ARDUINO, MATLAB & SIMULINK

Learn the basics of the Arduino IDE, the MATLAB® IDE, the programming language, and the Simulink® models. You will discover how these three tools can be connected to each other while making practical exercises.

2.1 The Arduino Environment

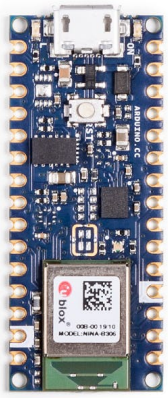
Before you get started, it's good to familiarise yourself with the arduino environment before you start using things. This section gives you some specific knowledge about the Arduino boards & carriers, the software you use to tell your boards what to do, principles and techniques that are behind the Arduino platform.

In this section, you will learn:

- About the Arduino Nano 33 IoT board and its key features.
- How to install and configure the Arduino IDE.
- Testing the Arduino board through the IDE.
- About the Arduino Nano Motor Carrier.
- To move a motor by using the Arduino Nano Motor Carrier.

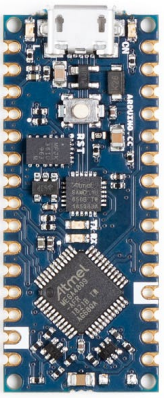
The Arduino Nano Family of Boards

Over the years, Arduino has developed a variety of **boards**, each one with different capabilities and functionalities. These boards are equipped with different processors, memory, input/output features and form factor. These different boards cater to the various needs of the user and to the complexity of the application that's being developed. The Arduino Nano family are a series of boards with a compact form factor and a powerful processor.



NANO 33 BLE

NANO 33 BLE SENSE

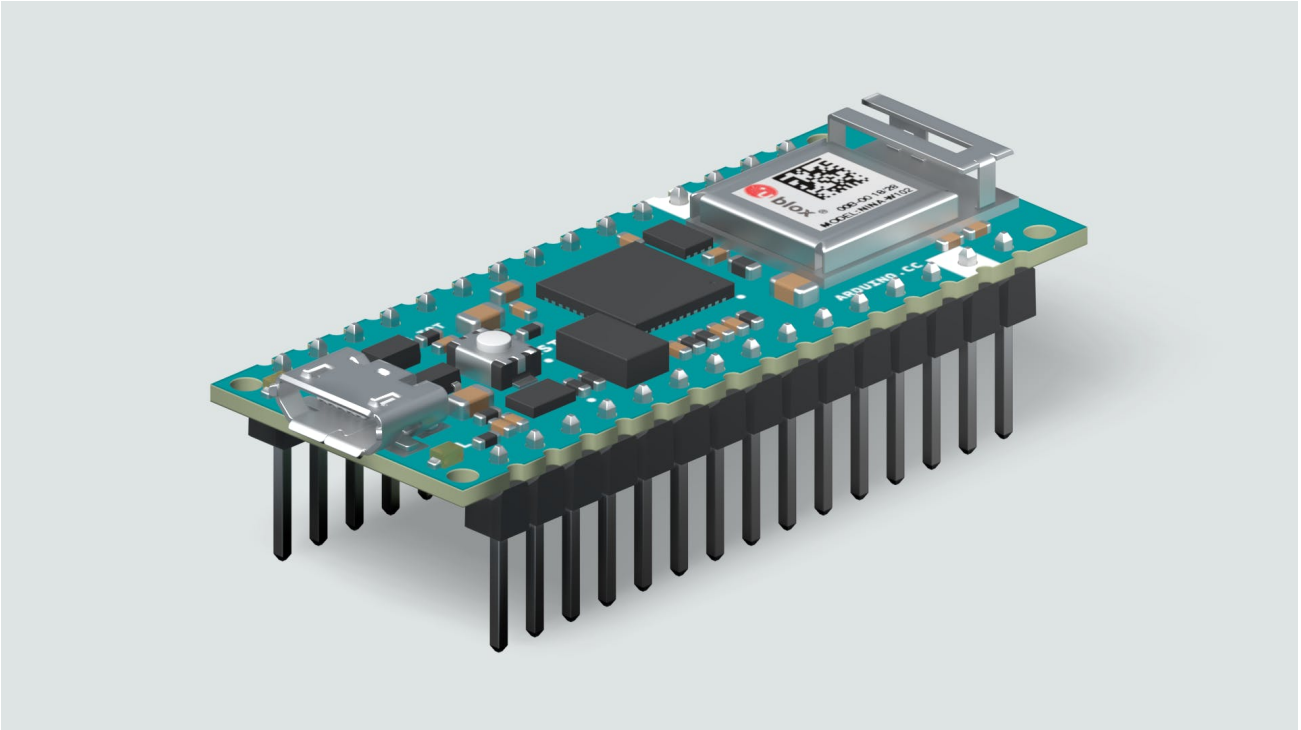


NANO EVERY

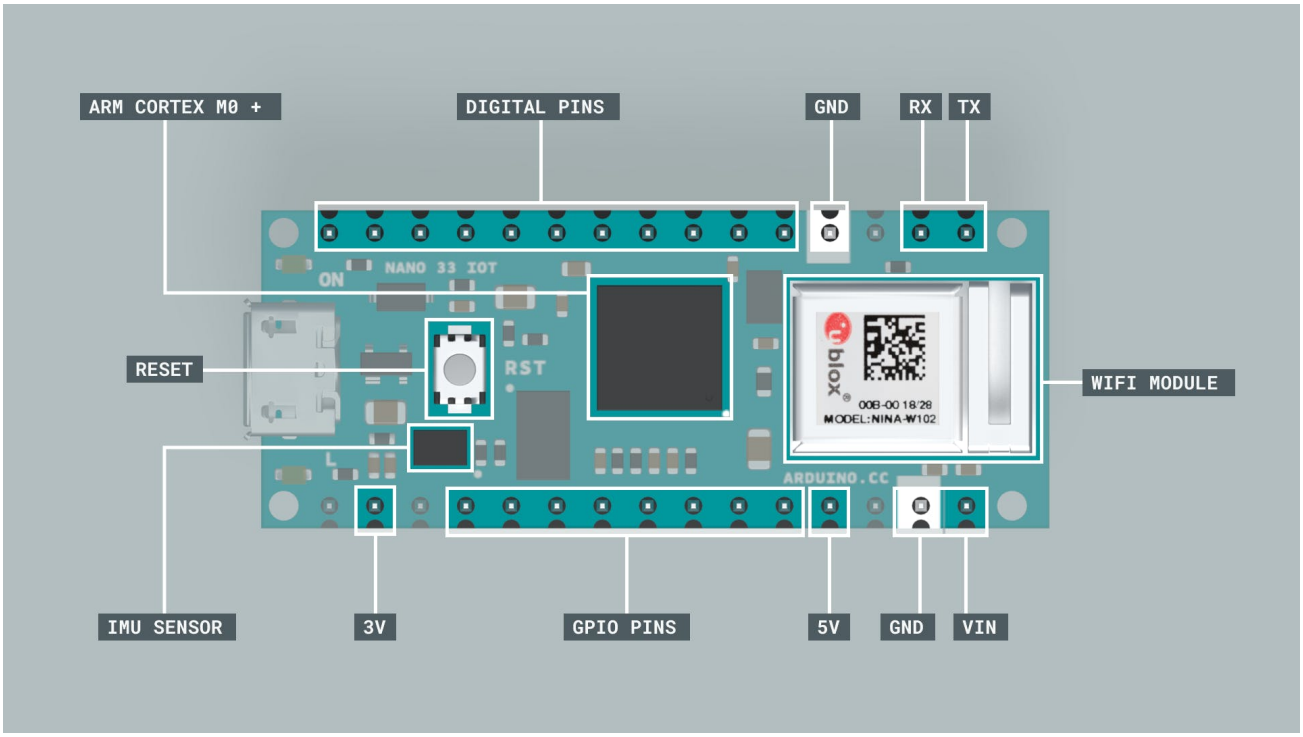
NANO 33 IOT

The Arduino Nano 33 IoT

The **Arduino Nano 33 IoT** is a powerful board that contains an ARM Cortex M0+, which has been designed to offer a powerful and cost-effective solution., has a Wi-Fi module with a crypto-chip to ensure secure communications, and can be configured to constrain its power consumption to keep it running for long periods. The Arduino Nano 33 IoT is the one of the most versatile boards for makers and engineering students seeking to develop robotics and IoT projects.



Key Features



Arm Cortex M0+

This high performance, 32 bit chip allows developers to build low-cost smart devices with lowest power requirements of all the Cortex-M processors. The Cortex-M0+ processor allows developers to optimize power usage for specific applications with built-in, low-power features.

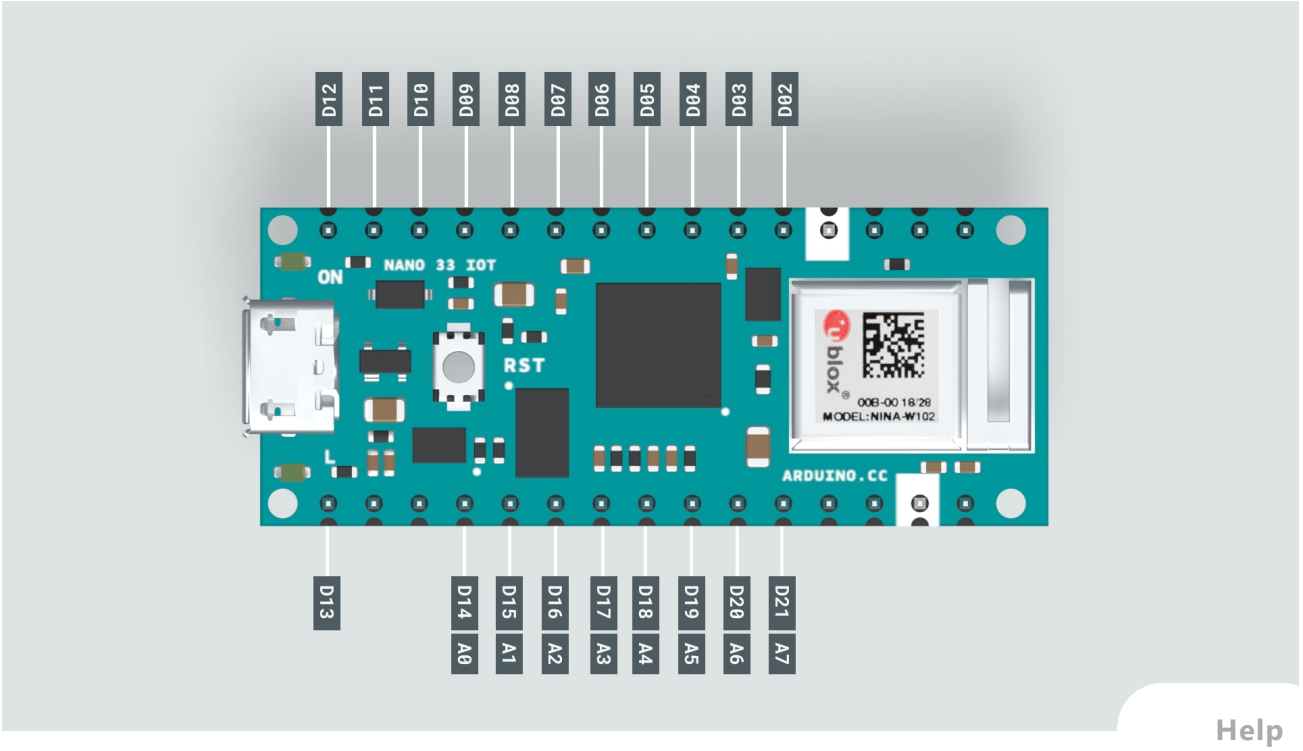
IMU Sensor

A 3D digital accelerometer and a 3D digital gyroscope performing at 1.25 mA in high performance mode and enabling always-on low-power features for an optimal motion experience for the consumer. The IMU which can be used to measure board orientation (by checking the gravity acceleration vector orientation) or to measure shocks, vibration, acceleration and rotation speed.

Wi-Fi Module

A multi radio module that has an integrated antenna that support both Wi-Fi and bluetooth communication. Nina W102 Wi-Fi module which is based on ESP32 and makes it the perfect board for any project related to the **Internet of Things**, convert the board to a Wi-Fi Access point, creating a local network that can run a small web server hosting websites.

Digital & Analog Pins



Help

The Arduino Nano 33 IoT board has 19 Digital Pins marked as D2 - D21, and 8 Analog pins, A0 - A7. The analog pins can transmit or receive voltage values between 0 and 3.3 volts, relative to GND whereas the digital pins can only transmit data and voltage values of 0 or 3.3 volts, relative to GND. When they receive data, the voltage is interpreted as HIGH or LOW, relative to some threshold between 0 and 3.3 volts.

Pins marked as D14-D21 or A0 - A7 can both transmit and receive voltage values and there are also called General purpose Input/Output (GPIO) pins. The GPIO pins (D14-D21) can behaves as Analog pin and Digital pins depending on how we configure it.

> **Note:** You must never apply more than 3.3 volts to the board's digital and analog pins. Care must be taken when connecting sensors and actuators to assure that this limit of 3.3 volts is never exceeded. Connecting higher voltage signals, like the 5V commonly used with the other Arduino boards, will damage the Arduino Nano 33 IoT.

GND, 5V, 3V and VIN

Besides the analog and digital pins, the board also has a few Fixed functionality pins. These include the ground denoted as GND and power, 5V & 3V. These pins provide access to the board's ground and 5-volt reference voltage lines respectively. An additional Vin pin can be used to provide power to the board from an external battery source.

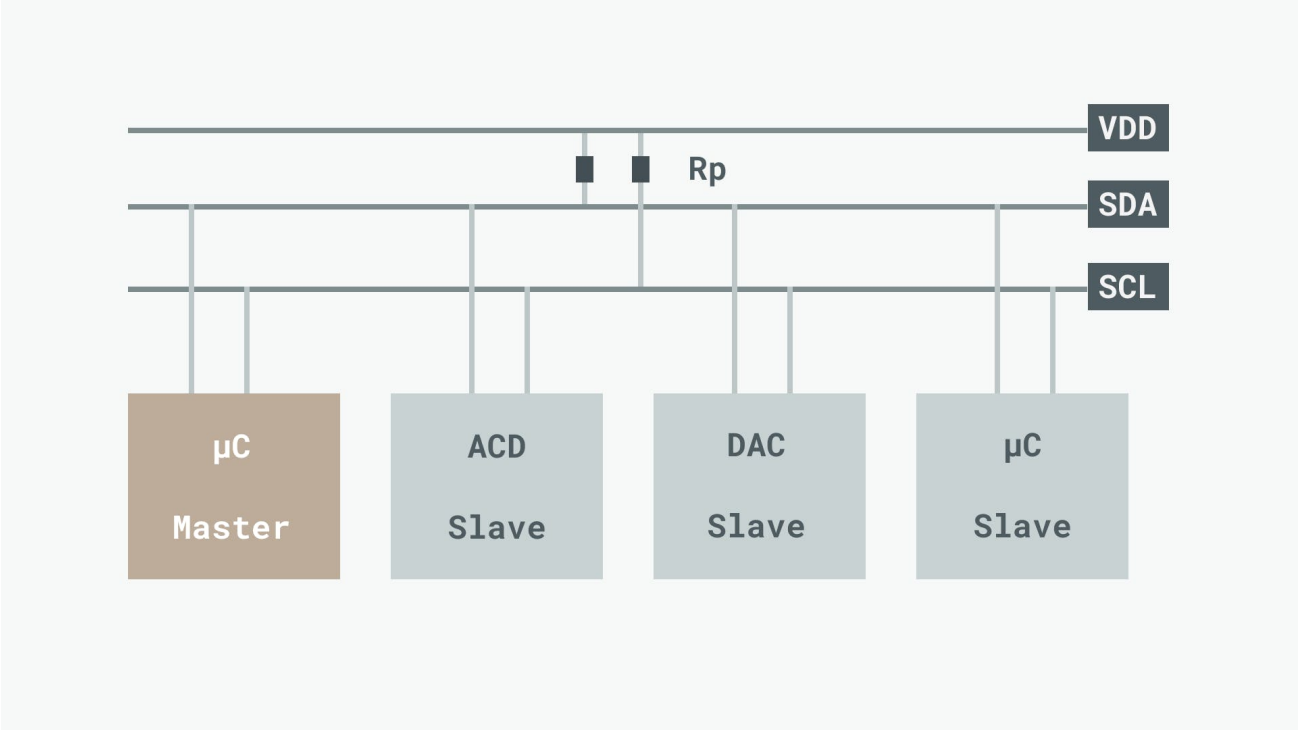
RX/TX

Used for communication between the Arduino board and a computer or other devices. All Arduino boards have at least one serial port (also known as a UART or USART), and some have several. It communicates on digital pins 0 (RX) and 1 (TX) as well as with the computer via USB. Thus, if you use these functions, you cannot also use pins 0 and 1 for digital input or output.

I2C

I2C (Inter-Integrated-Circuit) is a bus-based serial communication protocol that happens using two signal pins: SDA (data signal) and SCL (clock signal). Thi

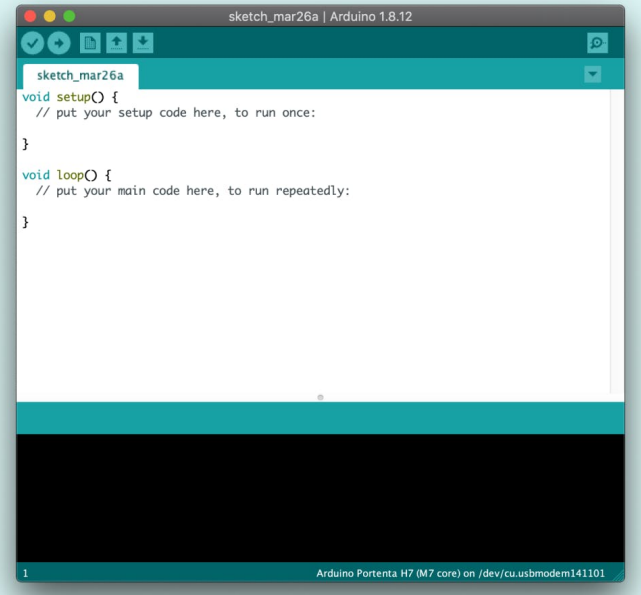
allows you to connect several devices to the same lines (SDA and SCL) and each device is selected by calling its unique address identifier. Typically, addresses are set up at the time of manufacturing, and many devices do not allow for address changes. The communication happens in a master-slave fashion, i.e. there has to be at least one device (usually a microcontroller) that leads the communication (master) while the rest (sensors, other uC etc) acts as slaves.



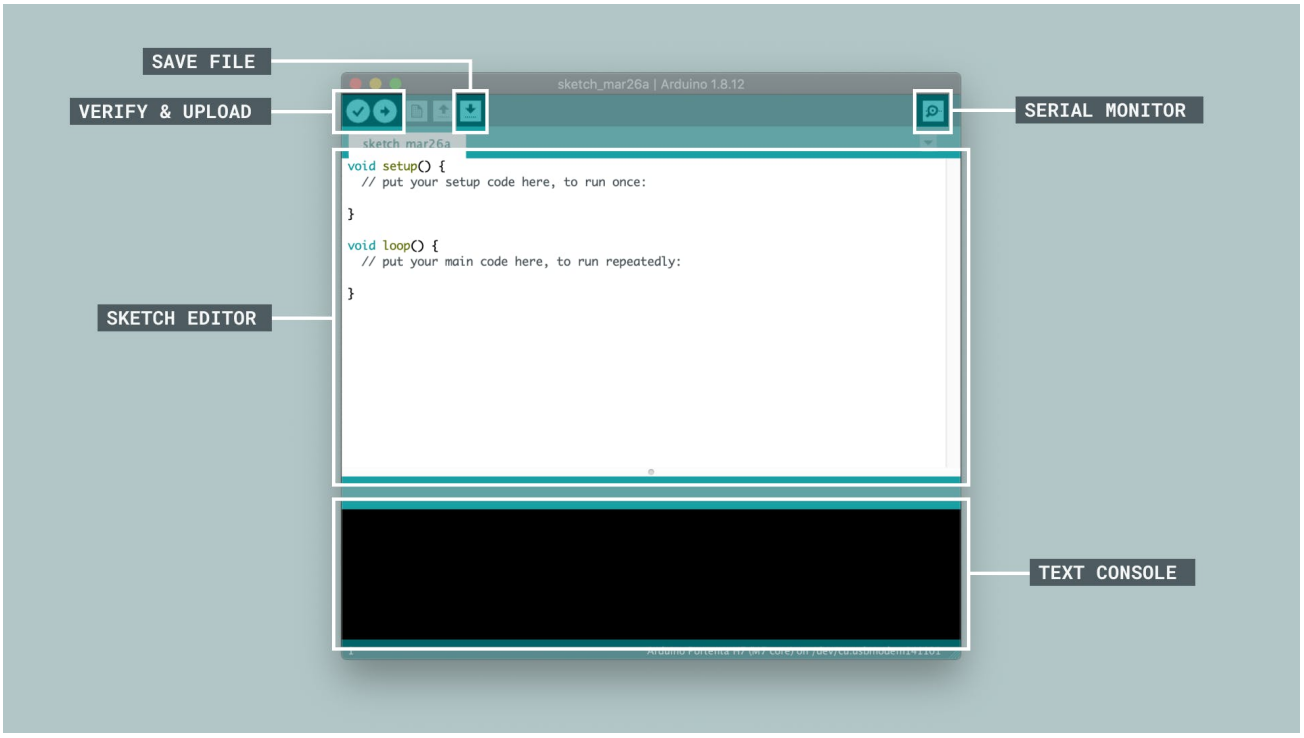
SDA and SCL bus wires require a pull-up resistor, which typically is 10 KOhm. Many microcontrollers that are used as masters in I2C implementations come with their own internal pull-up resistors as in the case of most Arduino boards.

The Arduino IDE

The Arduino Boards are programmed through Arduino's Integrated Development Environments. The open-source Arduino Software (IDE) makes it easy to write code and upload it to the board. It runs on Windows, Mac OS X, and Linux. The environment is written in Java and based on Processing and other open-source software. All the essential libraries, cores and software updates are contained in the environment making it easier for you to handle and manage your applications.



Parts of the IDE



Sketch Editor

Programs written using Arduino Software (IDE) are called sketches. These sketches are written in the text editor and are saved with the file extension .ino. The editor has features for cutting/pasting and for searching/replacing text.

There are two special functions that are a part of every Arduino sketch: `setup()` and `loop()`. The `setup()` is called once, when the sketch starts. It's a good place to do setup tasks like setting pin modes or initializing libraries. The `loop()` function is called over and over and is heart of most sketches. You need to include both functions in your sketch, even if you don't need them for anything.

Verify & Upload

Verify checks your code for errors by compiling it. Upload compiles your code and uploads it to the configured board. Before uploading your sketch, you need to select the correct items from the **Tools > Board and Tools > Port** menus.

Serial Monitor

This displays serial data sent from the Arduino board over USB or serial connector. You can also send data to the board by entering text and click on the "send" button found on the serial monitor. Choose the baud rate from the drop-down menu that matches the rate passed to Serial.begin in your sketch. Note that on Windows, Mac or Linux the board will reset (it will rerun your sketch) when you connect with the serial monitor.

Text Console

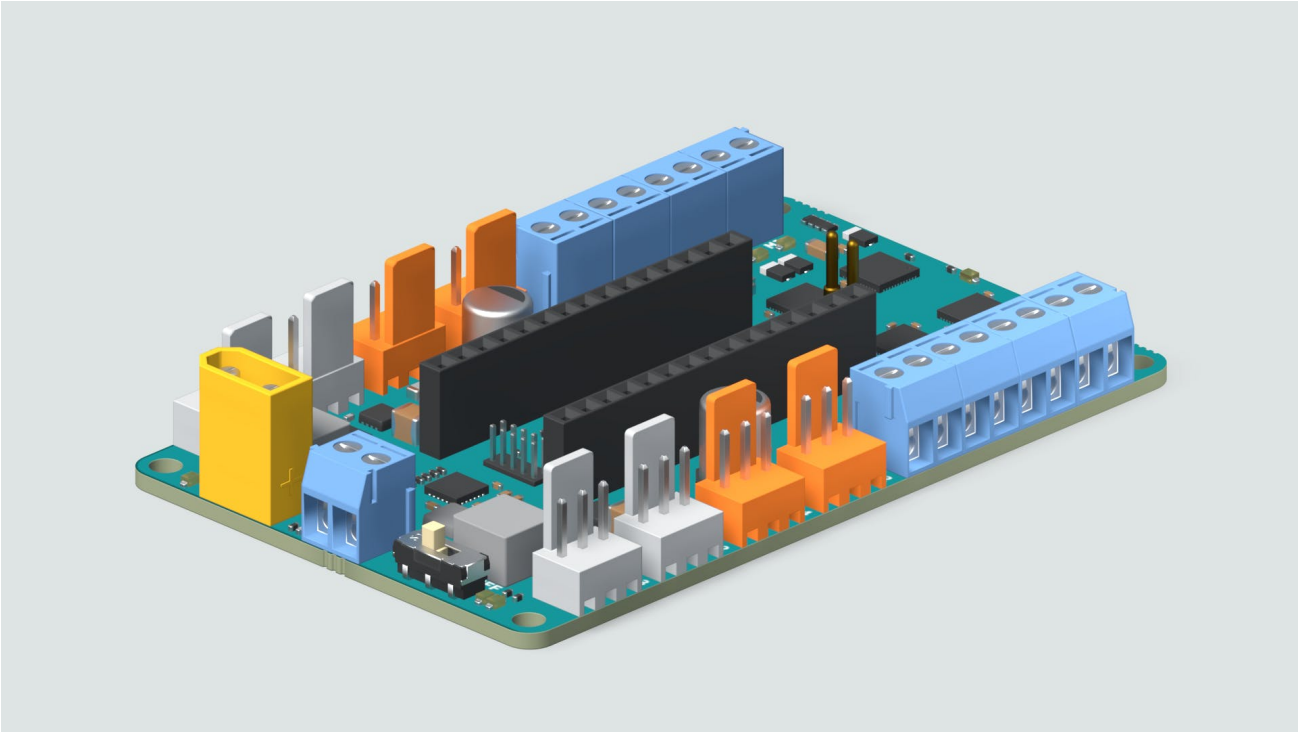
The console displays text output by the Arduino Software (IDE), including complete error messages and other information. The bottom right hand corner of the window displays the configured board and serial port.

To learn more about the Arduino IDE, visit our **guide**.

The Arduino Nano Motor Carrier

The Arduino Nano Motor Carrier is an Nano add-on board designed to control servo, DC, and stepper motors. The Carrier also allows you to connect other actuators and sensors via a series of 3-pin male headers. The carrier board can be plugge

underneath the Arduino Nano 33 IoT board and simplifies connecting various motors and analog sensors.



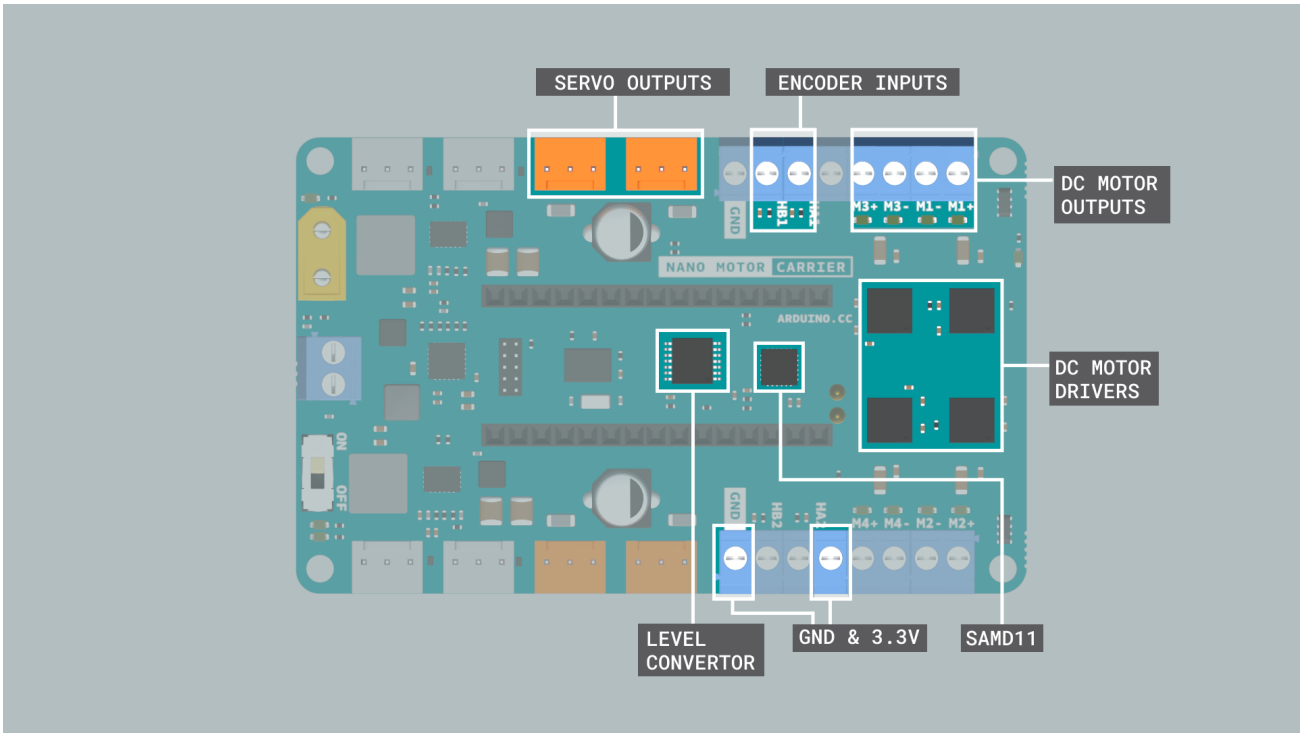
Tech specs

The summary of features is shown below:

Feature	Description
Microcontroller	ATSAMD11 (Arm Cortex-M0+ @48 Mhz)
Max Input voltage (power terminals)	4V (1S Li-Ion Battery)
Max output current per motor driver	1 Amp
Motor driver output voltage	12V
Over Temperature shutdown protection (for DC motor drivers)	Yes
Battery type	Li-ion battery (1S)
Battery charging	Yes
Max battery charging current	500mA (configurable)
Power terminals (connectors)	XT-30 and 2POS terminal block
Interface	Terminal block and 3 pin/4 pin header connector
Servo connector	4 terminals
Stepper connector	2 terminal
Encoder inputs	2 ports
DC motor control	4 ports
3V digital/analog sensor input/output	4 ports
IMU	BNO055 9axis Accelerometer / Gyroscope/ Magnetometer

The Arduino Nano Motor Carrier is compatible with all the boards from the Arduino Nano Family

Key Features



The SAMD11 Microcontroller

The SAMD11 is used to control the servos, read values from the encoders, and read the battery voltage in an autonomous way. This microcontroller receives commands and send information to the Arduino Nano 33 IoT via I2C.

Servo Outputs

The Servo pins handle all the communications between the servos and the Arduino Nano 33 IoT. Accessing the connected servos can be easily done with their corresponding Servo objects Servo1, Servo2, Servo3 and Servo4 from the IDE.

DC Motor Outputs

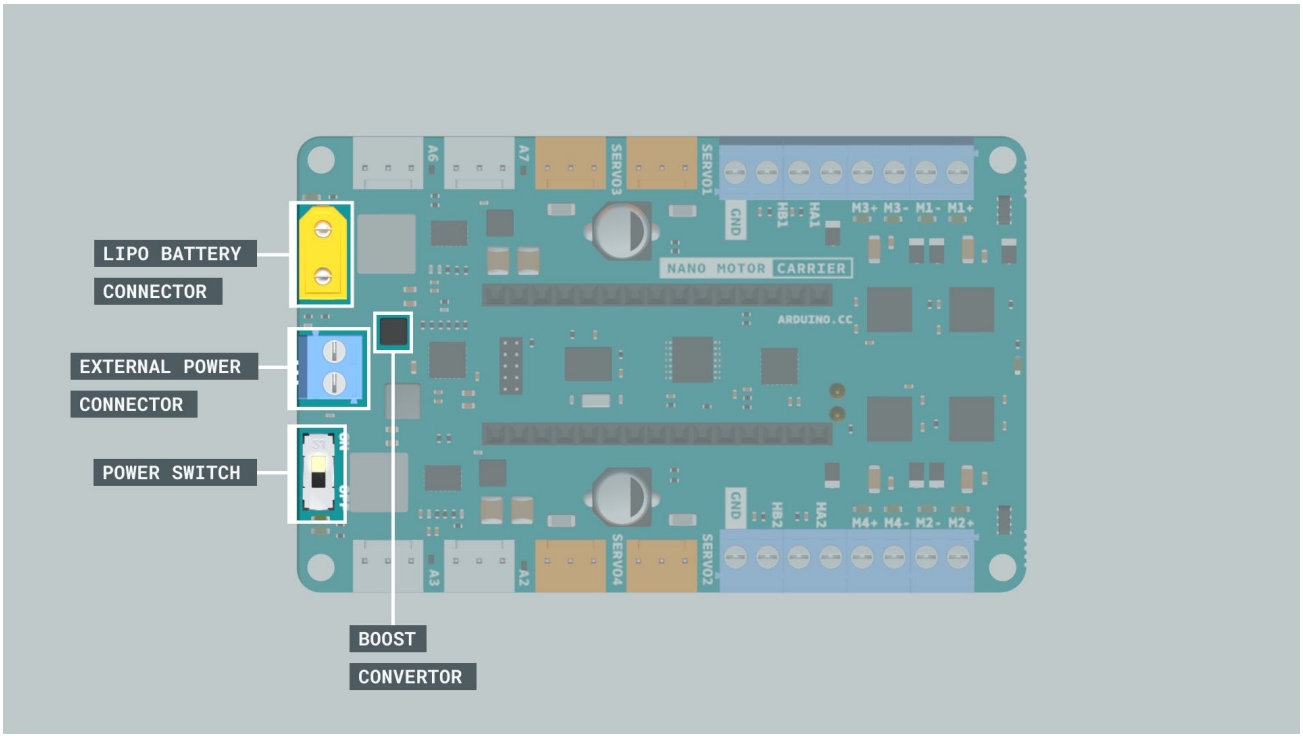
The DC motor terminal pins are used to connect any DC motor to the Carrier board. There are 4 terminals, M1, M2, M3 & M4, marked with a +ve and -ve sign. You can connect 4 DC motors at the same time but there are only 2 encoder inputs. The polarity signs signifies the direction of rotation of the connected motors.

DC Motor Drivers

The Motor Carrier features 4 motor drivers for high-performance DC motor control with direct connection to the Arduino Nano 33 IoT. Each driver allows you to control one DC Motor. The DC motors are connected to the Arduino Nano Motor Carrier using the blue connectors (terminal blocks) on each side of the board. The motors have to be connected in the pins that are labeled (M1+ M1-, M2+ M2-, M3+ M3- and M4+ M4-).

Encoder Inputs

These terminal blocks adjacent to the DC motor terminals can be used to read the encoder values from the attached DC motors through the terminals HA1-HB1 & HA2-HB2.



Lipo Battery Connector

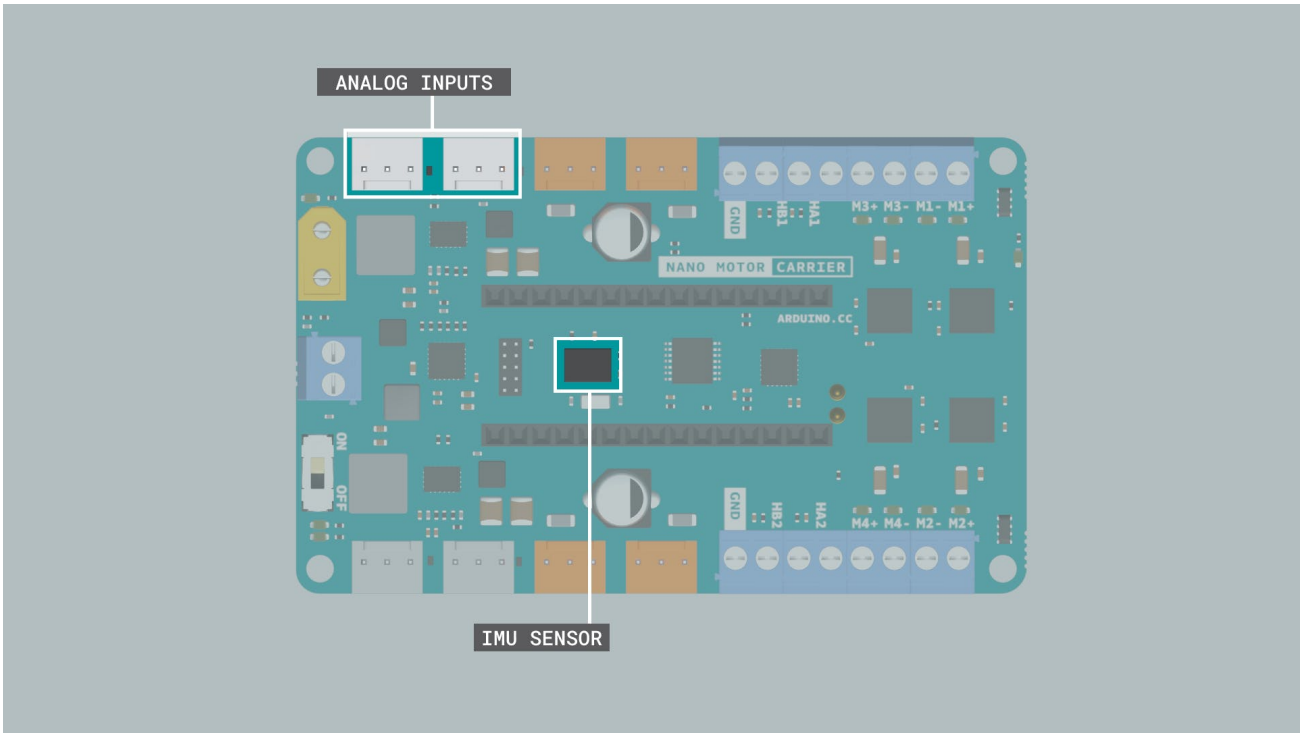
When working with motors, you will need an external source to feed the motor drivers and power up the motors. You can do this by connecting a LiPo Battery to the battery connector. It is recommended to do these operations with the power switch at the OFF position. Once the external power is connected to the board, the power switch can be turned on.

Boost Convertor

A boost converter is a DC-to-DC power converter that steps up voltage from its input to its output. All it consists of is an inductor, a semiconductor switch, a diode and a capacitor. The USB port cannot provide enough current to drive motors at the necessary voltage, and it can be dangerous to attempt to do so, therefore the boost convertor increases the voltage from the lipo battery source.

External Power Connector

Other external power sources can be connected to the VIN input on the terminal block.



Analog Inputs

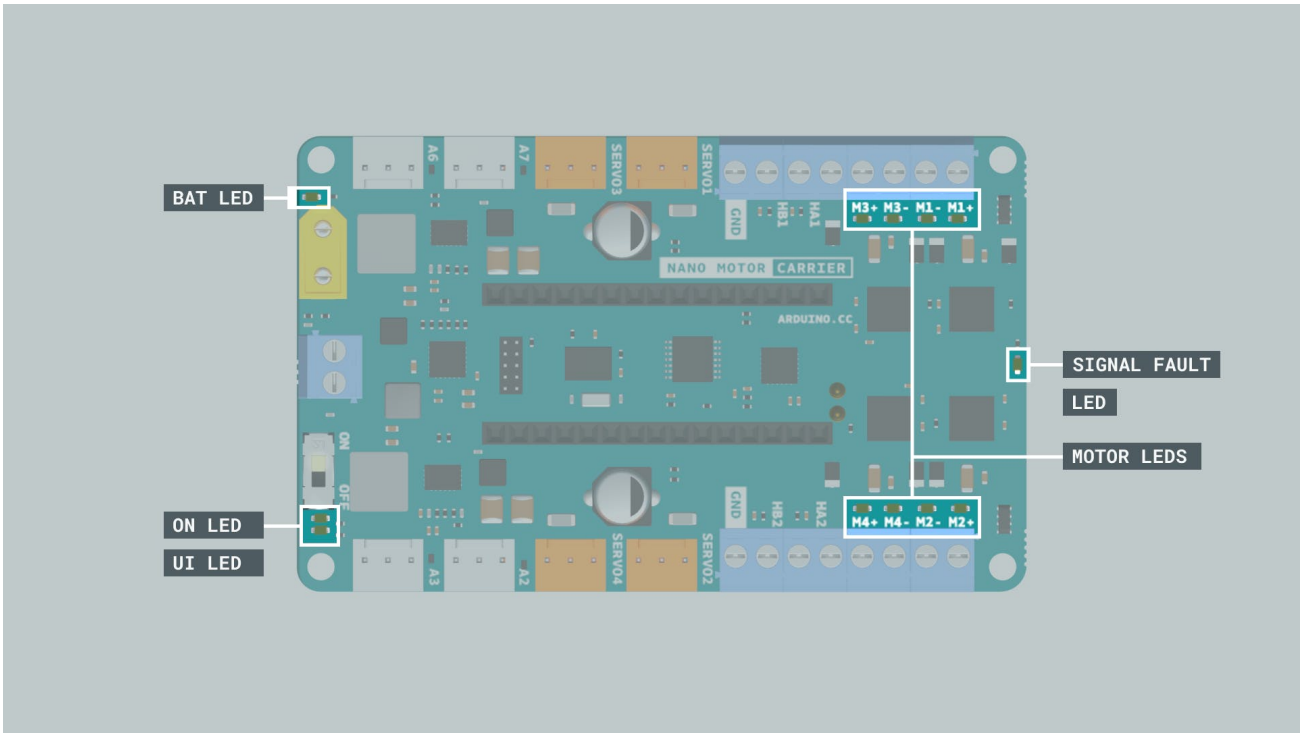
The carrier board also comes with 4 analog connectors used for connecting 3 pin analog sensors to the carrier board. These connectors are connected to the analog pins A2, 3, 7 and 6.

IMU Sensor

This carrier features the BN0055, a 9-axis (acc+gyro+magnetometer) orientation sensor. This IMU (Inertial measurement unit) module can measure how the carrier

board behaves under changes in linear acceleration, angular rotation, and, in some cases, the magnetic field around the module.

Onboard LEDS

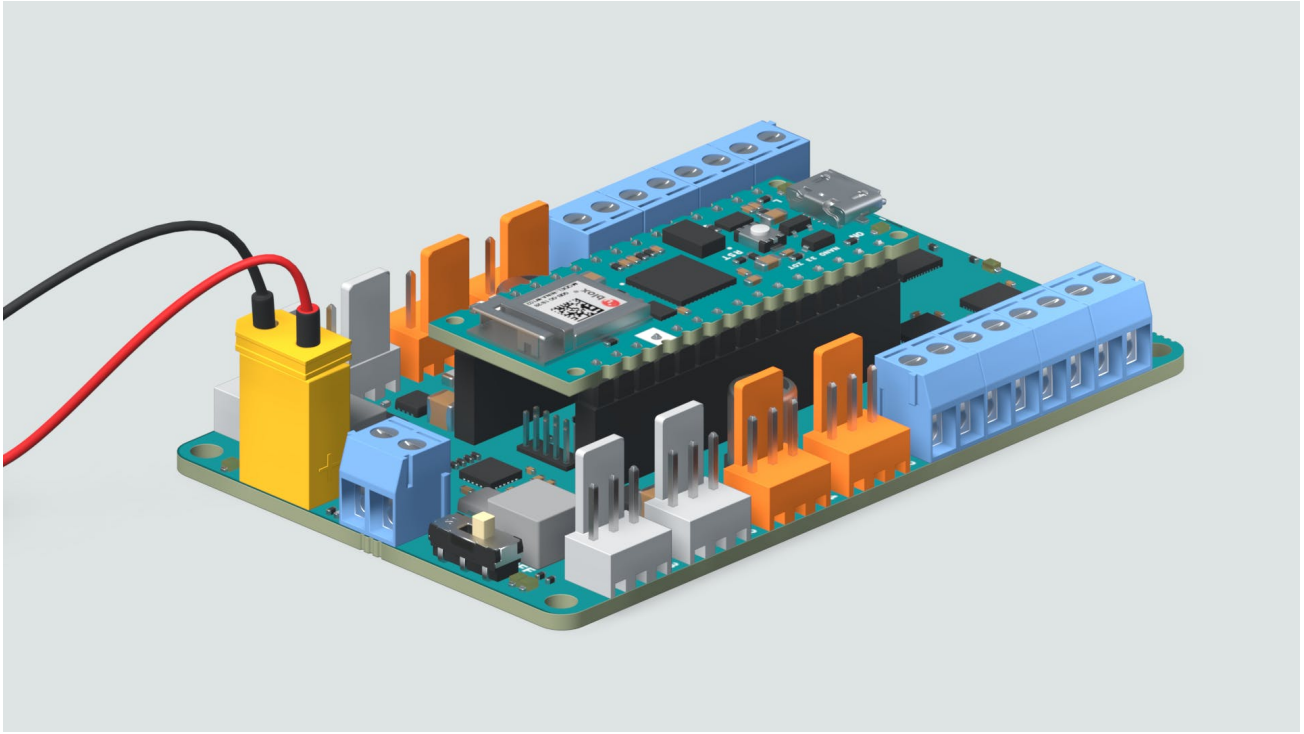


- **Motor LED** : This led indicates which motors are on and which directions they are spinning.
- **Signal Fault LED** : This lights up when there's a fault condition in the driver due to over current or over temperature.
- **On LED** : This LED signals when the board is powered.
- **UI LED** : User interface LED usually lights up when there is a problem with the library version that is incompatible with firmware version.
- **BAT LED** : The Battery LED Lights up when the battery is being charged.

Using the Carrier

To use the Carrier, you will need to plug an Arduino Nano 33 IoT board on the headers at the center of the board. Make sure the Arduino Nano 33 IoT is connected proper direction. You can do this by making sure the info printed on the sic

headers are matching for both the Arduino Nano 33 IoT and the Motor Carrier or by looking at the illustration below.



When plugging the Arduino Nano 33 IoT and the Motor Carrier, some of the pins will stop being available for you to use in your code, as they will be needed to control some of the features of the Carrier. For example, some of the pins of the Nano will be used to control 2 of the DC motors. The following list explains which pins of the Nano are used to control the Carrier:

- Digital pin D2 for IN2 signal for Motor3
- Digital pin D3 for IN1 signal for Motor3
- Digital pin D4 for IN2 signal for Motor4
- Digital pin D5 for IN1 signal for Motor4
- Digital pin D6 for Interrupt signal from the SAMD11 to the Arduino Nano 33 IoT
- Digital pin A4/SDA for the SDA signal (I2C)
- Digital pin A5/SCL for the SCL signal (I2C)

Once the board is properly connected to the Motor Carrier, you can start programming the board. To control the motors, you will need to import the Arduino Nano Motor Carrier library. You can find more information about the Arduino Nano Motor Carrier library library **here**. If you want to read more about how to install a library, follow this **link**.

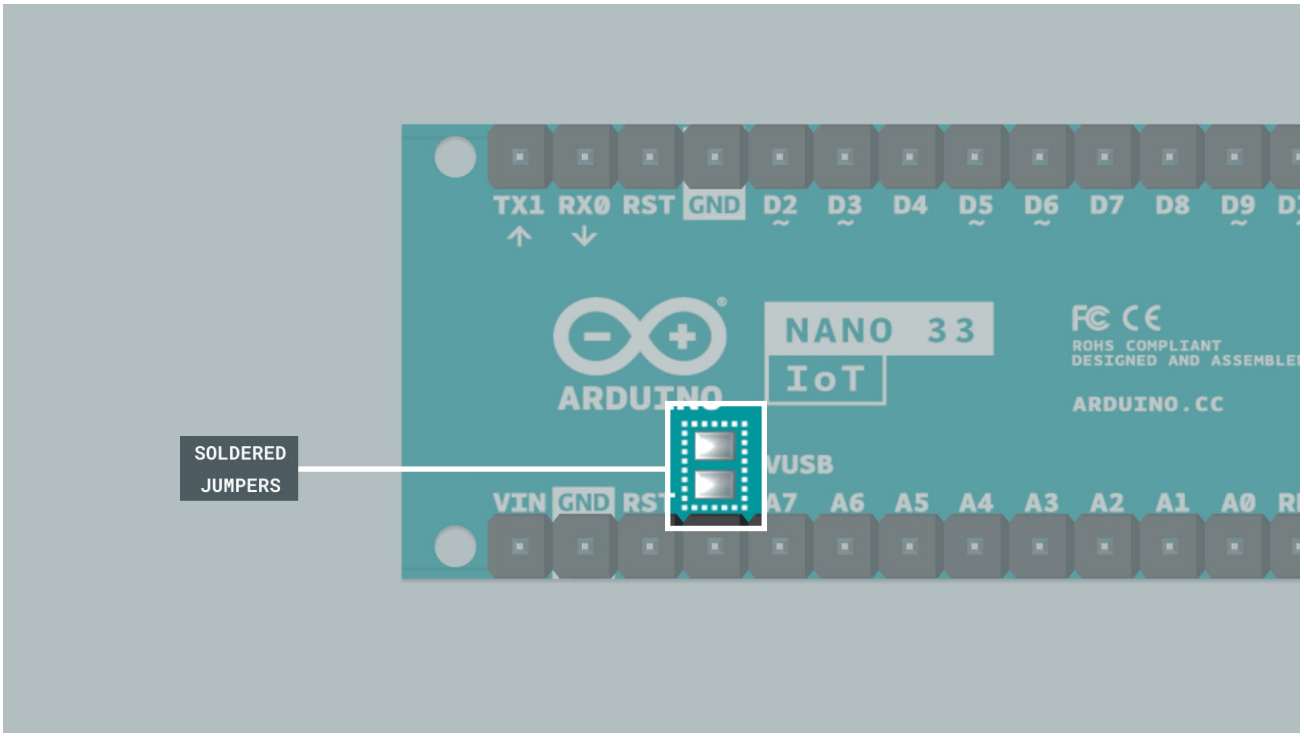
Charging the Battery

The Arduino Nano Motor Carrier comes with a built in battery charger that allows you re-charge your lipo batteries by connecting it to the board. When the Carrier is connected to matlab, it gets initialized first in order to start the charging process. This happens automatically everytime you load a sketch from MATLAB and Simulink through the `controller.begin()` command.

For your battery to charge fully without any failure ensure that

- The Arduino Nano 33 IoT Board is connected to the Arduino Nano Motor Carrier,
- The Arduino Nano 33 IoT Board is connected to the power source via the USB,
- The power switch on the Arduino Nano Motor Carrier is On.

There are cases where you might be using a Arduino Nano 33 IoT board not provided in the Kit. In such a scenario users must check that the solder jumper under the board is soldered (this is always the case for the Arduino Nano 33 IoT that comes with the Engineering Kit R2). In case you find out that the Jumpers arent soldered, please solder it yourself



The command controller.begin() must be called to enable battery charging. This is already taken care of by MATLAB and Simulink but must be added if using the Arduino IDE.

The Motor Carrier in Projects

In the projects of this kit, we will be using the Arduino Nano Motor Carrier with the Arduino Nano 33 IoT because of the Wi-Fi capabilities of that board. Thanks to a special configuration within MATLAB, the board can communicate directly to the software as long as they are on the same wireless network.

- **Self-Balancing Motorcycle**: In the self-balancing motorcycle, the Carrier will be used to control the inertia wheel in the center of the motorcycle. It will also command the geared DC motor in the back wheel to move forward or backward, and a servo in the front to steer the motorcycle.
- **Webcam controlled Rover**: In the rover, the Arduino Nano Motor Carri
used to control the two DC geared motors and the encoders that are cc

the rover's wheels. In addition, the Carrier will be used to control a servo motor in the front of the vehicle, providing the rover with a lifting mechanism.

- **Drawing Robot**: In the drawing robot, the Carrier will be used to control the two DC geared motors with an encoder to position the robot on the whiteboard. In addition, it will be used to control a servo motor involved in changing the marker color as well.

To know more about the components, connections and the pinouts you can refer to the **DATASHEET OF THE MOTOR CARRIER**

2.2 Getting Started with MATLAB

MATLAB is powerful software that allows you to make complex calculations on large datasets. MATLAB has its own textual programming language with instructions that can be collected into scripts. You could imagine MATLAB to be like a command-line interface for mathematical operations: you can execute commands directly on a terminal-like interface or make batch scripts that can be invoked from the same terminal.

In this section, you will learn:

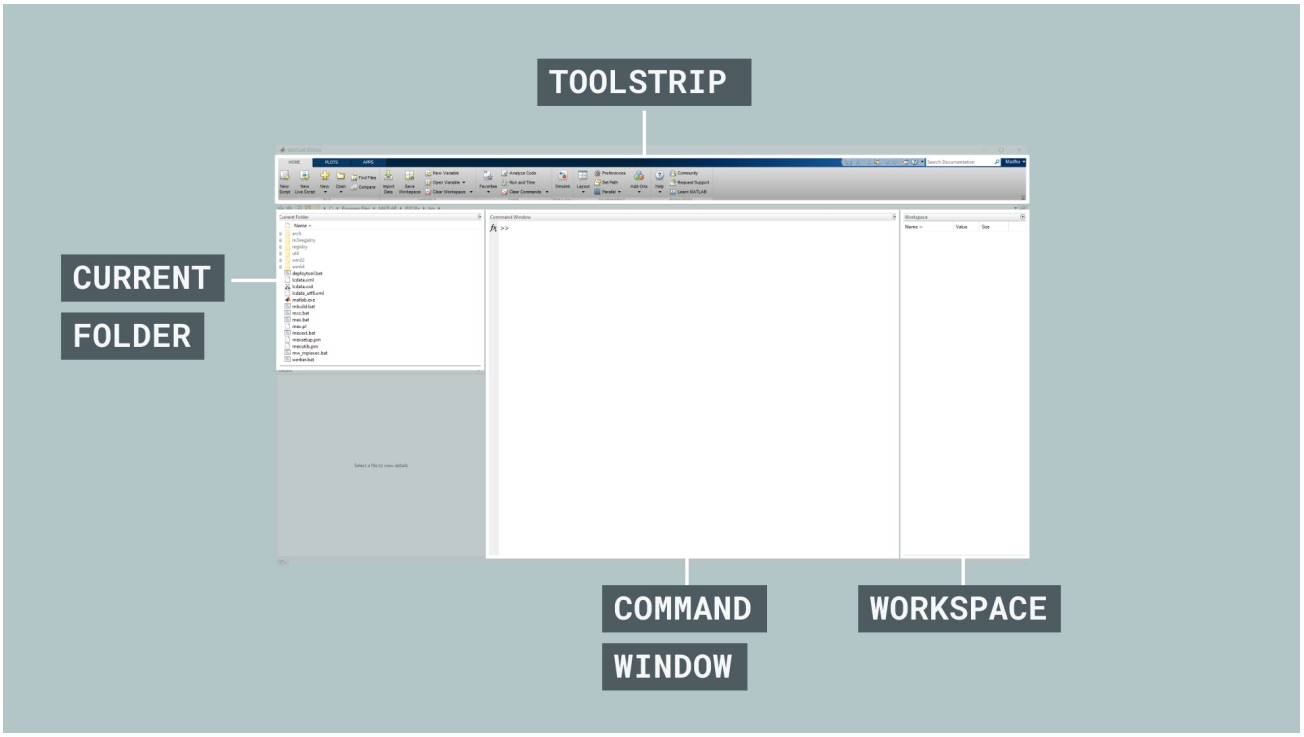
- How to identify the core components of the MATLAB desktop environment,
- How to issue MATLAB commands in the command window,
- How to create scalar and vector variables and apply functions and arithmetic operations to existing variables,
- How to access and manipulate data stored in vectors,
- How to create and label line plots of vector data,
- How to save and load MATLAB data to and from disk,
- To write MATLAB scripts and functions.

> **Note:** Install MATLAB, Simulink, and any other extensions before you start with this section. You can find more information about how to do that in "1.3 Installing MATLAB, Simulink & Add Ons".

The MATLAB IDE

Launch MATLAB from your Start menu or by double-clicking the executable file. For Windows users, the executable is called matlab.exe and is usually located in **C:\Program Files\MATLAB\R2021b\bin** where 2021b indicates the version the software in use.

Once you have launched the MATLAB Desktop, you will notice four panels in the MATLAB Desktop that you should familiarize yourself with. They are named **Current Folder**, **Command Window**, **Workspace**, and **Toolstrip**.



The **Current Folder** panel acts just like your file explorer. it indicates your current location in the computer's file structure and its contents. **Command Window** is the command-line interface (CLI for short) where you can experiment with commands in the MATLAB language before adding them to a MATLAB program. The **Workspace** will display a list of existing MATLAB variables and some of their properties. At the top of the MATLAB desktop is the colorful **Toolstrip**, which contains buttons for basic file operations, basic MATLAB operations, and software environment settings.

Just below the Toolstrip, you will see a horizontal white bar that indicates the current location within your file system. It is important to be mindful of your current folder (the folder from which you are executing MATLAB commands), because various operations in MATLAB and Simulink will generate files into the current folder. It is useful to have a designated "work" directory, to save your work into and keep all generated files in one place.

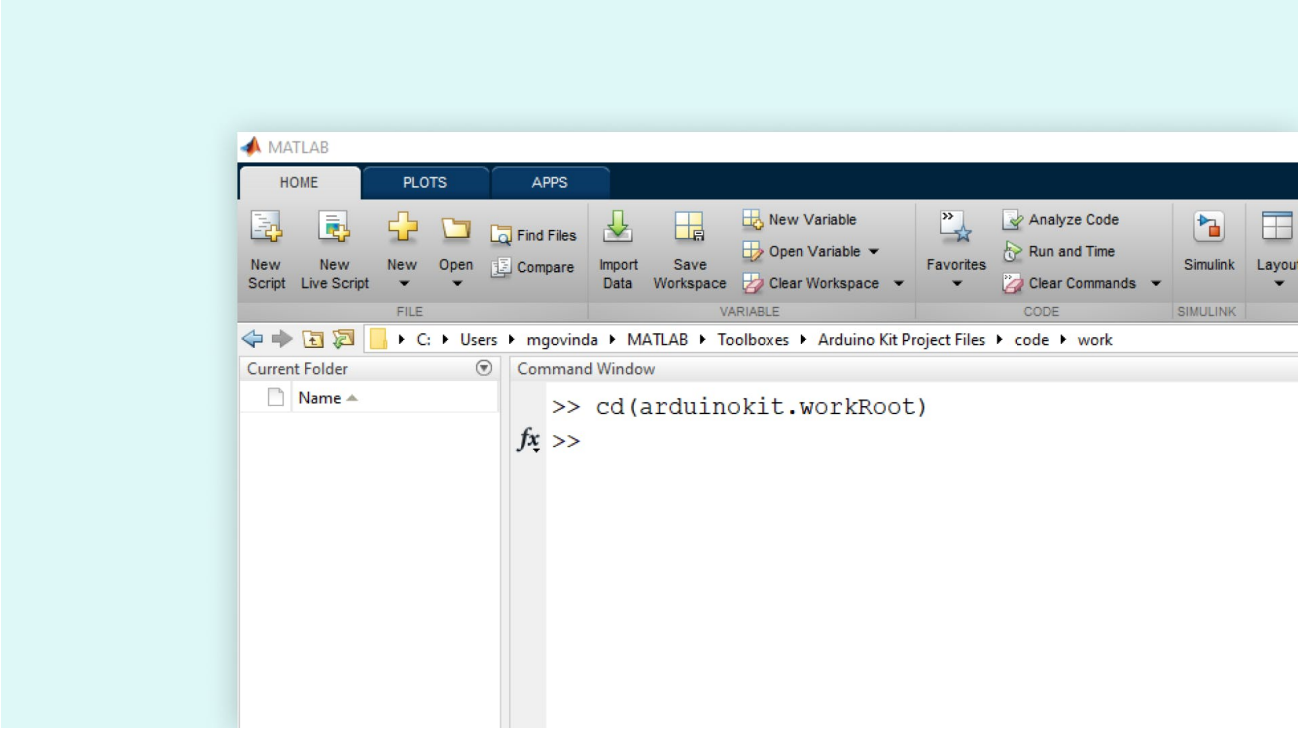
MATLAB Commands

Help

MATLAB commands are used for changing the working directory, creating variables and performing calculations with data.

Navigate to the "work" folder by typing the following command in MATLAB Command Window and pressing **Enter**.

```
>> cd(arduinokit.kitRoot)
```



Let's now get used to some basic MATLAB syntax. In the Command Window, type the following command to create a new variable `a`:

```
>> a=7
```

Next, create a variable called `b` by assigning a value to it.

```
>> b = 8;
```

You can revisit previous commands by using the **UP** button on your keyboa
the **UP** key until you get back to your original command, `a=7`, and press **E**

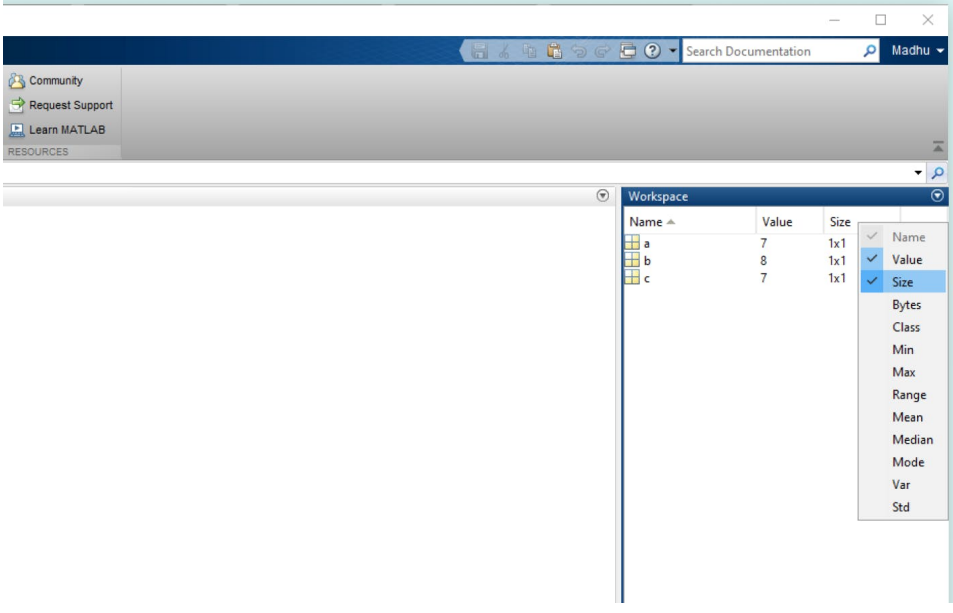
bring the same command back, but change the variable name to `c`. These operations will generate three variables (`a`, `b`, and `c`) that will be available for you to use. It is now possible to make operations using the variables as part of equations, vector declarations, etc.

As you can see, white space around operators is irrelevant in terms of MATLAB execution, but it is often easier to read code in which operators and operands are spaced out.

If you want to simply execute a line of code without seeing the result in the Command Window, you can end the line with a semicolon (`;`).

Scalar Variables

Look in the **Workspace** panel. You will see there the three MATLAB variables `a`, `b`, and `c`. By default, the **Value** of each variable is shown, provided it is not too large to be displayed in the **Workspace** panel. You can display various other attributes of your variables by right-clicking the column headers in the **Workspace** panel. Right-click the column headers, which will display a series of options in a list; locate the **Size** column and click to enable it. This will add a new column to the interface that shows the sizes of the variables.



In MATLAB, the size of a variable indicates its dimensions, in a rows x columns format. As you can see, all your current variables are 1 x 1 (1 column, 1 row), or scalars. Let's create another scalar. Type the following command, and then examine the **Workspace**:

```
>> A = 1.23;
```

You can see that the variables `A` and `a` are distinct in MATLAB. In other words, MATLAB is a case-sensitive language. Try the following commands, but this time do not peek at the **Workspace** yet:

```
>> B = 4.56;
>> A = B;
>> B = 3.14;
```

What do you think the value of `A` is? Check the **Workspace** to see if you were correct.

The correct answer is that the value of `A` is `4.56`. As you can see, MATLAB variables only store values. They do not propagate relationships with other variables that may have been used originally. If you would like `A` to reflect the new value of `B`, re-enter the command `A = B`.

Now let's delete some variables. The **clear** command deletes a subset or all the variables from the **Workspace**. Delete the variables `a` and `b` by typing the following command, and then examine the **Workspace**:

```
>> clear a b
```

You can delete all the variables by typing the following command:

```
>> clear
```

Vector Variables

While we can achieve a lot with scalars, most meaningful data consists of a 1-dimensional array of numbers, called a **vector**. A vector can be oriented as a row vector or a column vector. Create a row vector v1 of length 7 by entering the following command, and then examine your **Workspace**:

```
>> v1 = [1 3 2 6 4 4 9]
```

Now create a column vector v2 of length 5 as follows, and examine your **Workspace**:

```
>> v2 = [2;5;7;1;2]
```

When you have variables that are larger than a scalar, you may need to access a subset of that variable. This is known as indexing. Enter the following commands to index into a 1x1 portion of vector variables `v1` and `v2`:

```
>> x = v1(3)
>> x = v2(5)
>> x = v2(end)
```

Similarly, you can use variable indexing to modify a portion of an existing non-scalar variable. Enter the following command to change one value in `v2`:

```
>> v2(2) = -1
```

You can reorient a vector as a column or row using the transpose operator (`'`). Enter the following command to reorient v2 as a row vector:

```
>> v2 = v2'
```

Earlier you used square brackets (`[ ]`) with comma (`,`) or semicolon (`;`) delimiters to create row and column vectors respectively. This syntax is useful to create short vectors with arbitrary values. What about much longer vectors? Often you will need a vector of uniformly spaced values to represent a series of periodic time val
geometric coordinate. Other long vectors might be populated by a series o

values measured at those times or coordinates. Still other vectors might be the result of some processing of the raw measurements. For now, let's just generate vectors of uniformly spaced values. Try the following commands:

```
>> v3 = 0:6
>> v4 = 0:0.5:6
```

You can see that in the first case, you generated a row vector `v3` with an increment of 1 between the two endpoints 0 and 6. The resulting vector is of length 7. In the second case, you generated a vector `v4` with a specified increment of 0.5 between the same endpoints. The resulting vector is of length 13. What if you want a column vector? Repeat the previous command, but apply the transpose operator to the entire expression:

```
>> v4 = (0:0.5:6)'
```

Now let's create a vector with some real-world relevance:

```
>> theta = -2*pi:pi/10:2*pi;
```

The vector **theta** will represent an angle expressed in radians, spanning two full revolutions. We will use this vector in the following sub-section.

Built-in Functions

MATLAB comes with a vast library of built-in utilities, called functions, which can operate on your data to produce meaningful results. The functions introduced in this section are the ones that will be used later in the projects. Try the `sin` function, using the following syntax:

```
>> y = sin(pi/2)
```

The **sin()** function, and the other built-in trigonometric functions, operate on angles expressed in 3 radians. Now try taking the sine of many angles at once, an

examine `y` in your **Workspace**:

```
>> y = sin(theta);
```

When you input a vector into a function, and the output is a vector of the same size, the function is said to be vectorized. Vectorization is a key feature of the MATLAB language, and it allows you to write simple, concise lines of code that execute many operations at once, and which closely resemble mathematical expressions. You can input multiple operations and function calls in a single line of MATLAB code. Enter the following statement to apply a phase offset, a frequency scaling, a gain, and an amplitude bias to the sine wave:

```
>> y = 3*sin(2*theta + pi/2) - 1.5;
```

MATLAB has many built-in functions that generate outputs of arbitrary dimensions. Sometimes it is useful to produce an array of a given size that you will later populate with meaningful data. Try the following commands to generate two-dimensional matrices of all ones and all zeros:

```
>> z = ones(3,4)
>> z = zeros(4,3)
```

Since a given MATLAB expression can operate under many different circumstances, you may need to determine certain properties of your data before operating on it. Use the following commands to determine the size of `y`, `z`, and `theta`:

```
>> size(y)
>> size(z)
>> size(theta)
```

Now let's use this information to allocate an array `a` that is the same size as `theta`:

```
>> a = zeros(size(theta));
```

Help

Another function that operates similarly to zeros and ones is **randn()**, which generates an array of normally distributed random numbers with a mean of zero and standard deviation of 1. Try generating a list of 20 such numbers by calling the following function:

```
>> randn(20,1)
```

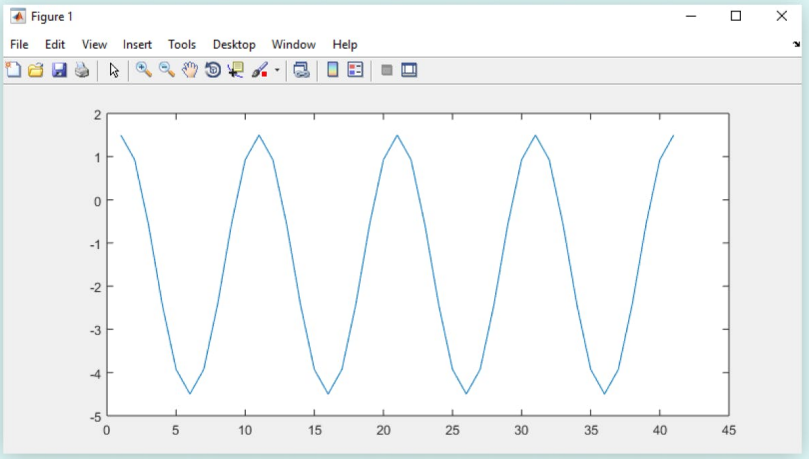
Finally, combine the previous techniques to add some random noise to your sine wave to produce a new vector `z`:

```
>> z = y + 0.3*randn(size(theta));
```

Visualising Data

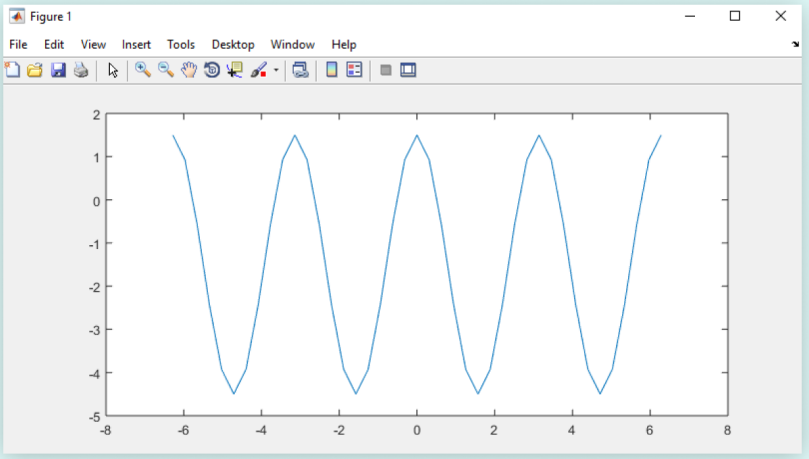
The quickest way to gain insight from your data is to visualize it using some type of plot. In this introduction you will create simple line plots of your vector data. Use the **plot()** command to visualize the values in the vector `y`:

```
>> plot(y)
```



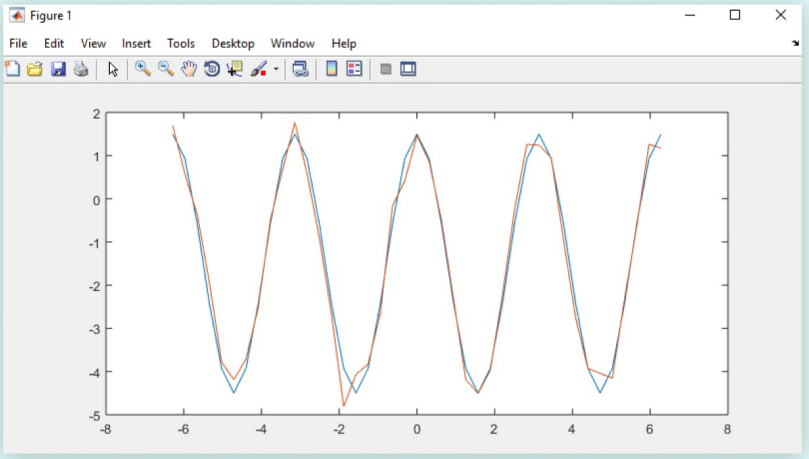
As you can see, when you plot a single vector, it renders as a continuous line joining the discrete values in that vector. The x-axis represents the indices of that vector from 1 to N. Now let's plot the vector `y` against a more meaningful variable, the `angle` theta in this case:

```
>> plot(theta,y)
```



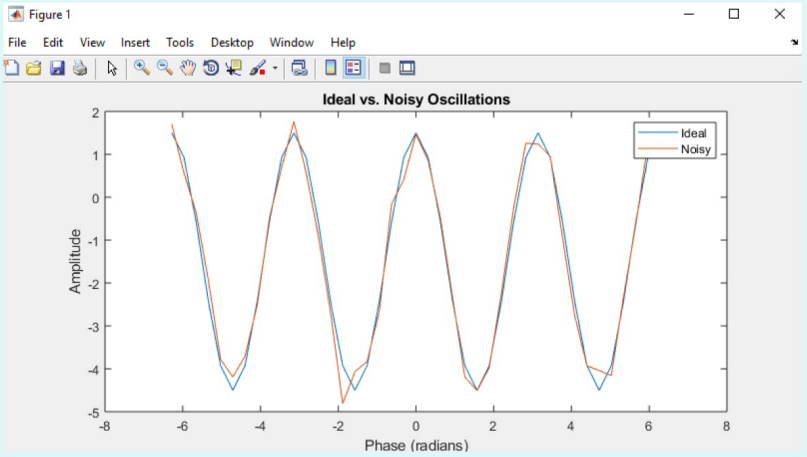
When you input two vectors to the `plot` function, the corresponding ordered pairs are plotted and connected by straight lines. Also notice that the new plot overwrote the old plot. Suppose you want to keep a plot in the figure window but plot a new line on top of it. You must **hold** the plot to avoid overwriting it. Try the following commands to plot a new line (taken from the variable `z`) on top of the sine wave:

```
>> hold on
>> plot(theta,z)
```



Finally, let's add some labels to the figure so that it is easier for your audience to understand. Use the following commands to add **axis labels**, a **title**, and a **legend** to distinguish the two lines:

```
>> xlabel('Phase (radians)')
>> ylabel('Amplitude')
>> title('Ideal vs. Noisy Oscillations')
>> legend('Ideal','Noisy')
```



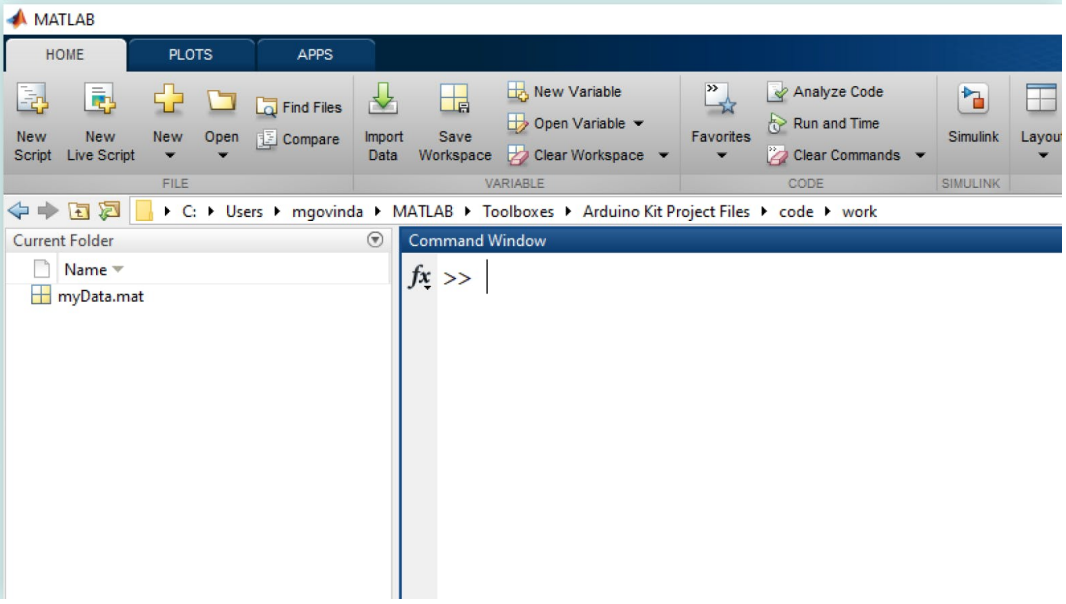
Note how the function `legend` works, as it assigns the labels to the plotted lines in the same order they were drawn.

Saving and Loading Data

You now have several variables in your workspace. Suppose you need to call it a day and shut down MATLAB. When you exit MATLAB, all your workspace data will be deleted. You can save important data to disk. Use the **save** command to save the variables `theta`, `a`, `y`, and `z`:

```
>> save myData theta a y z
```

Now, examine the Current Folder panel in the MATLAB desktop. You will see that MATLAB has created a new file called **myData.mat**:



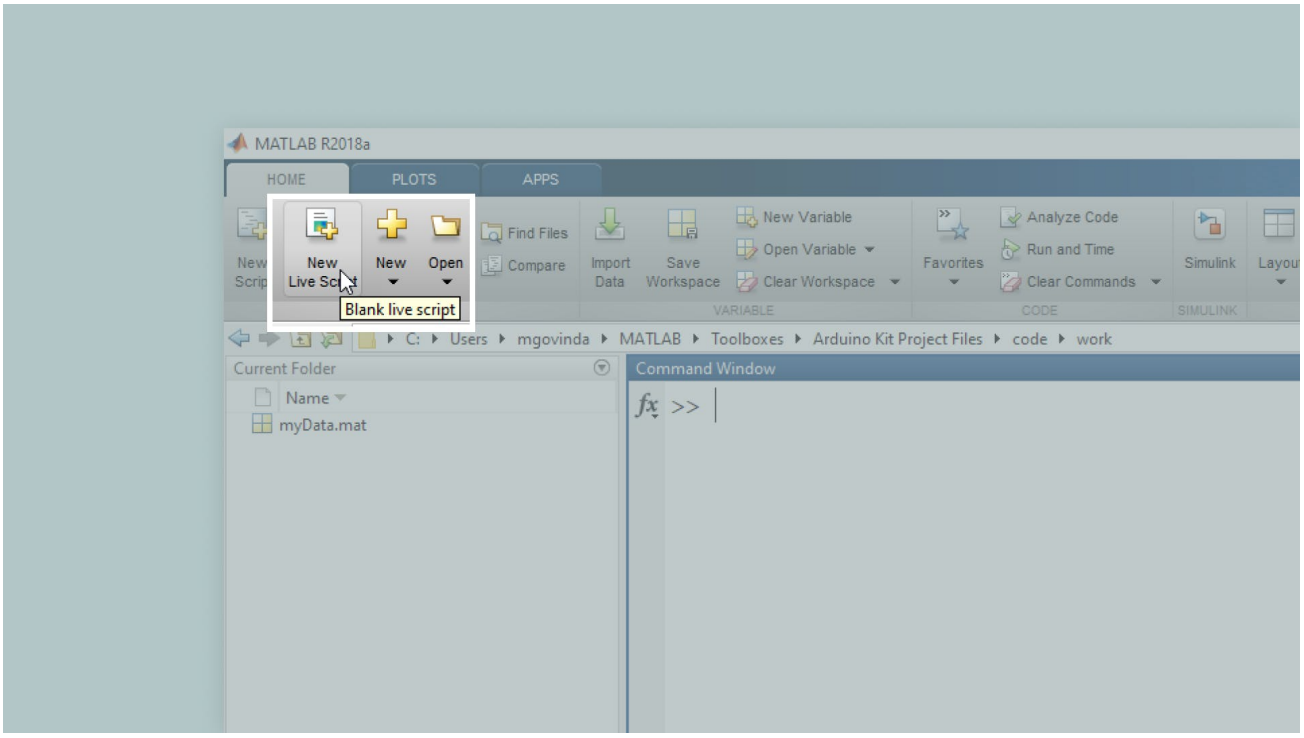
Let's test your new MAT-File. Use the following commands to clear all variables from the **Workspace** and then reload them using the **load** command:

```
>> clear
>> load myData
```

You will see now that the **Workspace** only contains the previously mentioned list of four variables and that all the others are gone.

Writing MATLAB Scripts

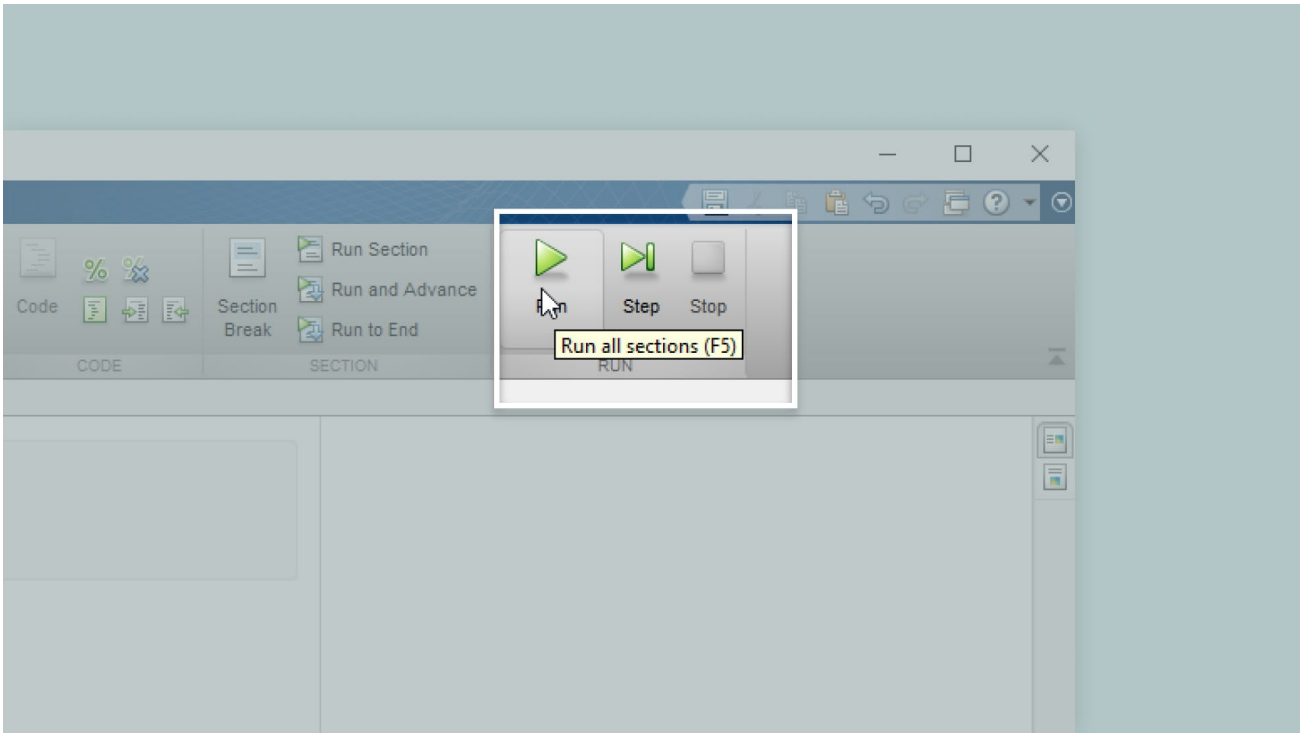
Up to this point, you have been typing commands directly on MATLAB's command line. If you perform certain operations many times, it might be useful to group those lines of code into scripts that you can easily call when needed. These are called **live scripts** in MATLAB's jargon. Begin by opening the **Live Editor** window. In the MATLAB Toolstrip , click **New Live Script**.



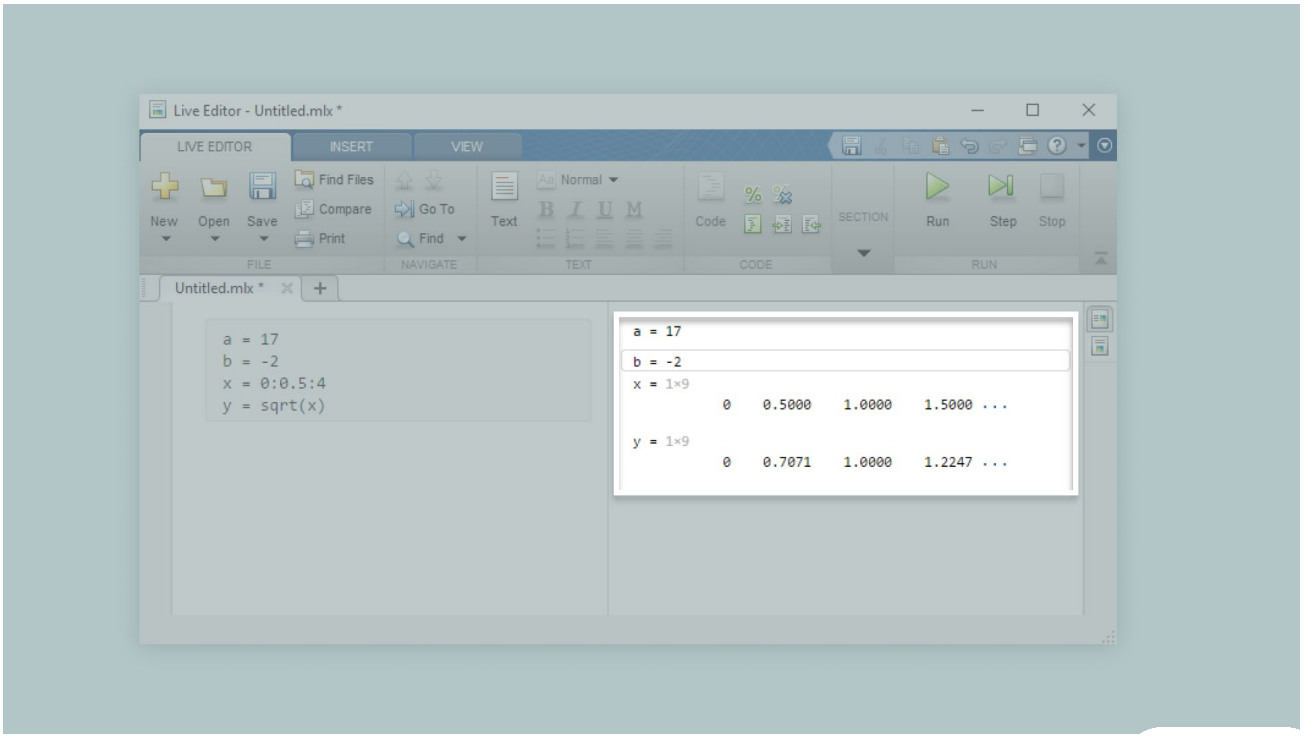
In the **Live Editor** window, you can type the same MATLAB commands as you would in the **Command Window**, and then execute them in batches. Start your **live script** by typing a few simple lines of code:

```
a = 17
b = -2
x = 0:0.5:4
y = sqrt(x)
```

Execute your live script by clicking **Run**:



Your commands execute sequentially, and assignment expressions modify variables in the MATLAB **Workspace**. Examine the **Workspace** and locate the new variables. As you can see, the results of expressions with unsuppressed output are displayed in the right column:

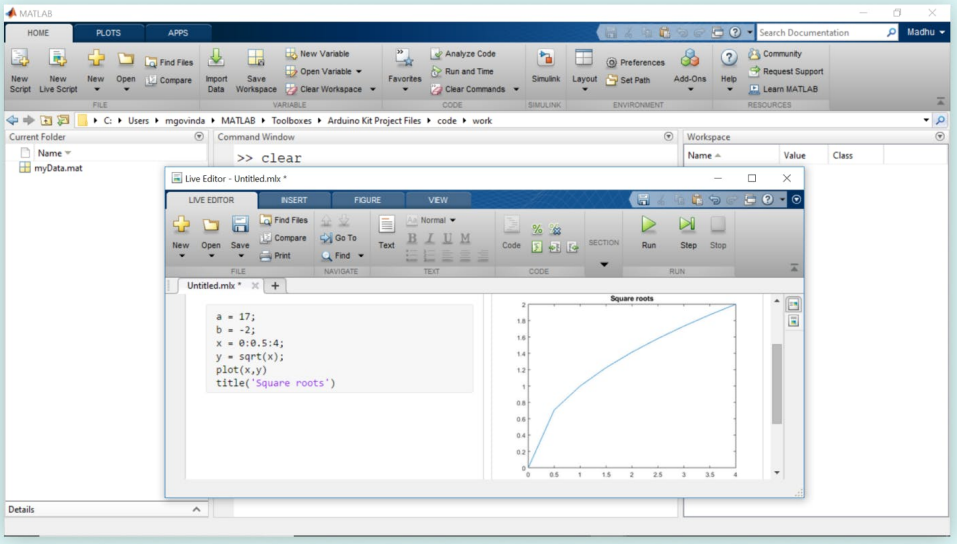


Help

Now suppress the output (which consists of adding a semicolon at the end of the command), and plot `y` against `x` by adding the code

```
a = 17;
b = -2;
x = 0:0.5:4;
y = sqrt(x);
plot(x,y)
title('Square roots')
```

Run the script and examine the embedded figure in the output column. Then clear your **Workspace** in the **Command Window**:




```
>> clear
```

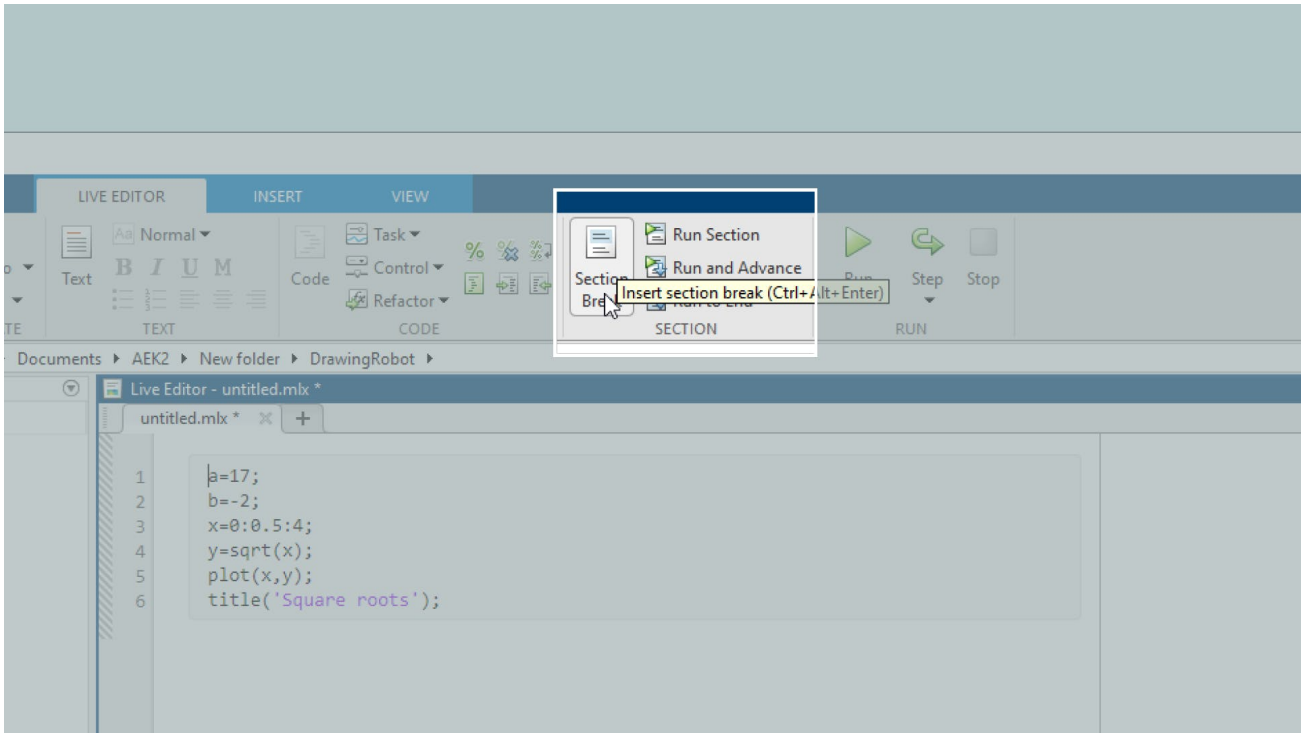
Now run your live script again. Examine the **Workspace** . You can see which information is being sent back to the command line and which isn't, depending on whether the variables are suppressed or not.

Now that you have learned how to use live scripts to execute a series of commands multiple times, we can use that knowledge to work with DC motors and ch

their behavior in terms of the PWM signal applied to them.

Sections, Texts and Comments

Matlab allows you to break down a script into sections, to divide it into logical steps. To partition your code, place the text cursor before the variables have been initialised and click **Section Break**.



Next, let's add some labels to the sections of the live script to explain what each one does. Place your text cursor at the beginning of the document and click **Text**:

```
%% 1. Initialize variables
a = 17;
b = -2;
x = 0:0.5:4;
y = sqrt(x);

%% 2. Plot the values
plot(x,y)
title('Square roots')
```

Any portions of your live script that are written in plain text have no impact on how the script executes.When you partition a live script into sections, you can e

section one at a time. This can be useful if you want to just test one section without running the whole live script.

You may also want to step through your code section by section, so that you can view and analyze intermediate results. Place your text cursor in the first section and click Run and Advance. Click **Run and Advance** once more (or click **Run Section** or **Run to End**) to complete execution of the live script. Now clear your **Workspace**, and run your live script section by section, examining the **Workspace** after each section executes.

You can also add comments to individual lines of code, if you want to add further explanation at a lower level. You can begin a comment directly in the MATLAB code by using the percent (%) character. Anything that follows % will not be executed in the live script. Add the following comments to existing lines of code:

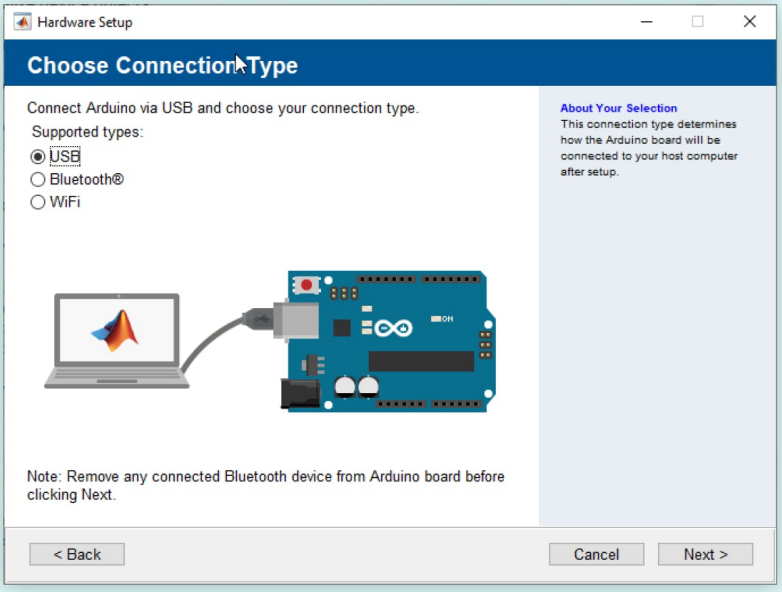
```
%% 1. Initialize variables
a = 17;
b = -2;
x = 0:0.5:4; %Array of values from 0 to 4 with an interval of 0.5
y = sqrt(x); %Calculate the square root of every value of X

%% 2. Plot the values
plot(x,y) %Plotting X and Y values
title('Square roots')
```

Connecting Matlab with Arduino

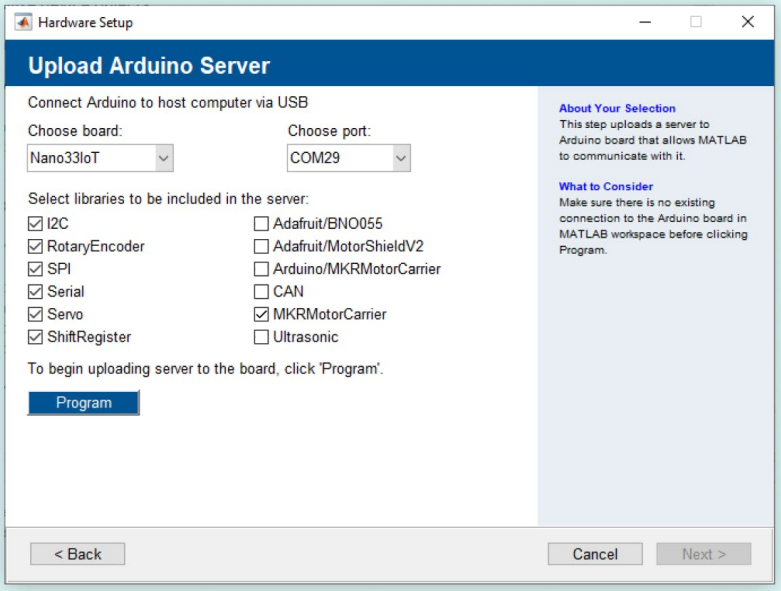
The next step is learning how to get your Arduino board to connect to MATLAB and learning enough of the basics to feel comfortable using the software on your own. Type the following command to set up the Arduino Nano 33 IoT in MATLAB.

```
>> arduinosetup
```

This will open the **Hardware Setup** window. For most of this lesson, you will communicate between your computer and the Arduino Nano 33 IoT board via USB connection. Therefore, you should select USB from the available options and click **Next**.



Next, you will select your Arduino board type, the port through which you will communicate with it, and the external devices you will need to have access to. Set your board to Arduino Nano 33 IoT. Set your communication port to the COM or ttyACM or usbmodem (depending on the OS) port you found earlier. Select **MotorCarrier**, **Servo**, **SPI** and **I2C**. Click **Program**.



Help

Note: This may take several minutes. If you receive an error message related to source not being found for the Arduino/ArduinoMotorCarrier library, then type edit ArduinoKitHardwareSupportReadMe.txt in MATLAB Command Window and follow the instructions provided on this text file.

After the libraries have successfully loaded to the Arduino Nano 33 IoT, click **Next** and complete the remainder of the dialog. You are uploading firmware to the Arduino Nano 33 IoT that will enable it to communicate back to your computer, where MATLAB will be executing commands with the help of the information that will be captured by sensors and sending it back to actuators via the same communication channel.

Now that the required resources have been loaded onto the Arduino Nano 33 IoT, we can access Nano and the devices from MATLAB. Enter the commands below to clear your **Workspace** and create an "arduino" object:

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

Unlike the variables in the previous section (which were numeric scalars, vectors, and matrices), the new variable `a` is a MATLAB object of type `arduino`. MATLAB objects are specialized variables with a fixed set of **properties**, which are meaningful values stored within the object, and **methods**, which are functions that operate on objects of that type. We will access and invoke some of these properties and methods shortly.

While the Arduino object (or any derived object) exists in your **Workspace**, MATLAB is connected to the Arduino board via a server utility. If you want to release this MATLAB server from Arduino so that you can run some other program on Arduino (for instance, from the Arduino software or Simulink), you need to delete the Arduino object in MATLAB. Enter the following commands to release Arduino from MATLAB and then re-establish the connection:

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

2.3 Getting Started with Simulink

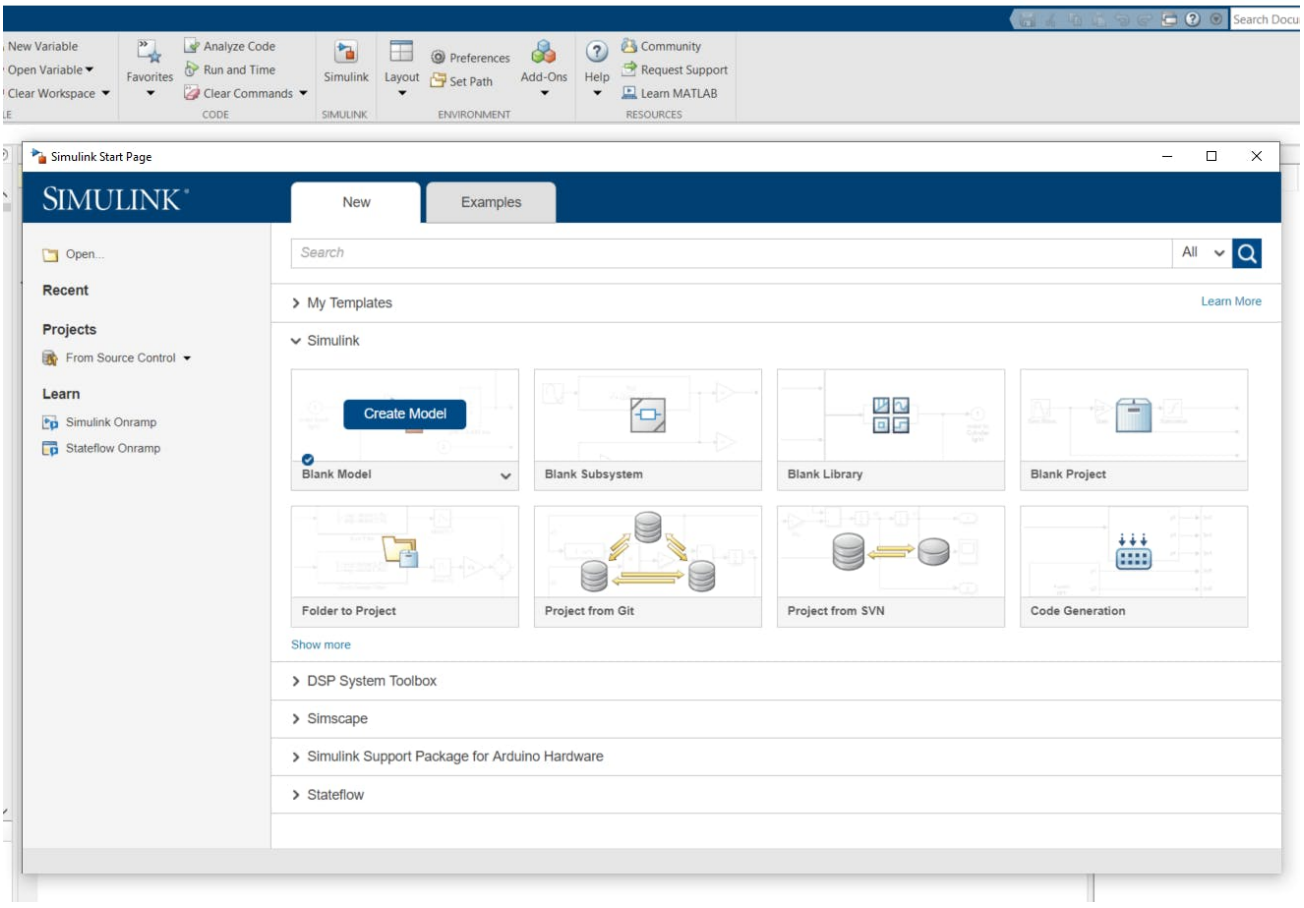
MATLAB allows you to perform a whole series of operations on data using code as an interface. Simulink offers a visual programming language (VPL) based on the use of a flow programming paradigm to route data, process it, save it, and send the outcomes to other programs, to MATLAB scripts, and even Arduino boards. One of the most exciting features of the Simulink VPL is the possibility of designing programs that could be uploaded to Arduino boards directly from within MATLAB IDE.

In this section you will learn:

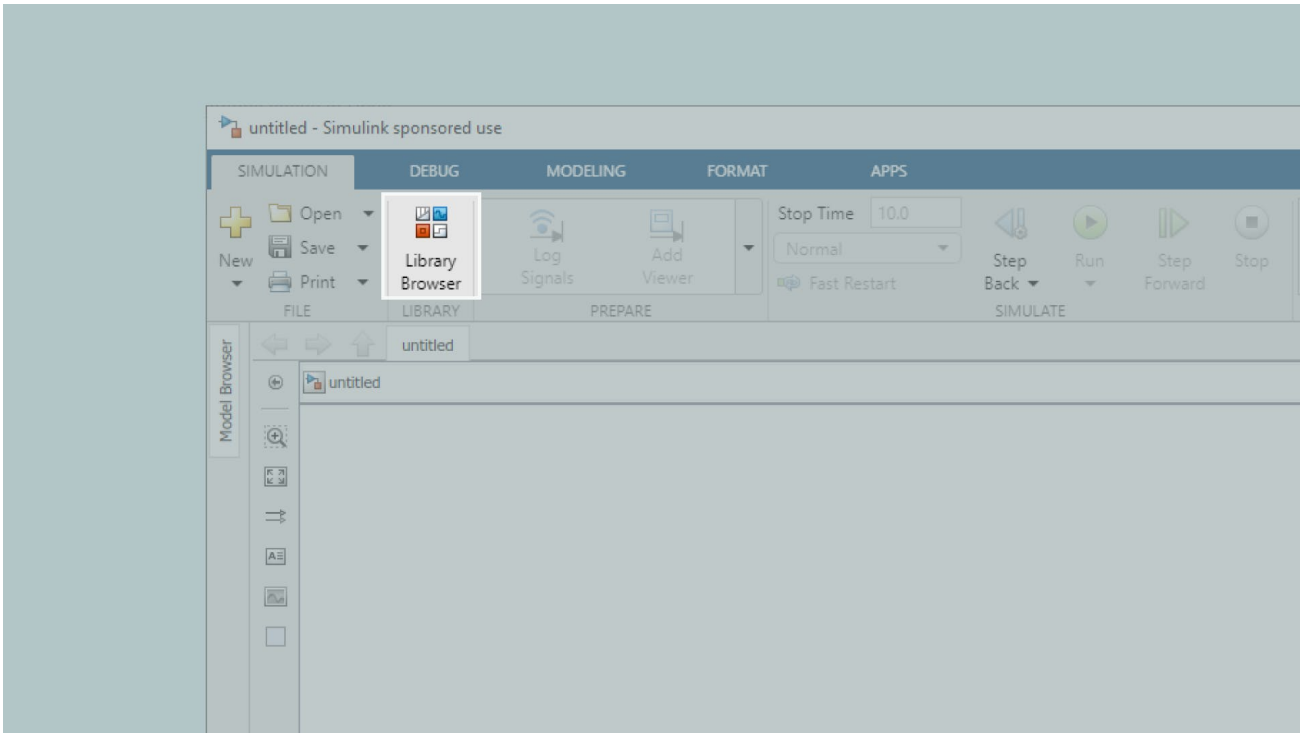
- How to create, simulate, and save a Simulink model.
- How to add and remove blocks from a Simulink model.
- How to connect blocks in a block diagram.
- How to label blocks and signals.
- How to set block parameter values.
- How to visualize simulation data in the Simulink environment.
- How to set the sampling rate of block in a Simulink model.
- How to add block hierarchy to a Simulink model using subsystems.

Blocks and Signals

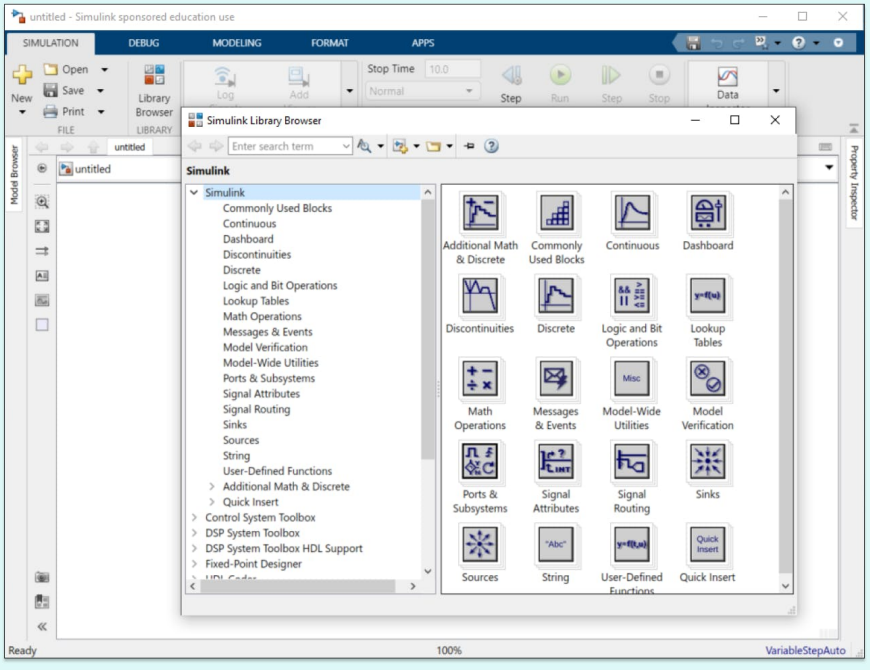
To bring up a blank canvas, in the **Home** ribbon open up the **Simulink Start Page** and then click on **Blank Model**. Here is where you will use use Simulink's VPL to program your Arduino.



There are two major classes of items in Simulink: blocks and signals. Blocks are used to generate, modify, combine, output, and display signals. Lines are used to transfer signals from one block to another. Now let's add some blocks to the model canvas. You can find all the built-in Simulink blocks by using the **Simulink Library Browser**. Open the **Simulink Library Browser** by clicking the **Library Browser button** in the toolstrip, as shown below:

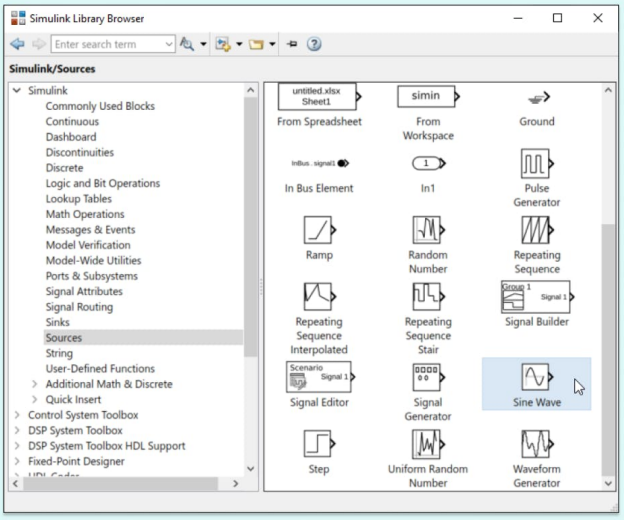


In the left panel, the **Simulink Library Browser** shows a list of the add-on products, and within each a number of **libraries** are made available. Throughout the projects in the Arduino Engineering Kit Rev2, you will use blocks from several libraries, including basic **Simulink blocks** and **Simulink Support Package** for Arduino Hardware.



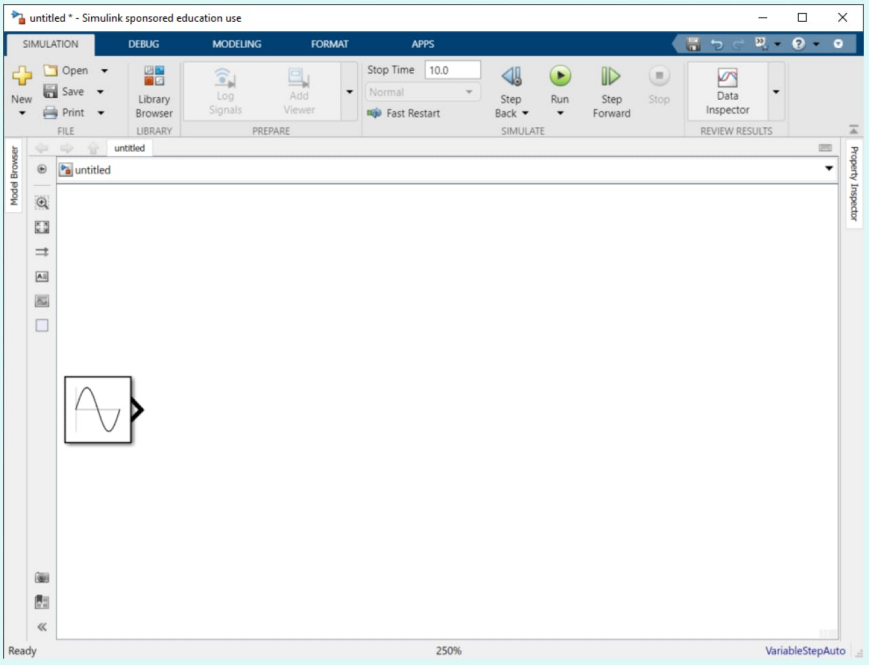
Help

Examine the **Sources** library under **Simulink**. These blocks function as inputs to a Simulink block diagram. Notice that they all have triangular ports on their right sides. These are called **outports** and allow you to connect signals at their origin. Let's select a sine wave block that generates a sine wave.

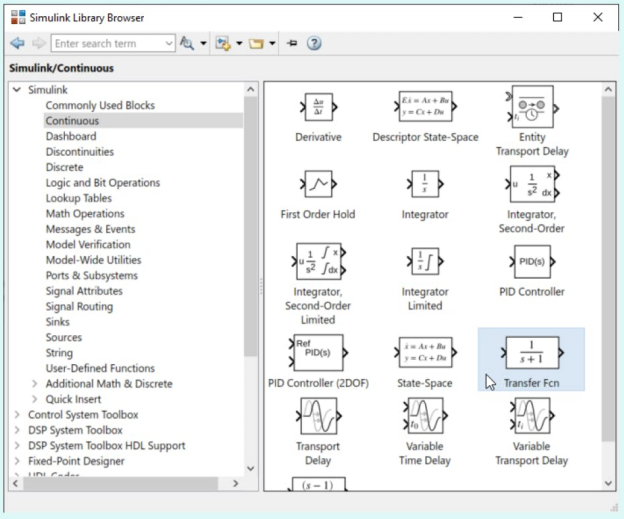


Locate the **Sine Wave** block and drag it from the **Simulink Library Browser** window to the **Simulink Editor** window.

Note: Use the scroll wheel to zoom in and out. Also, you can use move across the canvas by holding the middle mouse button.

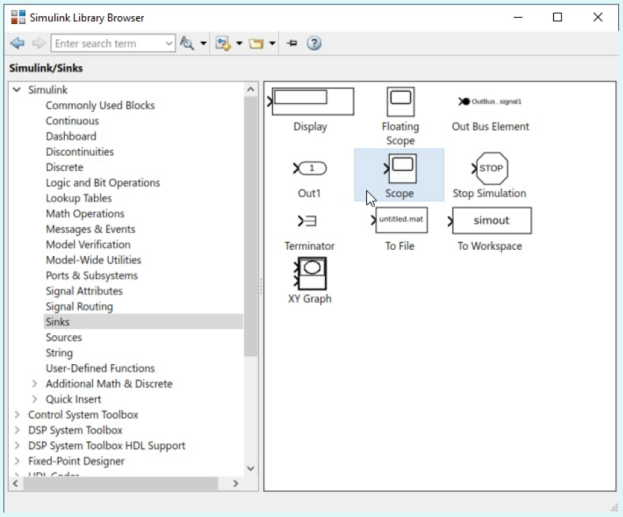


Next we will add a block from the **Continuous** library. These blocks modifiy the input signal and outputs a new signal on a line to the Scope. Navigate to **Continuous** on the Simulink Library Browser and drag and drop the **transfer fcn** block. We will modify this block later.

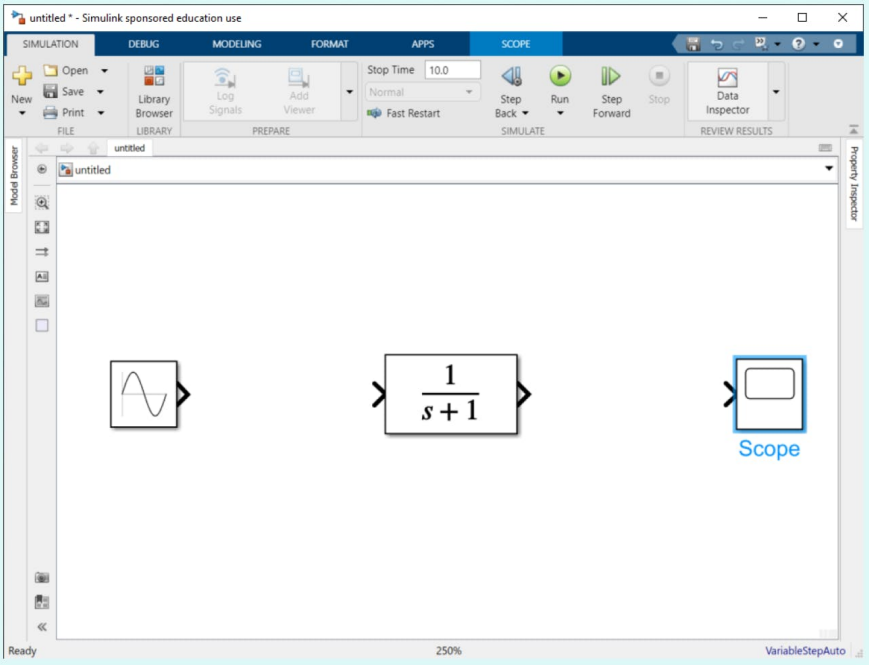


Help

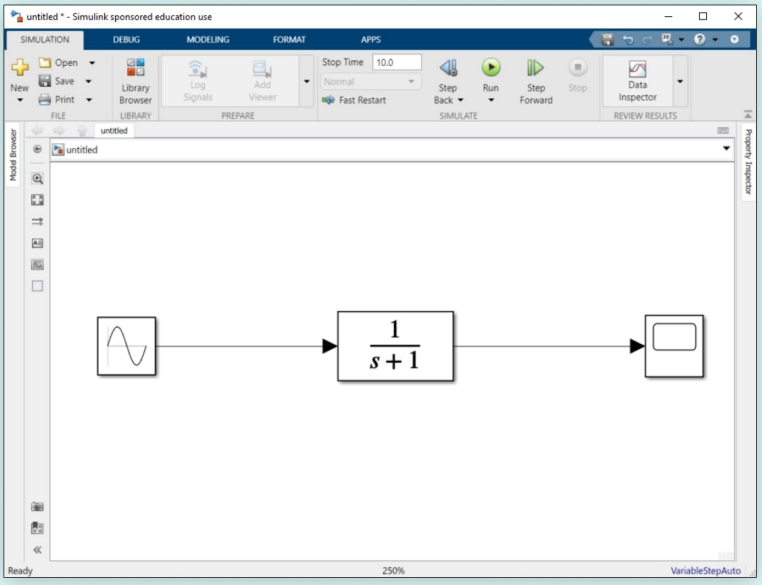
So far our model has a sine wave as a signal **source**, and a block that **continuously** modifies that signal. We can see how the sine wave has changed using a **scope** from the **sink** library. Sinks are blocks that represent outputs in your block diagram. Notice that these blocks generally have an inward triangular port on their left, known as an **import**. Block imports allow you to **Terminate** a signal. Go to the **Simulink > Sinks** library and drag a **Scope** block into your model:



Once you have added all the required blocks, your Simulink editor looks like the following image.

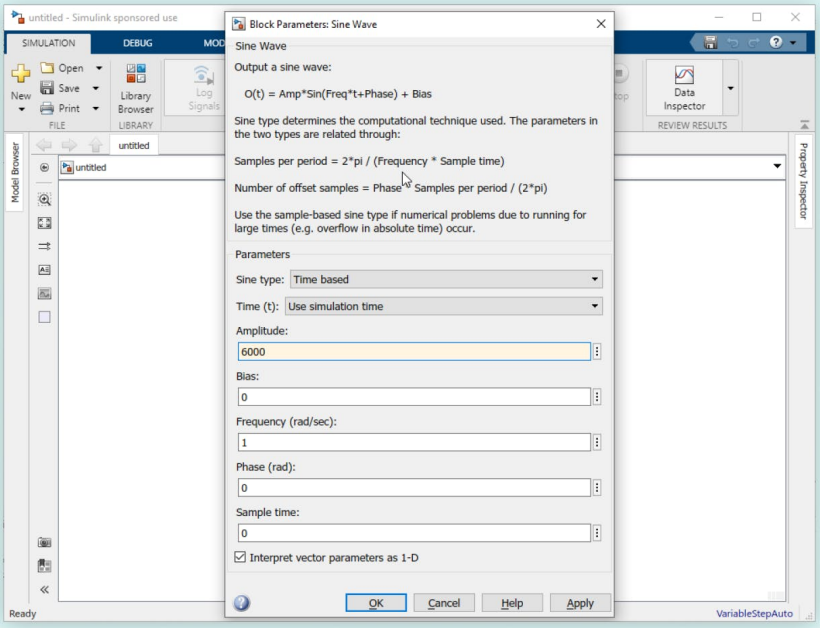


Now you need to connect the blocks using **signals**. Signals flow in the direction indicated by the arrow. They are represented as Lines and must always transmit signals from the output terminal of one block to the input terminal of another block. Left click on the outport and drag it to the import and when the **signal** line turns solid, you have connected the signal and you can stop the click and drag. Connect all the three blocks together as shown below.



Modifying Blocks

A blocks can be modified by double clicking it. Double-click the **Sine Wave** block to open its **block parameter dialog**. Configure the sine wave to have an amplitude of 6000, to cover the range of measured speeds. Then click **OK**.



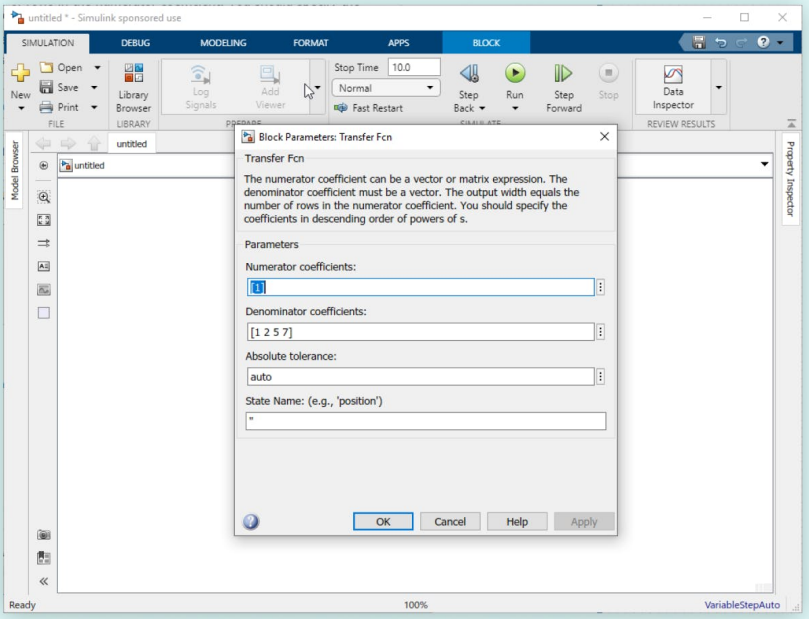
Help

Double click the **transfer fcn** block to change the transfer function. By entering a vector containing the coefficients of the desired numerator or denominator polynomial, the desired transfer function can be entered. For example, to change the denominator to

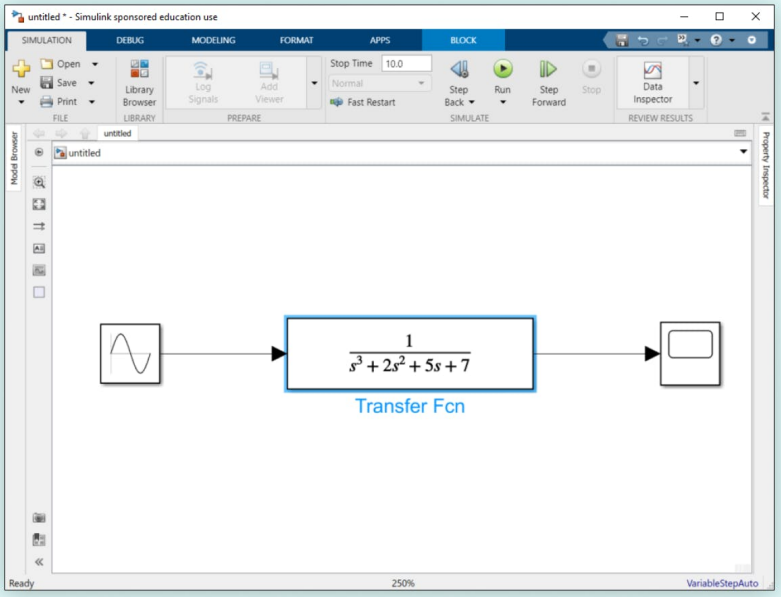
$$S^3 + 2S^2 + 5S + 7$$

Enter the following into the denominator field.

[1 2 5 7]

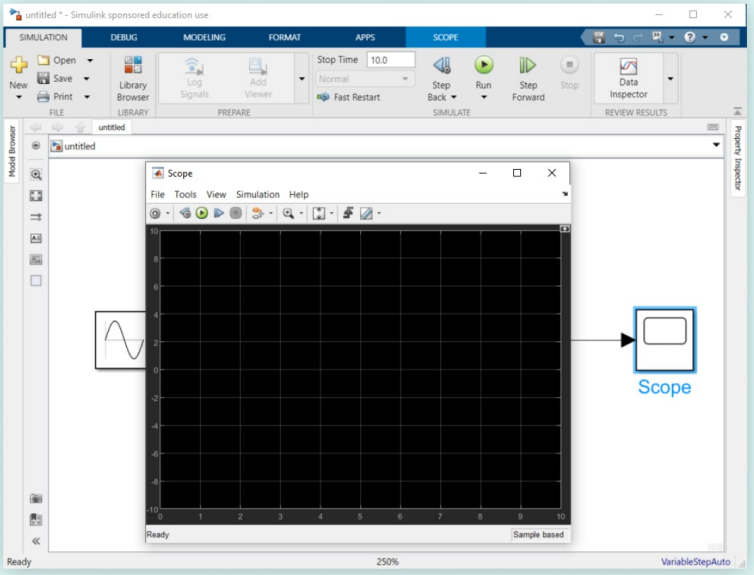


After clicking **OK**, you will see that the transfer function block changes to reflect the new denominator coefficients you entered.



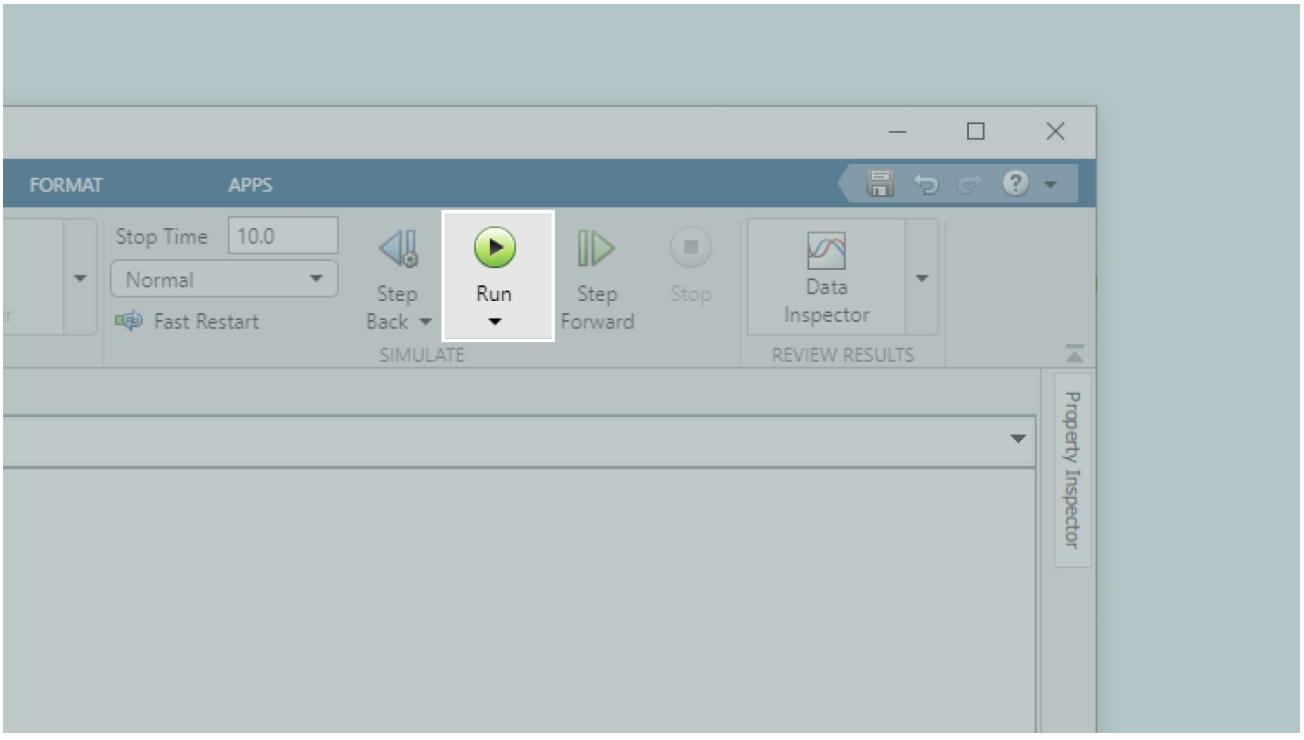
Note: The values may not be reflected in the transfer block if the size of the block is small. Expand the size of the block to see the change by draging one of the corners.

Double clicking the scope block brings up a blank oscilloscope screen. When a simulation is performed, the signal which feeds into the scope will be displayed in this window. We won't be making changes.

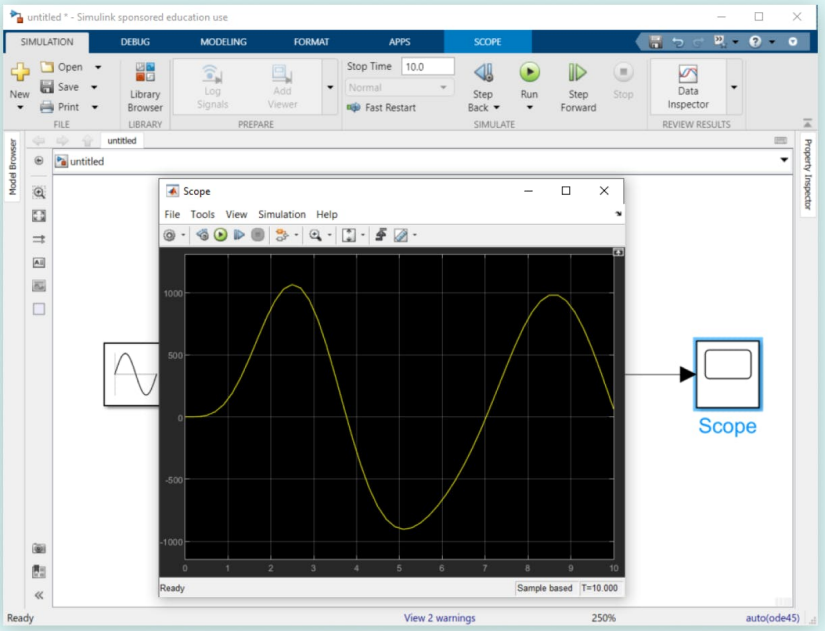


Simulating Models

Let's simulate the algorithm you just created. Click the **Run** button:



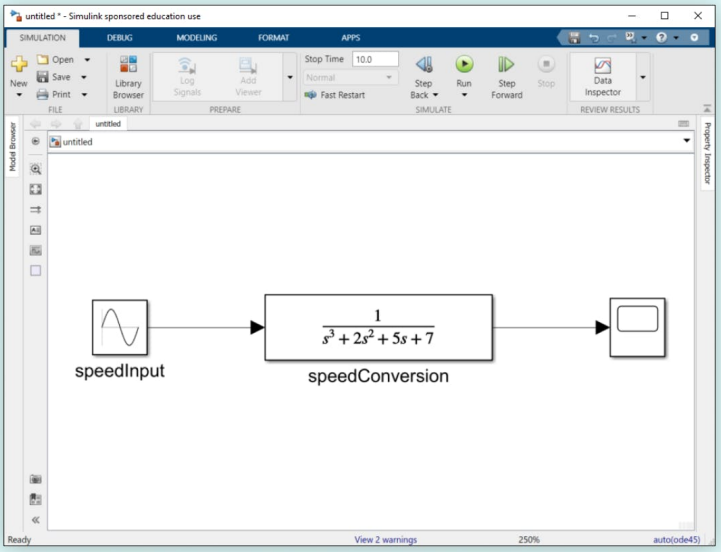
Once the simulation completes, double-click the **Scope** block to view the scaled Sin values: you will find that the sin wave has been slightly distorted in the oscillometer.



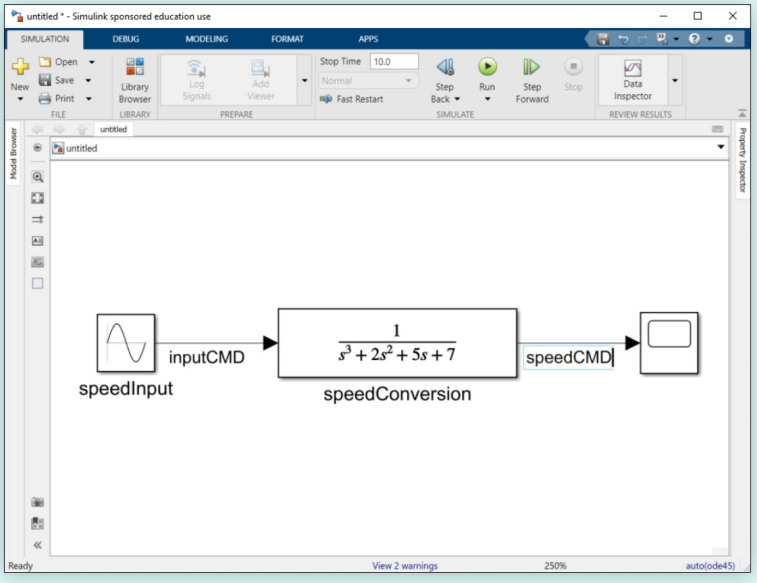
Annotating the blocks

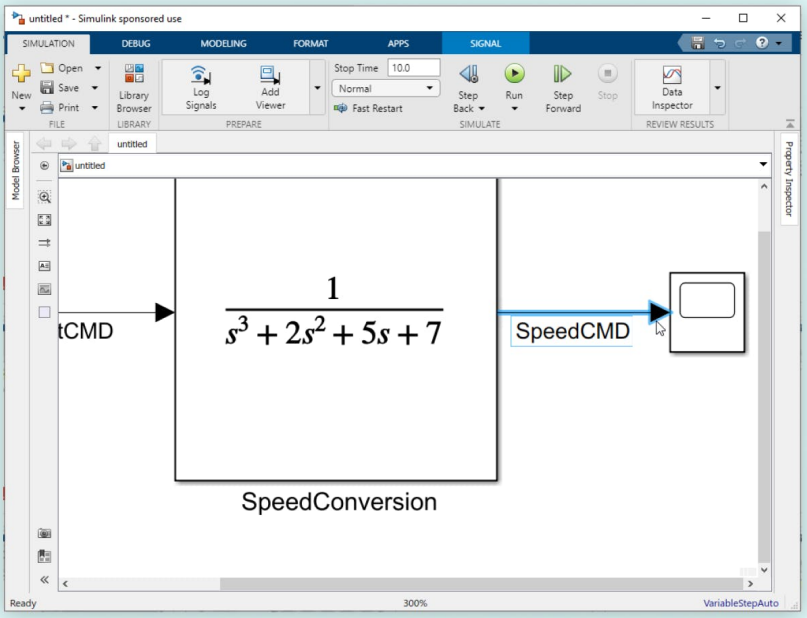
Just like you annotated your live script to communicate how your program works, you can similarly annotate a Simulink model to help explain it to others. First, let's label the blocks. By default, blocks do not display a block name. Click the **Sine Wave** block and change its label to `speedInput`.

Similarly, label the continuous block to `speedConversion` as shown in the following image:



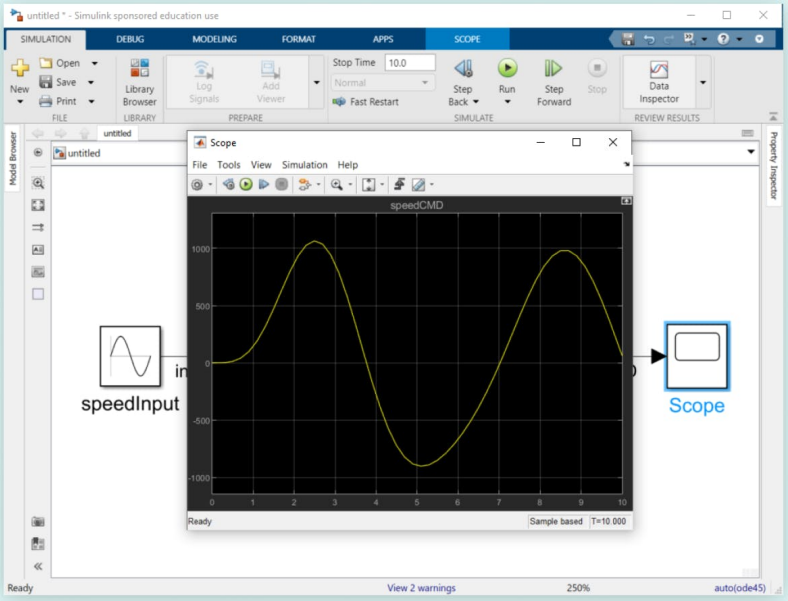
You can also label signals to indicate what quantity they represent. Label the `speedInput` to `speedConversion` signal as `inputCMD` and the `speedConversion` to `scope` signal as `speedCMD`.





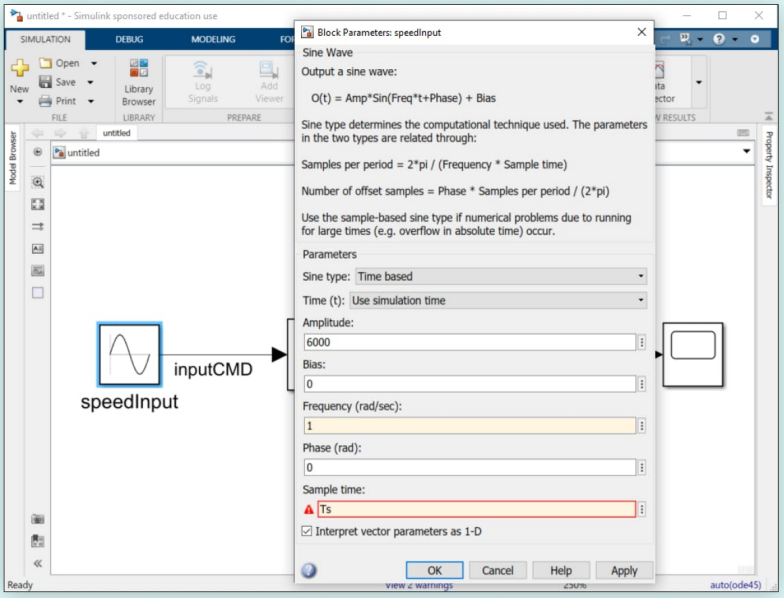
Note: You can click and drag signal labels to different locations along the signal line.

Run the simulation again. Notice that your signal labels now show as titles in the **Scope** window:



Right now, the simulation is running in continuous time. That means that all signals are treated as continuously varying values, and visualizations such as the **Scope** block show signals as smooth curves (although they are actually a series of discrete points connected by straight lines).

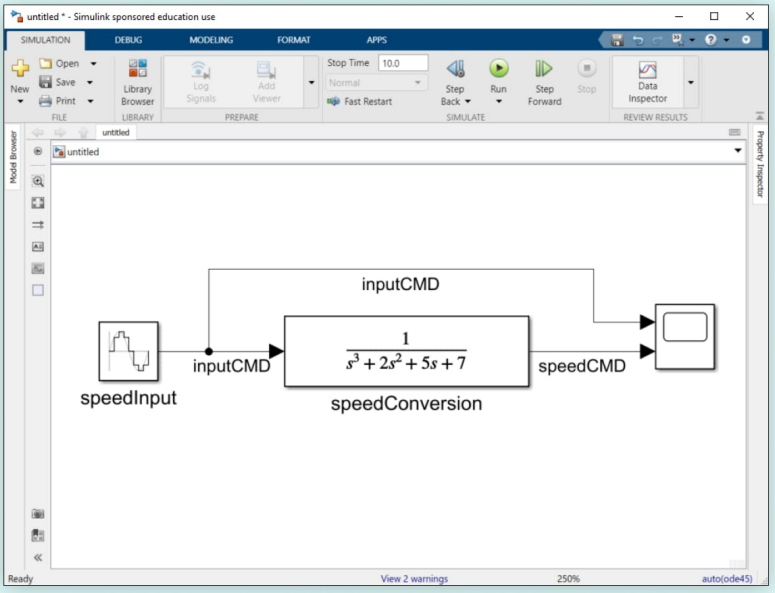
Soon you will upload and run this algorithm on the Arduino board, which executes in **discrete** time. That means signal values are calculated and updated at periodic intervals. Let's adjust the model to simulate in discrete time so that we understand how it will work on the Arduino Nano 33 IoT board later. In the **Sine Wave** block parameter dialog, set **Sample time** to Ts and click **OK**:

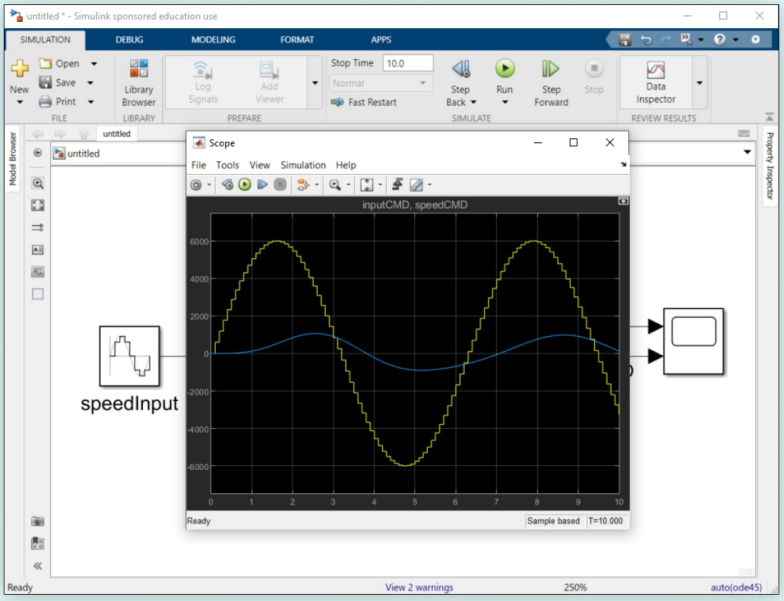


Now define Ts in the **Command Window**:

```
>> Ts = 0.1;
```

Add another signal from the source to the sink by right clicking the `inputCMD` line and dragging to the **Scope**. Double-click this new line, and it will automatically be named as `inputCMD` . **Run** the simulation, and examine the scope:





Running the Models

Run the model and view the scope to confirm it still behaves the same way as before. Now save your model as **simpleSimulinkModel.slx**.

While you can write C code to upload it to Arduino boards, Simulink offers a great interface to program boards directly from its UI using blocks. The Simulink blocks can translate into code that can be compiled and uploaded to the board. This feature will be introduced in chapter 3 and will be used later in the projects.