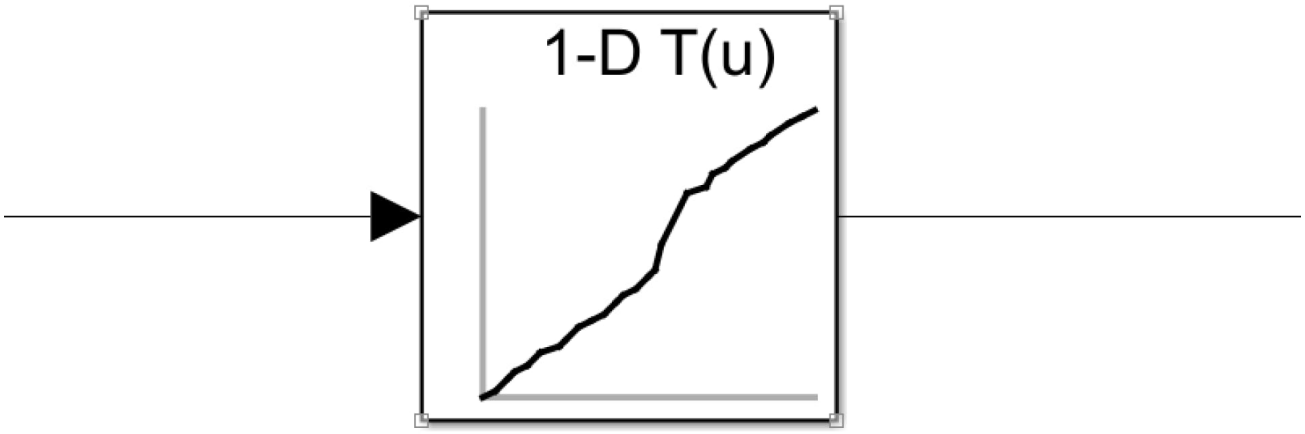
Setup the Look Up Table

Now let's set up the lookup table to map speeds to PWM commands. During simulation, the block will use linear interpolation to estimate the value of the PWM command to achieve an arbitrary speed.

Double-click the **1-D Lookup Table** block to open its block parameter dialog. Set **Table data** to PWMcmdMono and set **Breakpoints** to speedMono. Then click **OK**:

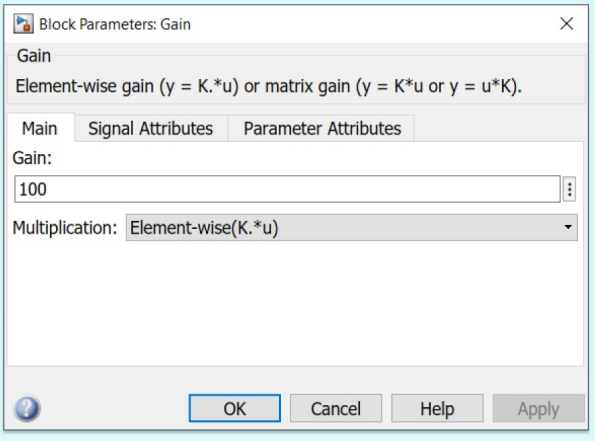


Once you set the lookup table data, the block will show a graph of your lookup table vectors:



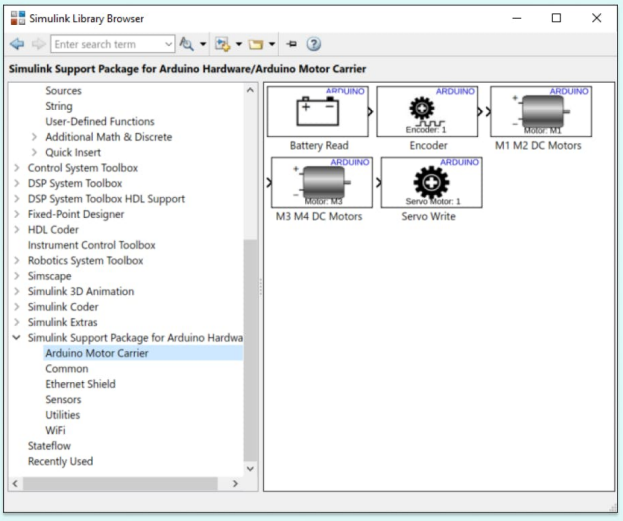
Amplify the PWM duty cycle

Amplify the PWM duty cycle to cover a range of -100 to 100, to be compatible with the motor driver later. Double-click the **Gain** block and set Gain to 100. Then click **OK**:

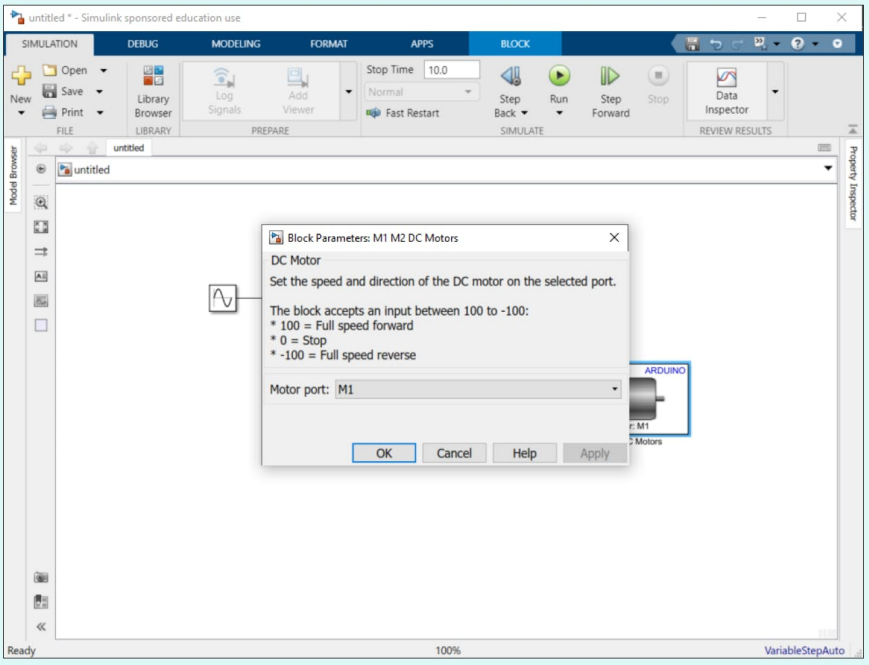


Add a Device Driver Block

To access the motor connected to the Arduino Nano Motor Carrier you need an **Arduino Device Driver** block. In the Simulink Library Browser, navigate to **Simulink Support for Arduino Hardware**, and examine the blocks inside the **Arduino Motor Carrier** section:

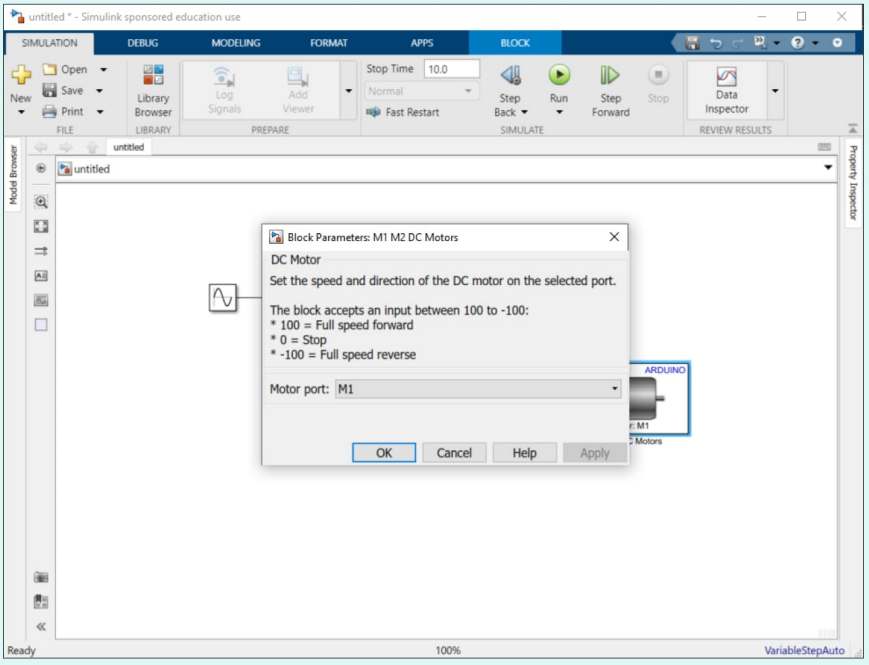


Notice that all the blocks are either source or sink blocks. This is because they represent the boundary between the Arduino processor application (which you are modeling in Simulink) and external devices, like the DC motor and magnetic encoder. Locate the **M1 M2 DC Motors** block, and drag it into your model:

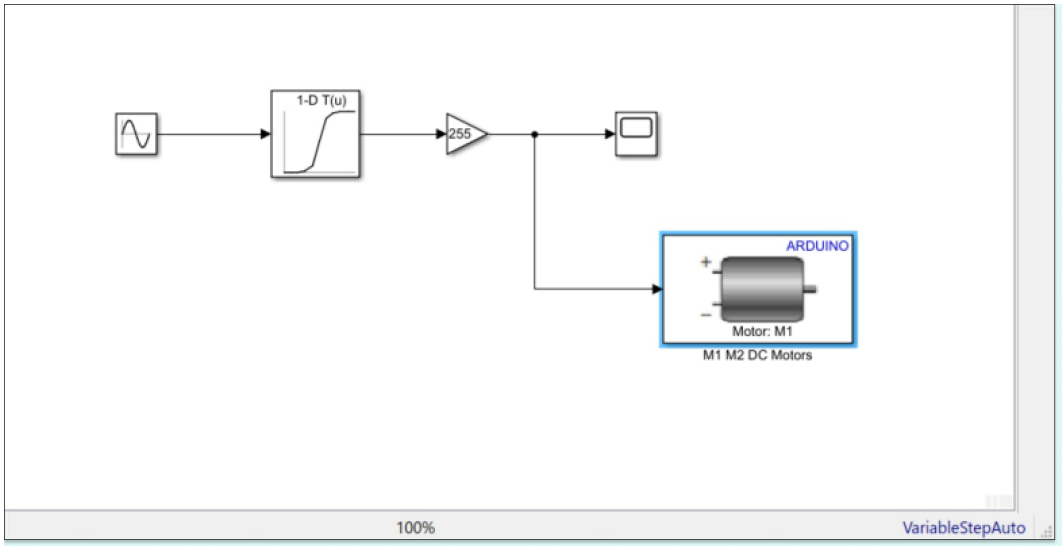


Configure the Device Driver Block

The **M1 M2 DC Motors** block requires one input signal, which is the drive command expressed as a value between -100 and 100. Double-click the **M1 M2 DC Motors** block, and make sure that the motor port is set to M1:

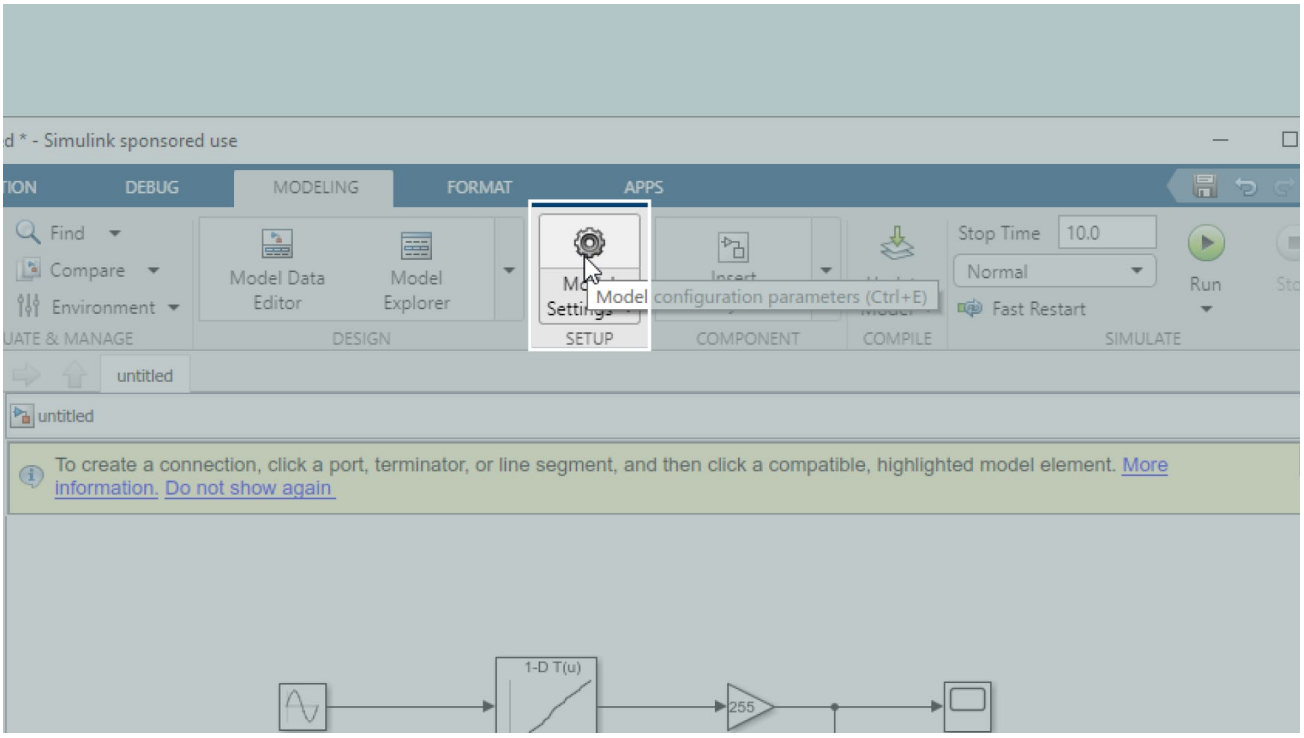


The **Motor port** property maps the block to one of two labeled DC motor ports (M1 or M2) on the Motor Carrier. Create a new branch of the `driveCmd` signal by right-clicking and dragging the mouse from the signal line, and route the signal into the **M1 M2 DC Motors** block:

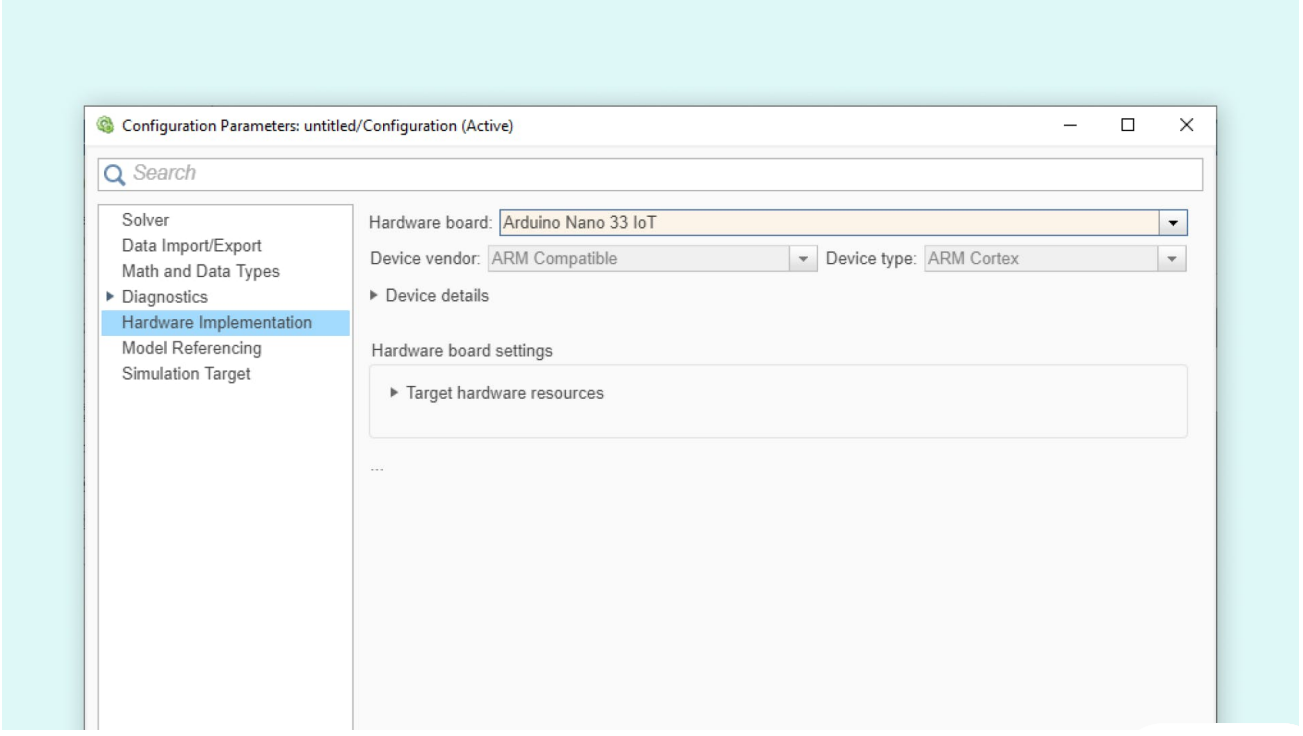


Deploy the model to Arduino

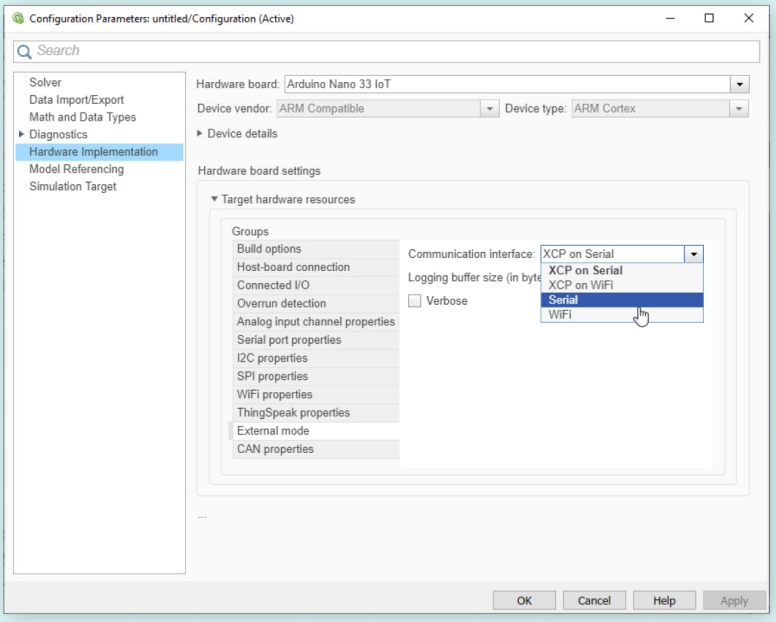
You're almost ready to run the software controller on the Arduino Nano 33 IoT board! First, you need to configure the model to run on Arduino. Open the **Configuration Parameters** window by clicking the "gear" button:



Configuration Parameters is a set of options where you configure how your simulation should run, what hardware you are running it on, and how the model algorithm should handle various run-time conditions. Navigate to the **Hardware Implementation** pane, and set **Hardware Board** to Arduino Nano 33 IoT:

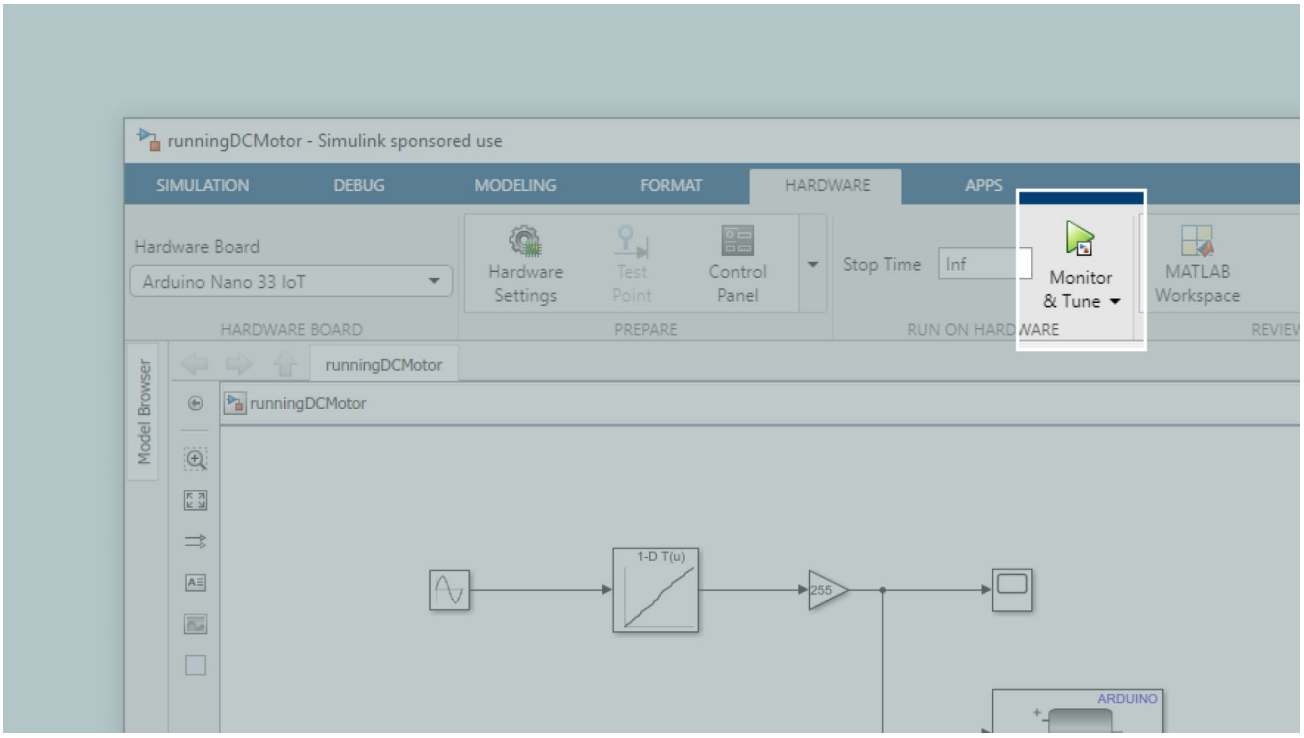


Go to **Target hardware resources > External mode > Communication interface** and choose `Serial` as shown below:

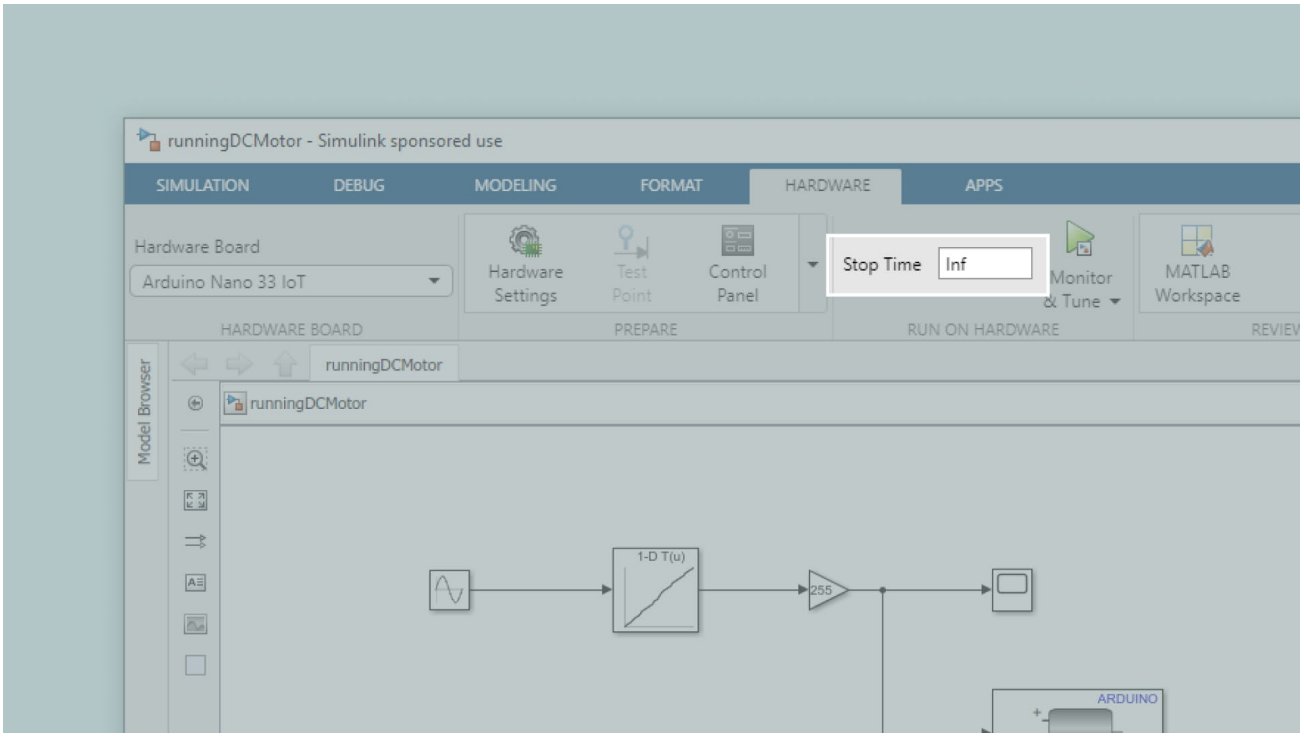


When you set the model to run on a specific board, some of the options throughout **Configuration Parameters** will set themselves automatically, as required. Click **OK** to return to the model window. Previously you have been simulating your model using normal mode. In normal mode, the default setting makes your model run on the computer without communicating with any external hardware. In normal mode, you can prove that your system works conceptually, before dealing with the details of implementing it on hardware.

To run the model on the Arduino Nano 33 IoT board, you will use **Monitor & Tune** under the **Hardware** tab. In external mode, Simulink builds an **executable** from your model and uploads it to the external hardware board. When the executable runs on the external hardware board, you can interact with the running application using the Simulink model. This enables you to monitor signals of interest and change parameter values in the model as it runs on the hardware.



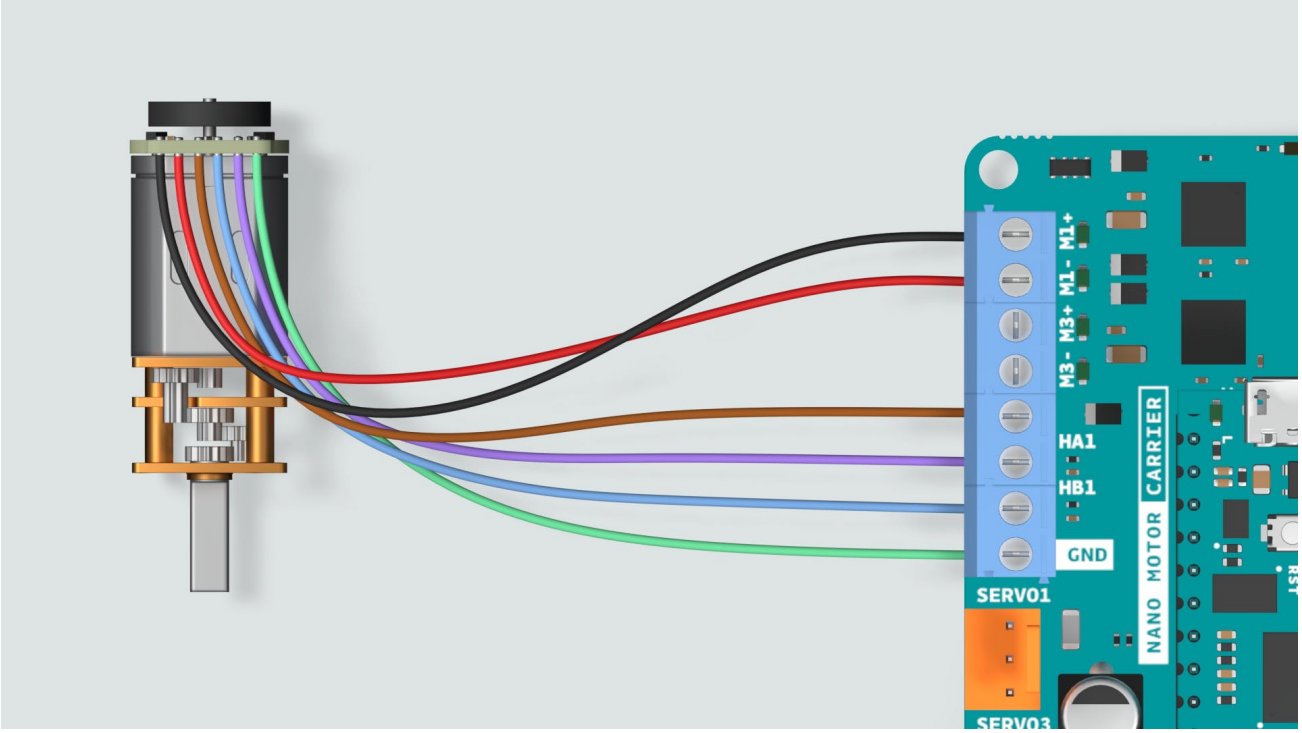
In the model window, change the simulation stop time to Inf so that it will run indefinitely.



Connect the DC Motor

Help

At this point you have finished making a Simulink model capable of generating a signal that you can send later to the Arduino Nano 33 IoT board to control the speed of a motor. Connect a micro-geared DC motor to the Arduino Nano Motor Carrier.



Turn on the battery power by using the ON-OFF switch on the Carrier board. Now click the **Run** button. You must wait for Simulink to build the executable and then initialize the external mode infrastructure.

> **Note:** If the model gets stuck in the code generation mode for more than a few minutes, a) try double pressing the reset button on your Arduino Nano 33 IoT or b) it could be because you have an "arduino object" (like in "a = arduino;"). You won't be able to upload any Simulink model to the Arduino Nano 33 IoT until you clear the "arduino object", because it is getting the COM port busy.

Once the application starts running, examine the oscillatory motion of the motor and magnetic encoder. Is the speed oscillating at the same rate as the input speed command? Stop the simulation, and then turn off the Motor Carrier power. Save the Simulink model.

Help