DRAWING ROBOT

Build a robot that extracts line traces from an image and program it to reproduce and duplicate the images as a drawing on a whiteboard. You will learn about physics, programming, and robotics.

YOU WILL LEARN ABOUT:

IMAGE PROCESSING, MATH COMPUTATIONS, COORDINATE GEOMETRY, ROBOT MOVEMENT, TRIGONOMETRY

4.0 Project Overview

In this project, you'll build a robot that can draw on a whiteboard. When you are done, your robot will be able to acquire an image (or load an existing image), extract the line traces from that image, scale the drawing to the drawable area of your whiteboard, and then reproduce the drawing by moving a marker along the same set of paths. Along the way, you'll learn important concepts in engineering, physics, and programming that are useful for a wide variety of applications.

In this project, you will learn to:

- Connect to an Arduino-based robot from MATLAB.
- Write MATLAB apps, functions, and scripts to control your robot.
- Learn and apply concepts for coordinating geometry and physics.
- Learn about symbolic math computations and image processing.
- Automate a complete application workflow from start to finish.

Project Assembly

Before you begin modeling and work on the project, you will need to assemble the drawing robot by following the instructions in the following video. The assembly of

this project will take about **45 mins** approximately.

Note: In order to command both the motors to move the drawing robot in forward or reverse directions simultaneously, make sure that you wind the strings on the two pulleys in the same direction (clockwise or anticlockwise). This way, you will ensure that the drawing robot moves in a particular direction (upwards or downwards) on the whiteboard with the help of both the motors using a positive or negative command signal.

4.1 Introduction to the Drawing Robot

Once you have assembled the robot, you'll want to connect to your robot and make sure you can control all its parts. This exercise introduces all the commands needed to do so.

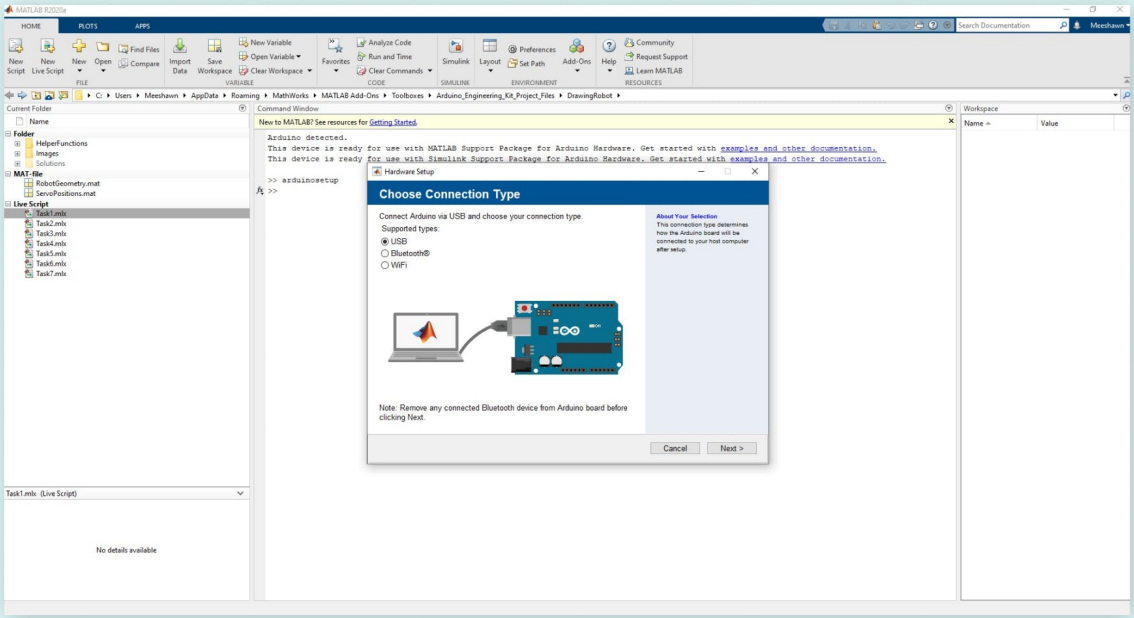
In this exercise, you will learn to:

- Connect to the assembled Arduino-based robot.
- Communicate and control the robot's various components.

Connect to the Robot

Let's get connected to the robot. You'll connect to your robot the same way you did in the Getting Started section. Connect the Arduino Nano 33 IoT board to your computer and execute the following command at the MATLAB command prompt to launch the Arduino hardware setup interface:

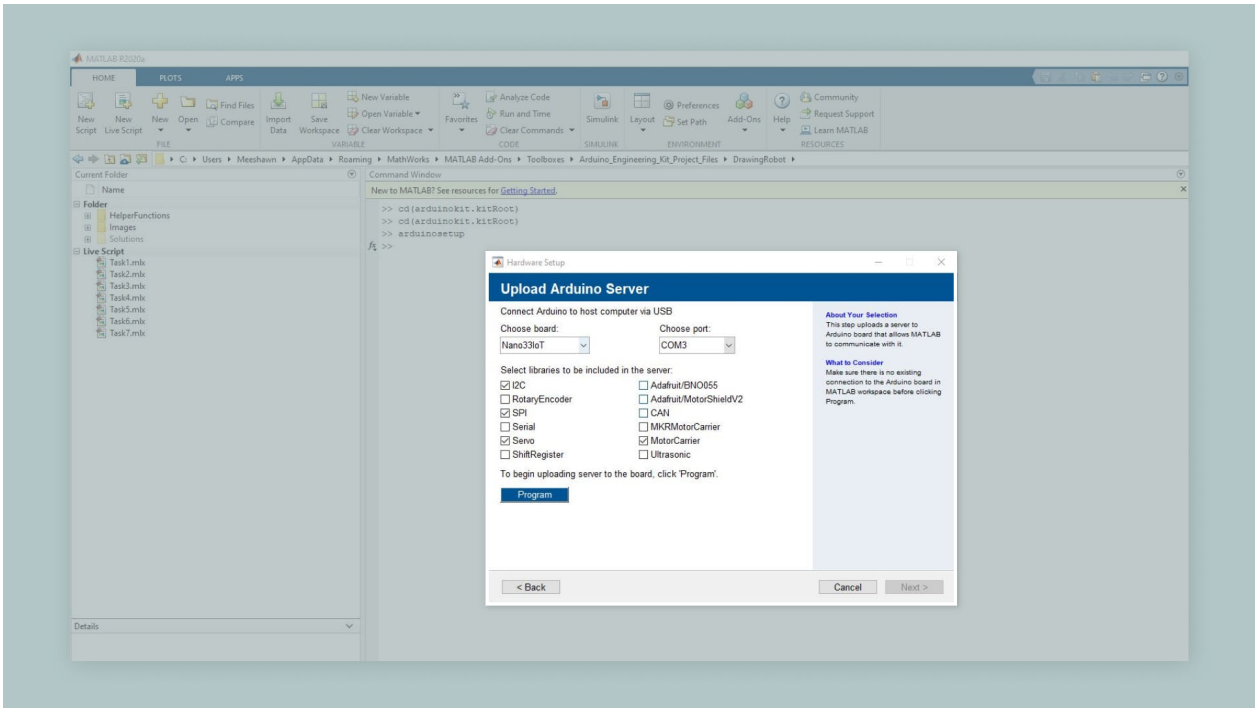
```
>> arduinosetup
```



Select **USB**. Click **Next**. On the following screen, include the **MotorCarrier** library as well as I2C, SPI, and Servo. From the **Port** dropdown list, choose the port that the Arduino is connected to

- **COM** for Windows
- **/dev/ttyACM** for Linux
- **/dev/tty.usbmodem** for OSX

Click **Program** to upload the program to the board.



Once the upload is complete, test the connection on the next screen using the **Test Connection** button. Close the setup interface.

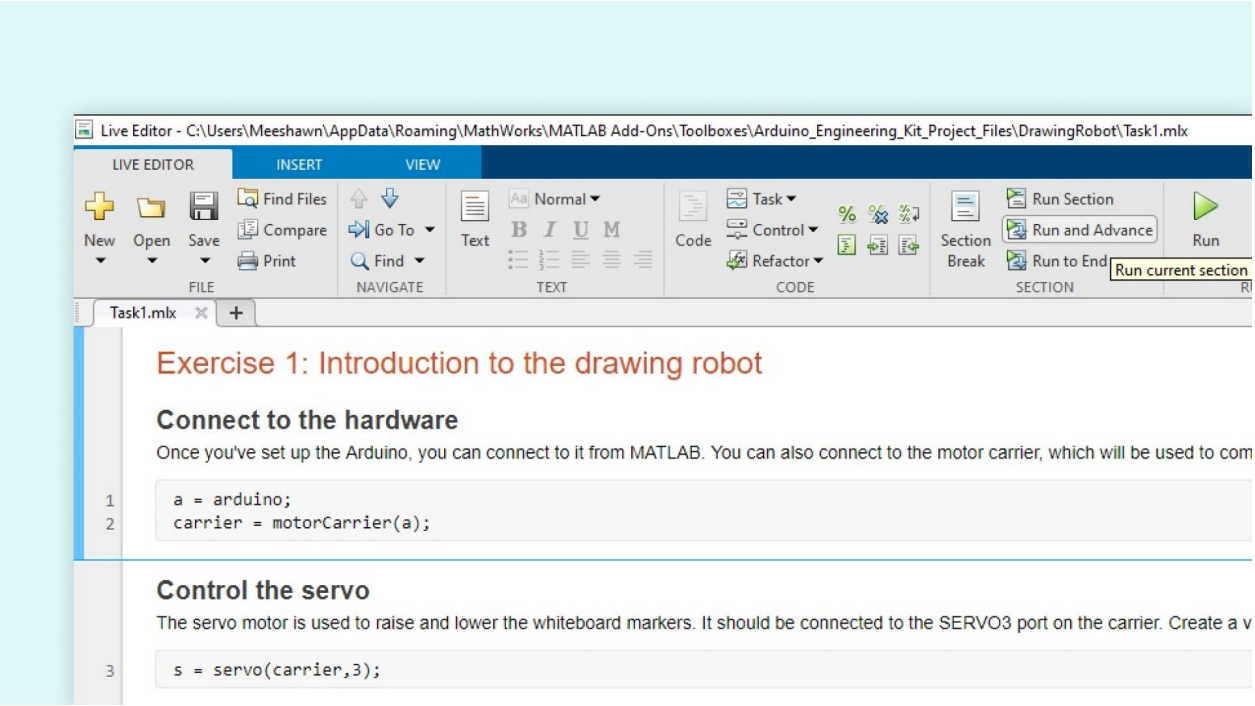
Connect to the Hardware

First, you will connect to the Arduino Nano 33 IoT board from MATLAB. Open the live script, **Task1.mlx** by typing:

```
>> edit Task1
```

> **Note:** Starting in MATLAB R2021a, the project files have been integrated within the MATLAB & Simulink Support Package for Arduino. If you have MATLAB R2021a or later, please make sure that you follow the instructions mentioned on **this** page to add the project files to the MATLAB path.

Then, execute the code in the **Connect to the hardware** section.

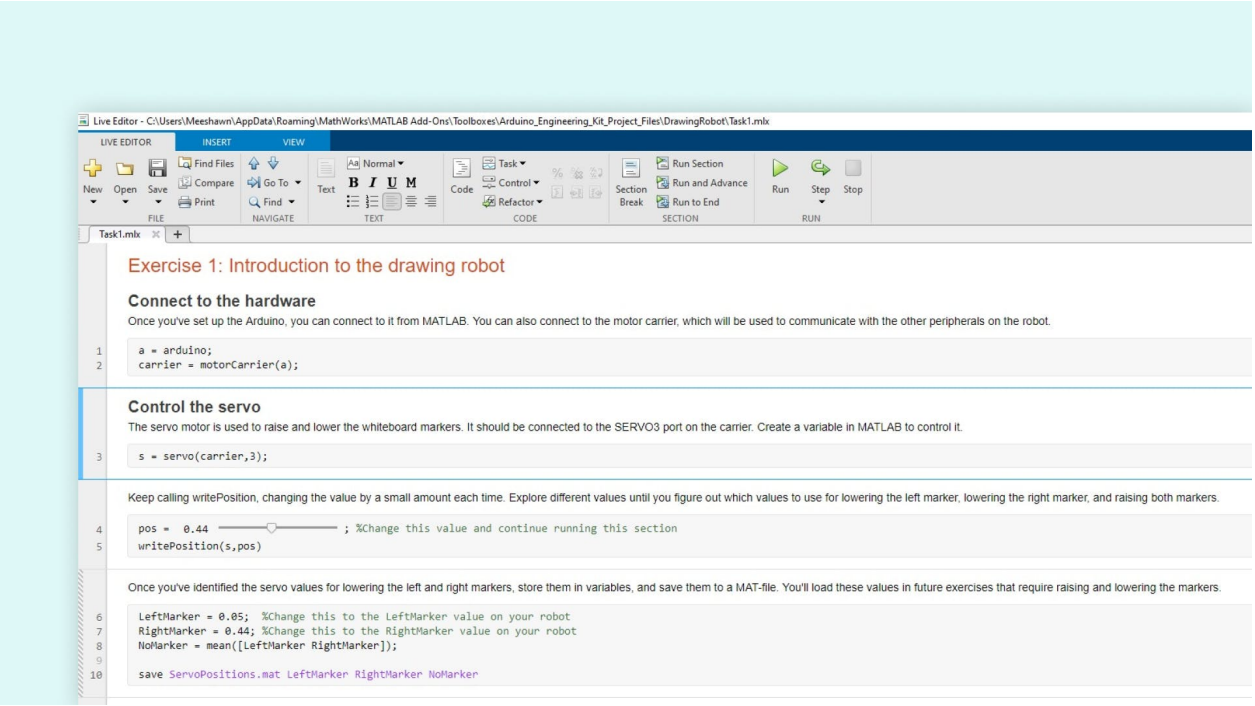


Control the Servo

The drawing robot has two different colored markers that can be raised and lowered by means of a servo motor. In this section, you will learn how to estimate the software limitations for driving the motor, given the physical limitations of the robot.

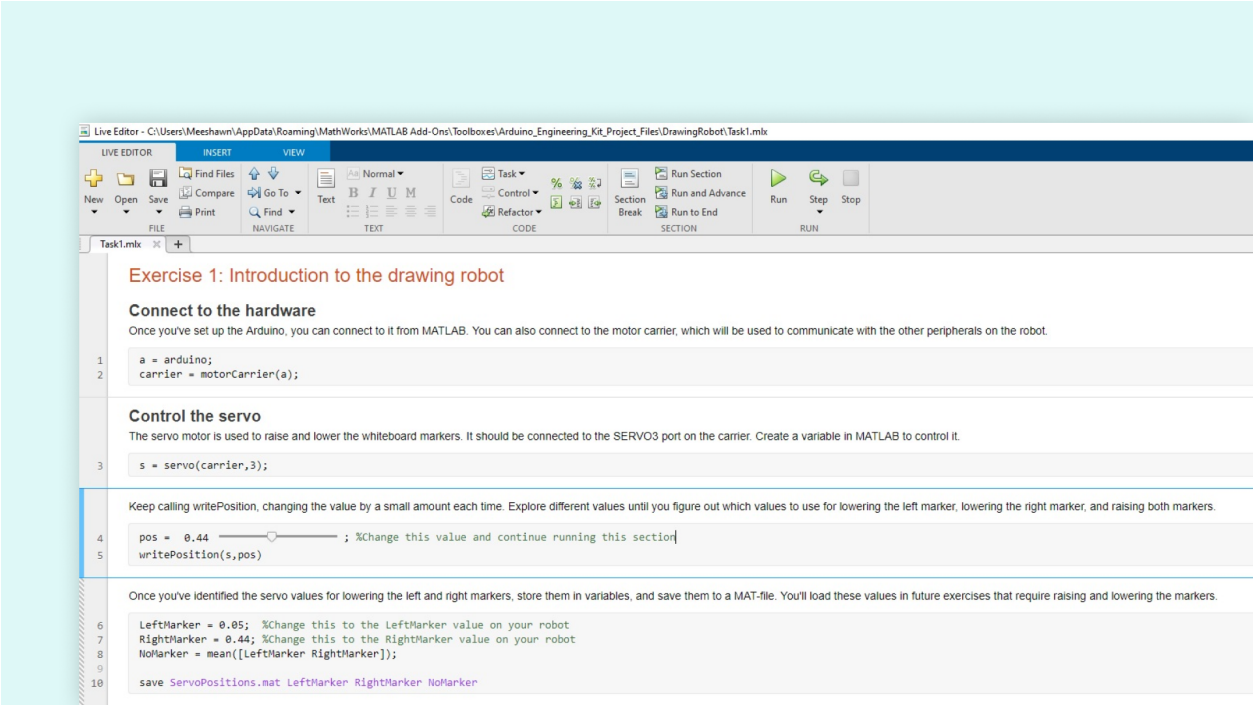
You need to be able to communicate with the servo motor and command it to raise and lower the two markers. This is a trial and error process where you'll need to experiment in order to determine the servo positions that lower the markers without trying to drive the motor beyond the limits of the marker channels.

The **Task1.mlx** live script contains code for you to connect to the servo motor and command it to move. First, connect to the motor by executing the sections under the heading **Control the servo** in the live script **Task1.mlx.**

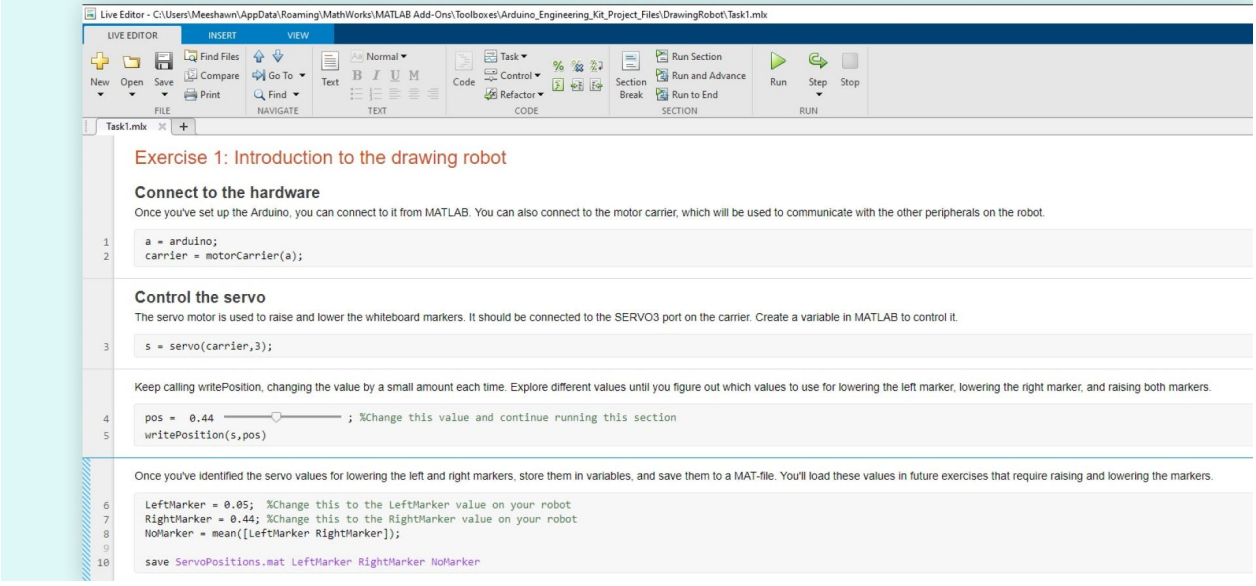


The following code section contains a numeric slider control that defines the `pos` variable. Whenever you change the value of the slider, this section of code will execute, commanding the servo to move to a new position. In order to figure out the limitations for the position, you will have to change the value of `pos` in small increments until you find the servo position corresponding to one of the markers being fully lowered. Record that value and then repeat the process for the other marker. Since the robot only uses one servo motor to control both the markers, you simply need to change the value of `pos` incrementally in the other direction until you find the servo position for lowering that marker.

Based on our observations, we found the servo position for lowering the **left** and **right** markers at 0.05 and 0.44 respectively. You may find a different set of servo positions suitable for your drawing robot around these values. After recording these values, you should also verify that a value of `pos` midway between these two extremes will have both markers up. In this way, you will have a single physical driver, the servo motor, to control three different states of the drawing mechanism: no markers drawing, marker 1 drawing, or marker 2 drawing.



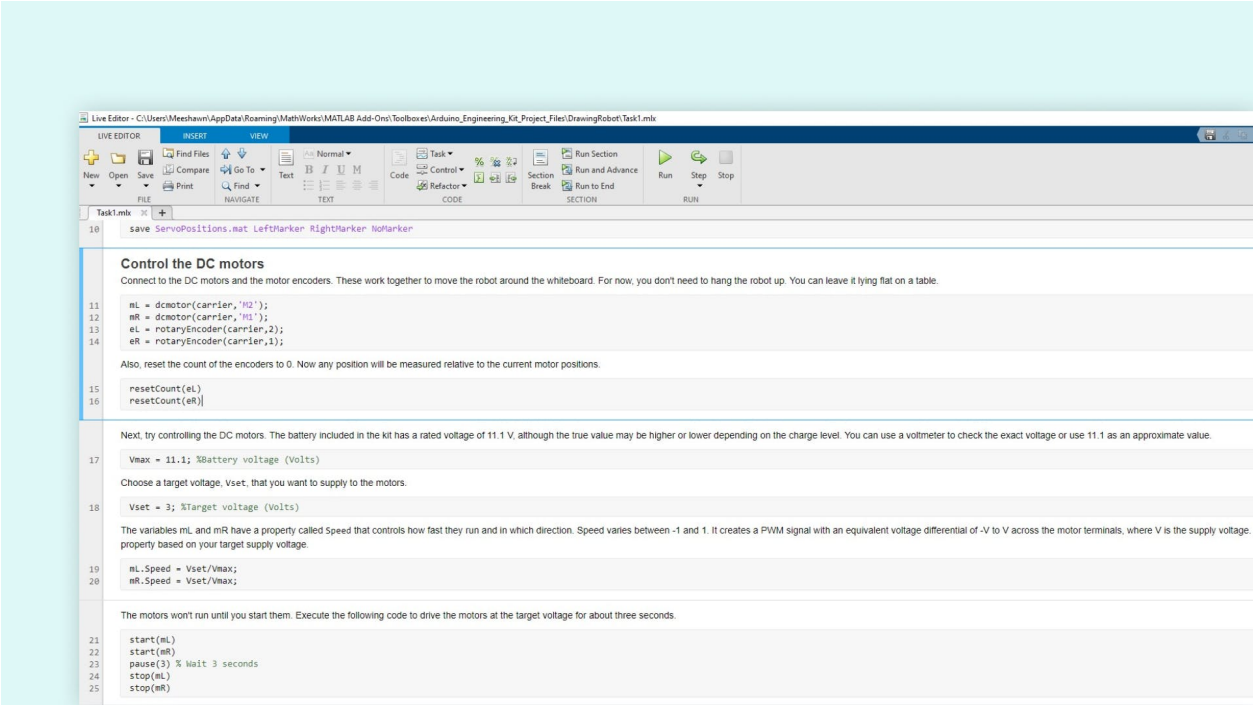
In the final code section under the heading **Control the servo**, enter your experimentally determined values for the servo position while lowering the left marker and the right marker. See the following screenshot for which values to update. Execute this section of code to save these values to a MAT-file so you can load them for use in future exercises.



Control the DC Motors

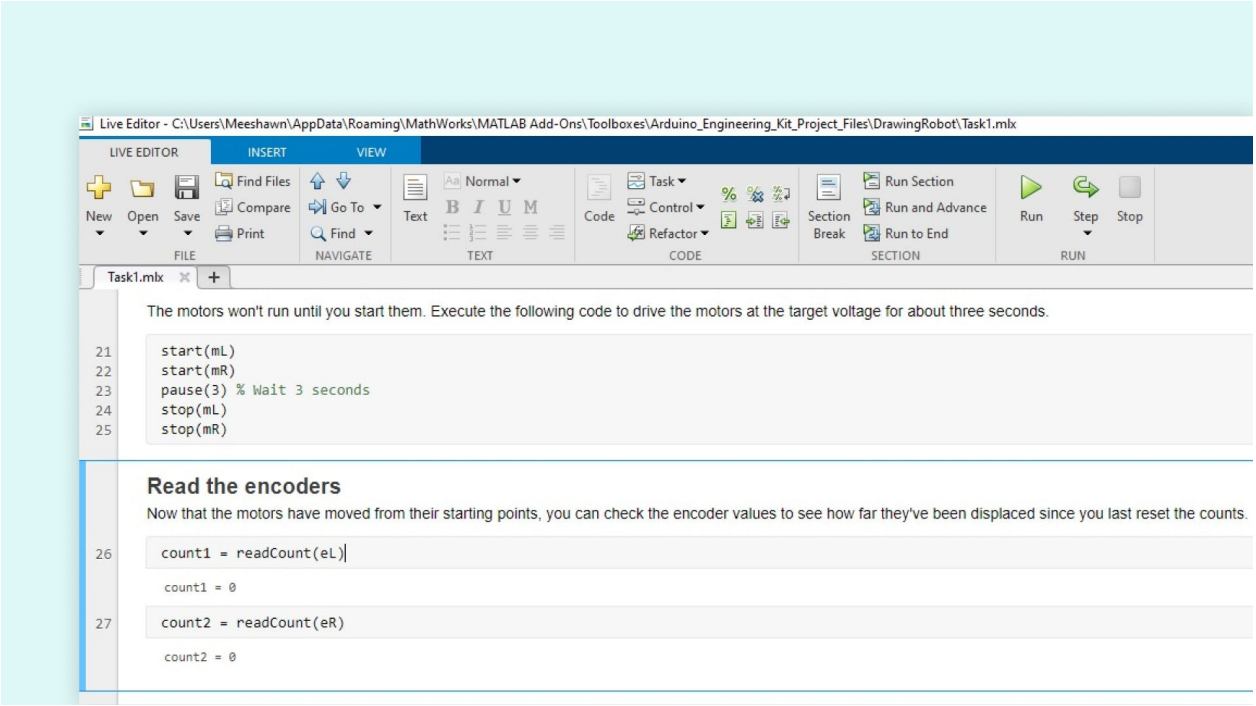
Once you have controlled the markers, the next step is to figure out how to control the two DC motors that make the drawing robot move over the whiteboard.

In this test, you will connect MATLAB to the motors and their encoders, specify a voltage to supply to the motors, and then run the motors for about three seconds. First, identify and read the sections of **Task1.mlx** under the heading **Control the DC motors**. In this code section, `Vset` is the target voltage that you want to supply to the motor (identified in the datasheet), and `Vmax` is the rated battery voltage. These parameters are used to compute the speed of the motors between 0 and 1, where 1 implies maximum speed and 0 supplies no voltage to the motor. Execute each of the sections of the live script under the heading **Control the DC motors**.



Read the Encoders

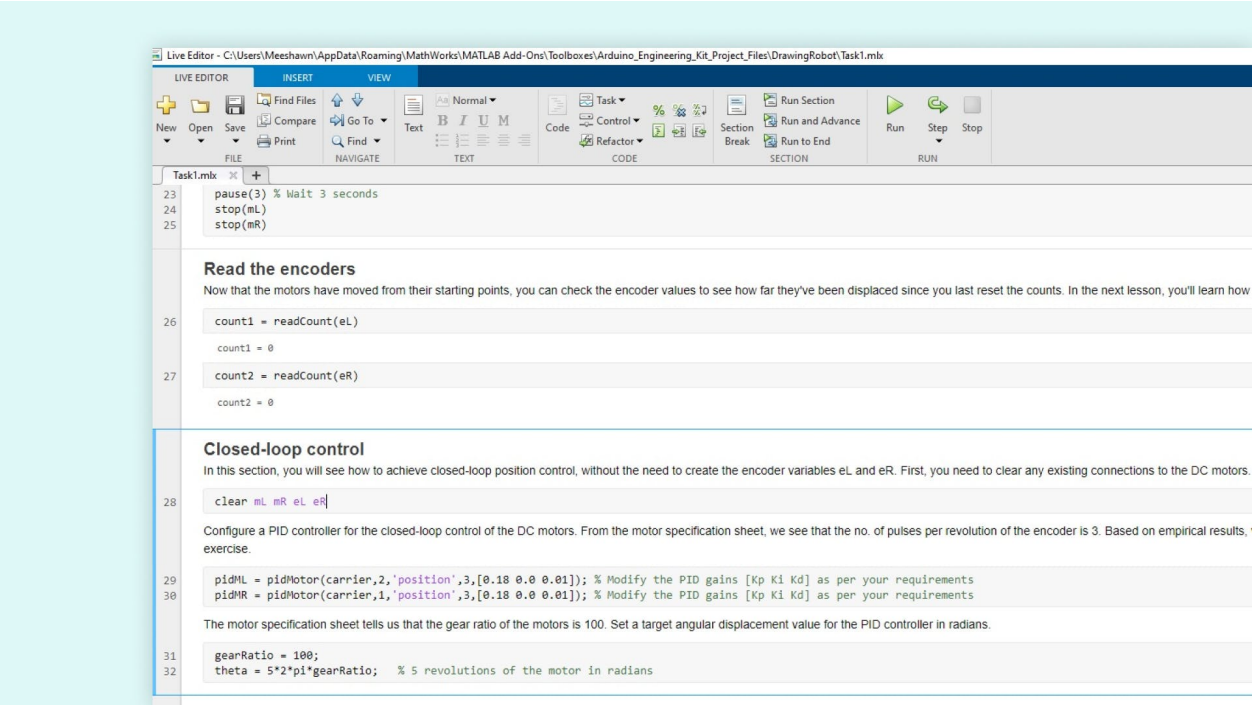
As you know by now, DC motors require external control mechanisms to ensure that they are behaving as expected. In this case, we will use rotary encoders to measure the angle and speed of the motors while operating. After displacing the motors from their initial positions, you can read the count value from the encoders to determine how far each motor has turned in units of encoder counts. The conversion from counts to physical distances and angular rotations will be calculated in a later exercise. Execute the code under the **Read the encoders** section of the live script **Task1.mlx**.



Closed-Loop Control

In the previous two sections, we looked at how to interactively control the DC motors and read back the encoder counts corresponding to the angular displacements. In this section, you will see how to achieve closed-loop position control of the motors, without the need to explicitly connect to the encoders.

Read through the code sections under the heading **Closed-loop control**. First, you will need to clear any existing connections to the DC motors in MATLAB. Next, you will create a new `pidMotor()` connection object which lets you configure a PID controller for closed loop control of the motors.



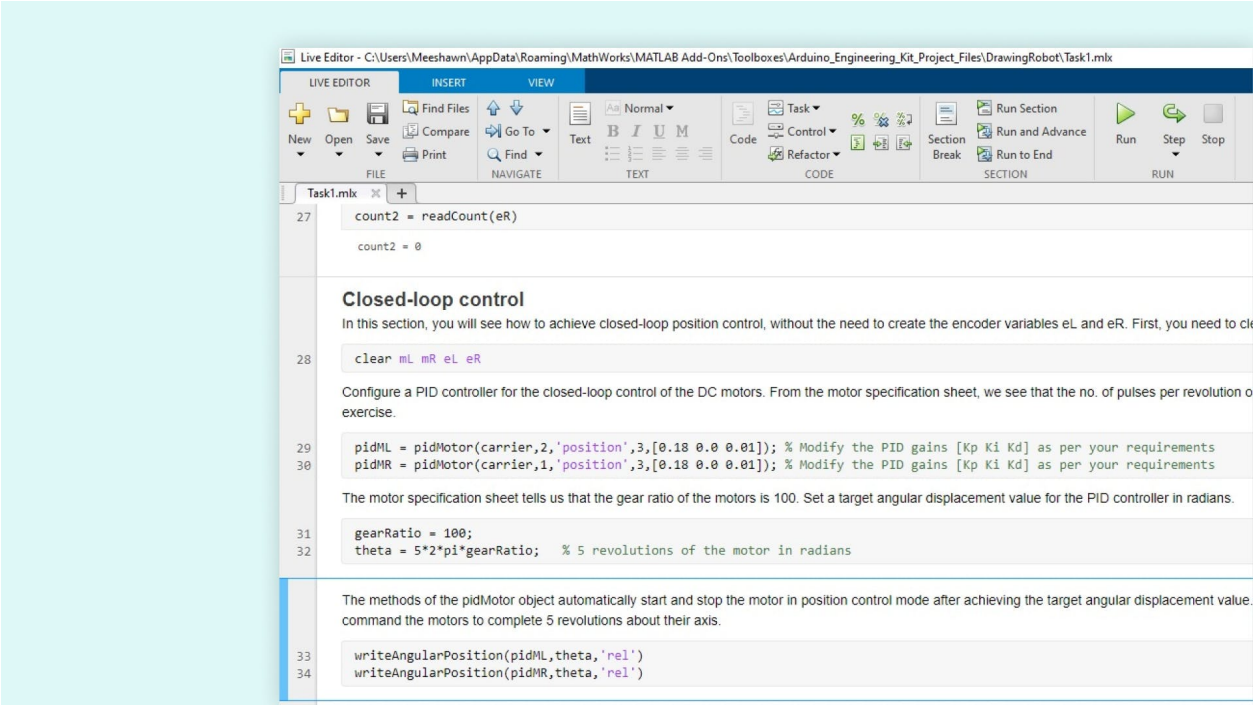
The standard form of a PID Controller is:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(t')dt' + K_d \cdot \frac{de(t)}{dt}$$

where `u(t)` is the control variable, `e(t)` is the error value, K_p, K_i and K_d are proportional, integral and derivative gain values respectively.

You can select either **position** or **speed** control mode and then specify the gains for the PID controller within the same `pidMotor()` connection object. Based on our tests, we found these PID gain values suitable: K_p = 0.18, K_i = 0.0, K_d = 0.01 for position control. You can come up with your own PID gain values based on your requirements and empirical results. You will need to set the number of pulses per revolution of the motor encoder as per the motor specification sheet.

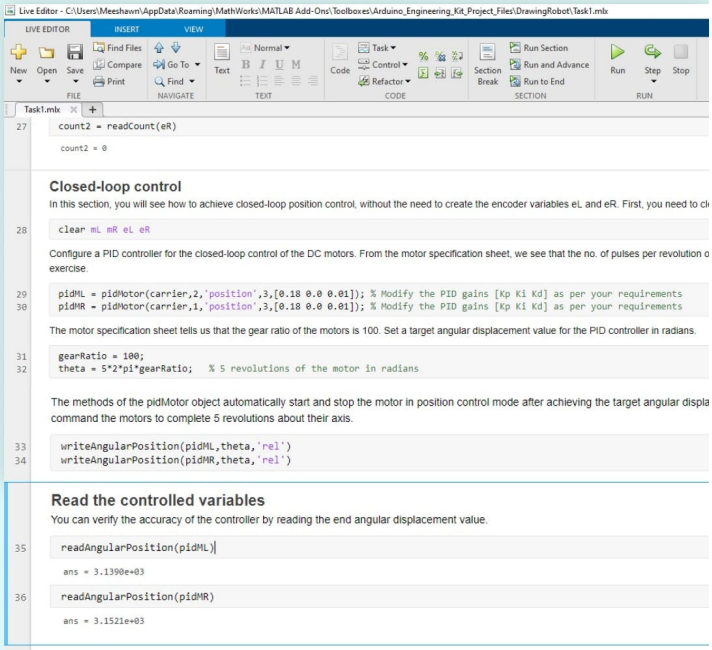
After configuring the `pidMotor()` object, you can command the motors to rotate to a specific angular position or maintain a constant speed using the functions `writeAngularPosition()` and `writeSpeed()` respectively. In the position control mode, you can set the mode of calculation of the angle of rotation to be either **absolute** or **relative**. In the absolute mode, the motor rotates by a difference in the angle between the current and the desired position. In the relative mode, the motor rotates by the input angular displacement value.



The `writeAngularPosition()` function realizes a closed-loop control action. Hence it's not possible to reset the encoder value since the PID controller continuously tries to achieve the set-point/target displacement value. You will need to clear the `pidMotor()` object from the MATLAB workspace in order to reset the encoder. However, successive commands to rotate the motor can still be given using the `writeAngularPosition()` function without the need to clear the `pidMotor()` object. Execute the code under the heading **Closed-loop control** of the live script **Task1.mlx**.

Read the Controlled Variables

You can verify the accuracy of the PID controller implemented in the previous section with the help of the functions `readAngularPosition()` and `readSpeed()` for **position** and **speed** control modes respectively. You might observe a discrepancy between the values obtained from these functions and the target values. You will need to redesign the PID controller by modifying the PID gains if the obtained accuracy is not within the acceptable performance level. Execute the code under the heading **Read the controlled variables** of the live script **Task1.mlx**.



Note: Unlike the `dcmotor()` object described in the **Control the DC motors** section of the live script **Task1.mlx**, the methods of the `pidMotor()` object automatically starts and stops the motor. Hence there are no `start()` and `stop()` methods for the `pidMotor()` function. Additionally, the `dcmotor()` and `pidmotor()` objects cannot coexist in the same MATLAB workspace.

Files

- Task1.mlx

Learn by doing

Create a new MATLAB script. Write a series of commands that will run the left motor for three seconds, then stop and run the right motor for three seconds, then run the left motor in reverse for three seconds, then run the right motor in reverse for three seconds, and finally stop the motors. Run your script and check that it performs as expected. Read the counts from each of the encoders. How close are the motors to their starting positions? What might account for the difference? Reset the encoder counts and run the script again to see how consistent this result is.

Hang the robot up on the whiteboard. Run the left motor forward with a voltage of 4 Volts for 2 seconds. Run it in reverse with a voltage of -4 Volts for 2 seconds. How far did it move each time? In which direction did it move faster? Why? How is the behavior different from when you have the robot flat on the table?

Repeat the same experiment in closed-loop control mode, where you'll input radians instead of voltage. Set an angular displacement target of 5 revolutions, i.e., θ = 5 * 2 * π * **gearRatio** to move the motors in forward direction. Measure the angular displacement. Now set θ = -5 * 2 * π * **gearRatio** to move the motors in reverse direction. Read the angular positions. How close are they to their starting positions? Experiment with different PID gain values to improve the closed-loop control accuracy.

4.2 Whiteboard Coordinate System

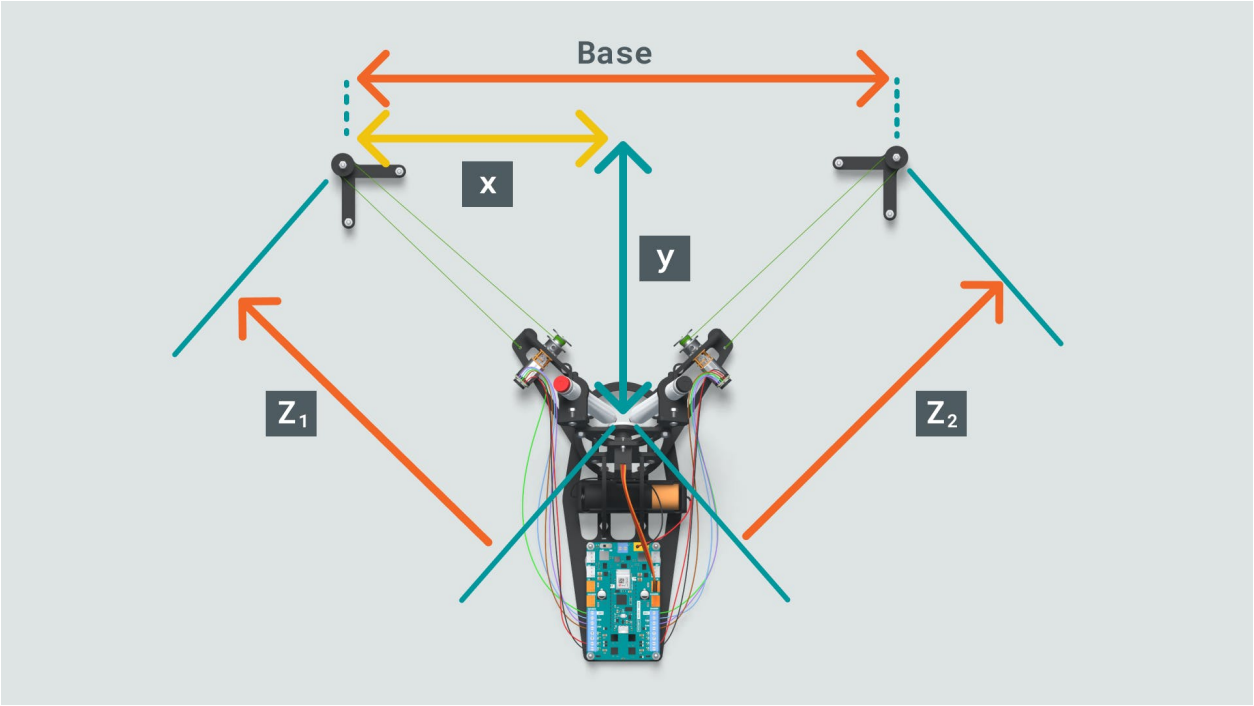
In this exercise, you will define distances that describe the position of the robot on the whiteboard. You will hang the robot up on the whiteboard and move it around using an app. Then you will compute the new position of the robot in x-y coordinates based on encoder measurements.

In this exercise, you will learn to:

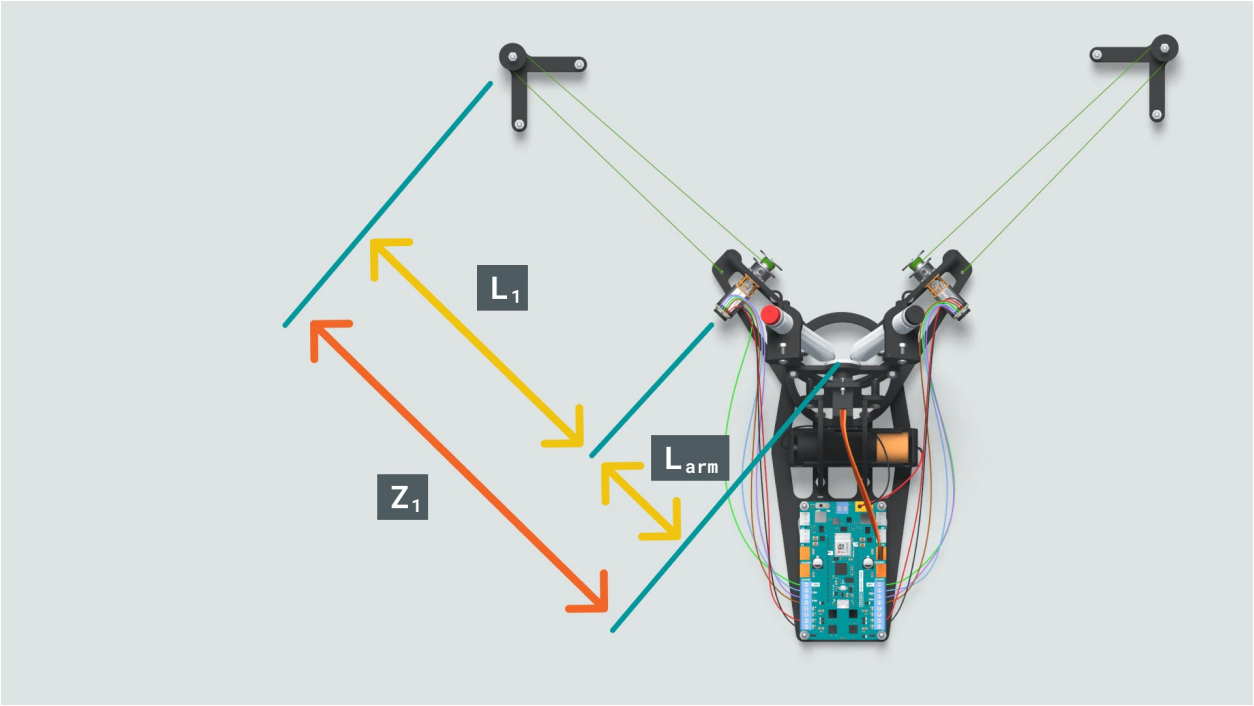
- Use built-in UI functions to get user input,
- Apply concepts from coordinate geometry to code,
- Use MATLAB apps,
- Write MATLAB functions.

Define Distances on the Whiteboard

Imagine hanging the robot up on the whiteboard. Define the values `x`, `y`, $Base$, Z_1, and Z_2 as shown in the following diagram. The black circles at both sides of the diagram represent the pulleys from which you hang the robot, while the robot is in the center of the image, hanging from two strings connected to the robot arms, where the motors are located. Note how the strings travel from the robot arm to the pulley and come back to the motor. Also observe that `x` and `y` positions on the whiteboard are measured with respect to the left-most pulley, with `x` being the horizontal distance to the right of the pulley and `y` being the vertical distance down from the pulley.



We can divide Z_1 into the motor arm length L_{arm} and the string length from the motor to the pulley L_1 We'll do the same for Z_2 using L_{arm} and L_2.

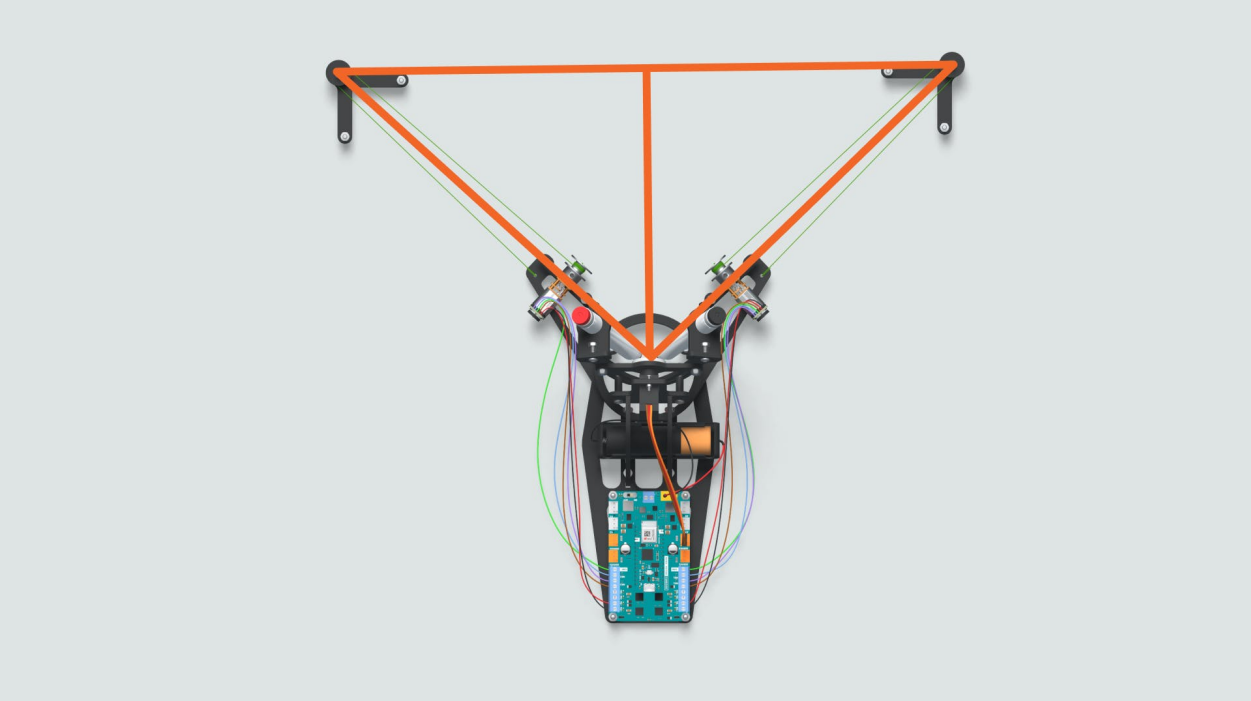


Note: In order to command both the motors to move the drawing robot in forward or reverse directions simultaneously, make sure that you wind the strings on the two pulleys in the same direction (clockwise or anticlockwise). This way, you will ensure that the drawing robot moves in a particular

direction (upwards or downwards) on the whiteboard with the help of both the motors using a positive or negative command signal.

Apply the Pythagorean theoram for the Distance

Before we start moving the robot, we need to measure the parameters $Base$, L_1, and L_2. We will use a known value of L_{arm} to compute Z_1 and Z_2. As the robot moves around the whiteboard, we'll update Z_1 and Z_2 based on the encoder measurements. To calculate the values of `x` and `y`, we can use these known values of Z_1, Z_2, and $Base$ along with the **Pythagorean Theorem**. If you drop a perpendicular line from the marker to the line between the two pulleys, two right-angled triangles are formed, as shown in the following diagram.



The sides of these triangles have the following relationships.

$$Z_1^2 = x^2 + y^2$$

$$Z_2^2 = (Base - x)^2 + y^2$$

If we solve the second equation for y^2, the first equation can be rewritten as,

$$Z_1^2 = x^2 + [Z_2^2 - (Base - x)^2]$$

Expanding this equation and solving for `x`, we end up with the following equations to solve for `x` and `y` sequentially.

$$x = \frac{Base^2 + Z_1^2 - Z_2^2}{2 \cdot Base}$$

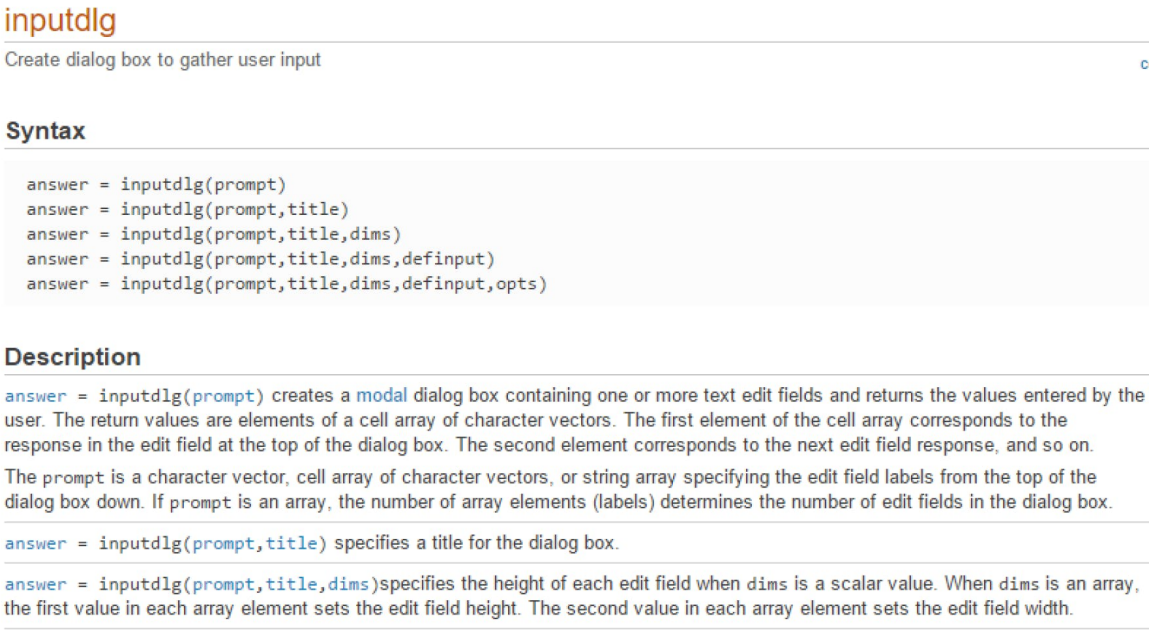
$$y = \sqrt{Z_1^2 - x^2}$$

Understand Input Dialogs

An easy way to allow a user to interact with your MATLAB program is with the help of a dialog box. Dialog boxes are simple apps that can report messages, gather input, or allow the user to interact with the file system. See the MATLAB documentation on **Dialog Boxes** for a list of dialog boxes available in MATLAB. To request input from a user, use an input dialog box. This can be created with the **inputdlg** function. There are several options for configuring this dialog box, so you should review the documentation page to better understand the available syntax. Open the documentation page directly from the MATLAB command prompt by typing the following:

```
>> doc inputdlg
```

Read the **Syntax** and **Description** sections of this documentation page to understand all the allowed input arguments for this function.



inputdlg

Create dialog box to gather user input

Syntax

```
answer = inputdlg(prompt)
answer = inputdlg(prompt,title)
answer = inputdlg(prompt,title,dims)
answer = inputdlg(prompt,title,dims,definput)
answer = inputdlg(prompt,title,dims,definput,opts)
```

Description

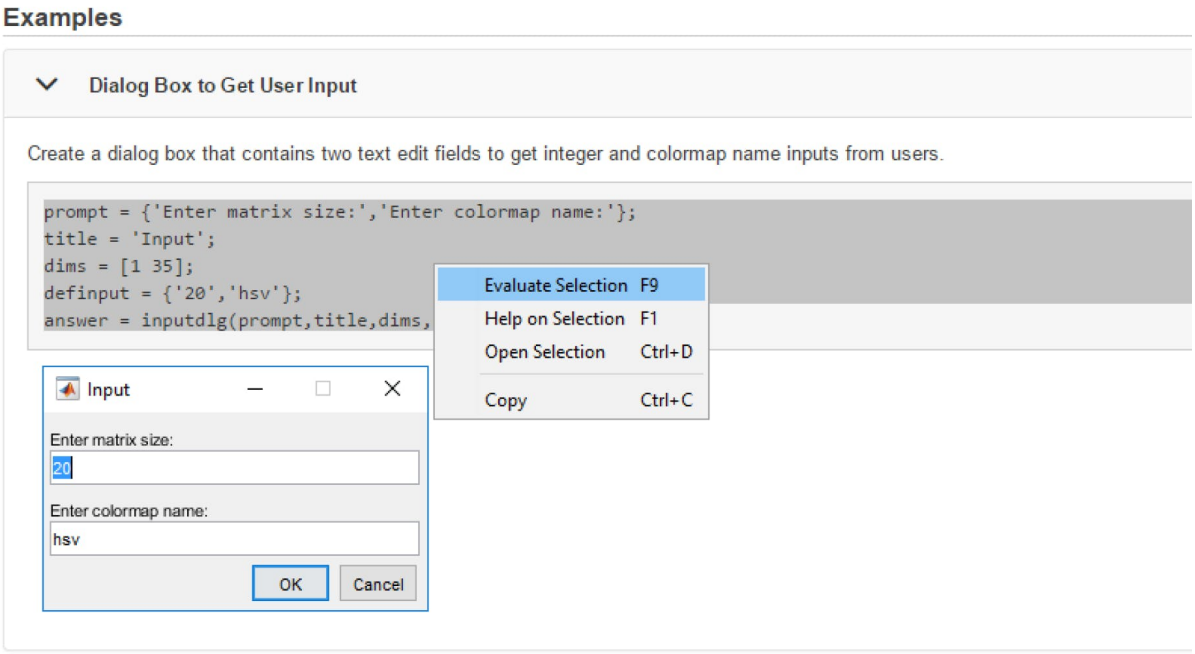
`answer = inputdlg(prompt)` creates a modal dialog box containing one or more text edit fields and returns the values entered by the user. The return values are elements of a cell array of character vectors. The first element of the cell array corresponds to the response in the edit field at the top of the dialog box. The second element corresponds to the next edit field response, and so on.

The `prompt` is a character vector, cell array of character vectors, or string array specifying the edit field labels from the top of the dialog box down. If `prompt` is an array, the number of array elements (labels) determines the number of edit fields in the dialog box.

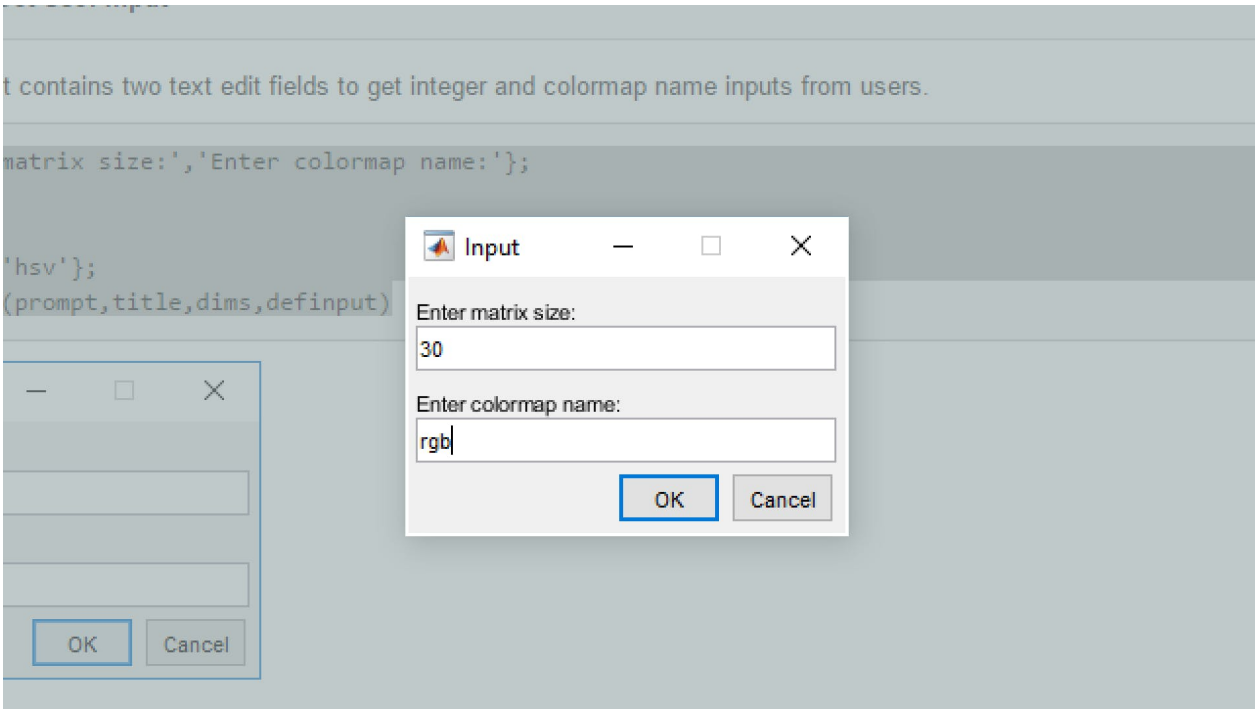
`answer = inputdlg(prompt,title)` specifies a title for the dialog box.

`answer = inputdlg(prompt,title,dims)`specifies the height of each edit field when `dims` is a scalar value. When `dims` is an array, the first value in each array element sets the edit field height. The second value in each array element sets the edit field width.

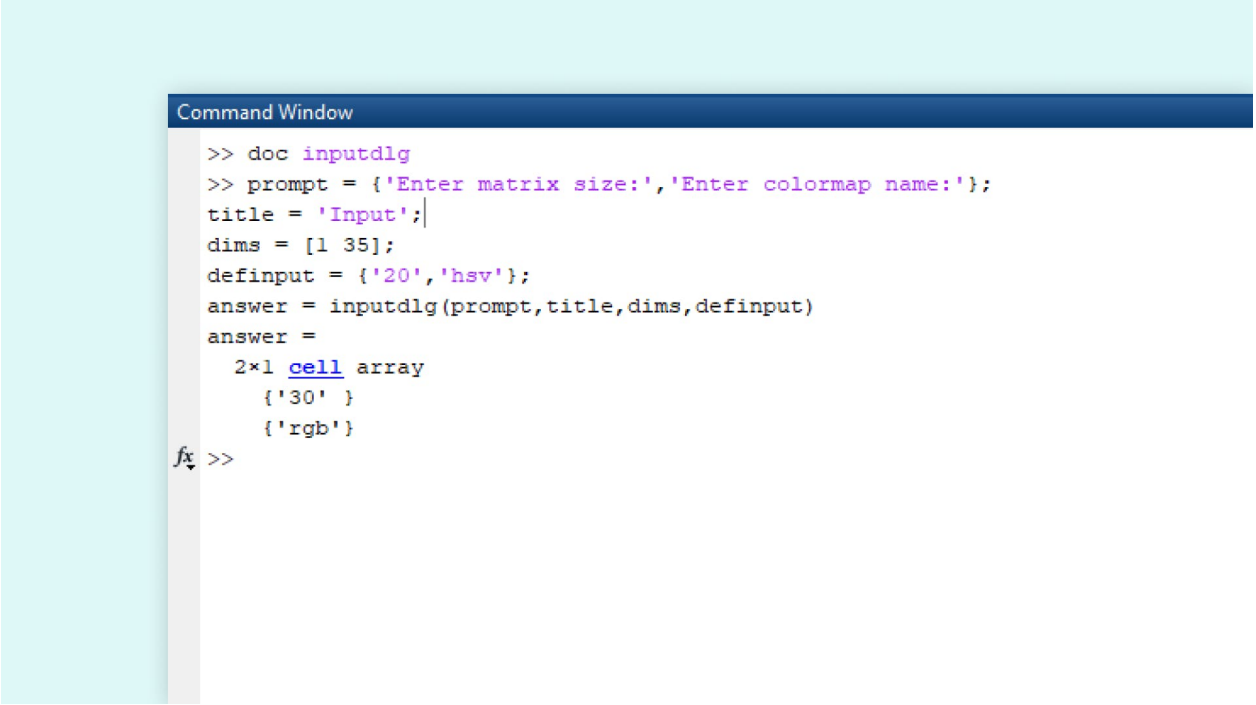
Then scroll down to the **Examples** section. One of the advantages of opening the documentation from within MATLAB rather than online is that you can run examples directly from the documentation page. Highlight the code from **Example 1**, then right-click and select **Evaluate Selection**.

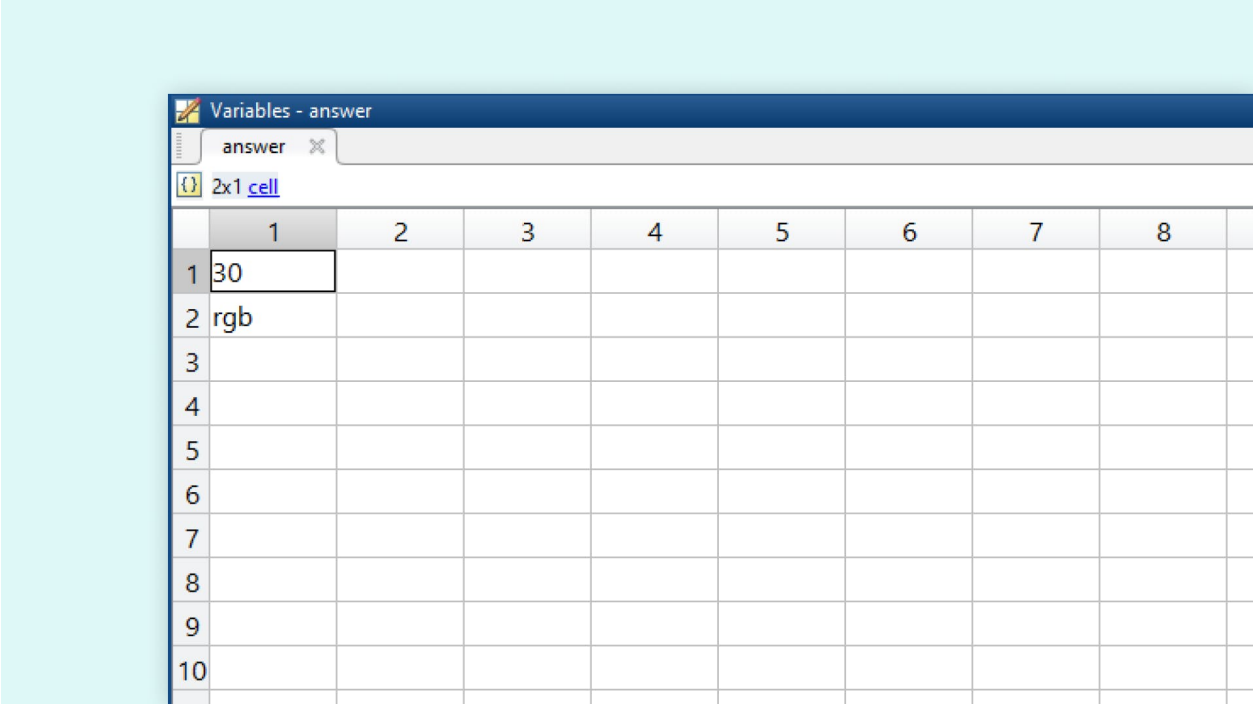


When the dialog launches, enter new values for matrix size (30) and colormap name (rgb). Click **OK**.



Return to the MATLAB Command Window. Note that the example code you evaluated appears here and in your Command History. Also, note there is a new variable `answer` in the **Workspace**. Check its value by double clicking on the variable.





Now try executing the code from other examples or write your own code that creates a dialog box to request input from a user. In the next section, you'll use an

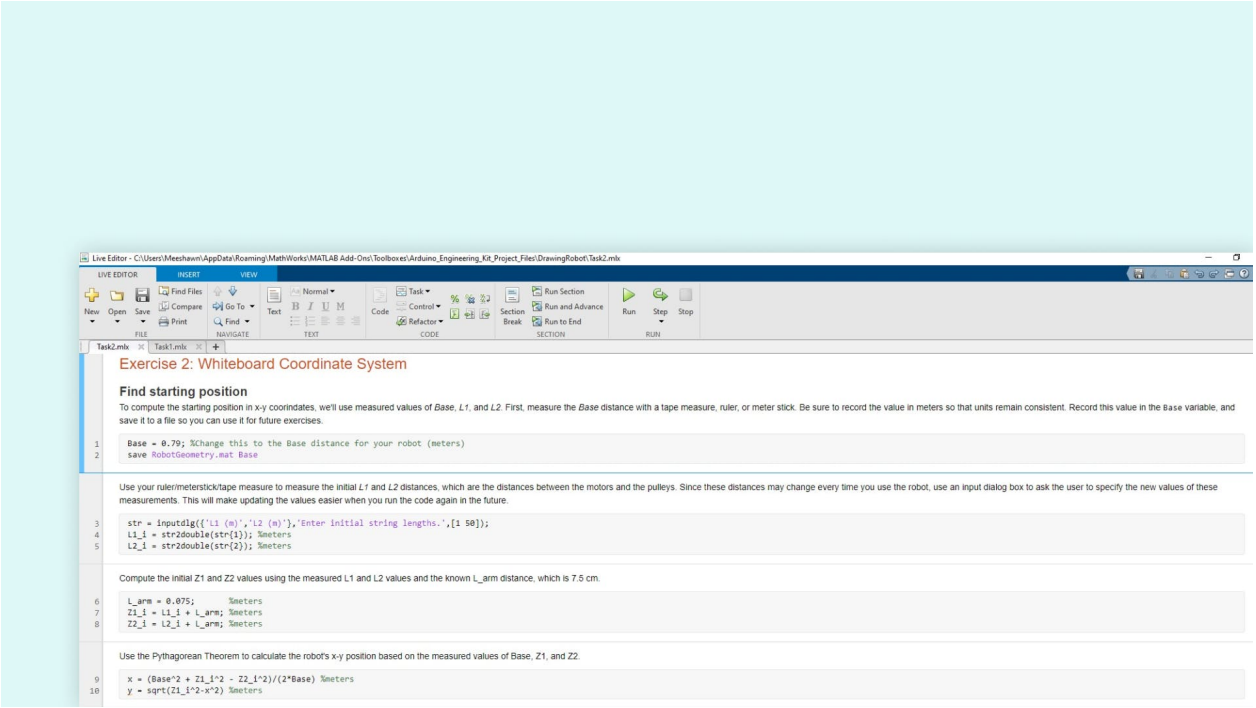
input dialog to update the robot measurements without having to edit your code each time you run it.

Find a Starting Position

Open the live script **Task2.mlx** by typing in the Command Window:

```
>> edit Task2
```

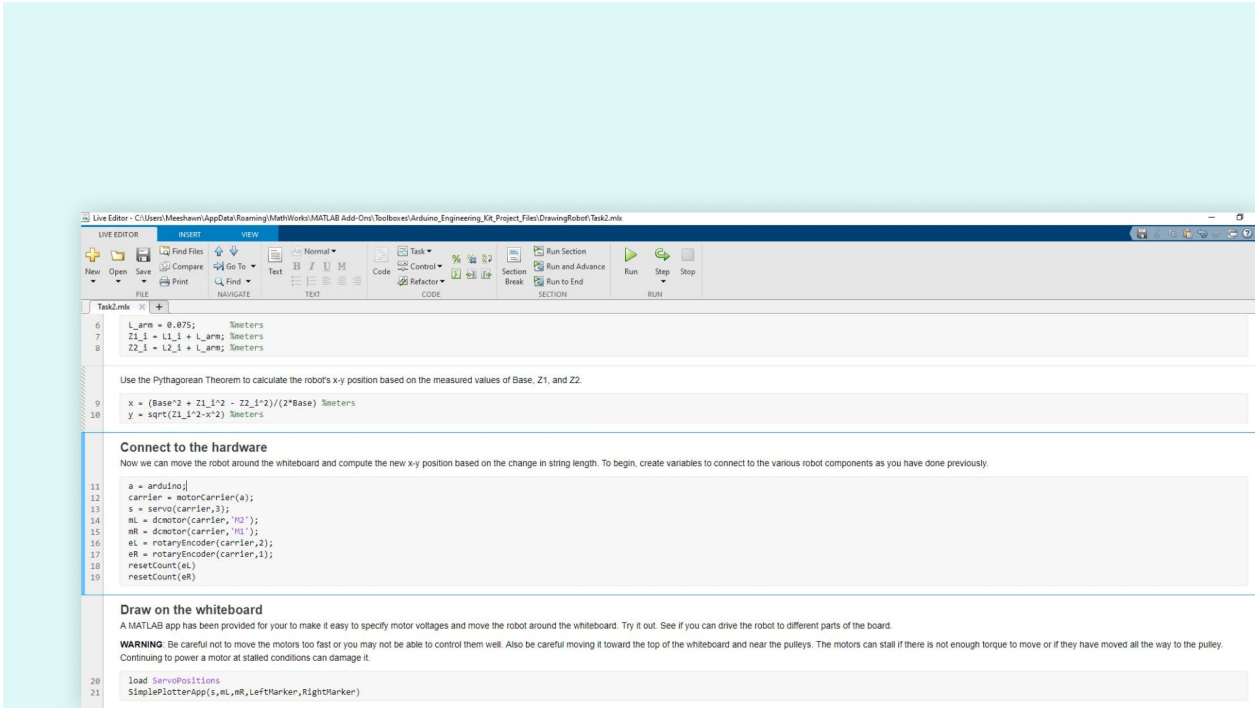
Measure the $Base$, L_1, and L_2 distances on your robot with a meter-stick, ruler, or tape measure. Then follow the instructions given in the first four sections of the live script under the heading **Find starting position**, entering these values wherever appropriate. Please note that all measurements should be written in **meters**. This will allow you to maintain consistency in the units and calculate the starting x-y position of the robot on the whiteboard.

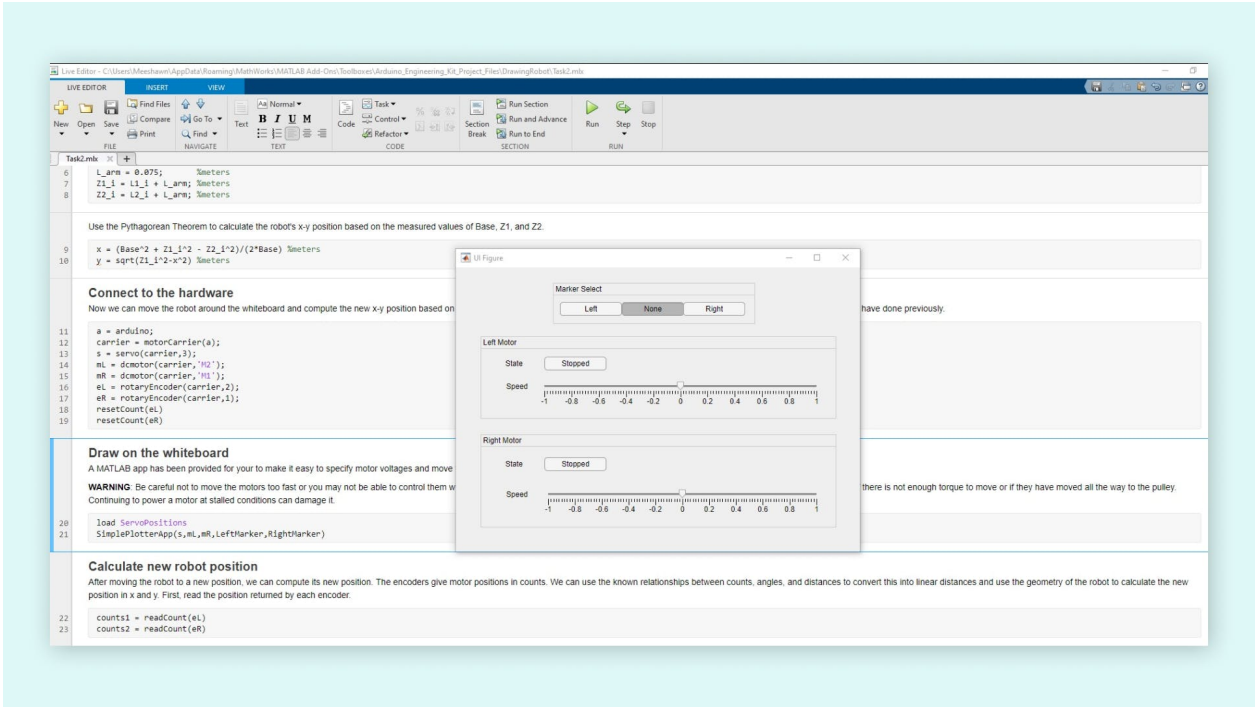


Connect to Hardware and Draw on Whiteboard

As you did in the previous exercise, you will now connect to the Arduino Nano 33 IoT board and control the motors. This time, however, you will use an interactive application (or app) in MATLAB to start and stop the motors and specify their voltages. MATLAB apps typically have a graphical user interface and code to perform a specific task. The drawing robot project provides an app called

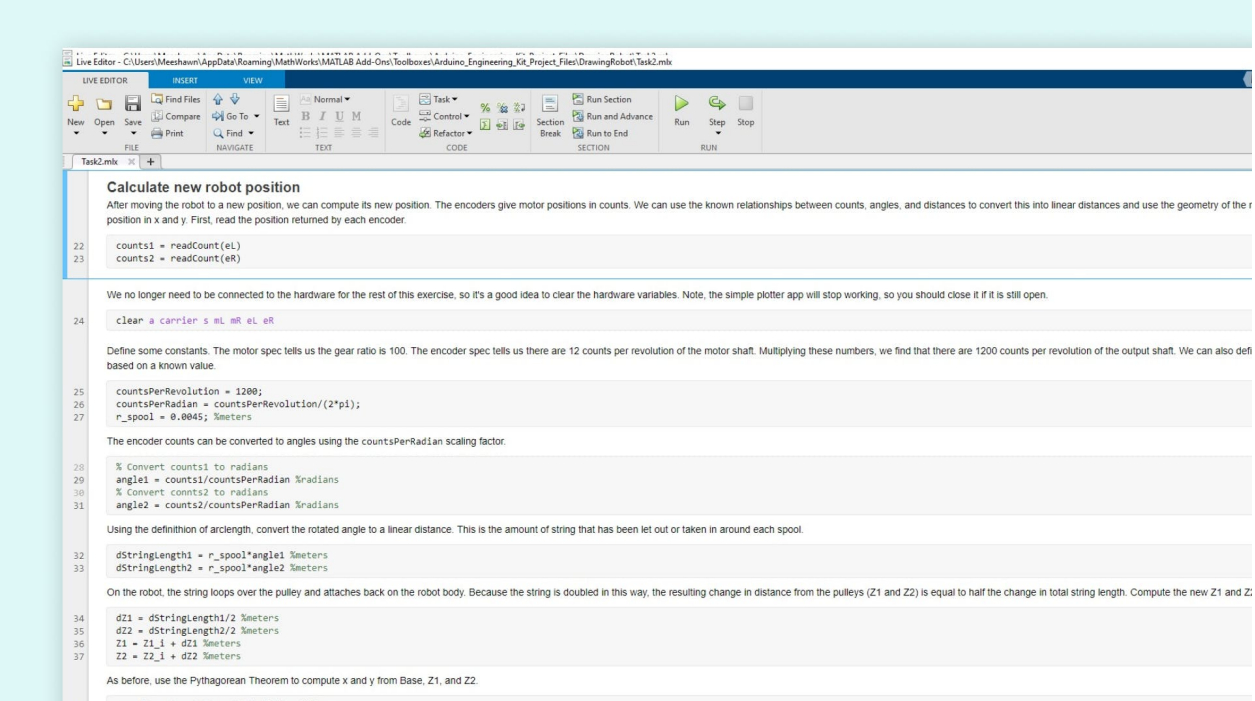
`SimplePlotterApp` that can be used to interactively control the robot's motors. Execute the code under the **Connect to the hardware** and **Draw on the whiteboard** sections in the live script **Task2.mlx**. This will launch an app with buttons allowing you to interactively start and stop each of the motors and change their speed in open-loop control mode. Be careful not to move the motors too fast or to pull the robot all the way up to the top of the whiteboard where the motors may stall.





Compute the New Position

With the information you have so far, it is now possible to compute the new position of the robot on the whiteboard in x-y coordinates. You can now read the new encoder counts, convert these values to angles, and then convert these angles to distances. Finally, you can use the robot geometry to convert the distances into x-y positions on the whiteboard. Start by reading the current count values for the two encoders. Note that you will need to revert the sign of the respective motor if you spooled in the clockwise direction. You can then clear the hardware variables in MATLAB, since you'll no longer need them for this exercise. Execute the first two sections of **Task2.mlx** under the heading **Calculate new robot position**.



Next, you will perform the necessary computations to calculate the corresponding x-y position on the whiteboard. You have the motor positions in units of counts. How do you convert this to an x-y position? First, you can convert the position in counts to an angular position. You know the number of counts per revolution (C_{rev}) of the output motor shaft from the motor specification sheet. The angular position of the motor (θ) is therefore computed from the counts per radian (C_{rad}) as follows:

$$C_{rad} = \frac{C_{rev}}{2\pi}$$

$$\theta = \frac{C_{rad}}{counts}$$

The amount of string spooled or unspooled by the motors (ΔL_{string}) is related to the angle it has rotated (in **radians**) through and the radius of the spool (r_{spool}). In fact, we can calculate it using the definition of arc length.

$$\Delta L_{string} = r_{spool} \cdot \theta$$

On the robot, the string loops over the pulley and then back to the robot body. Since the string is doubled in length, the resulting change in the distance from the pulleys (Z_1 or Z_2) equals half the change in the total string length (ΔL_{string}). We can compute the new lengths (Z_1 and Z_2) based on this relationship.

$$\Delta Z_{(1 \text{ or } 2)} = \frac{\Delta L_{string}(1 \text{ or } 2)}{2}$$

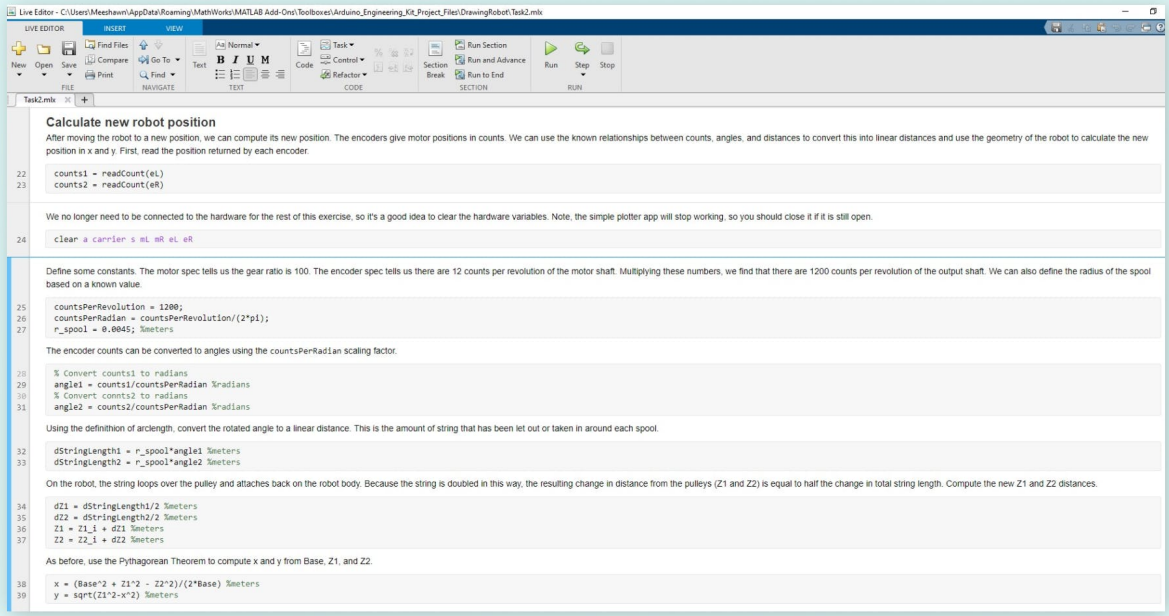
$$Z_{(1 \text{ or } 2)} = Z_i + \Delta Z_{(1 \text{ or } 2)}$$

The new robot position is fully defined by the variables $Base$, Z_1, and Z_2. As we did earlier in this exercise, we can use these values to compute the current values of `x` and `y`.

$$x = \frac{Base^2 + Z_1^2 - Z_2^2}{2 \cdot Base}$$

$$y = \sqrt{Z_1^2 - x^2}$$

To perform these calculations, execute the remaining sections of **Task2.mlx** under the heading **Calculate new robot position**.



Encapsulate code in Functions

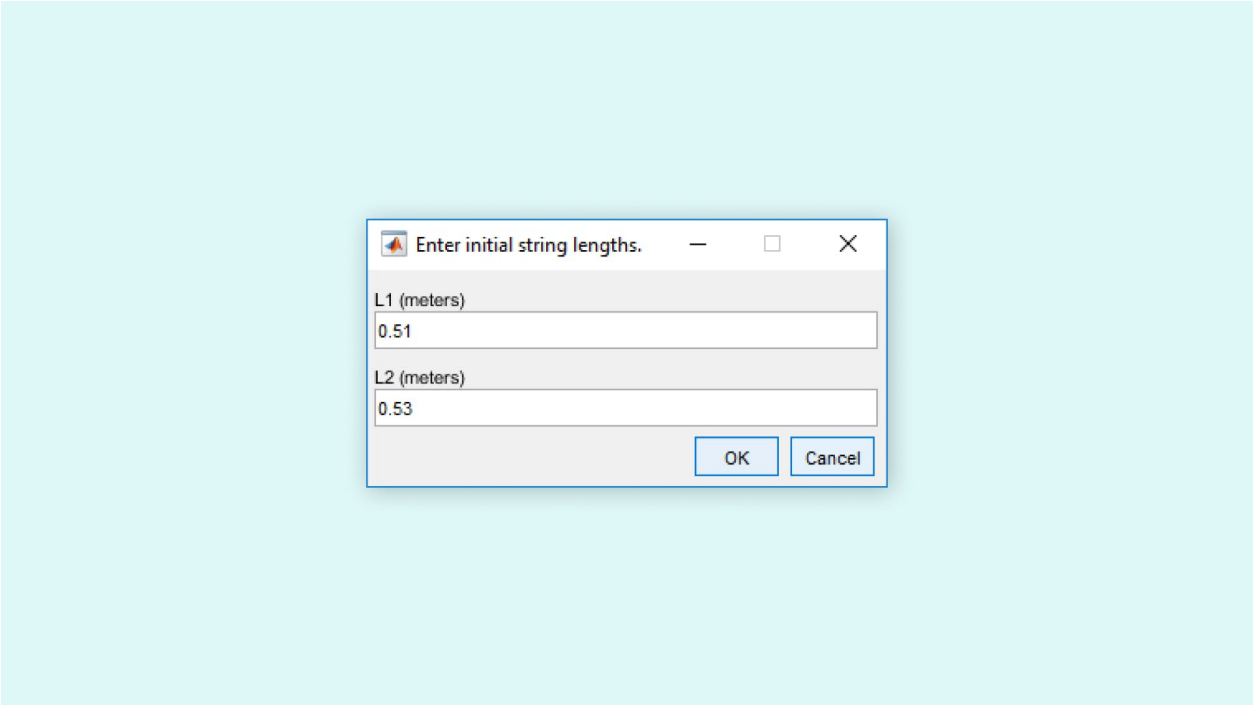
We'll want to repeat some of the tasks performed in this exercise again in the future. To better organize our code, reduce repetition, and make it easy to solve these problems in the future, we can create MATLAB functions that perform specific tasks.

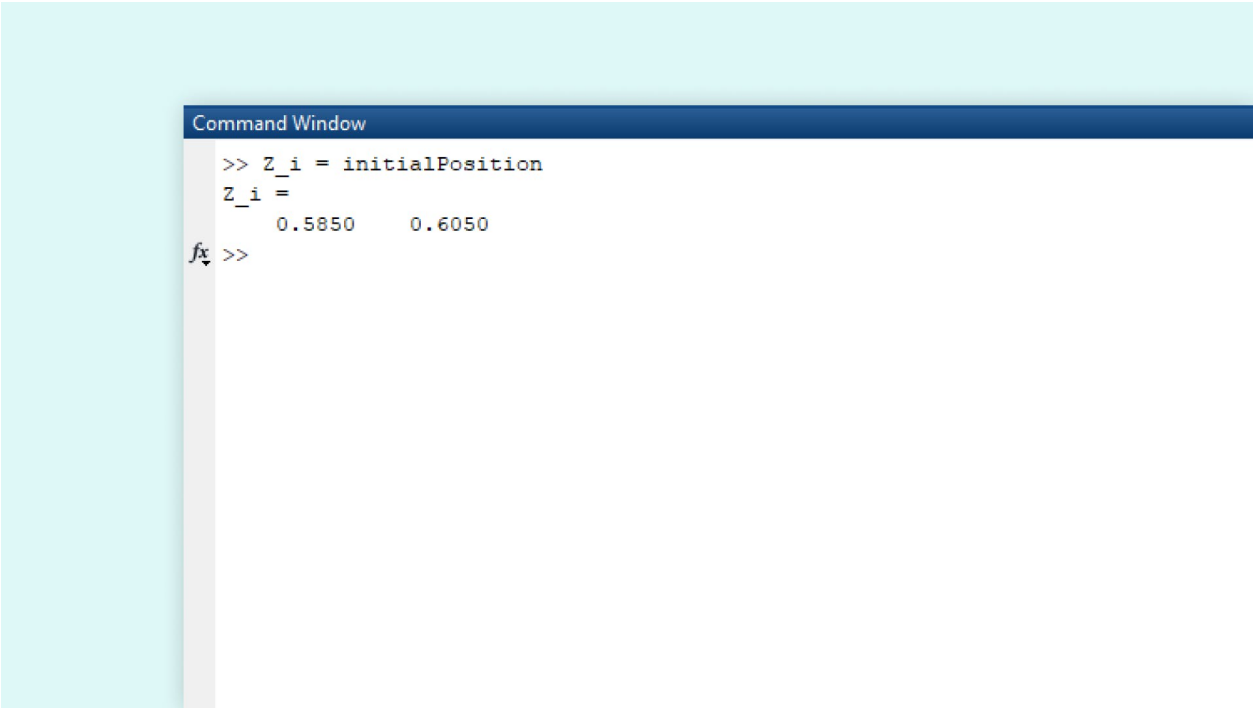
First, write a MATLAB function for requesting the initial string length measurements from the user. From your MATLAB window, select **New Script** under the Home tab. This will create an empty MATLAB file. Enter the following code and save the file as **initialPosition.m**.

```
function Z_i = initialPosition()
str = inputdlg({'L1 (meters)','L2 (meters)'},'Enter initial string
lengths:',[1 50]);
L1_i = str2double(str{1}); %meters
L2_i = str2double(str{2}); %meters
L_arm = 0.075;       %meters
Z_i = [L1_i L2_i] + L_arm; %meters
end
```

Call your new function from the MATLAB Command Window. Enter the string length values. Click on **OK**, and observe the result.

```
>> Z_i = initialPosition
```

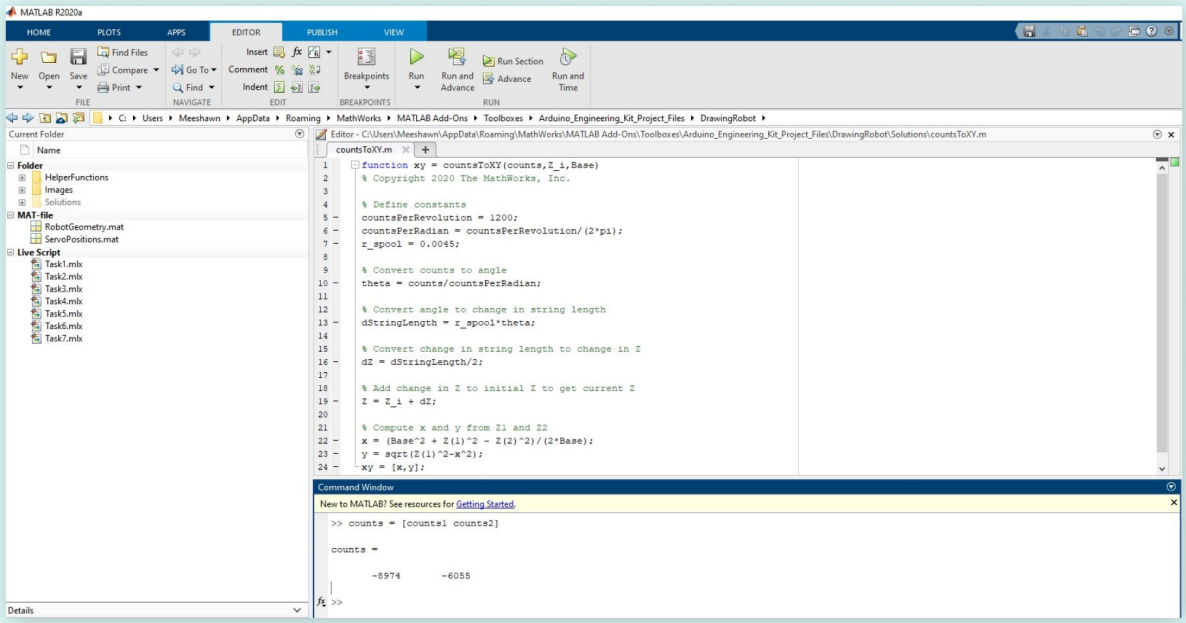




Note that we're now using a single variable Z_i to accommodate both Z_{1_i} and Z_{2_i}. This variable has size 1x2 and can be indexed with the syntax `Z _ i(1)` and `Z _ i(2)` to extract its components. Let's also place the `counts1` and `counts2` variables together under a single variable `counts`. This will make our code much simpler. Execute the following line at the MATLAB Command prompt to create the variable `counts`.

```
>> counts = [counts1 counts2]
```

Next, create a function for converting the measured encoder counts to an x-y position on the whiteboard. Create a new function called `countsToXY()` and include the following code.




```
function xy = countsToXY(counts,Z_i,Base)
% Define constants
countsPerRevolution = 1200;
countsPerRadian = countsPerRevolution/(2\*pi);
r_spool = 0.0045;
theta = counts/countsPerRadian;
dStringLength = r_spool*theta;
dZ = dStringLength/2;
Z = Z_i + dZ;
x = (Base^2 + Z(1)^2 - Z(2)^2)/(2*Base);
y= sqrt(z(1)^2-x^2);
xy = [x y];
end
```

Save the file as **countsToXY.m**. Verify that this function behaves correctly. You should still have the variables `counts`, Z_i, and $Base$ in your workspace. Execute the following at the MATLAB command prompt, and verify that the x-y position obtained is correct.

```
>> xy = countsToXY(counts,Z_i,Base)
```

Files

- Task2.mlx
- countsToXY.m
- initialPosition.m

Learn By Doing

If you click Cancel instead of OK on the input dialog box, what happens? What happens when you continue running the code? Is this what you would expect? Describe the desired behavior and how you would design your code to behave this way when a user clicks the Cancel button. Using just the `SimplePlotterApp` app, try to control the robot and draw a shape. What shapes are easy to draw? What shapes are difficult?

Run an experiment to measure the radius of the spool. Reset the encoder. Then write a script that will spin the motor for a few seconds in a direction that unspools the string. Hold onto the string and don't let it slip off the edge of the spool. When the motor stops, measure (with a ruler) how much string was let out. Read the encoder count to see how far the motor turned and convert this value to an angle. Now use the equation for arc length to compute the radius of the circle that unspooled that amount of thread and spun through that angle. Compare your experimentally determined radius to the radius provided in the exercise.

4.3 Move to Specified Points

In this exercise, you will choose specific locations on the whiteboard and move the robot to these locations. You will use functions from the **Closed-Loop Control** exercise to achieve position control. You will also learn to vectorize your MATLAB functions and use these functions to control the robot. At the end of this exercise, you'll have a shorter algorithm that can raise/lower the marker and draw simple shapes.

In this exercise, you will learn to:

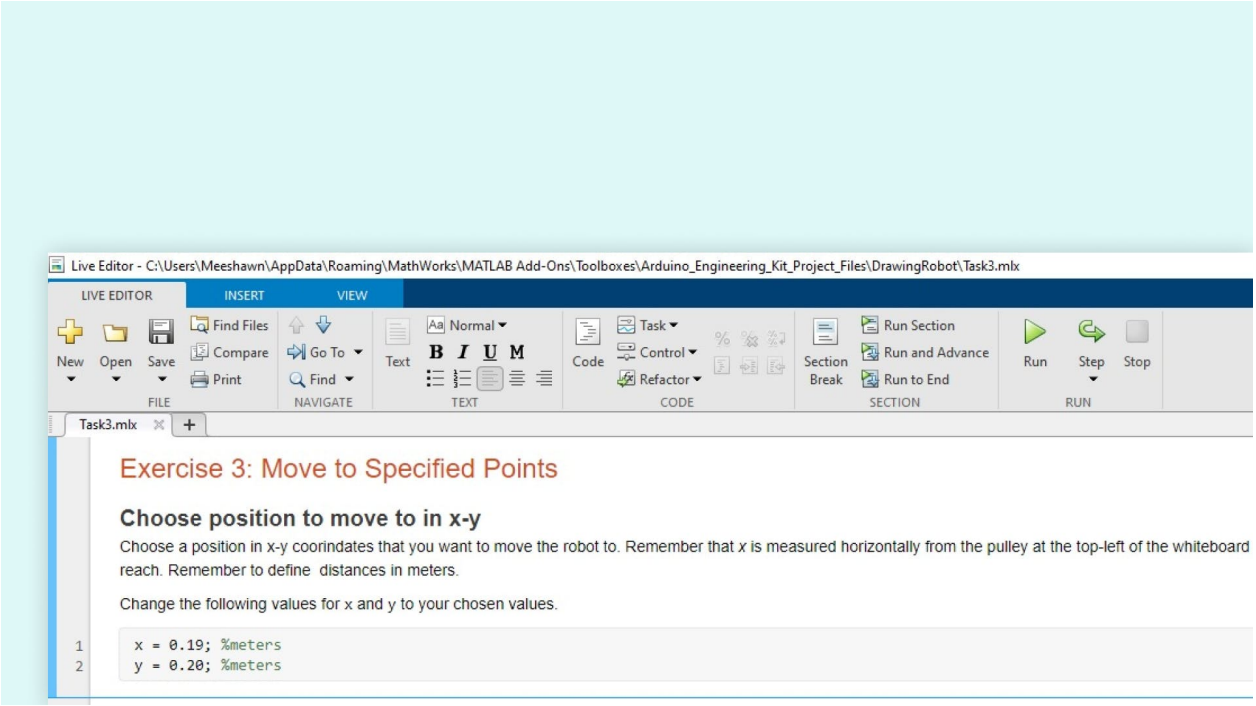
- construct simple MATLAB algorithms,
- write vectorized code.

Choose a Position to Move-to

Let's move on to the next exercise. Hang the robot up on the whiteboard, making sure that both pulleys are located over the same horizontal line. Choose a target position on the whiteboard where you'd like to move the marker, and make sure it's a point that is reachable by the robot. Measure the `x` and `y` coordinates of that point using the whiteboard coordinate system where `x` is the horizontal distance to the right of the top-left pulley and `y` is the vertical distance down from the line connecting the pulleys. Open the live script **Task3.mlx**.

```
>> edit Task3
```

In the first section, called **Choose position to move to in x-y**, replace the values of `x` and `y` with the values for the point you want your robot to move to. Make sure that all measurements of the coordinates are in **meters**. Execute this section.



Convert Target position to Angular Displacement

Previously, we saw how to interactively control the motors on the whiteboard and compute the new x-y positions of the robot with the help of the encoder count values. In this exercise, we intend to do the opposite. You will use the same geometric relationships between counts, angles and whiteboard distances that you used in the previous exercise. However, instead of taking a measured encoder count value and computing an x-y coordinate, you'll take a target x-y coordinate and compute the required angular displacement for that position. To do this, use the same equations from the previous exercise, but reverse the order of calculation.

In a later exercise, you will be using the closed-loop control function `writeAngularPosition()` to drive the motors to the specified x-y coordinates and hence you will only need to compute the required angular rotation and not the encoder counts.Starting with the x-y coordinate, you can use the **Pythagorean Theorem** to calculate the lengths of the desired Z_1 and Z_2 variables.

$$Z_1 = \sqrt{x^2 + y^2}$$

$$Z_2 = \sqrt{(Base - x)^2 + y^2}$$

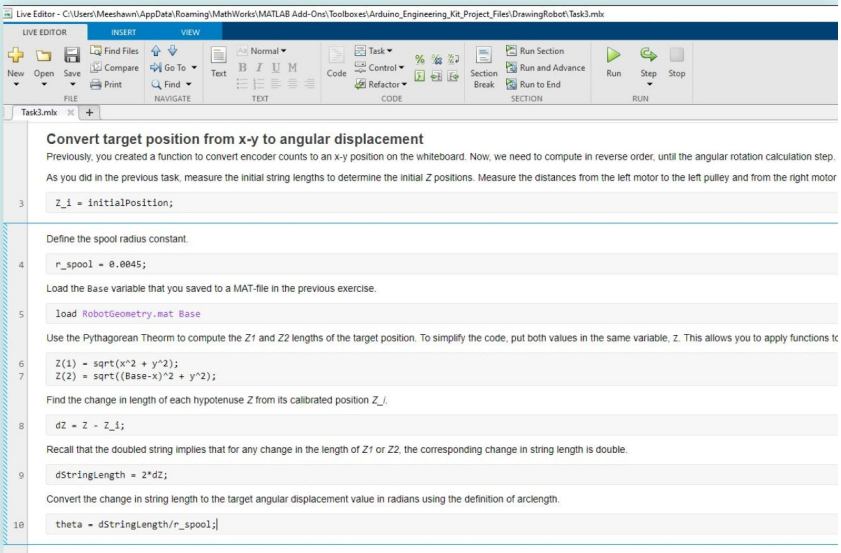
Next, you can compute the change in `Z` and the equivalent change in the string length.

$$\Delta Z = Z - Z_i$$

$$\Delta L_{string} = 2 \cdot \Delta Z$$ Finally, use the arc-length definition to convert the change in string length to the desired angular displacement value in **radians**.

$$\theta = \frac{\Delta L_{string}}{r_{spool}}$$

To perform these calculations in MATLAB, execute the sections of code under the **Convert target position to angular displacement** heading under **Task3.mlx**.



Converting Position to Angular Rotations

As you did in a previous exercise, you can take code that you've written to solve a specific problem and encapsulate it in a function. Just as you did for converting counts to position, you can now write a function that converts position to angle. Create a new function called `xyToRadians()` and include the following code:

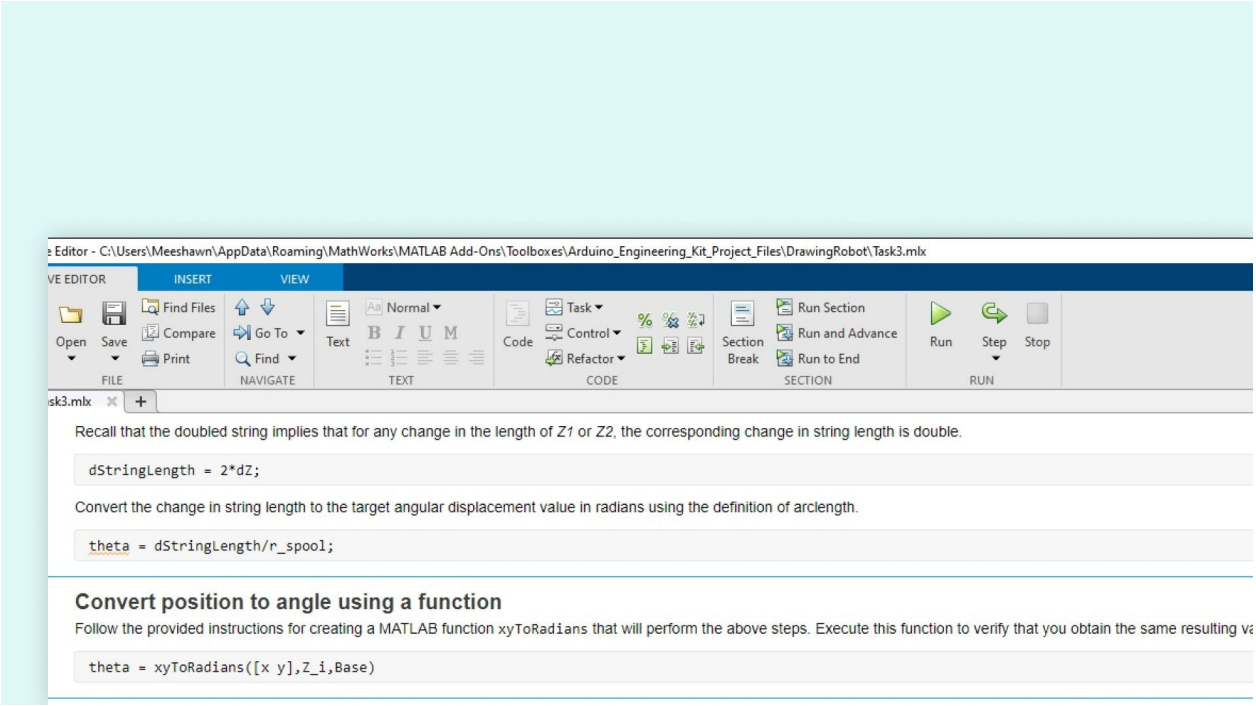
```
function theta = xyToRadians(xy,Z_i,Base)

% Define constants
r_spool = 0.0045;

% Convert x and y to Z1 and Z2
x = xy(:,1);
```

```
y = xy(:,2);
Z(:,1) = sqrt(x.^2 + y.^2);g
Z(:,2) = sqrt((Base-x).^2 + y.^2);

% Subtract initial Z to get change in Z
dZ = Z - Z_i;

% Convert change in Z to change in string length
dStringLength = 2*dZ;

% Compute change in string length to angle
theta = dStringLength/r_spool;
```

Note that there is a slight difference between the code in this function and the code that you executed in **Task3.mlx**. Here, we index the `xy` and `Z` variables with subscripted indices `(:,1)` and `(:,2)` in the following lines:

```
y = xy(:,2);
Z(:,1) = hypot(x,y);
Z(:,2) = hypot(Base-x,y);
```

This syntax allows you to refer to all the values in the first or second column of these variables respectively, which enables you to use the `xyToRadians()` function on not just one x-y coordinate, but on a set of 'N' x-y positions defined in an Nx2 array where the first column has all the x-positions and the second column has all the y-positions. You'll get to apply this workflow at the end of this exercise. Save the function `xyToRadians.m` and then test it out by executing the section of the live script **Task3.mlx** under the heading **Convert position to angle using a function**. Make sure that the angular displacements values obtained from this function match with the values obtained previously. This will prove that your function works as expected.



Understand moveToRadians() Function, Line by Line

To move the robot through a series of specified positions on the whiteboard, the function `moveToRadians()` has been provided for you. This function makes use of the angular displacement values obtained from the previous exercise to drive the motors to the specified locations. You will be using functions from the **Closed-Loop Control** section to achieve the position control.

A 'For-Loop' has been implemented to vectorize the function `moveToRadians()`, thereby allowing it to accept a vector input containing a series of target angular displacement values. As seen in the Closed-Loop Control section, the function `writeAngularPosition()` automatically starts and stops the motors after achieving the target angular displacement value. The PID algorithm implemented underneath constantly tracks the input values and hence responds to any changes in the input immediately. This implies that, if the target angular position is modified while the motor is moving towards this target setpoint, then it will instantly respond to the new target setpoint, thereby aborting the movement to the previous target setpoint. In order to achieve sequential tracking of all the target angular displacement values, the function should be able to cater to all the target setpoints from the input vector, one at a time.

Read through the following description of the function to see how it continuously measures the current positions of the motors and compares them with their past

values to ensure sequential tracking of a series of input angular positions, thereby moving the robot in the right direction. In this manner, the motors respond to one target angular position command at a time and execute the motion before responding to the next target input.

```
function moveToRadians(theta,pidML,pidMR)
% Copyright 2020 The MathWorks, Inc.**

% Define the gear ratio constant`
gearRatio = 100;

% Convert the motor shaft rotations to the actual motor rotations
thetaL = theta(:,1)\*gearRatio;
thetaR = theta(:,2)\*gearRatio;
% Implement an nth order delay 'nD' to store the past angular position
values nD = 3;

% Initializing delay vectors of length 'nD' for storing the past angular
position values for both the motors`
thetaDelayL = zeros(nD,1);`
thetaDelayR = zeros(nD,1);

% Size of the Input Sequence
N = size(theta,1);
```

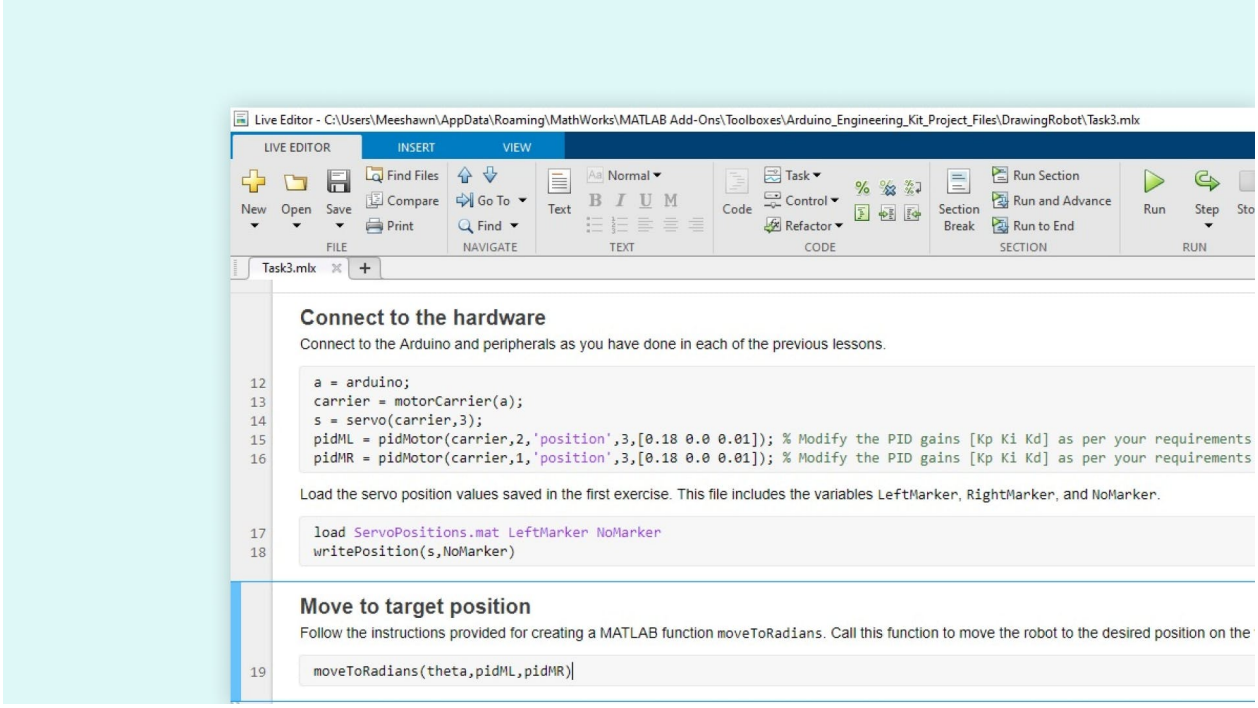
As seen above, the function `moveToRadians()` accepts a list of angular displacement values and PID control objects as inputs. It then makes use of the closed-loop control functions to move the robot to a list of desired x-y positions and monitors current and past position values to execute sequential tracking of inputs. Note that, unlike the exercise in **Task1.mlx** under the **Closed-Loop Control** section, the mode of calculation of the angle of rotation is '**abs**' (absolute) instead of '**rel**' (relative). In the next section, you will see how this feature helps in moving the robot to multiple positions.

A delay vector has been implemented to store the past '**nD**' angular displacement values (nDth order delay). The value selected for the Delay parameter `nD` in the function worked well with the robot during our tests. You can edit this parameter (via the command `edit moveToRadians()`) and adjust it to tune the robot for your environment, using the value provided as a starting point.

Move Robot to Target Position

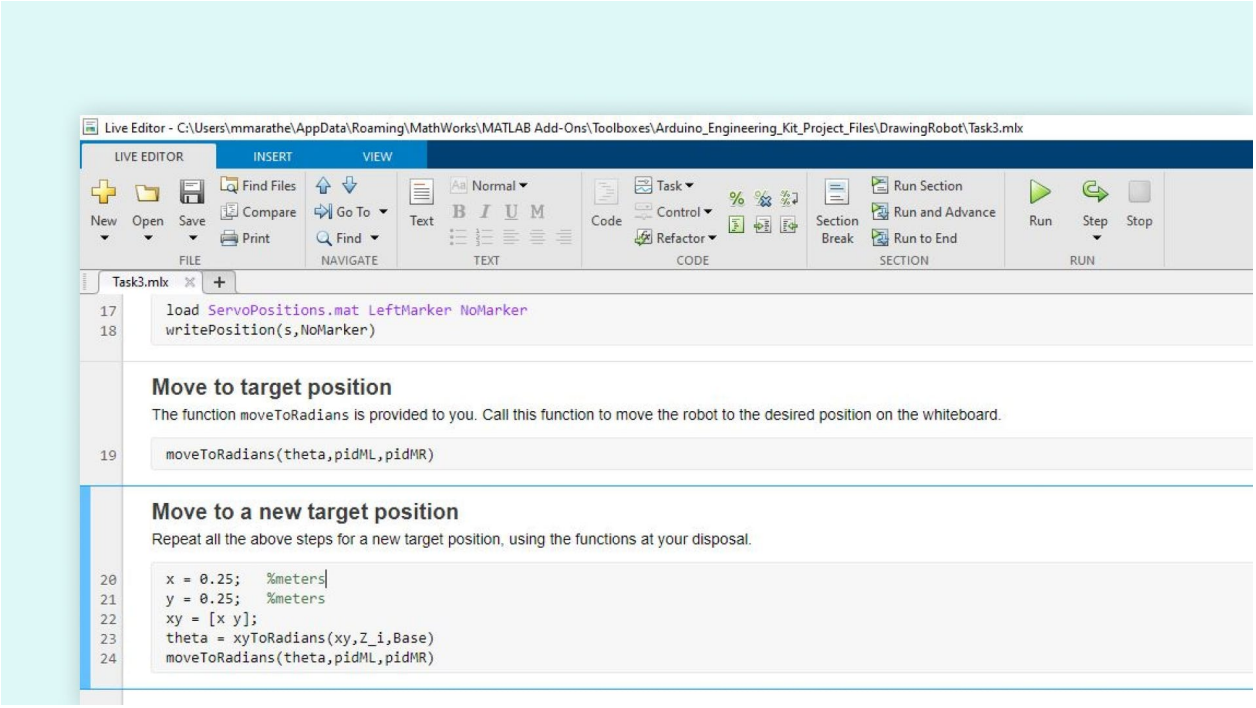
Start by connecting to the motors and creating a PID controller for the motors. Under the **Closed-Loop Control** subsection, we saw how to configure and implement a PID controller in order to achieve closed-loop position control of the motors. The values for the parameters [K_p, K_i, K_d] worked well with the robot

during our tests. You can edit the parameters and adjust these to tune the robot for your environment using these values as a starting point. Execute code under the sections **Connect to the hardware** and **Move to target position** of the live script **Task3.mlx**.



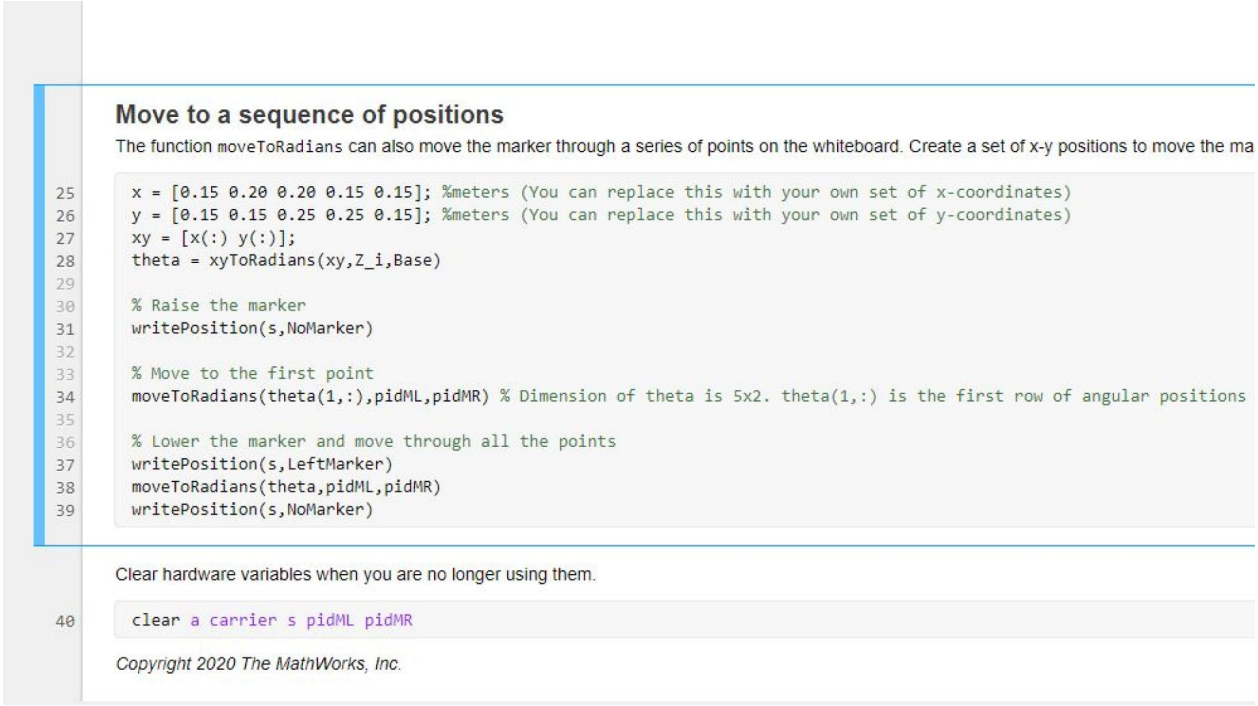
Move to a New Target Position

You can now direct the robot to move to different parts of the whiteboard. This only takes a few lines of code now that you have the functions `xyToRadians()` and `moveToRadians()` at your disposal. Since the function `writeAngularPosition()` is configured in **absolute** mode, you need not use the function `initialPosition()` to measure Z_i every time the robot moves to a new x-y position. It is needed only once before connecting to the hardware. If you disconnect the hardware by clearing the arduino and motor carrier connections from the MATLAB Workspace then you will need to supply the initial position again. Under the section Move to a new target position, choose new values for `x` and `y`. Execute code in this section to move the robot to a new position. You can do this multiple times with different `x` and `y` positions.



Move to a Sequence of Positions

As seen in the previous sections, the functions `xyToRadians` and `moveToRadians` are capable of working on a vector of target angular position values. If you define your target x-y positions with the help of an Nx2 array where the first column contains the list of x-coordinates and the second column holds the y-coordinates, you can direct the robot to move through a series of points on the whiteboard. The section **Move to a sequence of positions** under **Task3.mlx** has an example of coordinates that form a rectangle. Check whether these points fall in the drawable area of your whiteboard and change them if they do not. Notice that this code also includes a sequence of actions required for drawing shapes. It raises the marker, moves to the first point, lowers the marker, moves through all the points, and then raises the marker again. Execute the code under this section.



Move to a sequence of positions

The function `moveToRadians` can also move the marker through a series of points on the whiteboard. Create a set of x-y positions to move the ma

```
x = [0.15 0.20 0.20 0.15 0.15]; %meters (You can replace this with your own set of x-coordinates)
y = [0.15 0.15 0.25 0.25 0.15]; %meters (You can replace this with your own set of y-coordinates)
xy = [x(:) y(:)];
theta = xyToRadians(xy,Z_i,Base)

% Raise the marker
writePosition(s,NoMarker)

% Move to the first point
moveToRadians(theta(1,:),pidML,pidMR) % Dimension of theta is 5x2. theta(1,:) is the first row of angular positions

% Lower the marker and move through all the points
writePosition(s,LeftMarker)
moveToRadians(theta,pidML,pidMR)
writePosition(s,NoMarker)
```

Clear hardware variables when you are no longer using them.

```
clear a carrier s pidML pidMR
```

Copyright 2020 The MathWorks, Inc.

What did you observe after executing the code? Did the robot draw the rectangle properly? Was it able to reach the target x-y positions with decent accuracy? You will notice that although the robot is able to reach the target x-y positions, the end figure is not a rectangle, but a closed-sided polygon which passes through all the target positions. This is because the position control strategy focuses on achieving the target position, irrespective of the trajectory taken to achieve the goal. In order to achieve a perfectly straight line between two x-y positions on the whiteboard that are far apart, one needs to implement a speed control strategy in conjunction with position control.

Alternatively, you can divide the distance between the two x-y positions into smaller line segments and use the same position control strategy to obtain a fairly accurate straight line. In a later section, you will see how an image is eventually broken down into numerous equally spaced line segments and in this way, the robot is able to draw straight lines and even complex curves with decent accuracy.

Files

- Task3.mlx
- xyToRadians.m
- moveToRadians.m

Learn By Doing

In the last section of this exercise, you saw how you could raise and lower the marker between moving to positions. Try to draw a multi-line segment with this approach, raising the marker between lines. Draw a capital letter E. Define the coordinates of all the six segment endpoints. You can draw the three outer segments together, then raise the marker and draw the middle line. In MATLAB, use the equation of a circle to define x-y positions for eight points on a circle (this will form a closed-sided polygon). Draw this shape on the whiteboard.

4.4 Identify Position Limits

If you try to move the robot to the top of the whiteboard, it won't have enough power to move above a certain point. If you try to move it too close to the pulleys, it will collide with them and get stuck. At the bottom of your whiteboard, there may be a marker tray you don't want the robot to run into. In this exercise, you will determine what parts of the whiteboard the robot can reach and what parts you should avoid moving the robot to.

You'll learn a few important concepts in Physics that will help you determine how much load the motors are under at any point on the whiteboard. You'll perform computations and create visualizations to see the load distribution across the whiteboard and find locations where the load is the greatest.

In this exercise, you will learn to:

- Apply motor stall equations,
- Use Symbolic Math APIs to convert units,
- Create free-body diagrams,
- Compute forces on static body,
- Apply a function at every point on a grid,
- Plot surfaces and images in MATLAB.

Understand Motor Equations and Stall Conditions

For a DC motor, a mathematical equation can be used to describe the relationship between motor load (torque), supply voltage, and rotational speed. This is sometimes referred to as the DC motor torque equation:

$\tau = \frac{V - \omega \cdot k}{R} \cdot k$

In this equation, τ is the motor torque or motor load, V is the supply voltage, and ω is the angular speed (note that in some versions of this equation, the variable T is used for torque. For our project, we'll reserve T to use for tension). R refers to

the resistance in the motor windings, and k is the motor constant. The values of both R and k are constant for a given motor. There are two special cases that we can consider for this equation. One case refers to the situation when there is no load on the motor. In this case, τ = 0, and the equation simplifies to the following:

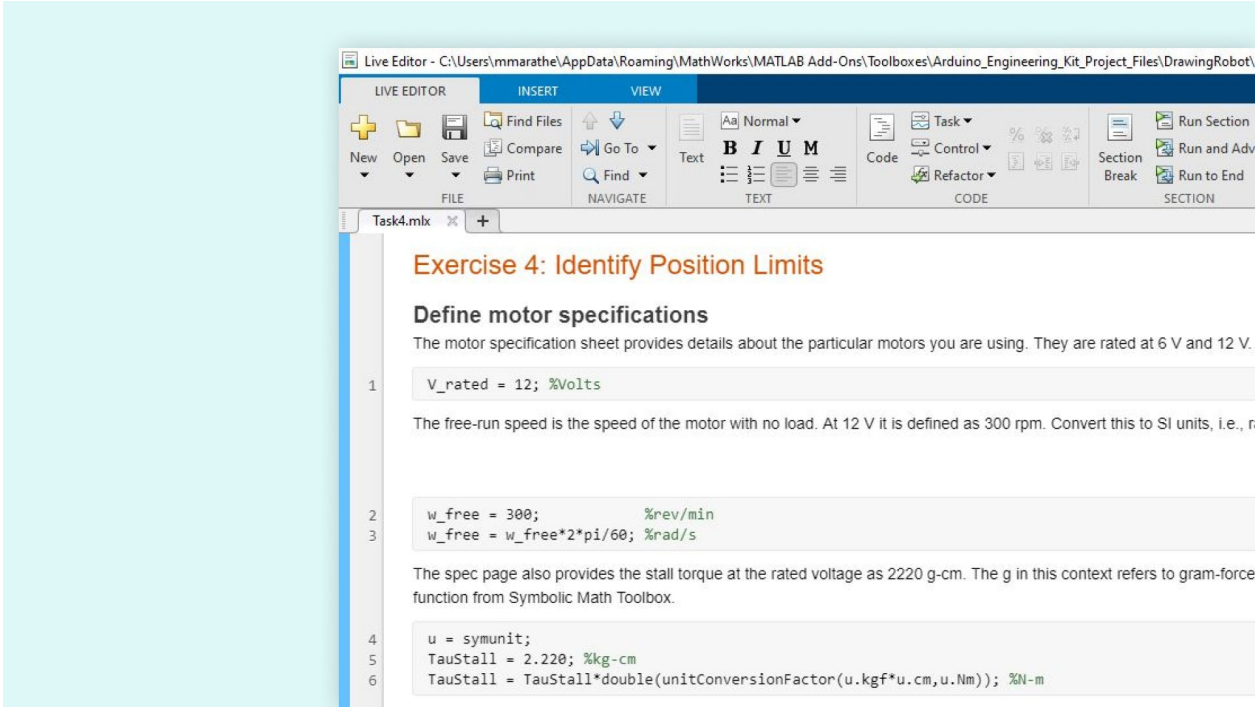
$$V = \omega_{free} \cdot k$$

The second special case is when the motor reaches **stall** conditions. This is when the load is so high that the motor cannot spin. In this case, ω = 0, and the equation simplifies to the following:

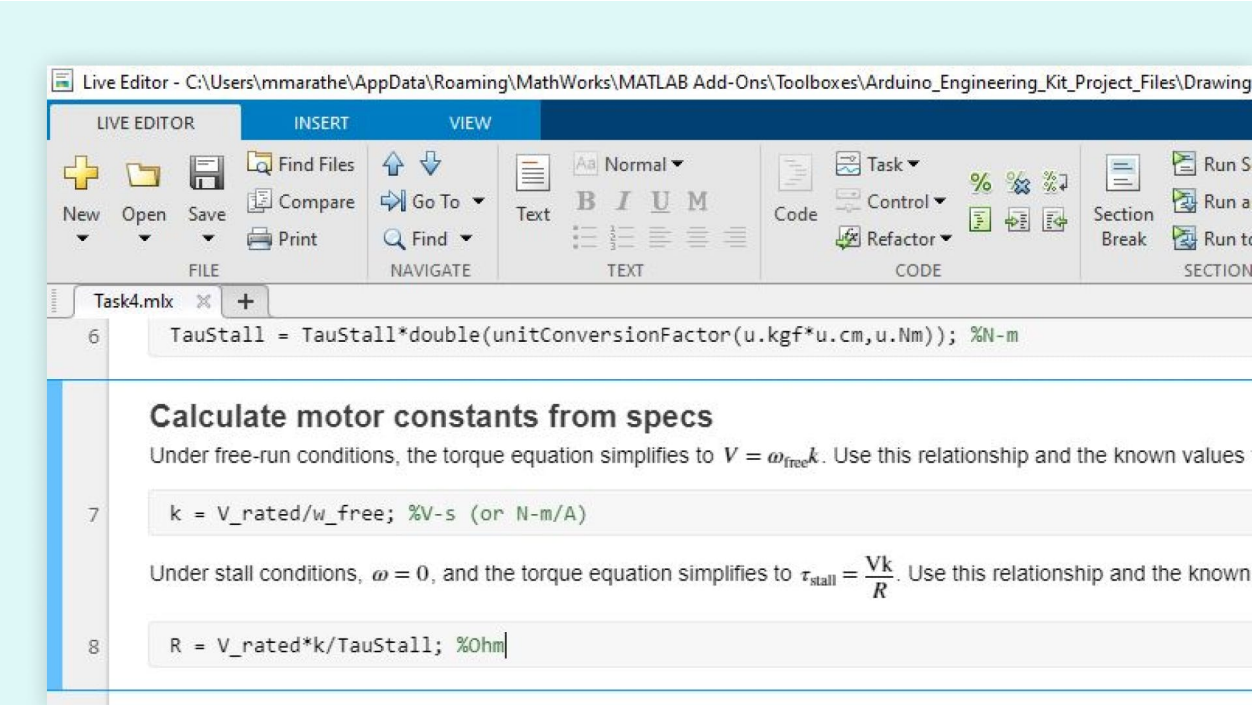
$$\tau_{stall} = \frac{V \cdot k}{R}$$

Calculate Maximum Allowable Load

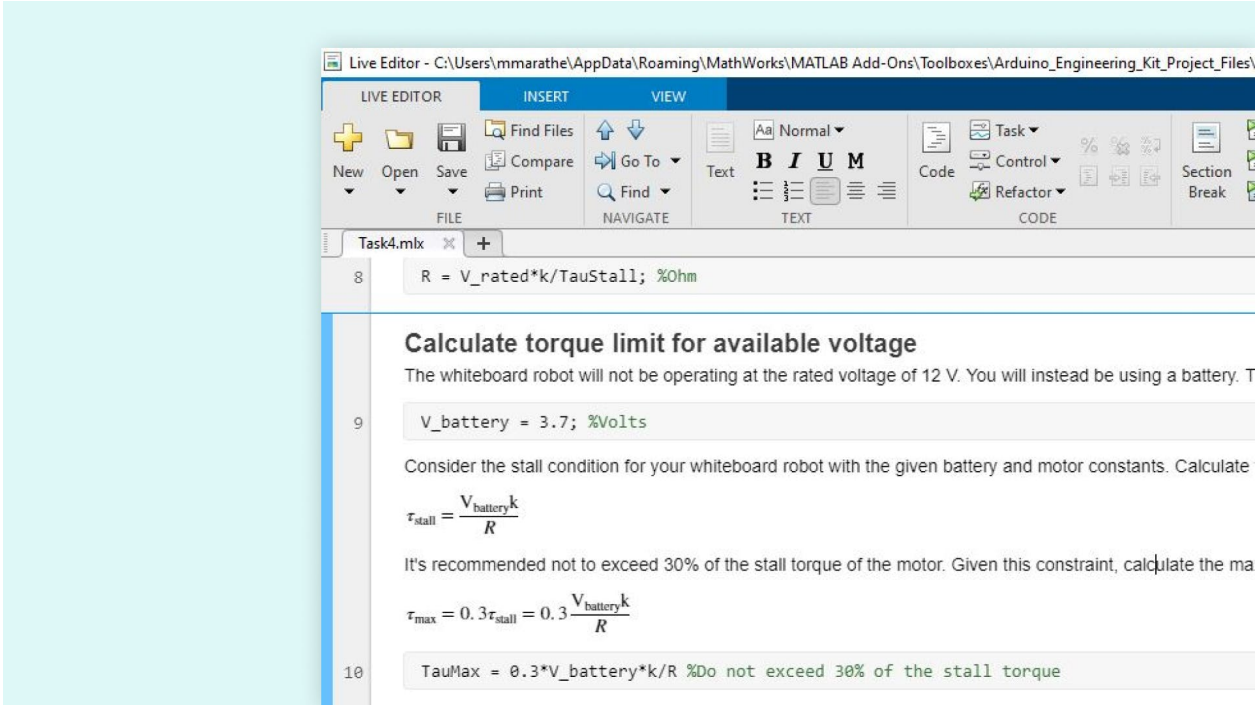
First, check out the specifications provided for the motor. They are available **here**. The motor is rated at 12 V. Make a note of the values for the motor specified voltage, free-run (no-load) speed, and stall torque. Convert them to SI units. The Symbolic Math Toolbox provides a useful function called `unitConversionFactor` that provides you the conversion factor to convert kg-cm to N-m. Open the live script **Task4.mlx**. Under the heading **Define motor specifications**, you'll find all the steps discussed so far. Execute code under this section.



You can utilize the motor's voltage-speed-torque relationship to determine the constants k and R for the motor. First, use the motor equation at free-run conditions (where τ is 0) to calculate the constant k. Next, use the motor equation at stall conditions (where ω is 0) to calculate the constant R. Execute the section of code under the heading **Calculate motor constants from specs** from the live script **Task4.mlx**.



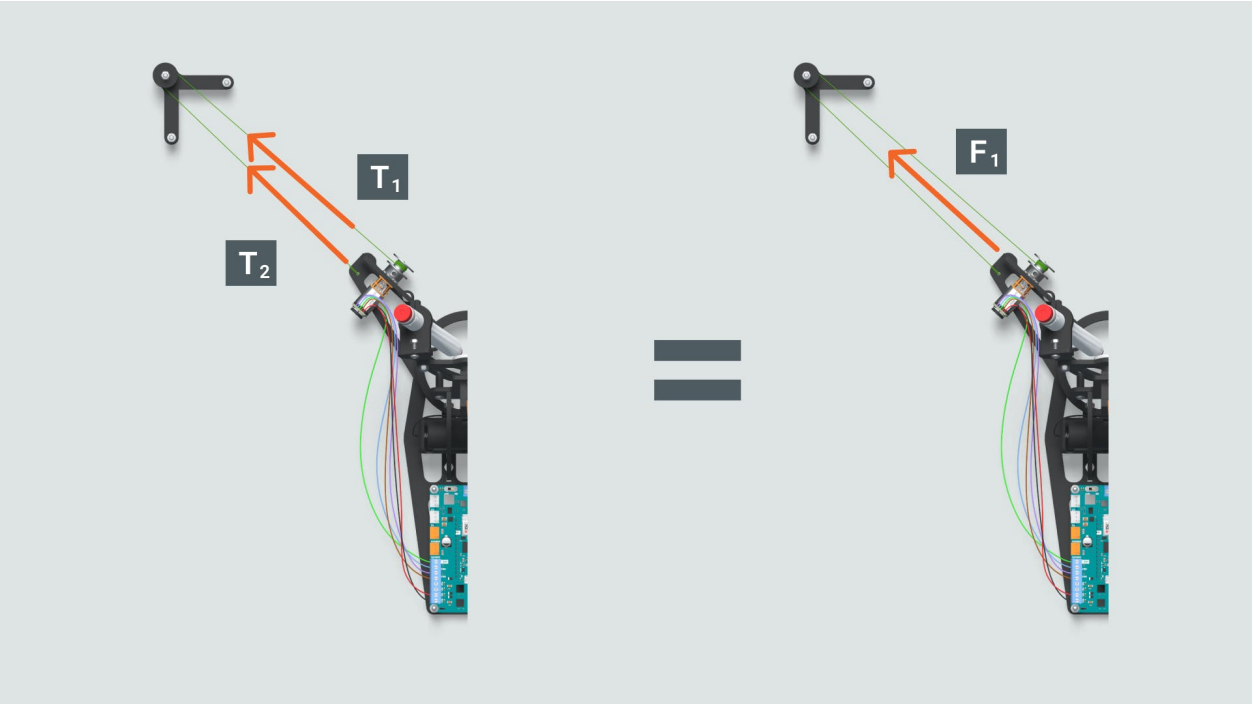
Your robot will not be running the motors at the rated voltage. Instead, in the kit, you have a battery with a rating of 3.7 V. This will put a restriction on the stall torque for your operating conditions. Additionally, it's good practice not to run the motor at more than 30% of the stall torque (more information **here**). Use the motor equation under stall conditions to calculate the stall torque for the given battery rating; then choose 30% of this value as the maximum allowed torque on your robot's motors. Let's call this variable `TauMax`. To compute this value, execute code under the heading **Calculate torque limit for available voltage** from the live script **Task4.mlx**.

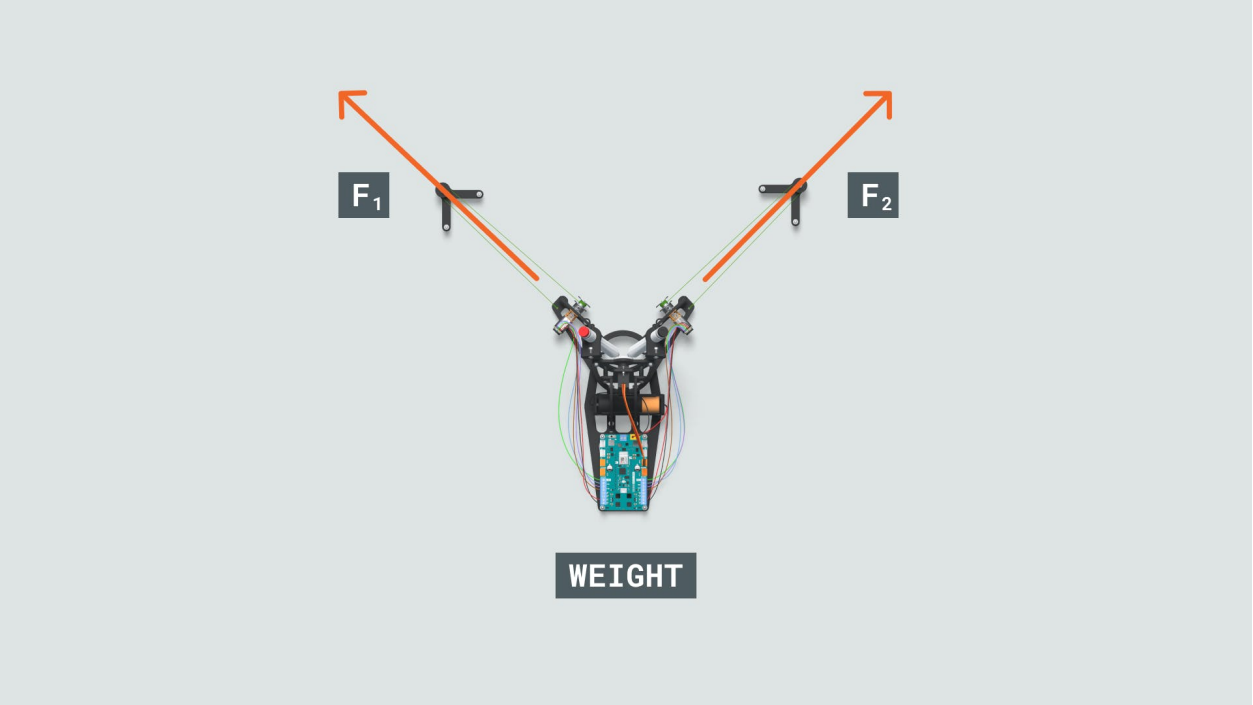


Understand the free Body Diagram

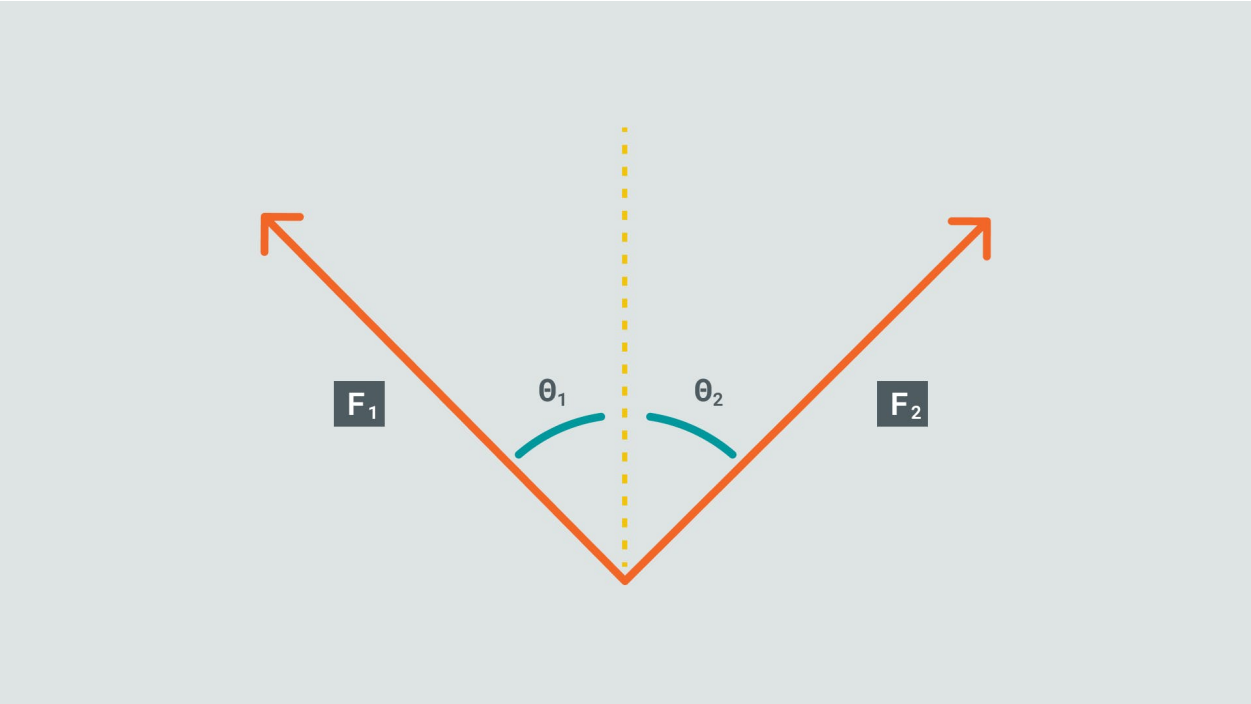
Think about hanging the robot at various parts of the whiteboard. Do you think the motors will have a harder time turning if they are at the top or at the bottom? Is it easier to move up or down the whiteboard? Which motor will have a greater load when the robot is all the way at the left edge of the whiteboard?

In Physics and Engineering, it's often useful to be able to calculate the forces acting on an object. For the whiteboard robot, these forces can tell us how much load is acting on each of the motors. This is achieved by creating what is known as a free-body diagram, which is a diagram showing the various forces acting on a body from different directions. Take a look at the figure described below. On the robot, there are two strings that experience a pull force or tension toward the left pulley. Each one of these will experience the same tension `T1`, so we can treat them as a single force `F1` whose value is twice the amount of the tension force `T1`. Similarly, the tension `T2` in the two strings directed towards the right pulley can be treated as a single force `F2` where `F2 = 2 * T2`. Finally, the weight of the bot acts as a downward force.

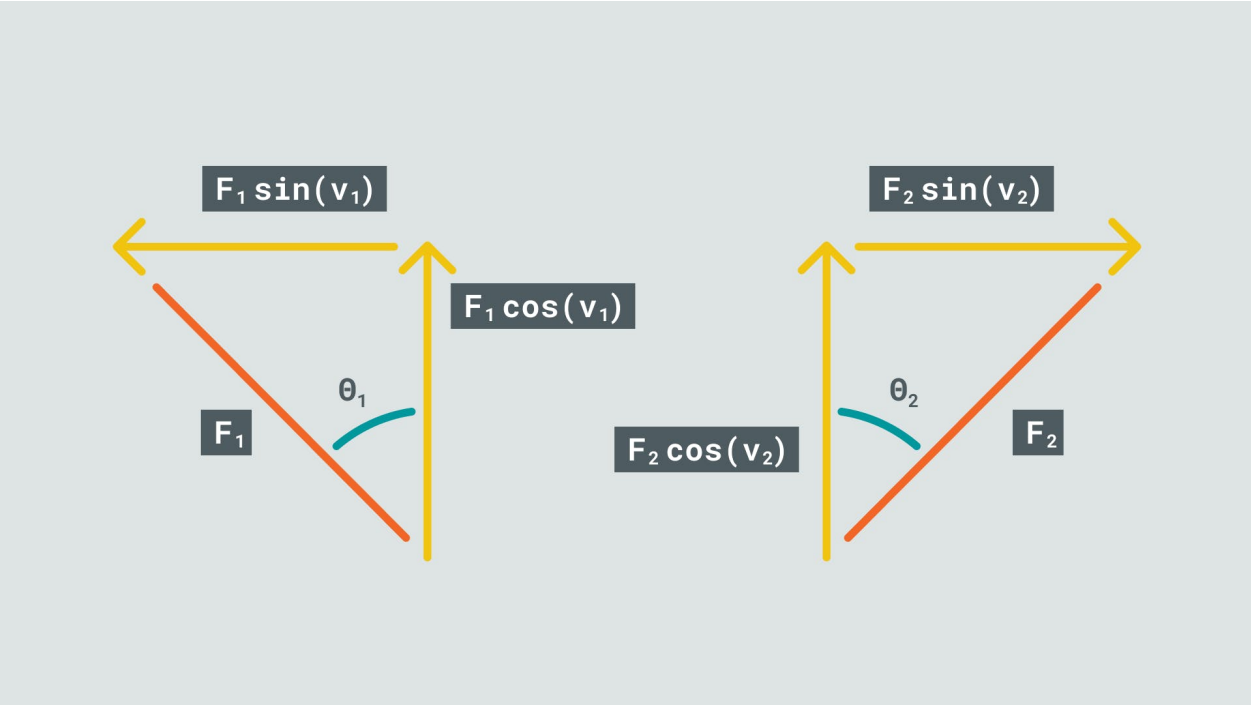




On a free-body diagram, it’s also useful to break down the forces into their x and y components. First, let’s define θ_1 and θ_2 as the angle from the vertical axis for F1 and F2 respectively, as shown in the following diagram. The relationship between F1 , F2 , and the weight of the bot will be explained in the subsequent sections. For now, let's focus on how to express the values of these forces as functions of the vertical angles, based on the robot's location on the whiteboard.



We will use trigonometry to break down each force `F1` and `F2` into its x and y components, as shown below:



$$F_{1,x} = -F_1 \cdot \sin\theta_1$$

$$F_{1,y} = F_1 \cdot \cos\theta_1$$

$$F_{2,x} = F_2 \cdot \sin\theta_2$$

$$F_{2,y} = F_2 \cdot \cos\theta_2$$

With the help of these equations, it's possible to determine the amount of force the motors will be subjected to at different locations on the whiteboard, at steady-state. The next section will cover more details on this.

The Force Balance on a Static Body

Newton's 3rd law of motion tells us how an object behaves with a number of forces acting on it. The sum of the forces acting on the object is equal to the mass of the object times the acceleration of the object.

$$\Sigma\vec{F} = m\vec{a}$$

This equation can be broken into its x and y component:

$$\Sigma\vec{F_x} = m\vec{a_x}$$

$$\Sigma\vec{F_y} = m\vec{a_y}$$

The LHS of the equations represents the sum of the forces in their respective directions. When the robot is not moving, the acceleration in every direction is zero and hence, the RHS of the equations equal zero. For the whiteboard robot, the equations reduce to the following:

$$-F_1 \cdot \sin\theta_1 + F_2 \cdot \sin\theta_2 = 0$$

$$F_1 \cdot \cos\theta_1 + F_2 \cdot \cos\theta_2 - Weight = 0$$

Solving this system of simultaneous equations for the unknown forces `F1` and `F2` gives us the following:

$$F_1 = Weight \cdot \frac{\sin\theta_2}{\sin\theta_1 \cdot \cos\theta_2 + \sin\theta_2 \cdot \cos\theta_1}$$

$$F_2 = Weight \cdot \frac{\sin\theta_1}{\sin\theta_1 \cdot \cos\theta_2 + \sin\theta_2 \cdot \cos\theta_1}$$

So far, we've explored the relationships between the geometry of the robot and the various forces acting on it. However, the motor equations are expressed in terms of torque, and not force. In the next section, we'll bring together all the equations discussed so far and introduce a relationship between torque and force that will

bind everything together. This exercise will allow you to determine how much load the motors are under at every point on the whiteboard.

Derive Equations to Compute Torque From X-Y Position

Given an **x-y** position on the whiteboard, it is possible to compute the torque requirements for the motors. First, use the `x` and `y` positions to compute the angles !θ_1 and !θ_2 formed by the robot's hanging position. Next, compute `F1`, `F2` and `T1`, `T2`. Finally, use `T1`, `T2` to compute τ_1, τ_2.

To compute θ_1 and θ_2 from `x` and `y`, use the trigonometric formula for computing the tangent of an angle:

$$\theta_1 = \tan^{-1}\left(\frac{x}{y}\right)$$

$$\theta_2 = \tan^{-1}\left(\frac{Base-x}{y}\right)$$

To compute `F1`, `F2` from θ_1, θ_2, use the equations derived in the previous section:

$$F_1 = Weight \cdot \frac{\sin\theta_2}{\sin\theta_1 \cdot \cos\theta_2 + \sin\theta_2 \cdot \cos\theta_1}$$

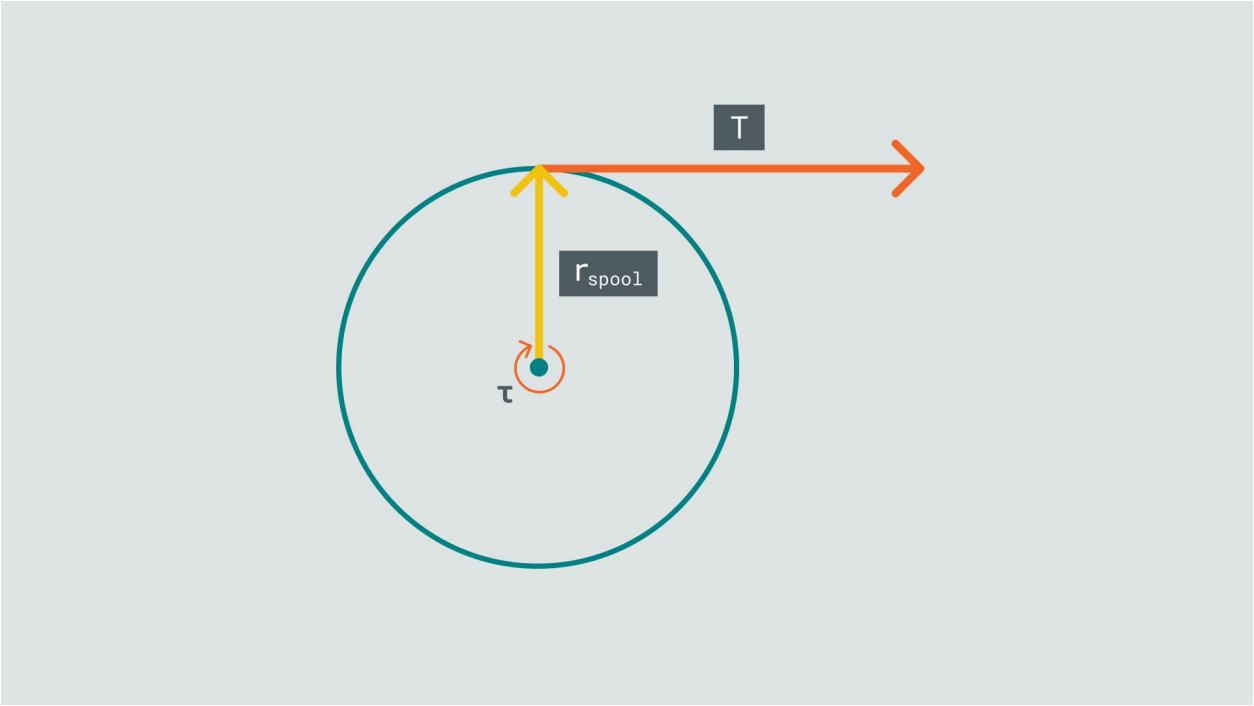
$$F_2 = Weight \cdot \frac{\sin\theta_1}{\sin\theta_1 \cdot \cos\theta_2 + \sin\theta_2 \cdot \cos\theta_1}$$

To compute `T1`, `T2` from `F1`, `F2`, recall that the resultant force pulling the robot in the direction of a pulley equals twice the amount of tension in the string over that pulley. This is because for a given pulley, the string supports the robot at two different points.

$$T_1 = \frac{F_1}{2}$$

$$T_2 = \frac{F_2}{2}$$

Finally, compute the required torque from the tension in the string. Each one of the two strings pulls the motor in a direction perpendicular to the line joining the center of the spool to the point where the string touches the motor, as shown in the following diagram.



The magnitude of the torque is then given as the product of the perpendicular force and the distance between the axis of rotation and the point where the force is applied. In our case, this distance refers to the radius of the spool (r_{spool}), and hence the resulting equations are as follows:

$$\tau_1 = T_1 \cdot r_{spool}$$

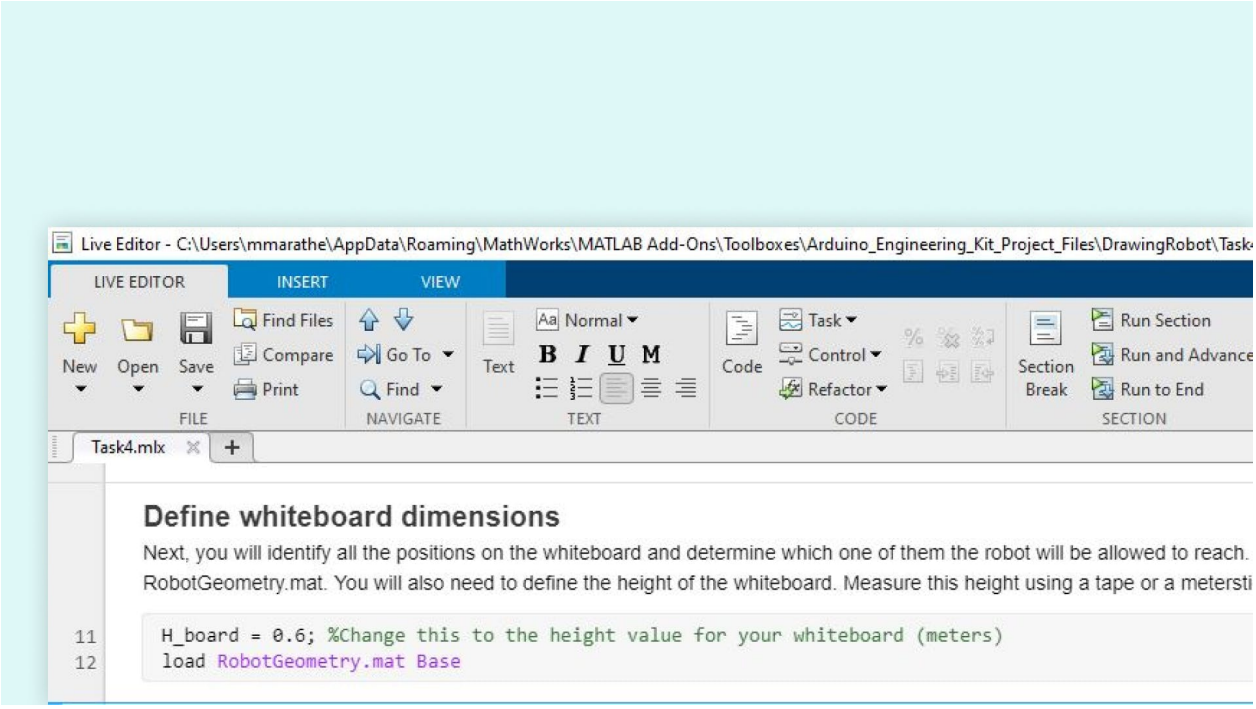
$$\tau_2 = T_2 \cdot r_{spool}$$

With the help of the equations introduced in this section, it is possible to compute angles using locations, forces using angles, tension in the string using forces, and torque using the string tension, which implies that there is a way to relate torque with the location on the whiteboard. As you can imagine, constructing a map of the torques on the robot for the entire whiteboard can be quite tedious but, with the help of these equations and MATLAB, these operations could be automated, as shown in the following sections.

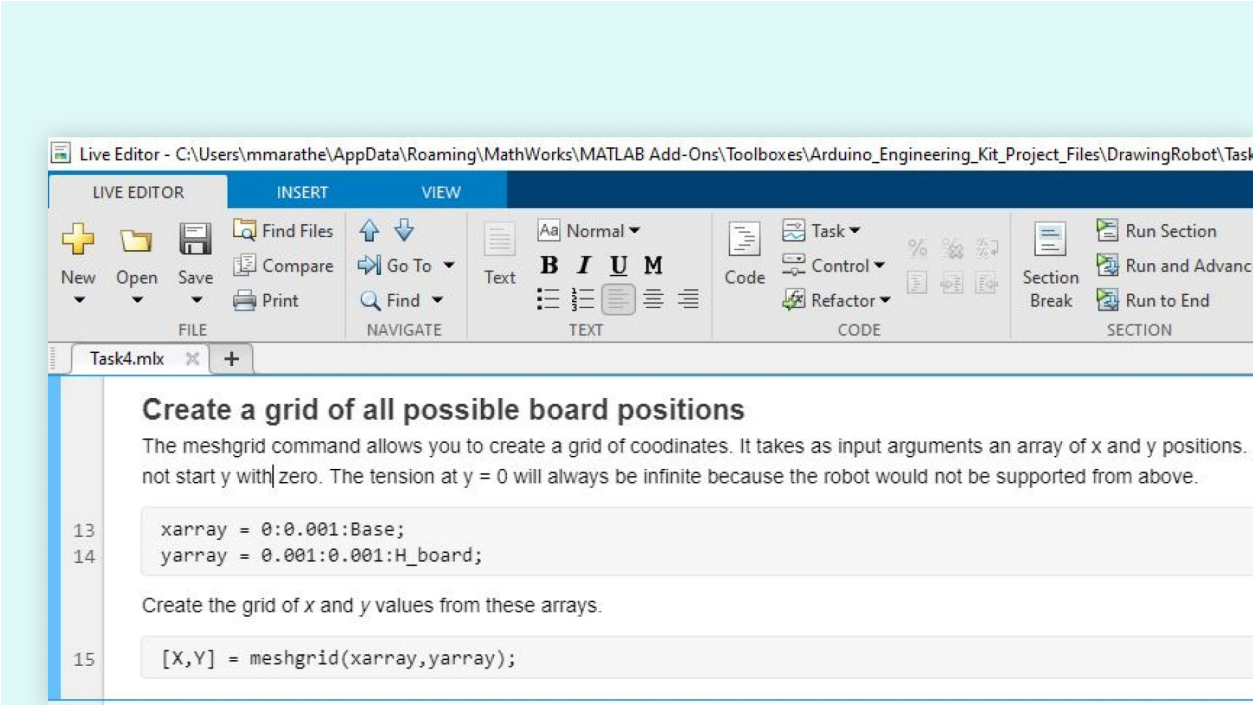
Create a Grid of All Possible Board Positions

To visualize which parts of the whiteboard the robot should be allowed to move to, you'll define a grid that covers the entire whiteboard and calculate the torque everywhere on that grid. The first step in creating this grid is to define the dimensions of the whiteboard.

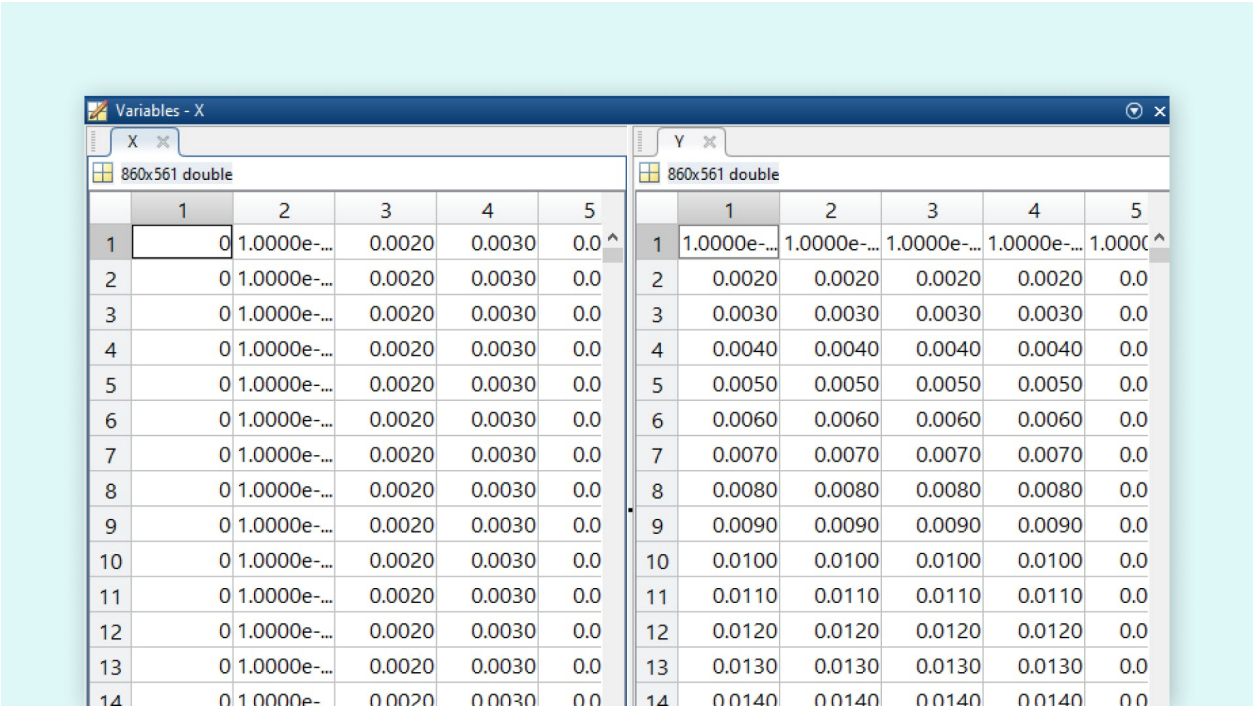
Using a meter-stick or tape, measure the height of the whiteboard, from the pulleys down to the bottom of the whiteboard. Navigate to the section under the heading **Define whiteboard dimensions** of the live-script **Task4.mlx**. Change the value of `H_board` according to the measured height of your whiteboard (defined in meters).



You previously measured the distance between the two pulleys. This is defined in the `Base` variable which you should have saved in a MAT-file. Execute code under the section **Define whiteboard dimensions** to define the whiteboard height (`H_board`) and load the measured base (`Base`). A data grid of coordinate positions can be created in MATLAB using the `meshgrid` function. An easy way to call this function is by defining arrays of x and y values to be contained in the grid. In the next section of code, you will define these arrays over the entire range of x and y values with an interval of 1 mm. Execute the section of code under the heading **Create a grid of all possible board positions** in the live-script **Task4.mlx**.



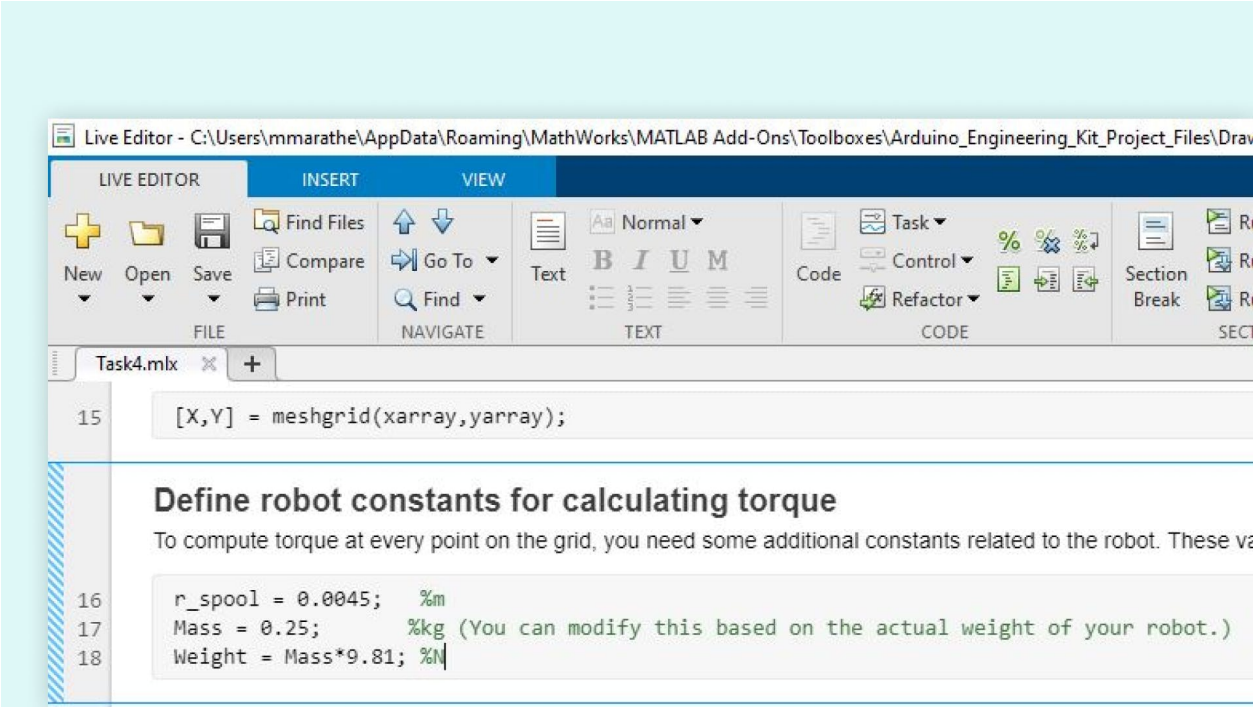
Take a look at the variables `X` and `Y` in the **Variable Editor** by double-clicking on them in the **Workspace**. Note that they are of the same size. Additionally, the rows of `X` are all the same, and the columns of `Y` are all the same. Therefore, each pair of positions `X(i,j)` , `Y(i,j)` defines a unique location on the whiteboard.



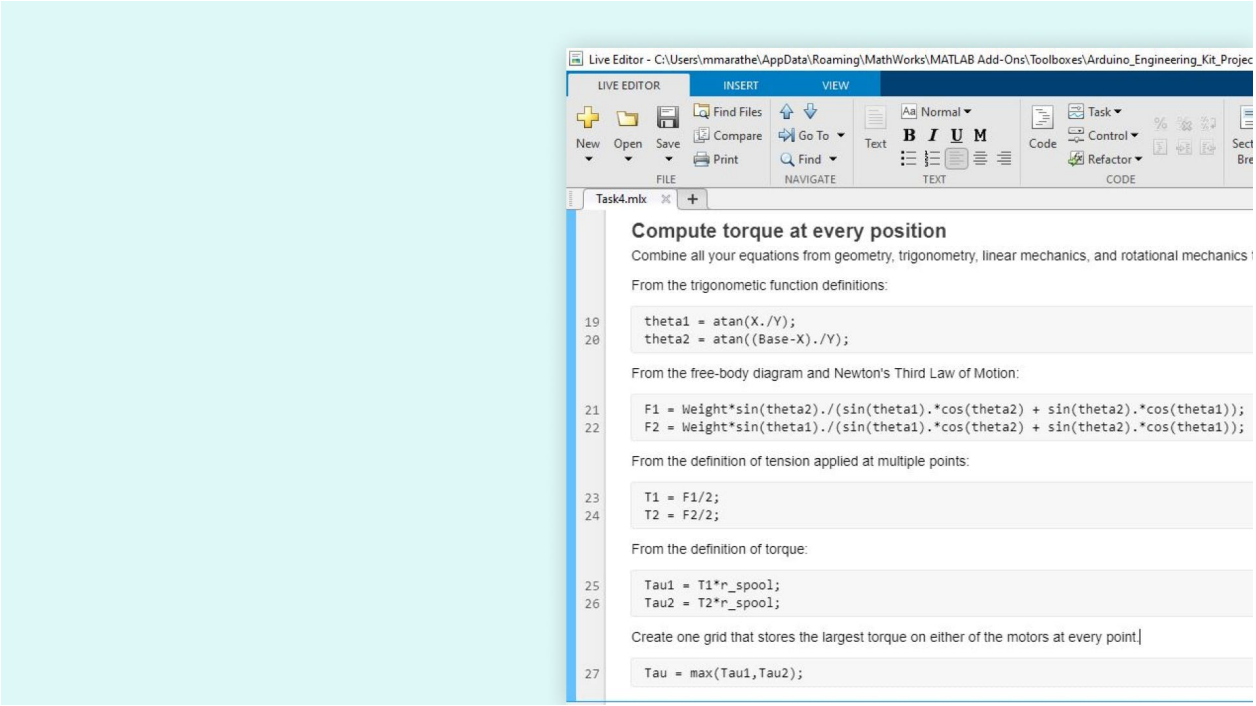
We can now use these grid points to evaluate the series of equations obtained earlier and visualize the results on a plot.

Compute Torque at Every Position and Create a Surface Plot

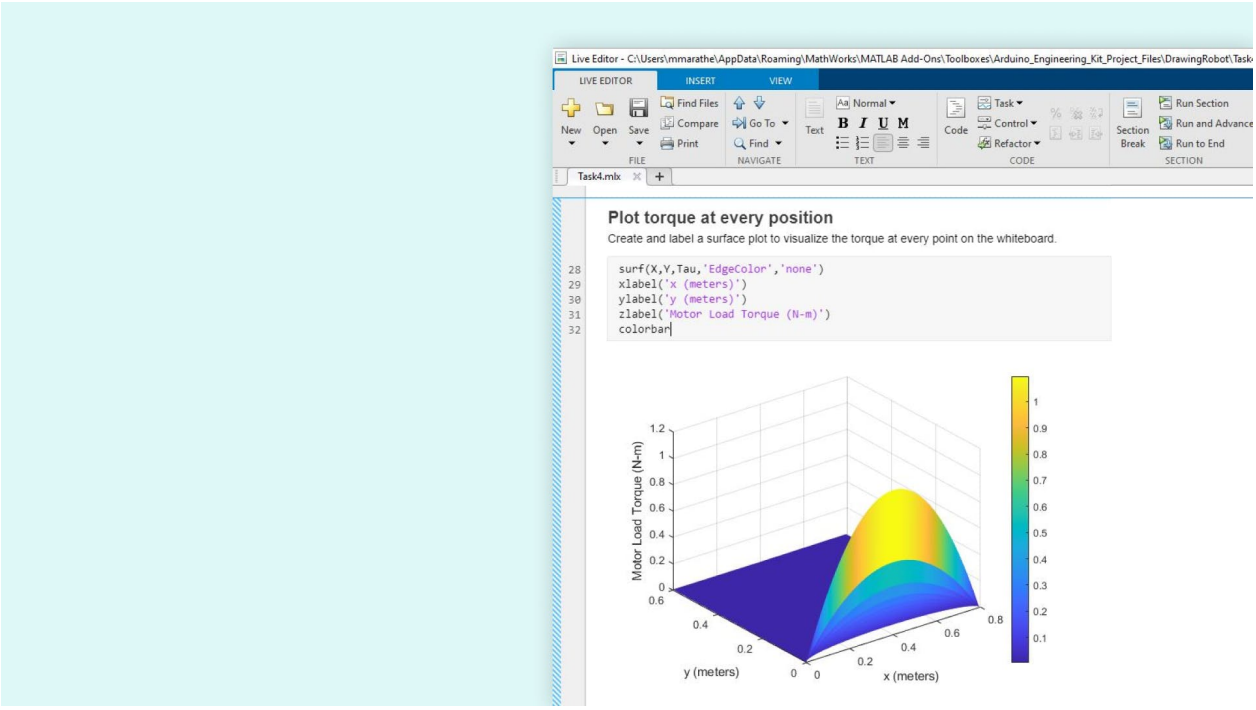
In addition to the variables defined previously, we also need to know a few additional values before we can compute the motor torques at every point on the whiteboard. To calculate the tension in each string, we need to know the robot's weight. To convert the tension into torque, we need to know the radius of the spool (r_{spool}). These are both defined for the given robot kit. To define these in the MATLAB code, execute the section of code under the heading **Define robot constants for calculating torque**.



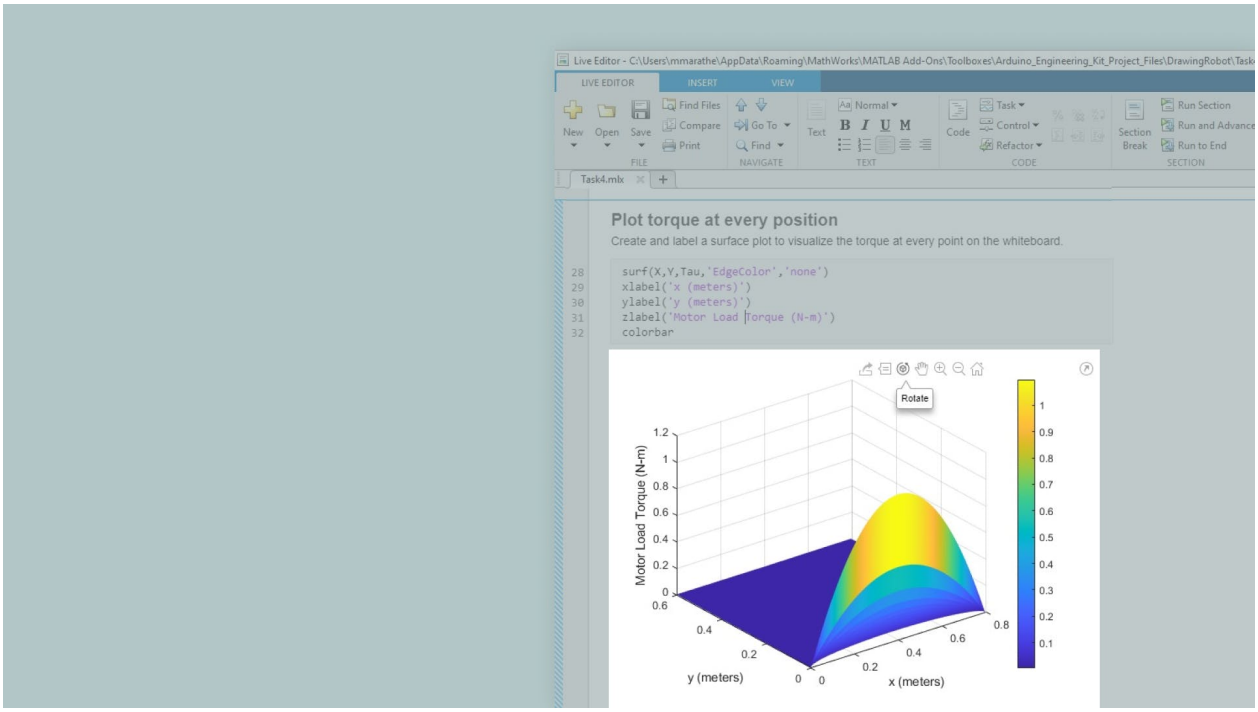
Next, use MATLAB to apply the derived equations for converting the x-y positions to torque. First, use the trigonometric equations to calculate the angles θ_1 and θ_2. Next, use the force-balance equations to calculate the force in each direction and the tension in each string. Then compute the torque load on each motor at every grid point. Finally, define a single variable `Tau` that contains the larger of the two torque values at every position. You must never move the robot to a location where this value is greater than `TauMax`, the maximum allowable torque derived in a previous section. Execute the section of code under the heading **Compute torque at every position** in the live-script **Task4.mlx**.

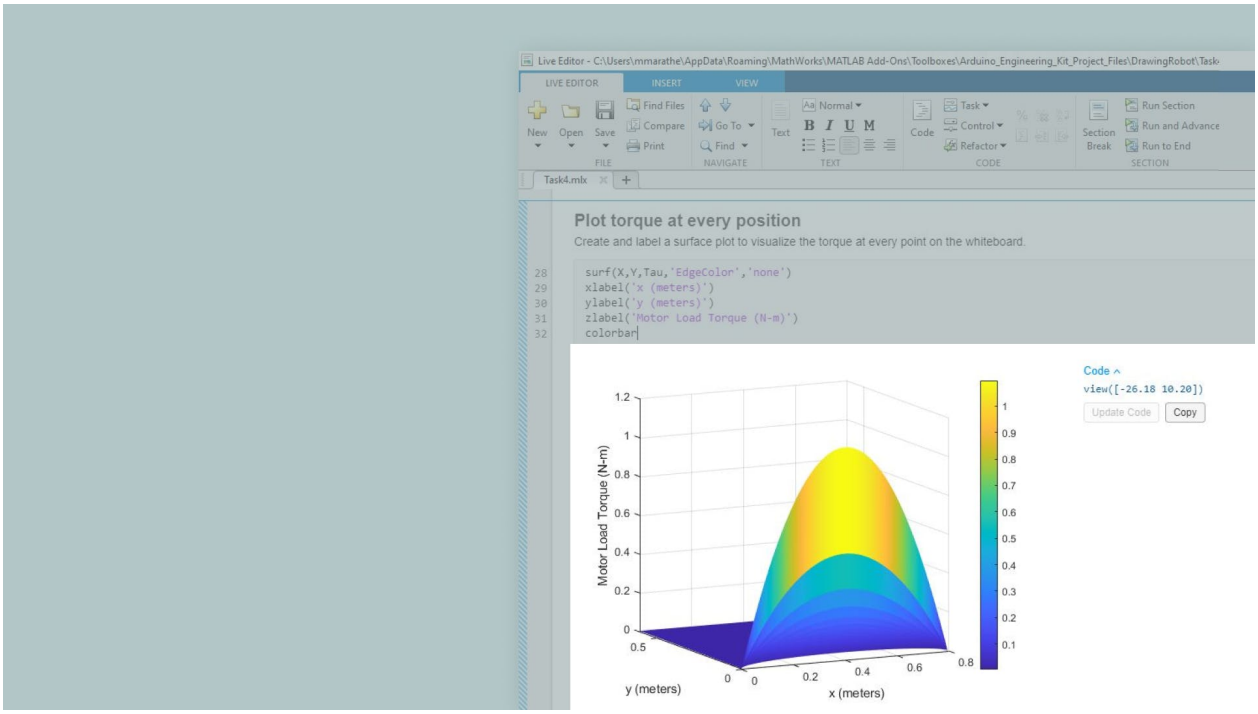


Now that the motor load `Tau` is defined at every position on the whiteboard, you can visualize it in MATLAB. Create a simple surface plot using the `surf` function in MATLAB. To do so, execute the section of code in the live-script **Task4.mlx** under the heading **Plot torque at every position**.



Explore this plot. Click the icon marked **Rotate**. Then click and drag the plot to examine the surface from all sides and get a better sense of what the torque is like at different positions on the whiteboard.

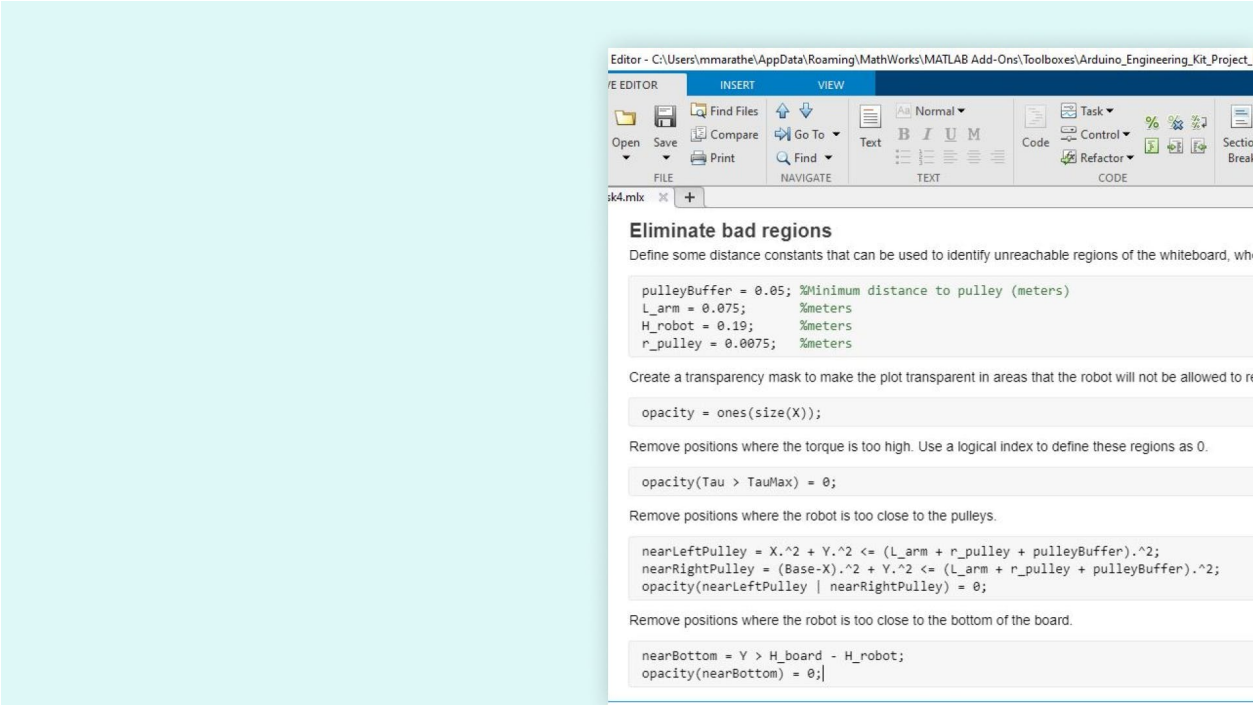




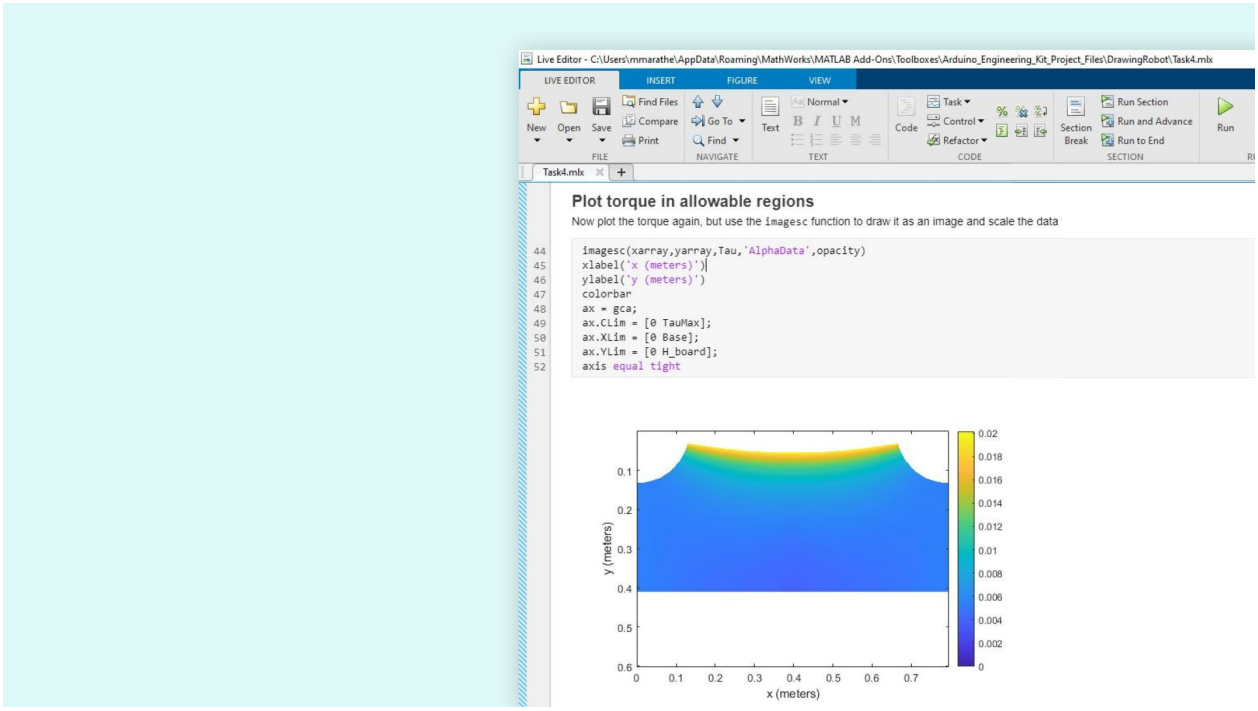
Eliminate Bad Regions and Plot the Final Torque Map

In this exercise, you will create a plot that removes the regions the robot should not move to. These include two types of regions. First, remove the regions where the torque is too high. Then remove the regions where the robot cannot go because it is too close to the pulleys or bottom of the whiteboard.

To "remove" these regions from our plot, we'll define a transparency mask for the plot. The plot will be visible at regions the robot can reach and transparent at regions the robot cannot reach. In MATLAB, the data that defines these regions is called AlphaData. Execute the section of code under the heading Eliminate bad regions in the live-script **Task4.mlx** to create an opacity variable and remove the unreachable areas.

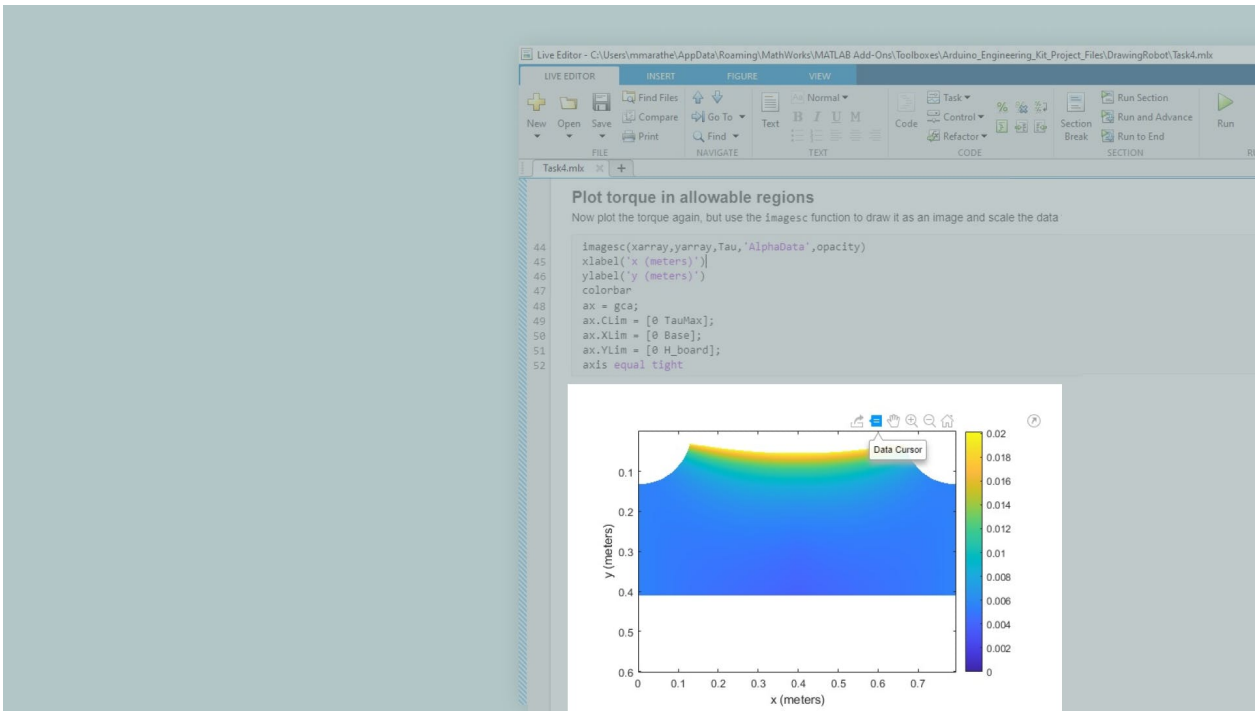


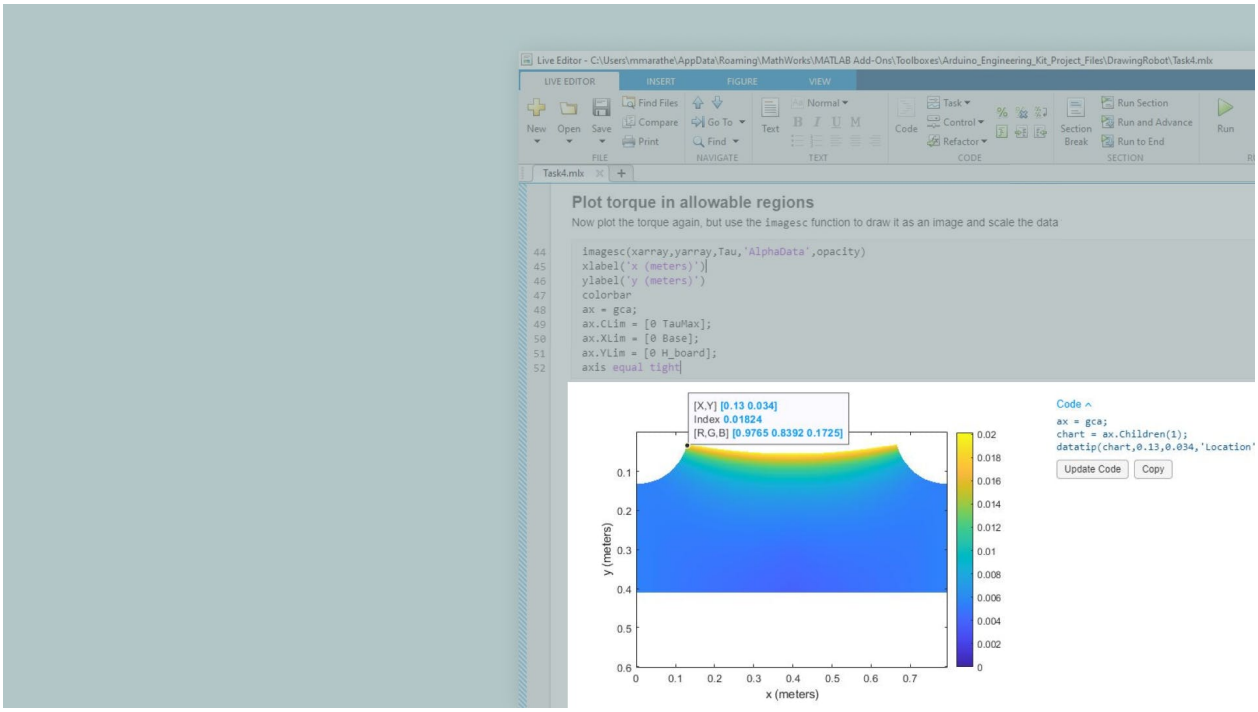
Now, you can use the `opacity` variable in a plot. Create an image plot to view the `Tau` data in 2D and include the `opacity` variable as the plot's **AlphaData**. Additionally, you can adjust the color limits so that they scale to the maximum value of the visible portion rather than the maximum value over all of `Tau`. This 2D plot will show you which parts of the whiteboard the robot can and cannot reach. The transparent regions represent those where the robot can't reach. Create this plot by executing the code under the section **Plot torque in allowable regions** in the live-script **Task4.mlx**.



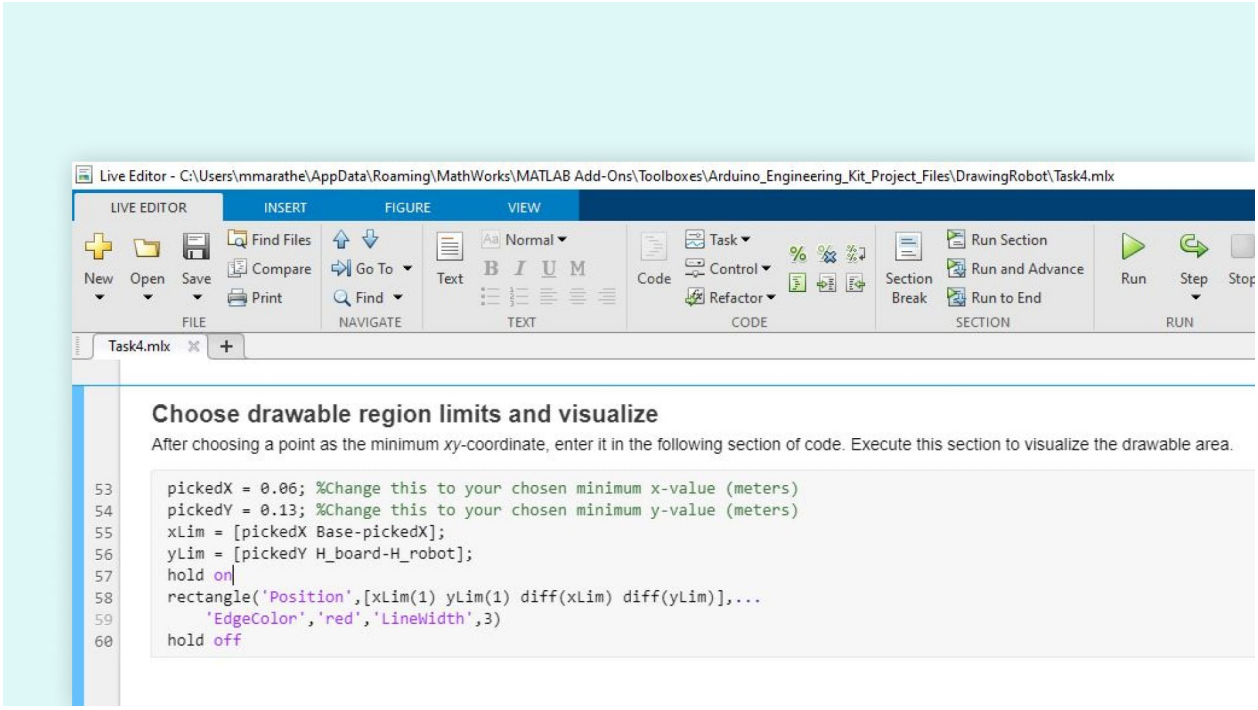
Define and Save Drawing Limits for your Whiteboard

Next, you'll define the part of the whiteboard that you will allow the robot to draw on. You can choose where you want the minimum x and y values to be for your drawing region. These are likely to fall somewhere along the curve that defines how close the robot can get to the left pulley. Use the **Data Tips** tool in the live script to select a point and note its x-y coordinate.



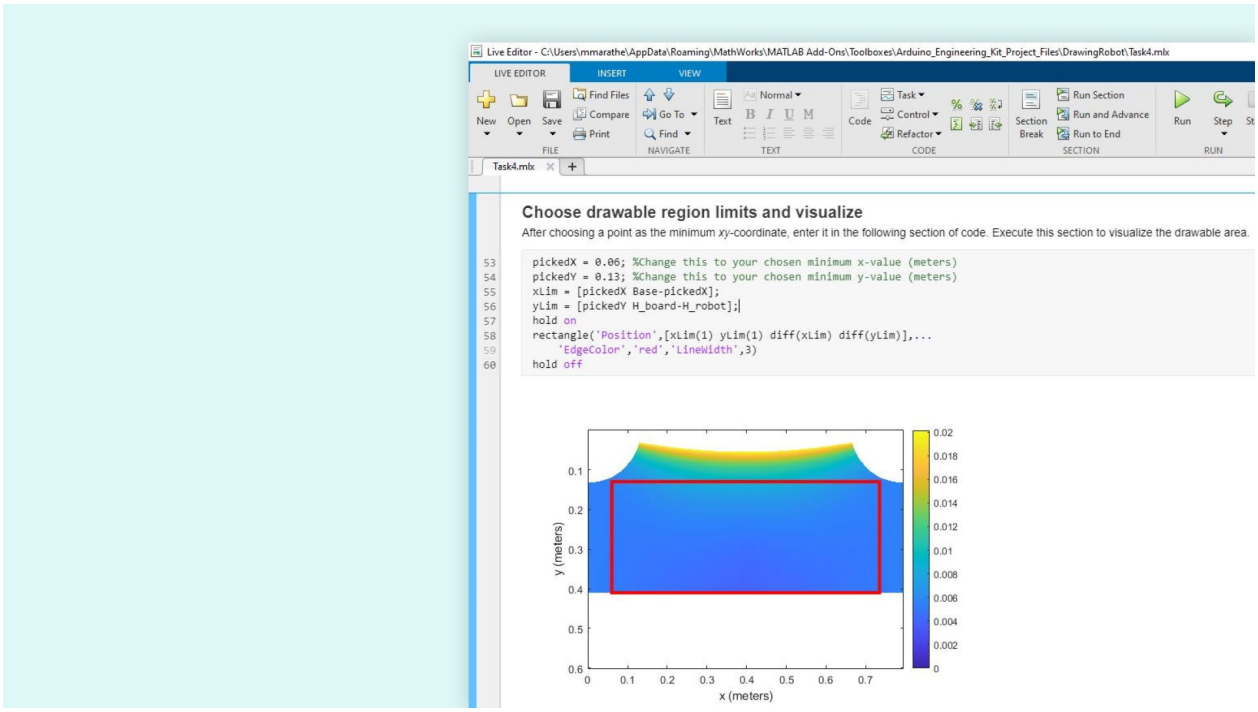


Record the `x` and `y` values from the plot in the code. Navigate to the next section **Choose drawable region limits and visualize** in the live-script **Task4.mlx**. Replace the values of the variables `pickedX` and `pickedY` with the values you selected in the plot.

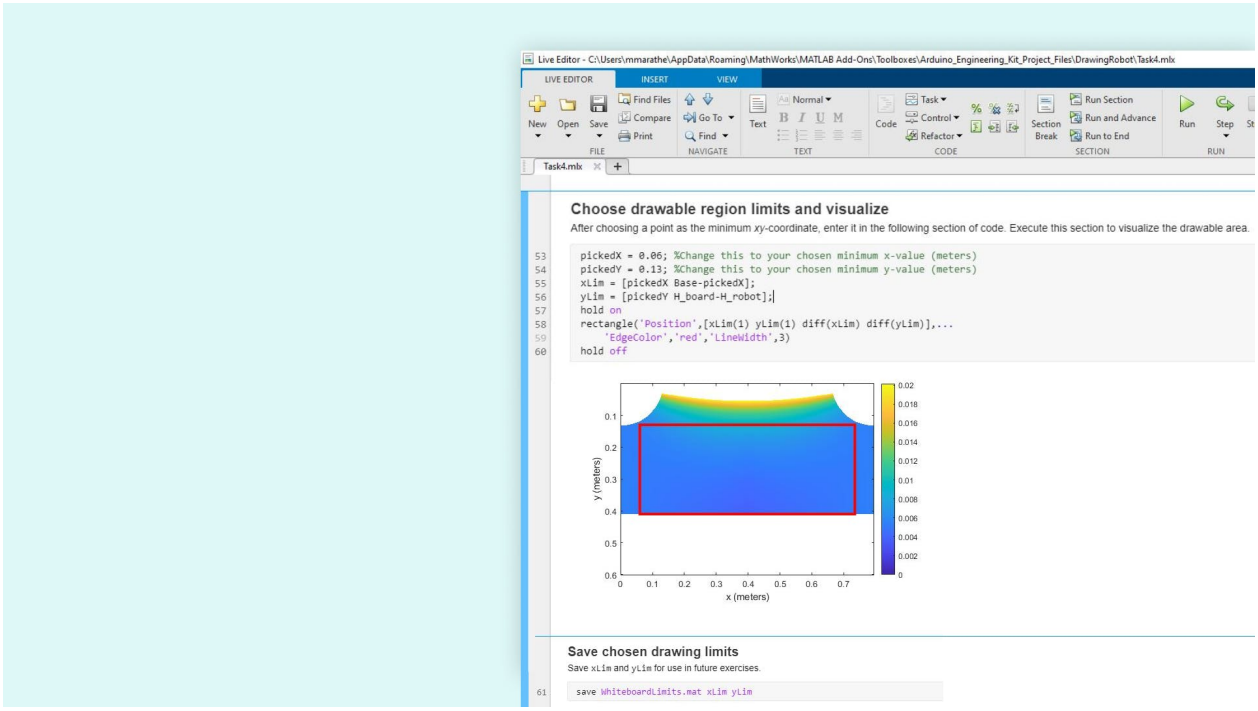


You can now visualize the drawable area of the whiteboard by overlaying a rectangle on the plot you created previously. To retain the previous plot and add data on top of it, use the `hold on` command. Use the `rectangle` function to

draw the rectangle. To perform these steps, and verify that your drawable region looks correct, execute the code under the heading **Choose drawable region limits and visualize** in the live-script **Task4.mlx**.



If you're happy with the size and shape of the rectangular region selected, then you should store the values of `xLim` and `yLim` in a MAT-file for later use. Execute the section of code under the heading **Save chosen drawing limits** in the live-script **Task4.mlx** so that you can use these values in future exercises.



Files

- Task4.mlx

Learn By Doing

We created the variable `Tau` to store the greater of the two motor torques at every position on the whiteboard. Create a new variable containing `Tau2 - Tau1` that describes the difference in load between the right and the left motors at every point on the whiteboard. Plot this just as you plotted `Tau`. Identify points where the load is greater on the right motor than the left motor and vice versa.

Use the app `SimplePlotterApp` to drive the robot around the whiteboard as you did in an earlier exercise. Confirm what you've seen in the plots in this exercise. For a given voltage, the motors will run slower if there is greater load. Check how fast the motors run at the top of the whiteboard versus the bottom. Check how fast each motor runs at the left of the whiteboard versus the right. Does this match the plots you've created?

The size and dimensions of the drawable whiteboard region can be defined by math equations. Let's say you wanted a square region where the width and height were the same. Use the defining equations to calculate the `pickedX` and `pickedY` values that would give you a square drawable region. Enter these values in the live script Task4.mlx, and then run the script again to check the accuracy of your calculations.

4.5 Draw Preprocessed Images

Let's draw an image of something real. In this lesson, you'll load some data that stores the line traces from a real image. You'll learn how to interpret this data and build an algorithm that instructs the robot to draw each line segment and then raise the marker and move to the next line segment. You'll learn how to scale the size of the line drawing so that it fills up the drawable space on your whiteboard. Finally, you'll create one MATLAB function that can perform all these steps.

In this exercise, you will learn to:

- Plot image data in MATLAB
- Loop through and manipulate data in a cell array
- Convert data from pixels to physical distances
- Design an algorithm to draw an image in multiple segments using a loop

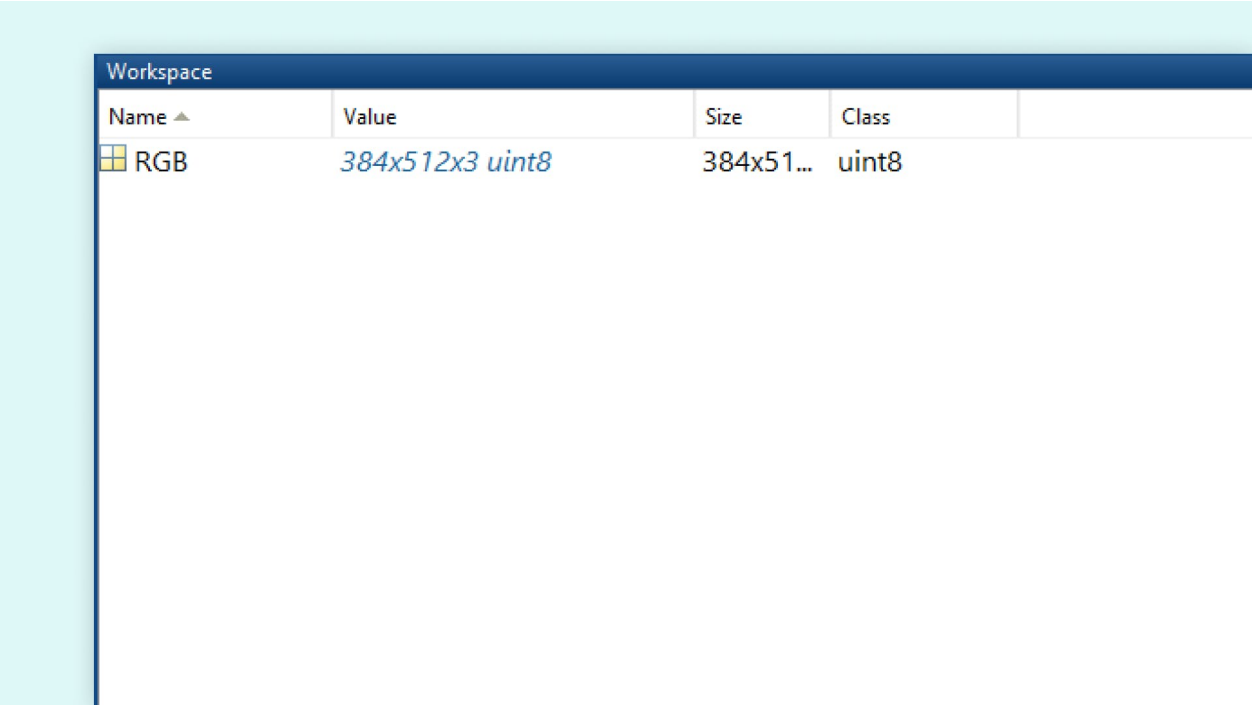
Understanding How to Read Images in MATLAB

MATLAB can work with images in common image file formats. The general workflow is to first read the image and store it in a MATLAB variable. You can then apply functions to modify the image or visualize it, and then write the image to a new file.

To read an image into MATLAB, use the `imread` function. An image that ships with MATLAB is the peppers.png file. Read this image by entering the following line of code at the MATLAB command line.

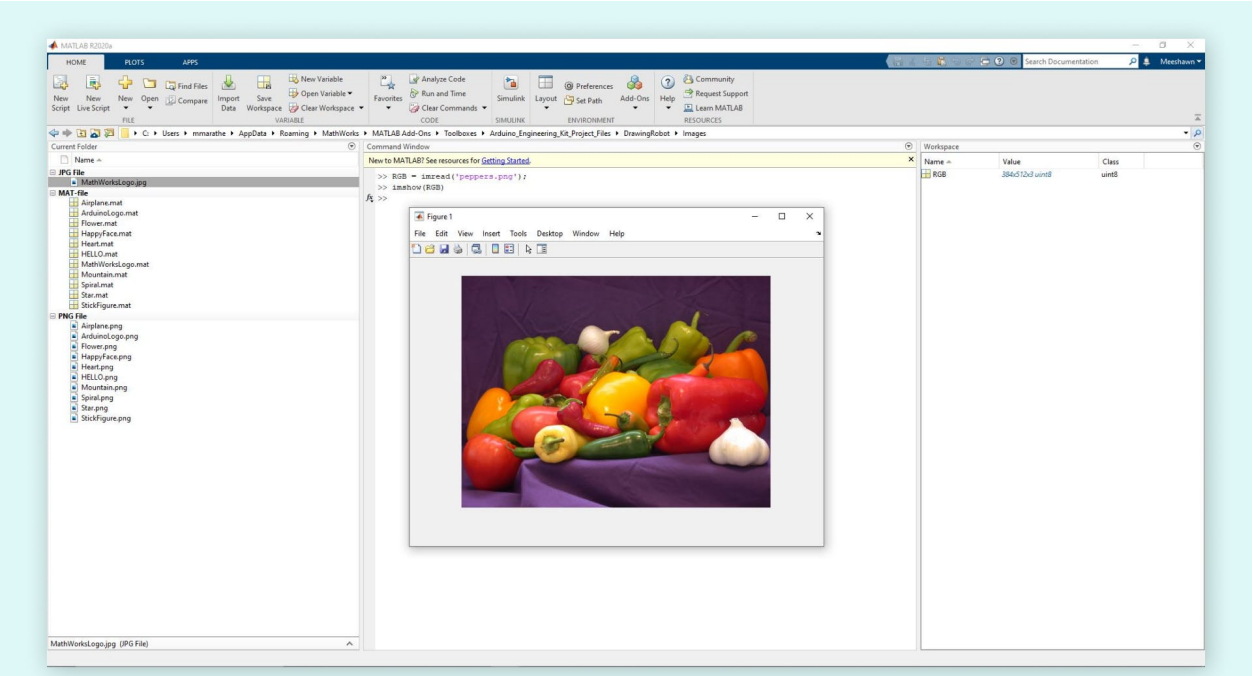
```
>> RGB = imread('peppers.png');
```

Examine the `RGB` variable in your **Workspace**. Note that the variable is three-dimensional and of type **uint8**. Each one of the components (R, G, B) representing one of the basic color components gets 8 bits to store its value in digital form. This means that you can work with images of 24-bit color depth (8 bits or 1 byte for each one of them).



You can view the image using the `imshow` function. Execute the following statement at the MATLAB command prompt.

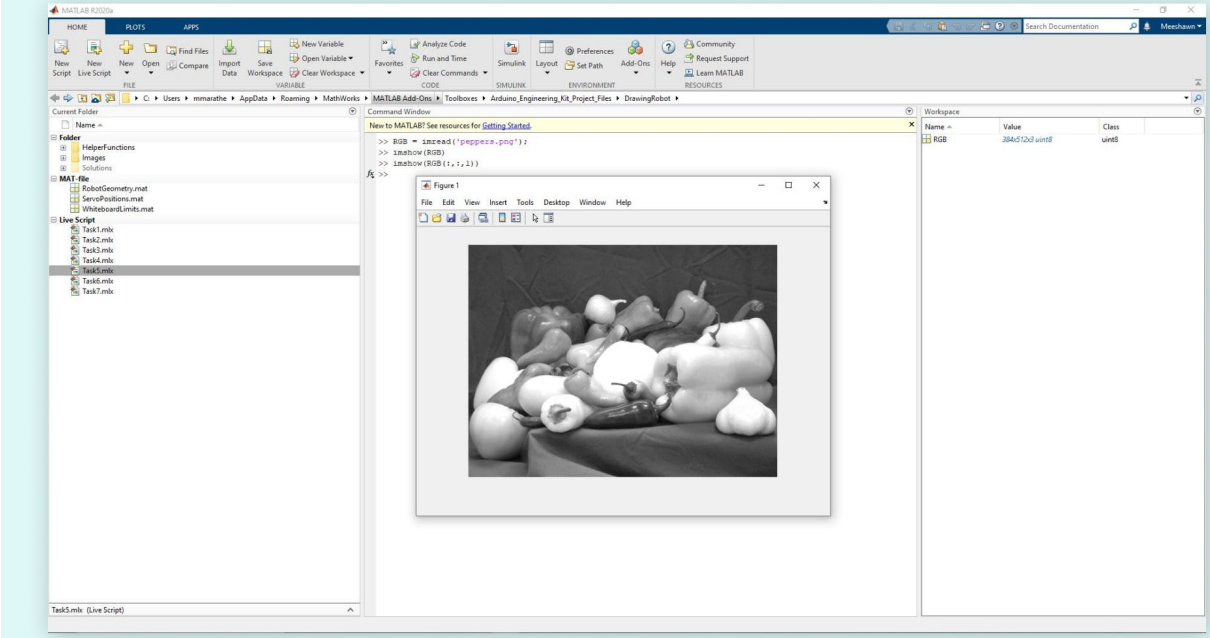
```
>> imshow(RGB)
```



The three components of the variable `RGB` represent the red (R), green (G), and blue (B) color information of the image respectively. You can visualize this as three grids of numbers stacked on top of each other, one for each color plane. To get a

sense of what this means, you can visualize one plane of the image at a time. To visualize a representation of the red plane data, use the following command to view the regions of the image that are the most red:

```
>> imshow(RGB(:,:,1))
```



Notice that how representing a single color plane of an image results in a grayscale image. This might seem confusing. Additionally, you can view an image in MATLAB without even loading it into a variable by calling the `imshow` function on the file itself. Execute the following command to implement this with the **peppers.png** image.

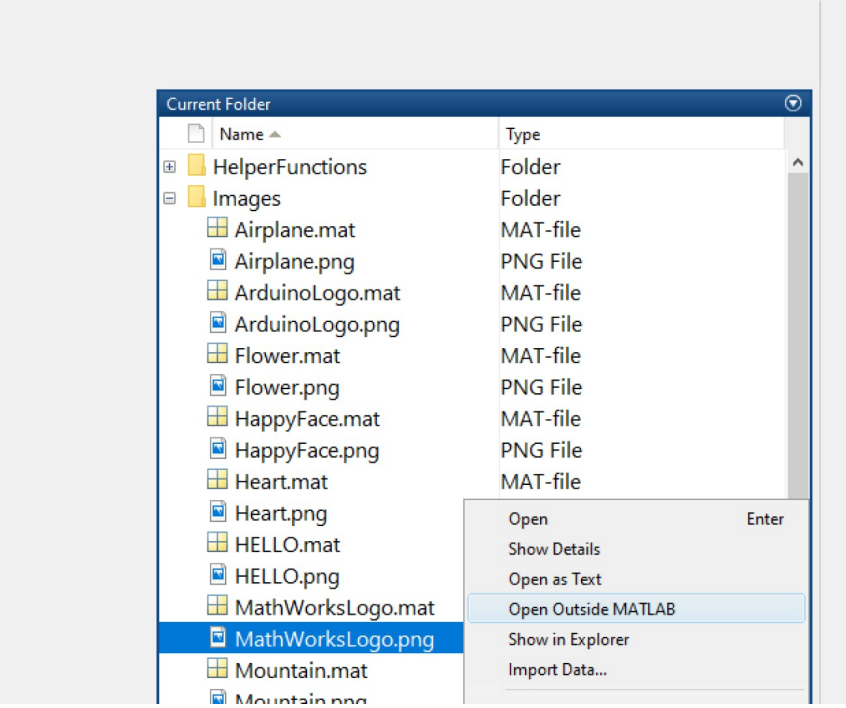
```
>> imshow('peppers.png')
```

The result is the same as the one obtained when we displayed the full color matrix.

Plot an Image of a Drawing to Replicate

The robot cannot draw full RGB planes and neither grayscale images. The current design needs vector graphics as input. Therefore, any image that you want to draw will need to go through some image processing initially to transform into an acceptable format.

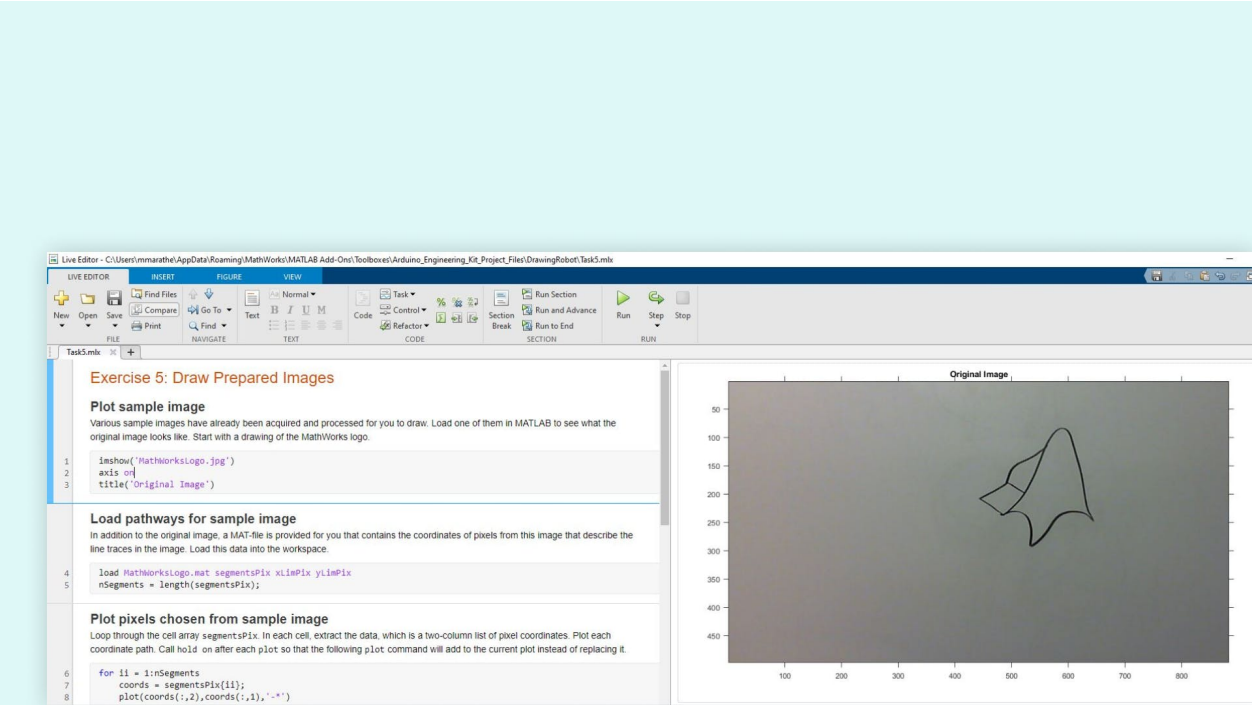
A few sample line drawings have been processed into a drawable form and provided to you. (In a later exercise, you'll learn how to generate these yourself). The **Images** folder, under the **DrawingRobot** folder, contains both the original images and their corresponding MAT-files containing the path to draw. Locate the file **MathWorksLogo.jpg** under this folder. Right click on this image, which is listed in the **Current Folder**, and select **Open Outside MATLAB** to see what it looks like in your system's image viewer. Note that different operating systems have different image viewers.



The next step is to check whether the processed file looks similar to the original image. To verify this, open the live script **Task5.mlx**:

```
>> edit Task5
```

Execute the code under the section **Plot sample image** to see this image plotted in MATLAB.



As you can see here, the outcome of the plot is a vectorized version of the original image. An image of this nature can easily be translated into motor movements. In this exercise, you will go through the required steps to arrive at these images from a given image. But first, we need to build the understanding of some basic tools within MATLAB that will let you perform the needed operations.

Understand Cell Arrays in MATLAB

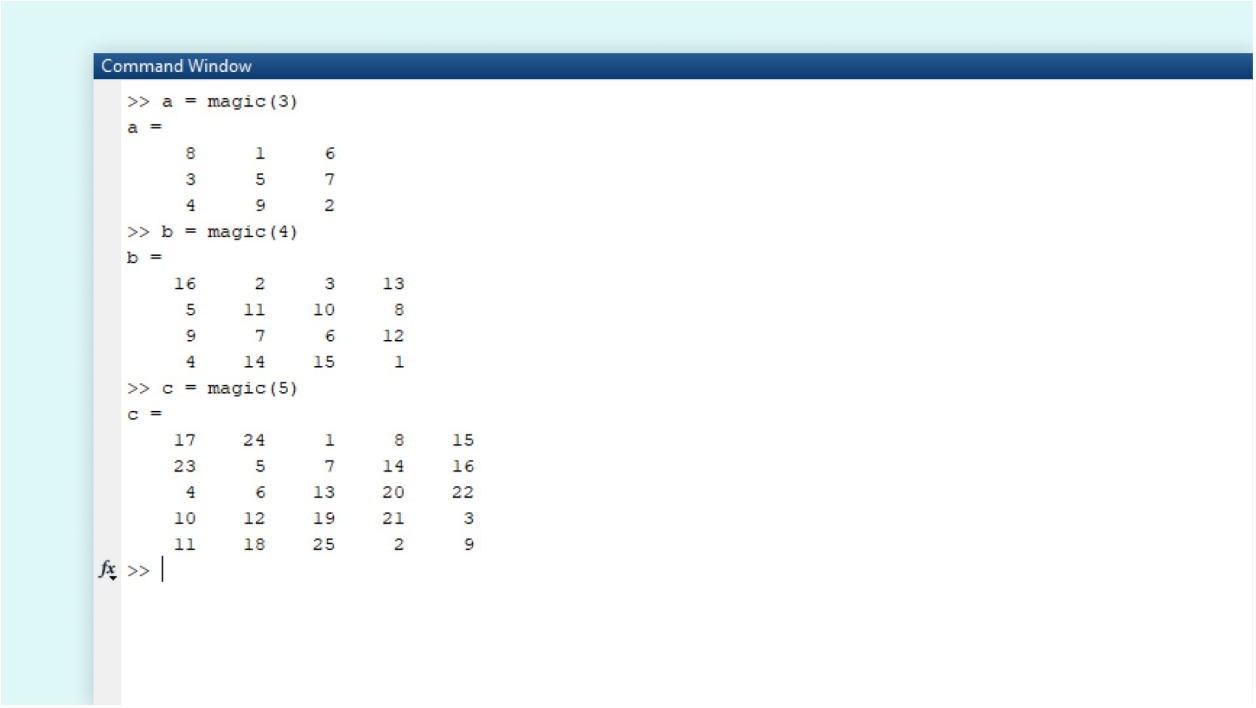
So far you have been working with different types of variables in MATLAB. You have created objects to represent the Arduino device and its components. You have created numeric scalars and vectors to store numeric values. You have also looked at 3-dimensional **uint8** data that represents images.

Sometimes it is useful to combine multiple pieces of data into a single variable. It can be useful for grouping data of different sizes or types. To do this, you can use a data container in MATLAB called a `cell array`. There are many different ways in which cell arrays can be useful, but for the purposes of this exercise, we'll focus on using them to store numeric arrays of different sizes.

To see how cell arrays work, first create some numeric arrays of different sizes. Execute the following code at the MATLAB Command Prompt to create three different magic squares.

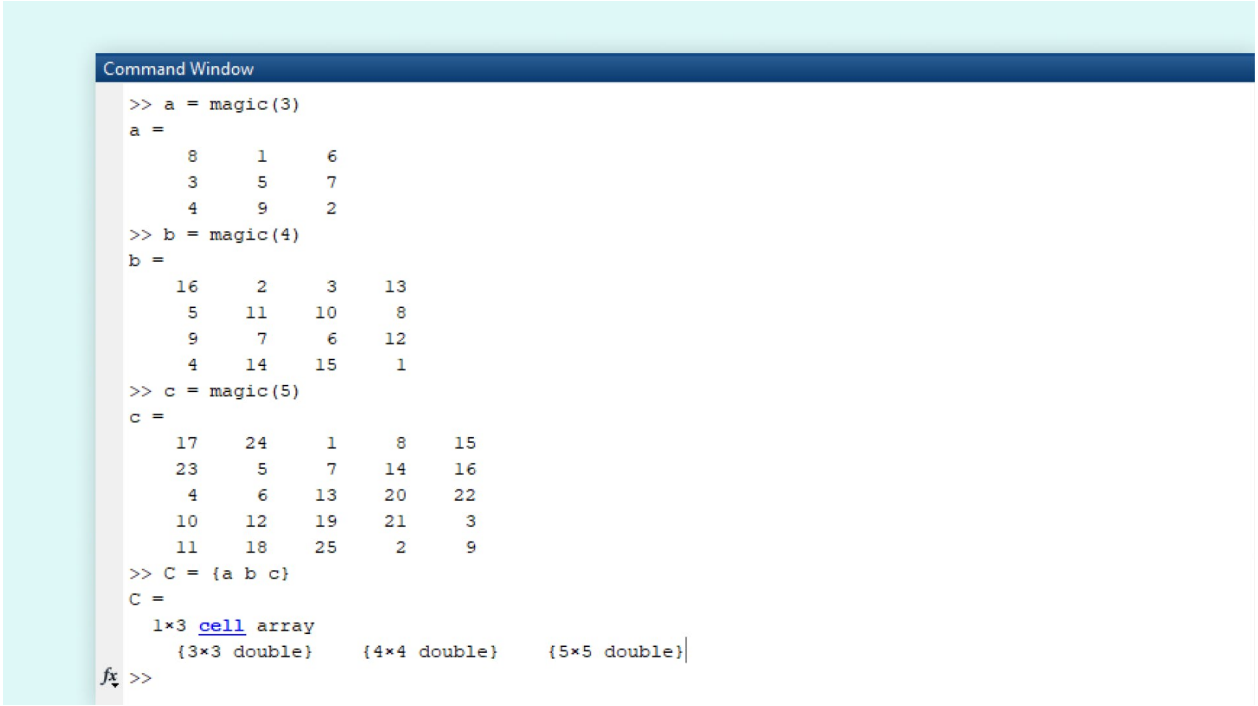
```
>> a = magic(3)
>> b = magic(4)
```

```
>> c = magic(5)
```



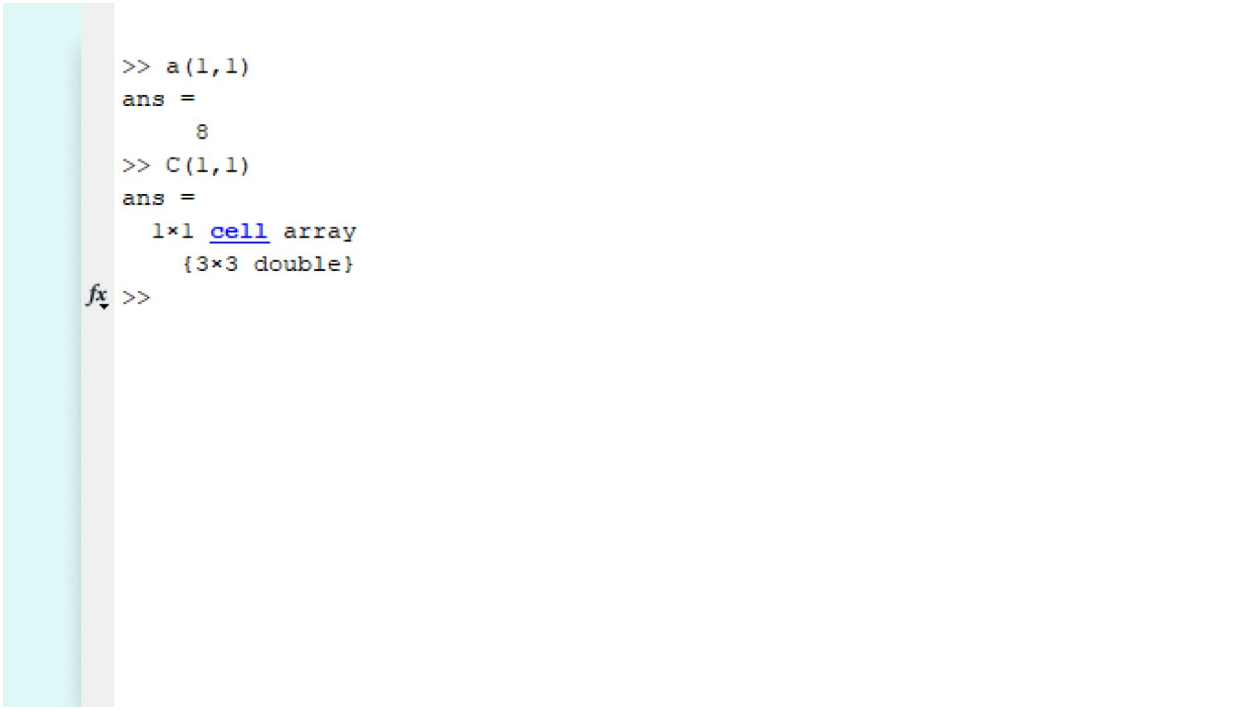
Note that the resulting arrays will have different dimensions. Now, create a single variable that stores the data from `a` , `b` , and `c` as first, second, and third elements respectively of another array. To do this, use a cell array. Note that curly braces `{ }` are used to construct cell arrays. Execute the following code:

```
>> C = {a b c}
```



The cell array `C` (capitalized) has size 1x3, but each element contains a numeric array. Reaching inside a cell array to pull data out of a variable is called indexing. With a numeric array, you use parentheses `()` to pull out the numeric elements. With a cell array, if you use parentheses, you'll pull out the cell elements. Compare the output of the following two statements:

```
>> a(1,1)
>> C(1,1)
```

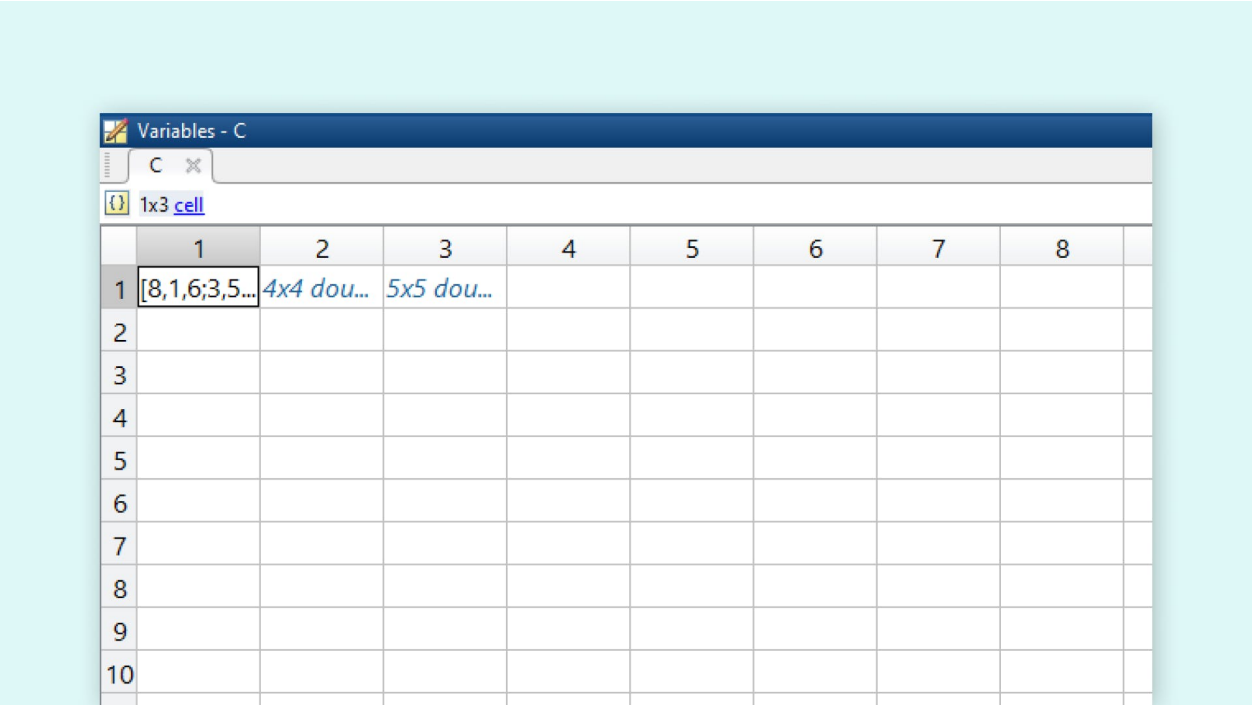


The call to `C(1,1)` returned a link to the object, which is the array a, but did not display the content in it. If you want to reach inside a cell array to pull out the data it contains, you will have to index with curly braces `{}`. To get to the data contained inside the cell `C(1,1)`, execute the following line of code:

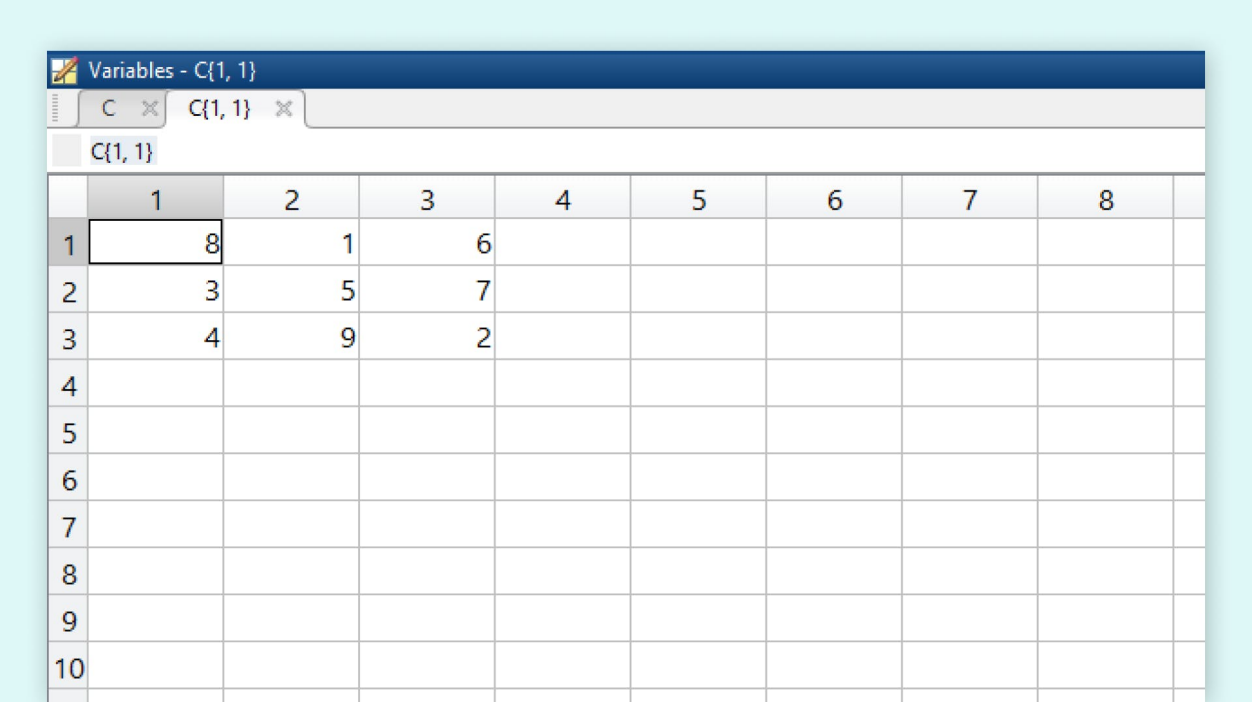
```
>> C{1,1}
```

```
>> C(1,1)
ans =
  1×1 cell array
    {3×3 double}
>> C{1,1}
ans =
     8     1     6
     3     5     7
     4     9     2
>>
```

You can also examine the contents of a cell array by double-clicking on it in the **Workspace**. This action will bring up the contents in the **Variable Editor**.



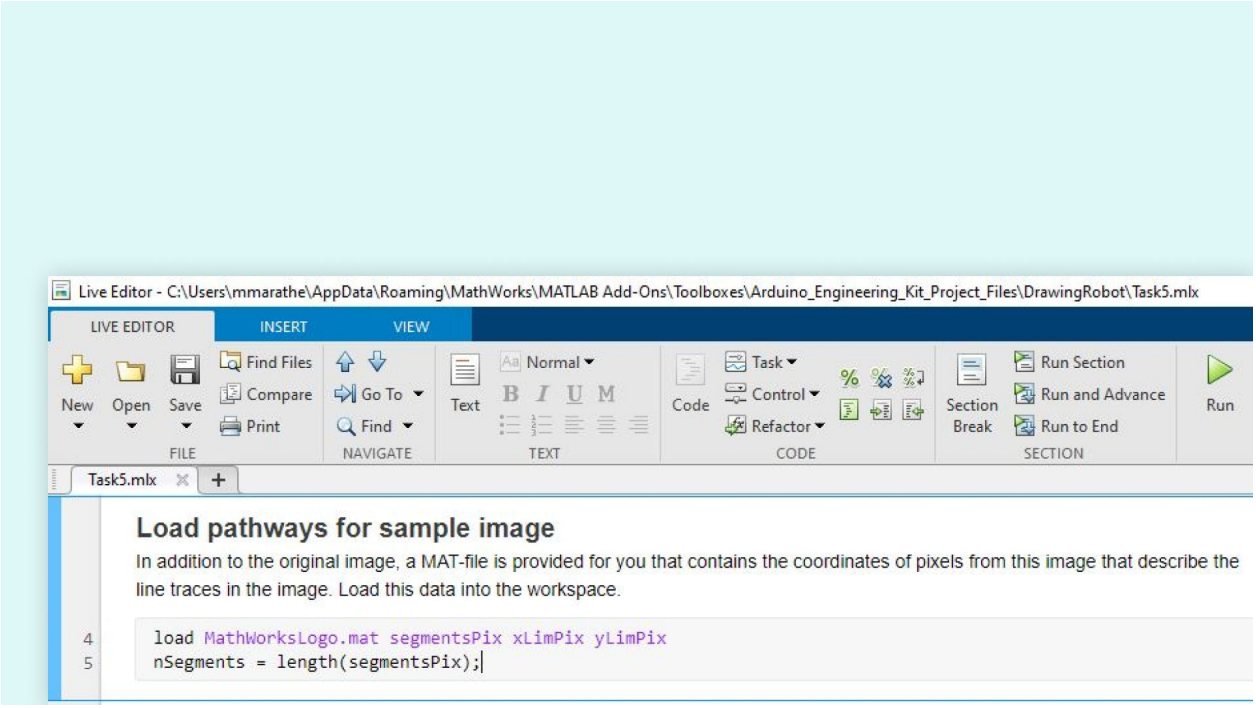
Double-click a cell to dig in and view the contents of that cell, in a separate tab.



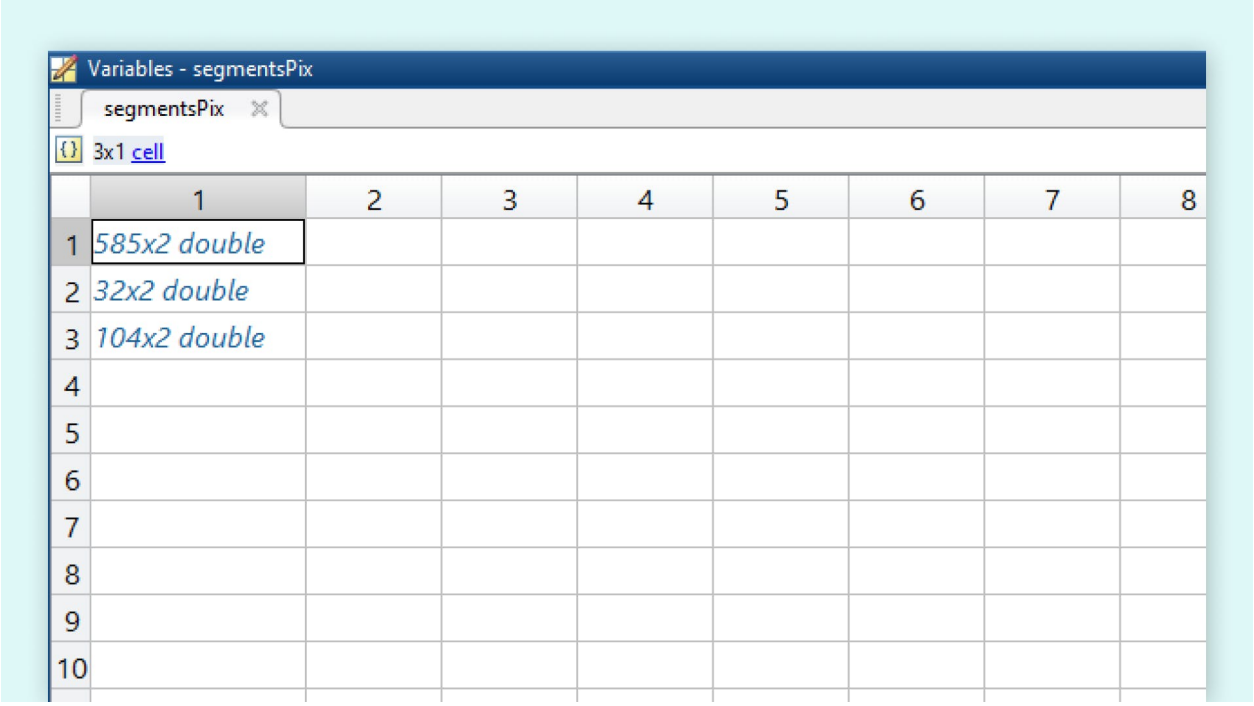
You will use cell arrays in the following sections because they allow you to store coordinate lists of different sizes in a single MATLAB variable. We will work in this way to keep our code tidy. Remember that it is good practice to have your code written in a way that is easy to read by others.

Load and Plot Coordinates for an Existing Image

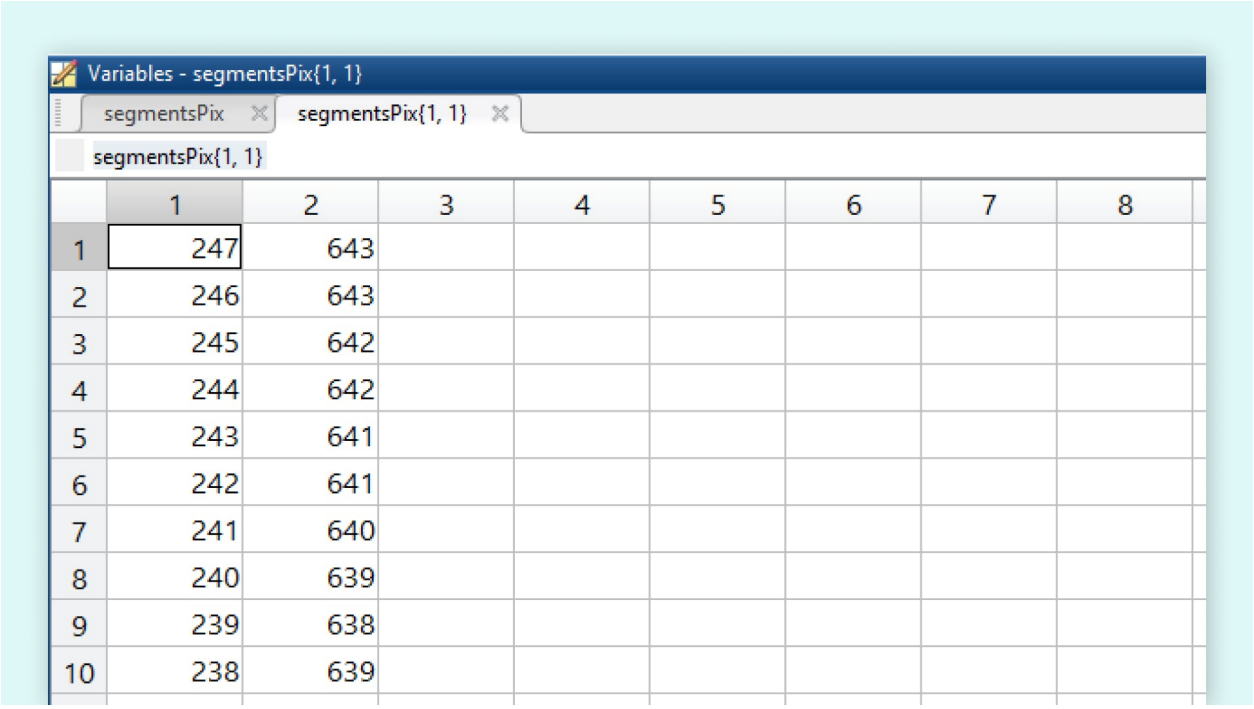
The coordinates of the points to plot are stored in a cell array. Execute the code under the section Load pathways for sample image in the live-script **Task5.mlx** to bring the data into MATLAB.



Double-click on the variable `segmentsPix` from **Workspace** to open it in the **Variable Editor**. Notice that it is a cell array with three elements, each one of which contains a two-column array of doubles with different numbers of rows each.



In other words, there are three different line traces in this image file. Double-click on any one of these elements to see the data contained in that cell. These values represent the pixel coordinates in the image for the selected line trace.

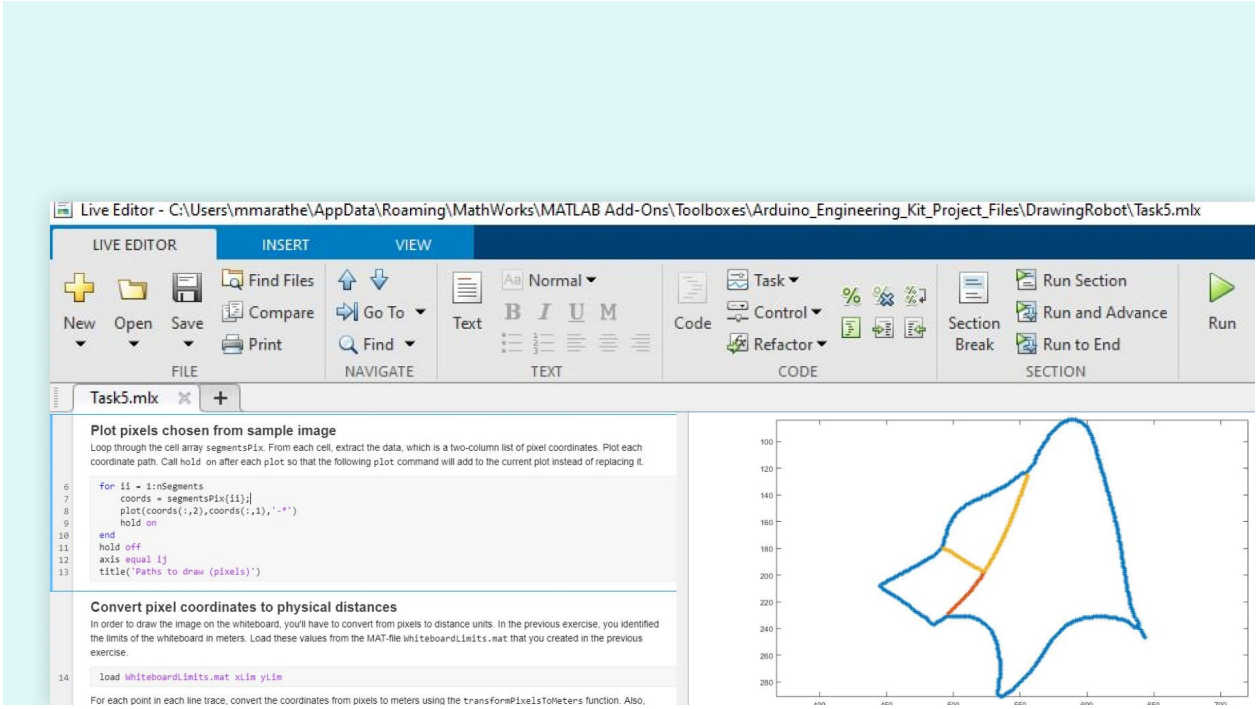


	1	2	3	4	5	6	7	8
1	247	643						
2	246	643						
3	245	642						
4	244	642						
5	243	641						
6	242	641						
7	241	640						
8	240	639						
9	239	638						
10	238	639						

The **nth** element of the `segmentsPix` cell array variable can be accessed using curly braces, as follows:

```
>> data = segmentsPix{n};
```

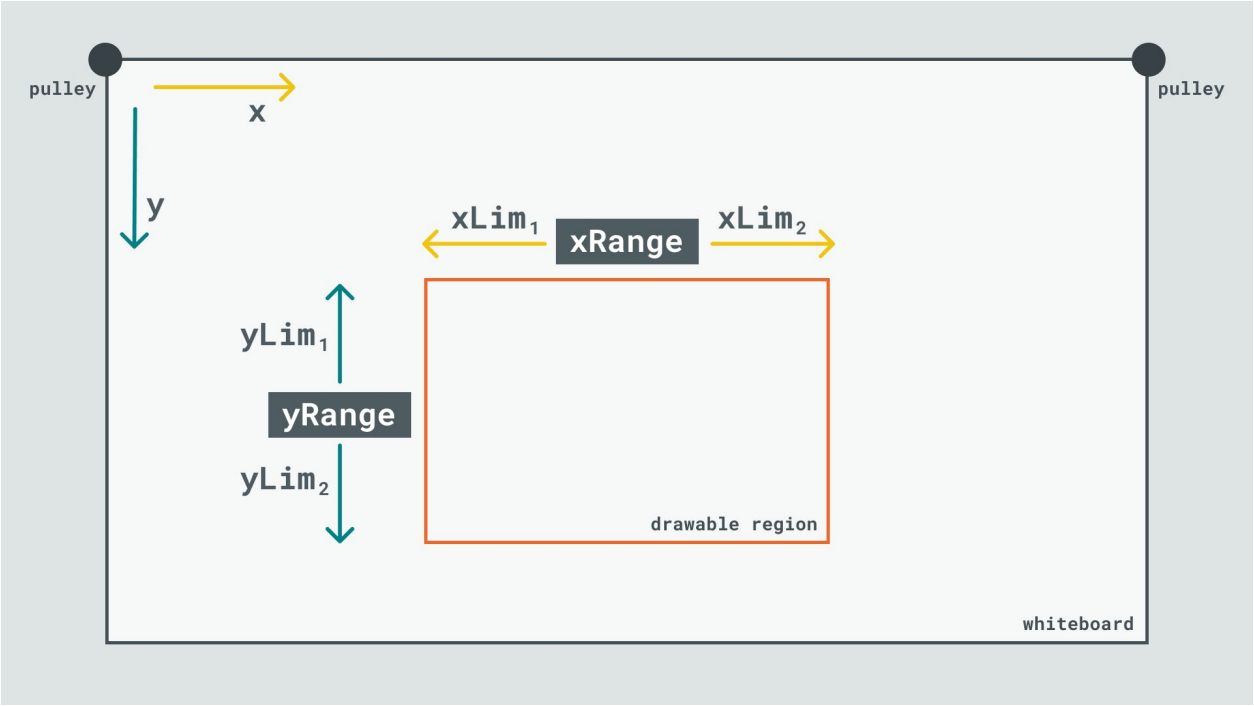
Navigate to the section of code under the heading **Plot pixels chosen from sample image** in the live-script **Task5.mlx**. Note that this section of code contains a `for-loop` with a loop index `ii`. In each iteration of the loop, the **iith** element of the `segmentsPix` variable is extracted and plotted. Execute the code under this section to see the pathways stored in this variable.



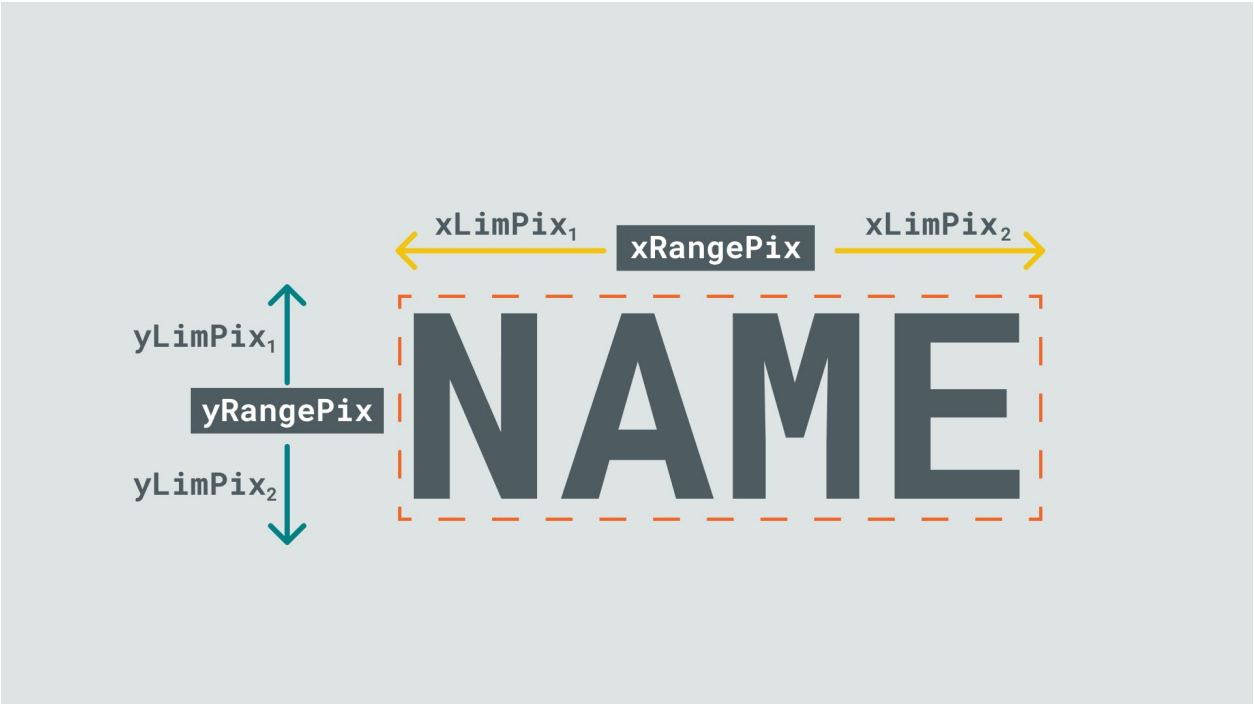
As you can see in the image, the different pathways are drawn in different colors. When we design our own algorithm in a later exercise, we will have a similar result to whatever image we capture, i.e., there will be different pathways which, when joined together, will represent the full image.

Understand Scaling to Physical Units

To decide where to draw an image on the whiteboard, you will have to convert the data from pixels to metric units (meters). The whiteboard dimensions will determine the maximum and minimum whiteboard positions in meters. These values are stored in variables that we will label as `xLim` and `yLim` for `x` and `y` respectively. There will be two limits in the x axis, and two in the y axis (a maximum and minimum position for moving on each of the axis). We define the variables `xRange` and `yRange` as the difference between these maximum and minimum values, as shown in the following diagram.



Similarly, for an image displayed on the screen (where we count distances in pixels), the maximum and minimum pixel positions along the `x` and `y` axes for the image we want to draw are stored in the `xLimPix` and `yLimPix` variables respectively. We define the variables `xRangePix` and `yRangePix` as the difference between these maximum and minimum values, as shown in the following diagram.



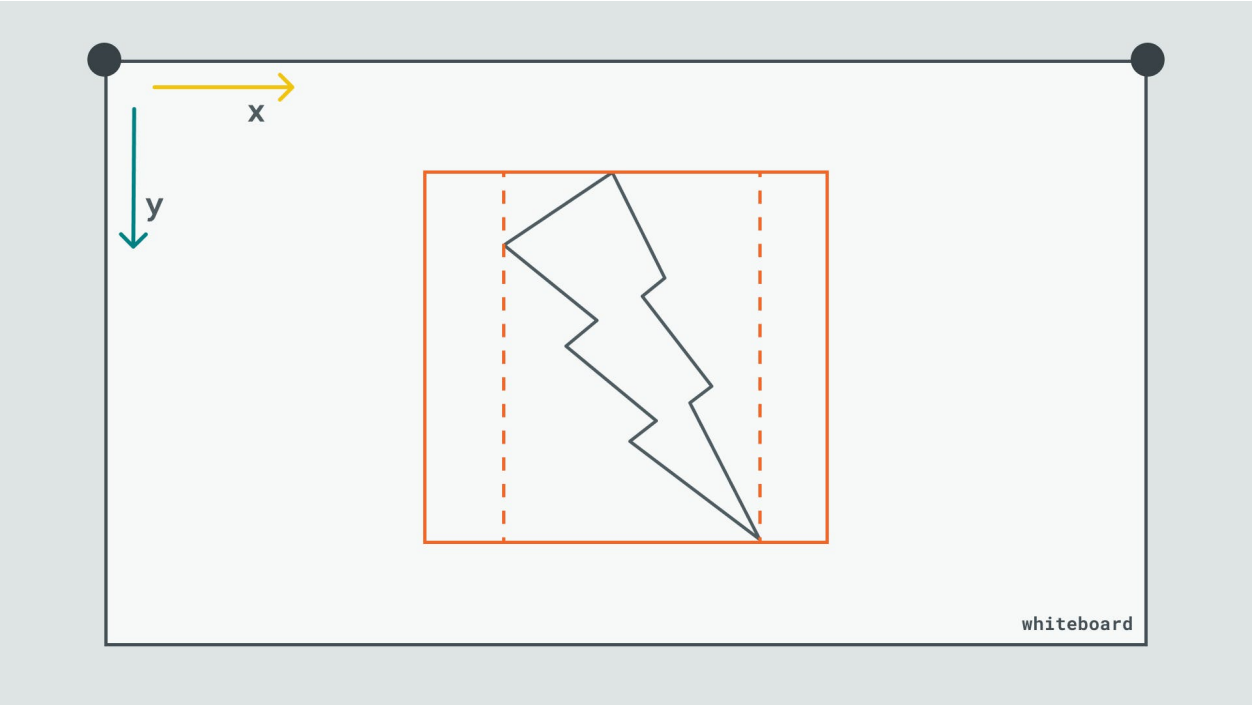
It seems obvious that to convert from pixels to distances, you'll need to multiply the pixel values by some scaling factor. That scaling factor will be one of the following two quantities:

$$xScaleFactor = \frac{xRange}{xRangePix}$$

$$yScaleFactor = \frac{yRange}{yRangePix}$$

As we have a bi-dimensional image, and the drawing surface is not connected, by-design, to the image size, we will end up with two different values: one corresponding to the translation of the image to the whiteboard looking at the x-axis, and the other for the y-axis. We need to decide between one of the two possible scaling factors. The choice of which scaling factor to use depends on which one is smaller. This will be related to the aspect ratio of the whiteboard drawing area versus the image to draw. Consider a relatively square drawing area and a short, wide image. In this case, you would want to use the `xScaleFactor`, as shown in the following diagram:

If the image is tall and narrow, you would be limited by the height of the image and would want to scale it using the `yScaleFactor`, as shown in the following diagram:



In other words, your algorithm will have to decide how to best convert the image from pixels to meters. The approach presented here scales the image to fill in as much of the drawable area as possible without distorting the image. But distortion is something that you should consider at a later stage, especially if your drawing area is a bit large.

The transformPixelsToMeters() function

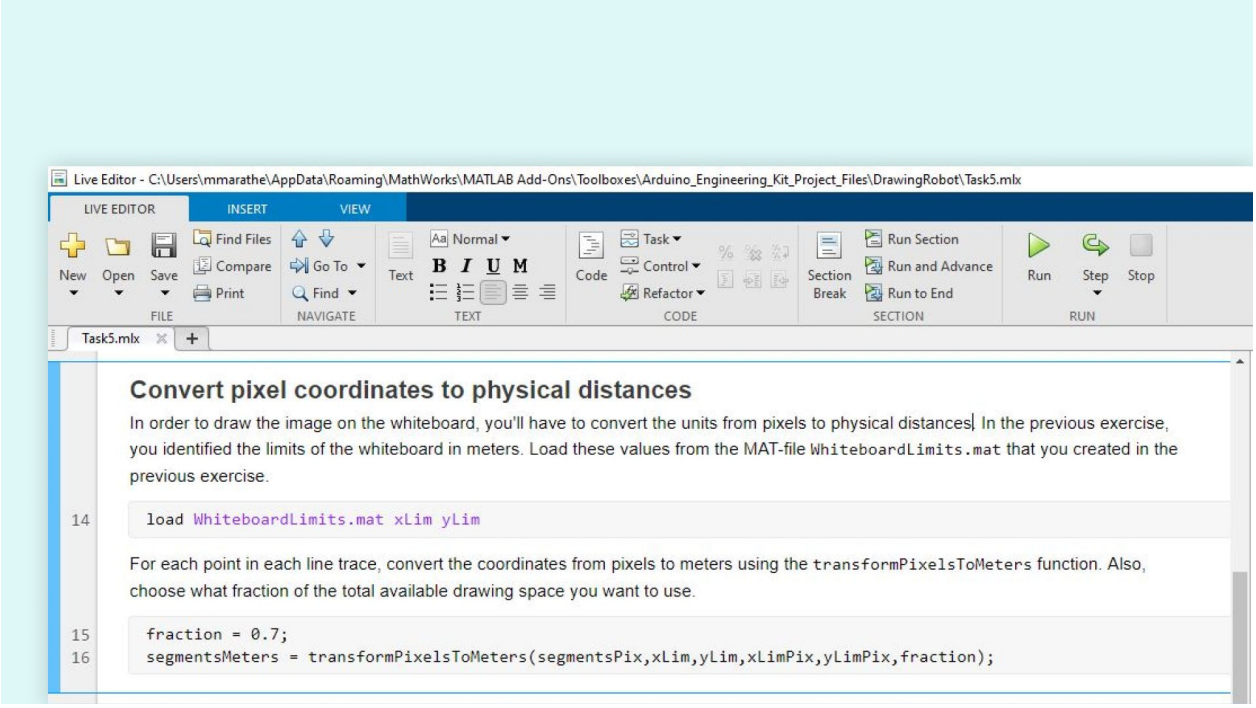
The function `transformPixelsToMeters()` converts pixels into meters. As you go through the function, notice how it decides which scaling factor to use, how it centers the image in the middle of the drawing area, and how it converts the measurements of each trace, translating them to the new coordinate system. The function also allows the user to specify what percentage of the available space should be used to draw the image. This is specified by the `fraction` input argument.

```
function segmentsMeters =
transformPixelsToMeters(segmentsPix,xLim,yLim,xLimPix,yLimPix,fraction)

% This function computes the scale factor for converting pixels to
meters, then converts all the segments in segmentsPix to meters.

% First, create variables to represent the ranges xRange, yRange,
xRangePix, and yRangePix.

xMinM = xLim(1);
yMinM = yLim(1);
xRangeM = diff(xLim);
```

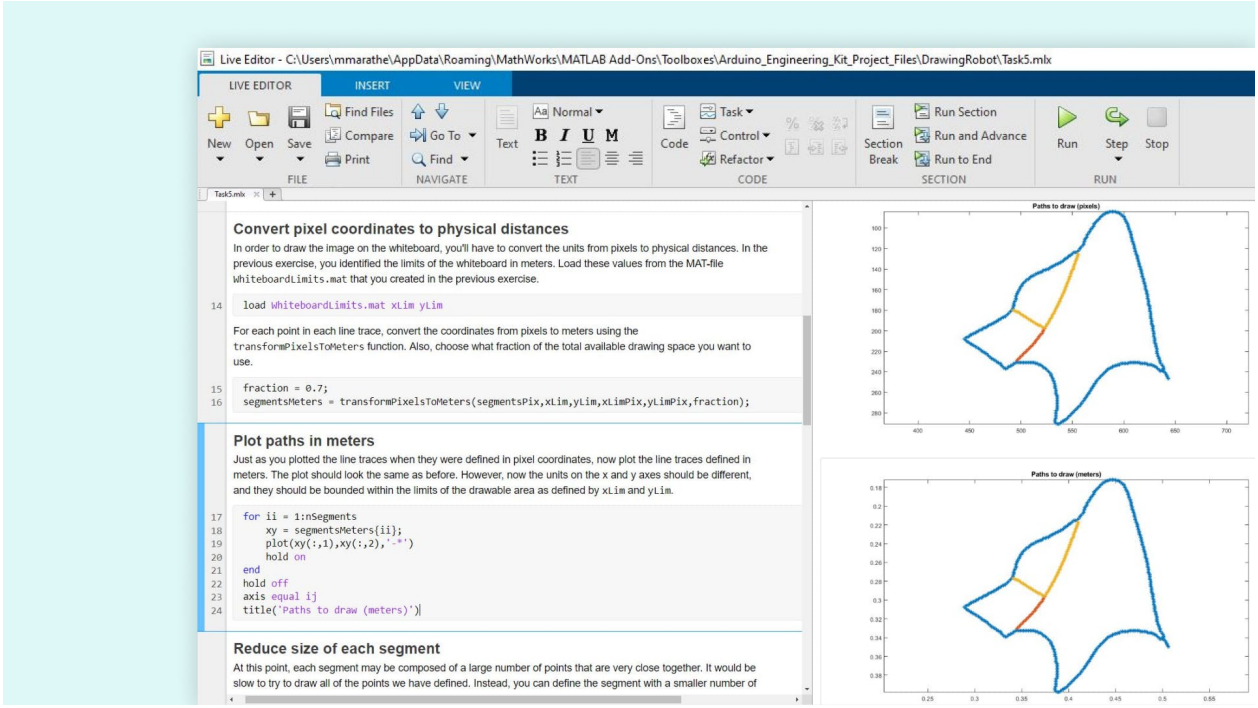
```
yRangeM = diff(yLim);

% Determine the range of the coordinates to draw
xMinPix = xLimPix(1);
yMinPix = yLimPix(1);
xRangePix = diff(xLimPix);
yRangePix = diff(yLimPix);
```

Convert Pixels to Physical Distance and Plot Images

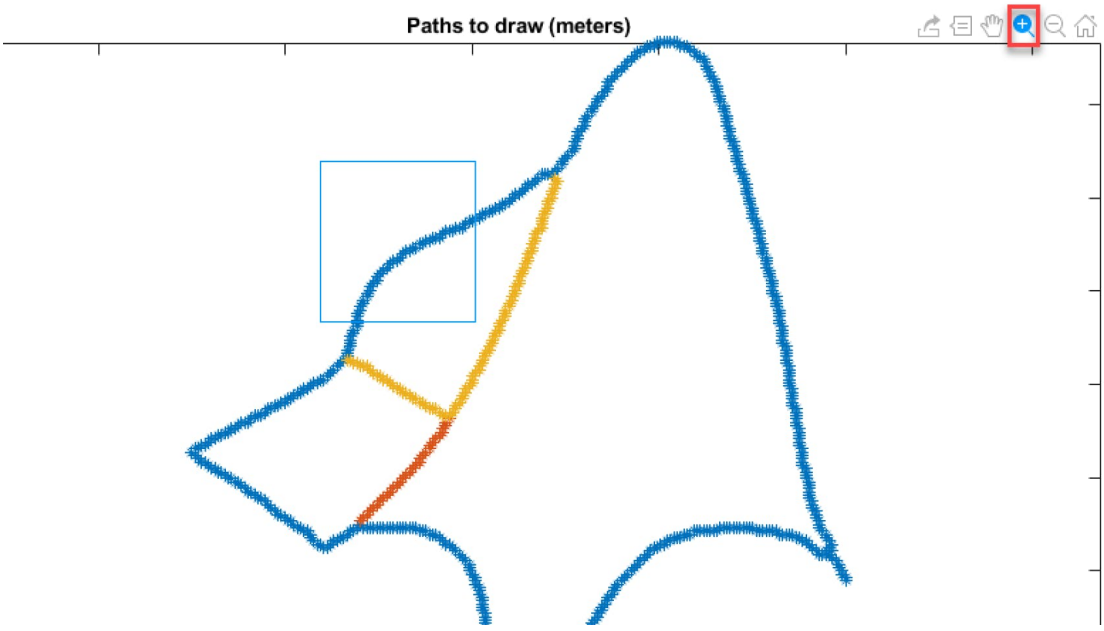
Now that you know how it works, you can put the `transformPixelsToMeters` function to use on the data you loaded from the sample image. You'll use the whiteboard x and y limits that you determined in the previous exercise. Then, for each segment, you'll convert the pixels from that segment to meters and store the new segment data in a cell array variable called `segmentsMeters` . To do this, execute the code under the heading **Convert pixel coordinates** to physical distances in the live-script **Task5.mlx**.

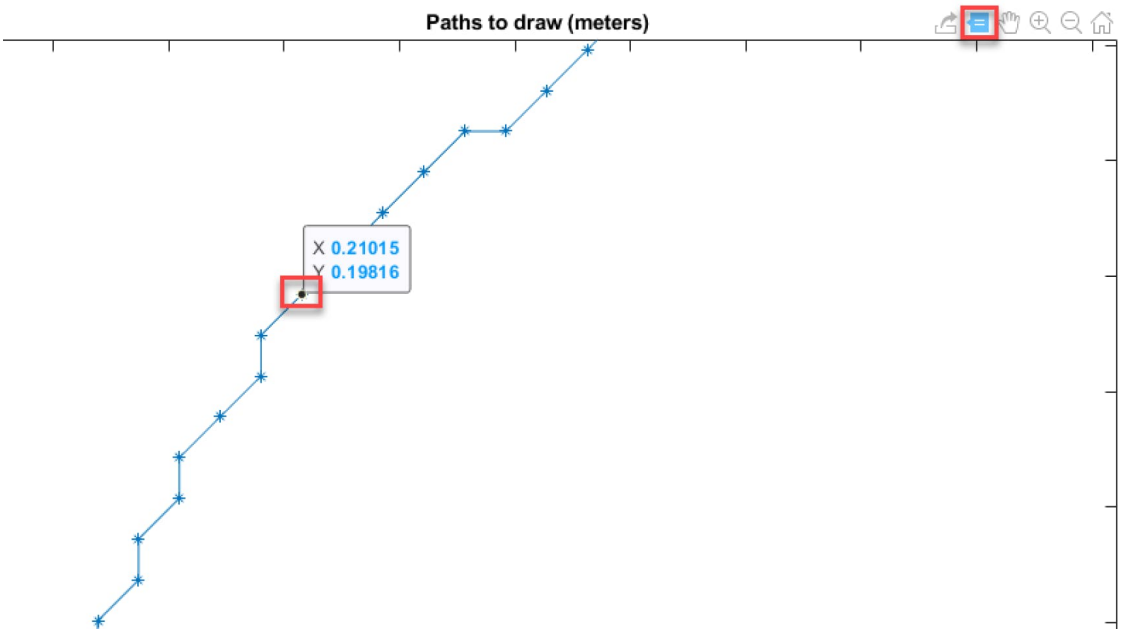


Next, you'll create a plot of the data contained in the `segmentsMeters` variable just as you plotted the data in the `segmentsPix` variable earlier. The resulting plot should look the same but use units of meters and a range within the limits of your whiteboard. Execute the code under the heading **Plot paths in meters** in the live script **Task5.mlx**.



If you zoom in on this plot and use the **Data Cursor**, you'll see that the points to be drawn are very close to each other (in this case most points are separated by less than 1 mm).





If the robot tries to draw each and every point on all the segments, it will take a lot of time to complete the drawing. In the next section, we will see how to filter the segments to reduce the drawing time, while maintaining the fundamental shape of the image.

The reduceSegment() function

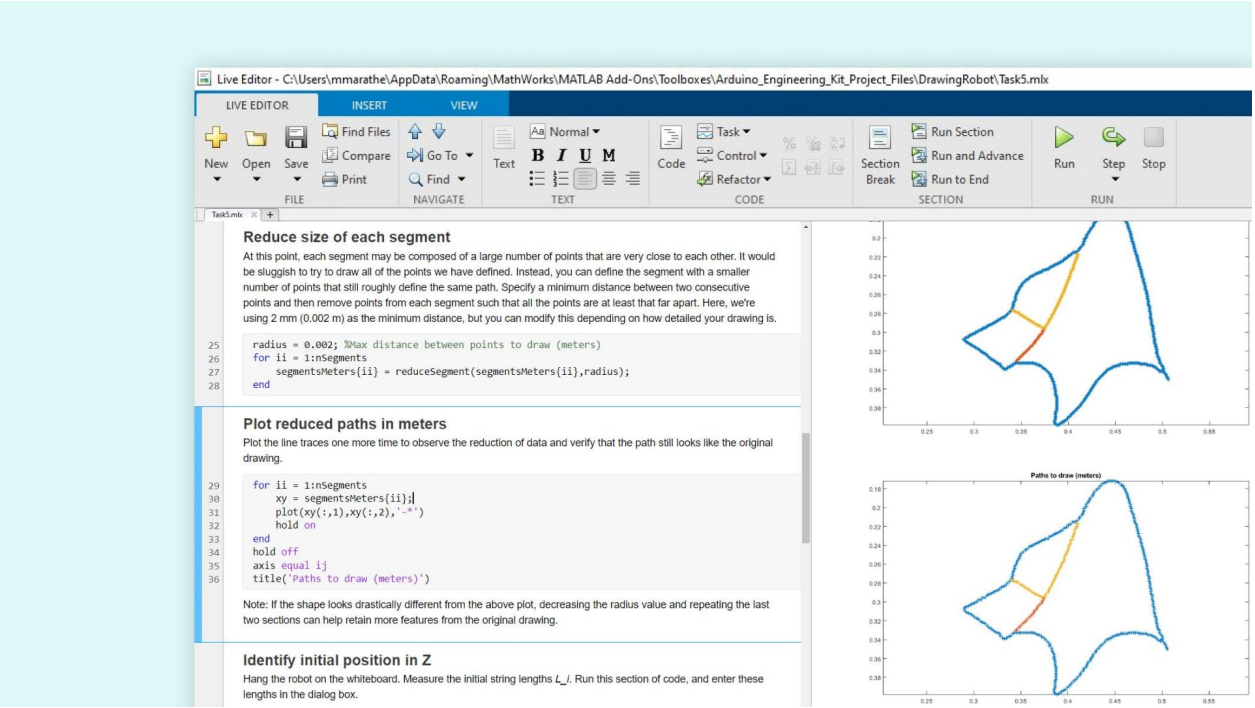
The `reduceSegment` function reduces the number of points in a segment by removing points within a specified radius. In other words, it filters out points, which reduces the number of subsegments to draw. The function accepts two inputs: the segment that should be reduced and a radius value. The radius value is the desired minimum distance between any two points on the segment. As you read through, observe how the function applies some of the concepts you applied previously, such as: `while loop` and `logical indexing`.

```
function reducedSegment = reduceSegment(segment, radius)
% First, create variables that will be used to decide which points to
keep in the reducedSegment
nPoints = size(segment,1);
keepPoint = true(nPoints,1);
reference = 1;
test = 2;

% Loop through the segment and decide which points to remove. By
default, all points will be included in the output. Select points to
keep and remove such that each consecutive point remaining is separated
by a minimum distance from the previous point.
while test < nPoints
```

```
% Check if the distance between the reference point and the current test
point is less than the specified minimum radius. If it is, mark the test
point for removal and test the next point in the segment.

if norm(segment(test,:) - segment(reference,:)) < radius
```

Reduce Segment Sizes and Plot the Images

Let's execute the `reduceSegment` function in a `For-Loop` to reduce the number of points in each segment of the image. Execute the section of code under the heading Reduce size of each segment in the live-script **Task5.mlx**. Then execute the next section of code under the heading **Plot reduced path in meters** to visualize the updated paths.



This will represent a third plot, where you will see a lot less number of points, which will significantly reduce the time taken to print such an image by the robot.

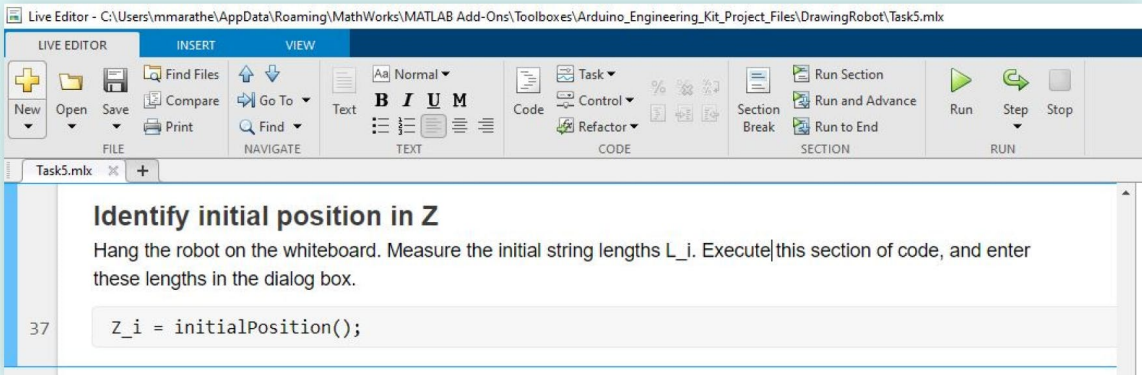
> **Note:** The empirically-obtained radius value of 2 mm worked well for the images on our whiteboard. If the image on your whiteboard looks like it's missing important features, try decreasing the radius and executing the last two sections of code again.
>
> Alternatively, if your image doesn't have much fine detail, you may be able to draw it even faster by choosing a larger radius. Choosing a radius value that

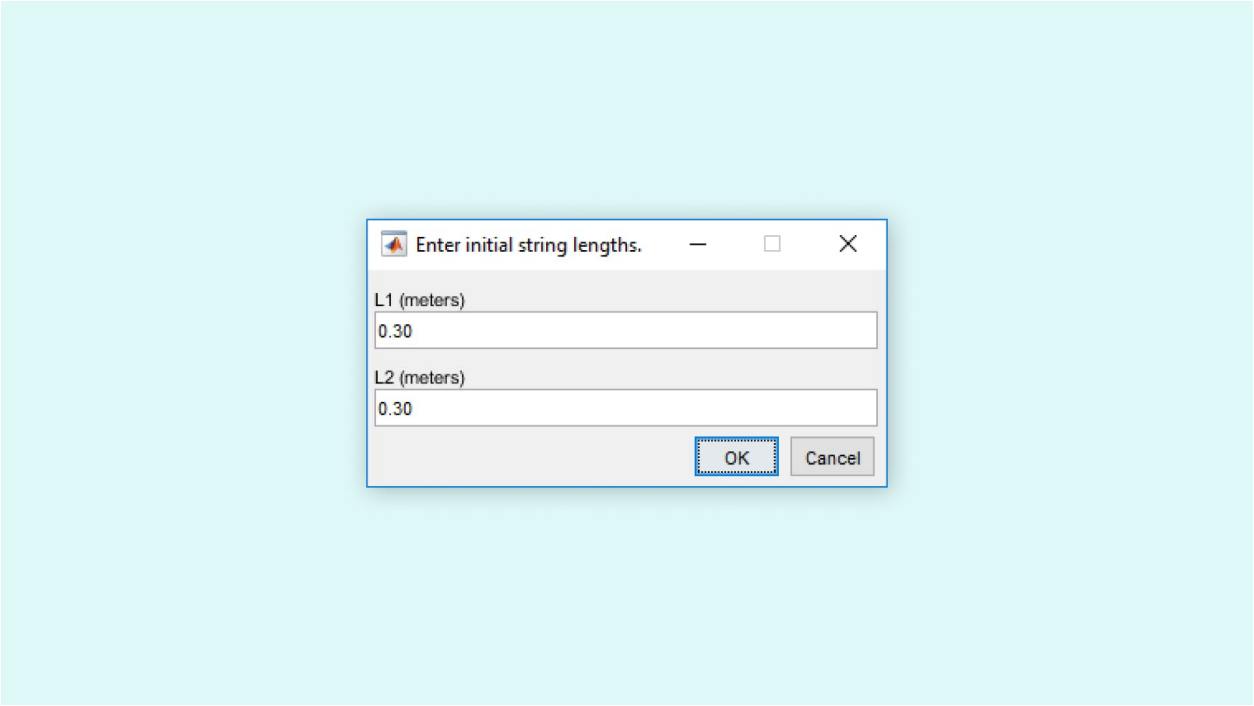
balances drawing speed and reproduction accuracy is at this point a process of trial and error.

You could consider improving this algorithm in several ways: by figuring out how different an image is from one another, and by having a dynamic radius depending on the estimated error between original and reduced images.

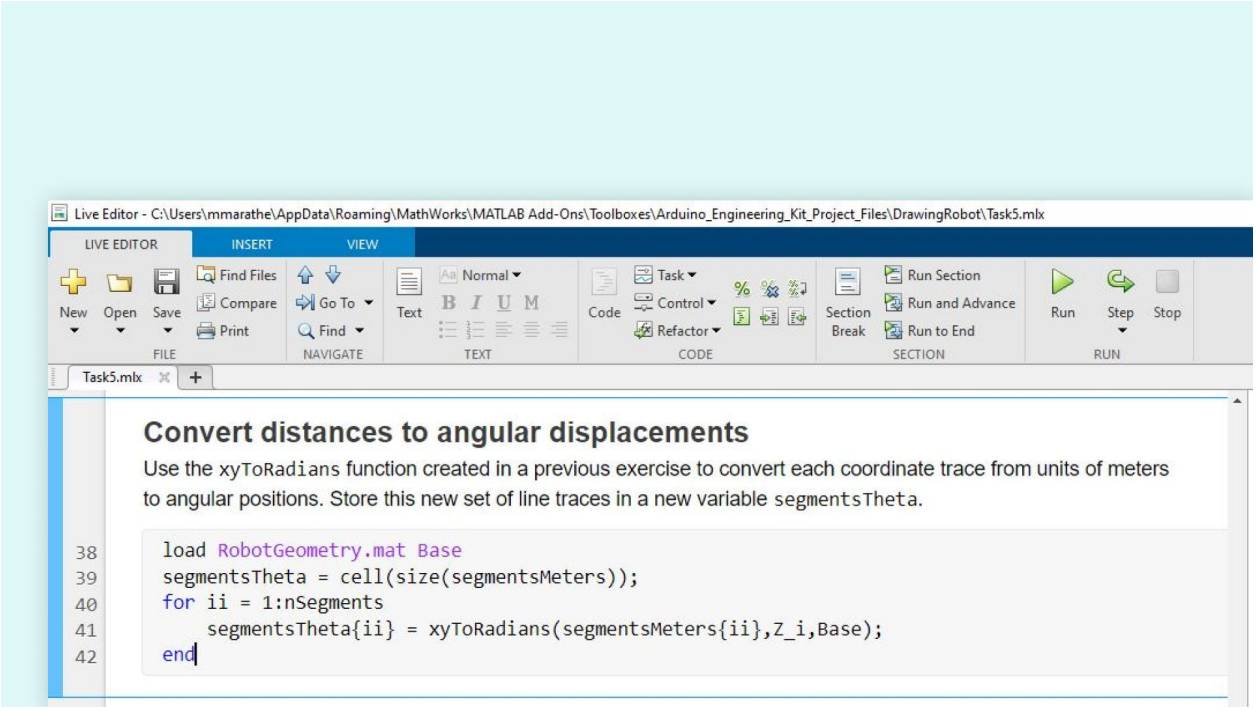
Convert Desired Positions to Angular Displacements

As you did in a previous exercise, you'll now use the `xyToRadians` function to convert the target positions in meters to target angular displacements of the motor shaft. Before calling this function, you'll need to determine the initial position of the robot. Measure the length of each string as you've done in the previous exercises. Then execute the section of code under the heading **Identify initial position in Z** in the live-script **Task5.mlx** and enter these lengths in the dialog box.



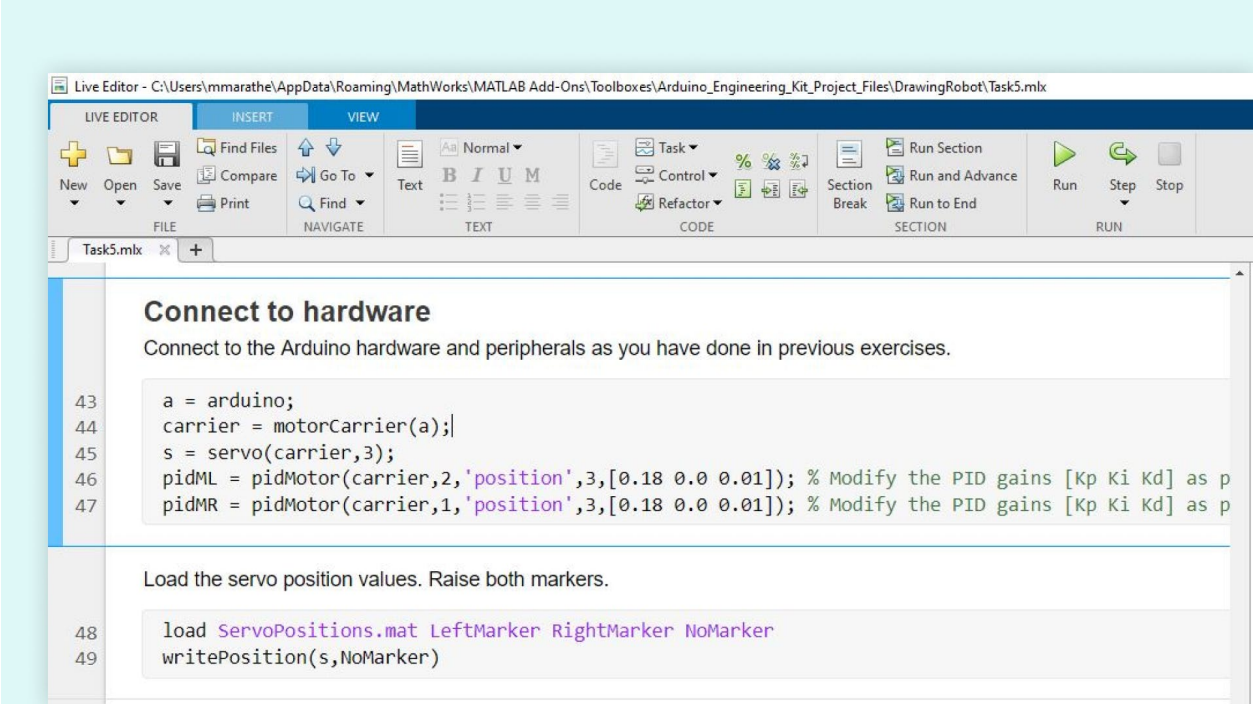


Since the data is stored in a cell array, you'll have to loop through the array using a `For-Loop` and convert the pathways one at a time. Store the resulting target angular position values in a cell array called `segmentsTheta`. Execute the section of code under the heading **Convert distances to angular displacements** in the live-script **Task5.mlx**.

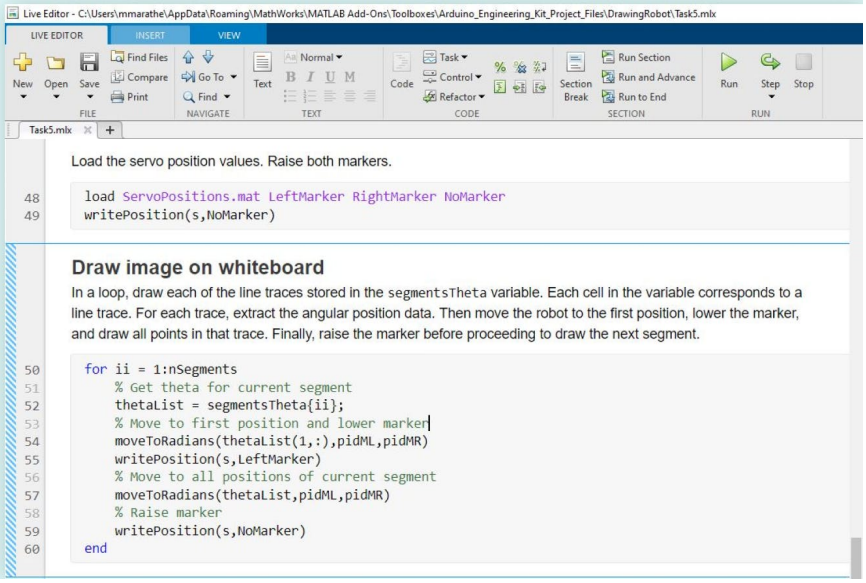


Draw Line Segments onto the Whiteboard

Now that you have the angular displacement values for all the positions you want to move the robot to, you can use them to draw each path just as you did in a previous exercise. As before, first, **connect to the hardware**. Execute the section of code under the heading Connect to hardware in the live-script **Task5.mlx**.



Navigate to the section of code under the heading **Draw image on whiteboard**. It uses a `For-Loop` to step through the `segmentsTheta` variable. Think about what's happening in the loop. Each iteration of the loop, at a time, works on a single line trace of the image. It extracts the target angular position values of the motor shaft for that line trace, moves the robot to the first position, lowers the marker and moves the robot to all positions in that line trace, and then finally raises the marker. This is repeated for each of the three line traces in the image. Execute this section of code to draw the image on the whiteboard.



As discussed earlier, if the image on your whiteboard looks like it's missing important features or appears distorted compared to the original image, try experimenting with the following variables: `radius` under the heading **Reduce size of each segment**, and the delay parameter `nD` from the function `moveToRadians()`, and the PID gain values K_p, K_i, K_d under the heading **Connect to hardware**.

Write a Function for Drawing an Image from Pixels

You've just completed a task that began with gathering pixel data information from an image and ended with a drawing of the image on a whiteboard. If you want to perform all these steps again, then it's useful to have the individual code sections bundled together in a function. Create a new MATLAB function and call it `drawImageFromPix()`. The function will accept pixel data and whiteboard starting position as inputs and then draw the image. Include the following code in the function and save it as **drawImageFromPix.m**.

```
function drawImageFromPix(segmentsPix,xLimPix,yLimPix,Z_i)
nSegments = length(segmentsPix);

% Define whiteboard limits
load WhiteboardLimits.mat xLim yLim

% Convert pixel coordinates to physical distances and then to encoder
counts
```

```
fraction = 0.7;
segmentsMeters =
transformPixelsToMeters(segmentsPix,xLim,yLim,xLimPix,yLimPix,fraction);

% Reduce size of each segment
radius = 0.002;

%Max distance between points to draw (meters)
for ii = 1:nSegments
    segmentsMeters{ii} = reduceSegment(segmentsMeters{ii},radius);
end
```

The variables `segmentsPix`, `xLimPix`, `yLimPix`, and Z_i should still be in your **Workspace**. Call your new function from the MATLAB command prompt to draw the image again and confirm that the function works as expected.

```
>> drawImageFromPix(segmentsPix,xLimPix,yLimPix,Z_i)
```

Files

- Task5.mlx
- transformPixelsToMeters.m
- drawImageFromPix.m

Learn by Doing

In this exercise, you drew an image using data from the MathWorksLogo.mat MAT-file. Other MAT-files have been provided for you, containing the line traces from other images. Update your code to draw some of these other images. First, preview the images in MATLAB, and then draw them on the whiteboard.

For a more advanced challenge, try drawing two images side by side. Load the MAT-file data for both images. Then update the data from the second image so that its pixel coordinates are to the right of all points in the first image. Then concatenate the cell arrays of pixel positions. Finally, update the `xLimPix` and `yLimPix` variables so they accurately describe the pixel limits for the new combined set of paths.

4.6 Draw any Image

In the previous exercise, you took a set of pixel positions that described pathways in an image and used the drawing robot to draw those pathways. Now you'll learn how to take a photograph and process the image to identify those pixel positions and pathways to draw. You'll use image conversion, filtering, and analysis techniques. At the end of the exercise, you'll be able to use your robot to recreate any image with a line drawing. Try drawing the included sample images or make your own.

In this exercise, you will learn to:

- Convert, filter, and analyze images using image processing functions
- Construct recursive functions
- Simplify code by operating on entire array at once
- Write a main script to execute the complete workflow

Understand Image Conversion, Filtering and Analysis

The Image Processing Toolbox, from MATLAB, provides a large number of functionalities for image processing, analysis, visualization, and algorithm development. For this project, you'll leverage some of these capabilities to extract line traces from an image containing line drawings.

In a previous exercise, you saw that an image can be represented in MATLAB as an **MxNx3** array of **uint8** values, which is great if you want to preserve all the information about an image. This is just one of the several ways to represent an image. We can reduce the complexity of an image to extract the specific information we are interested in. Here are some other forms an image can take in MATLAB:

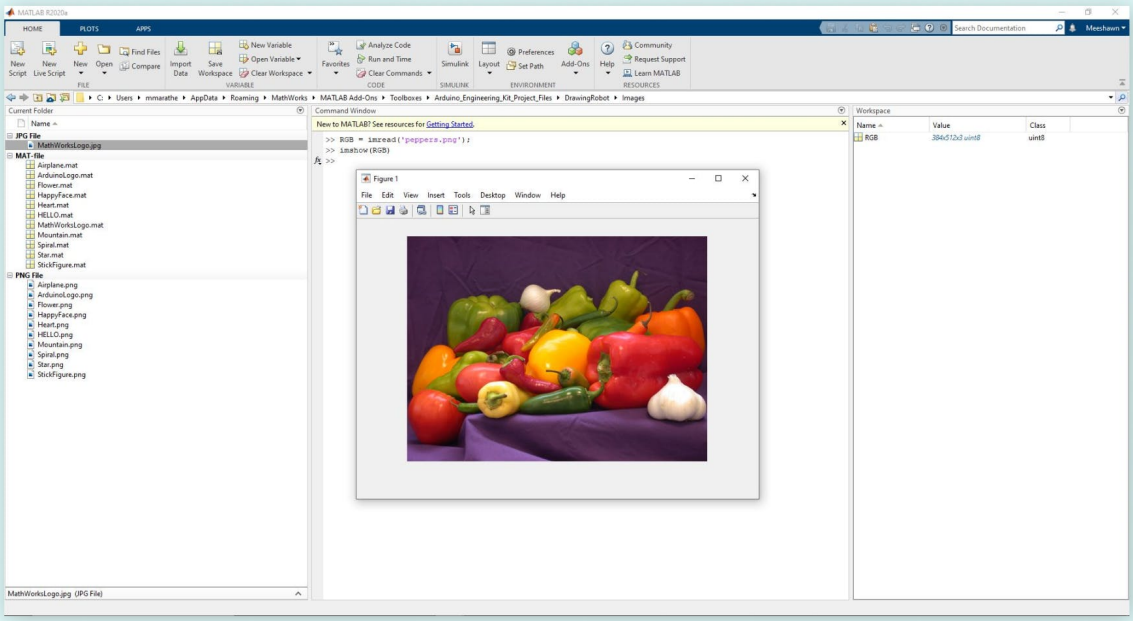
- A color image can also be represented as a 2-dimensional MxN array with a corresponding colormap. We won't be working with this type of color image.

- A grayscale image can be represented simply as an MxN array. This is useful for image processing functions that operate on a 2-D grid of data, such as filtering.
- A binary image can be represented as an MxN array of logical (true or false) values. This is useful for extracting the characteristics of particular regions of interest in an image.

If you want more information on image processing, you can check out **this book.**

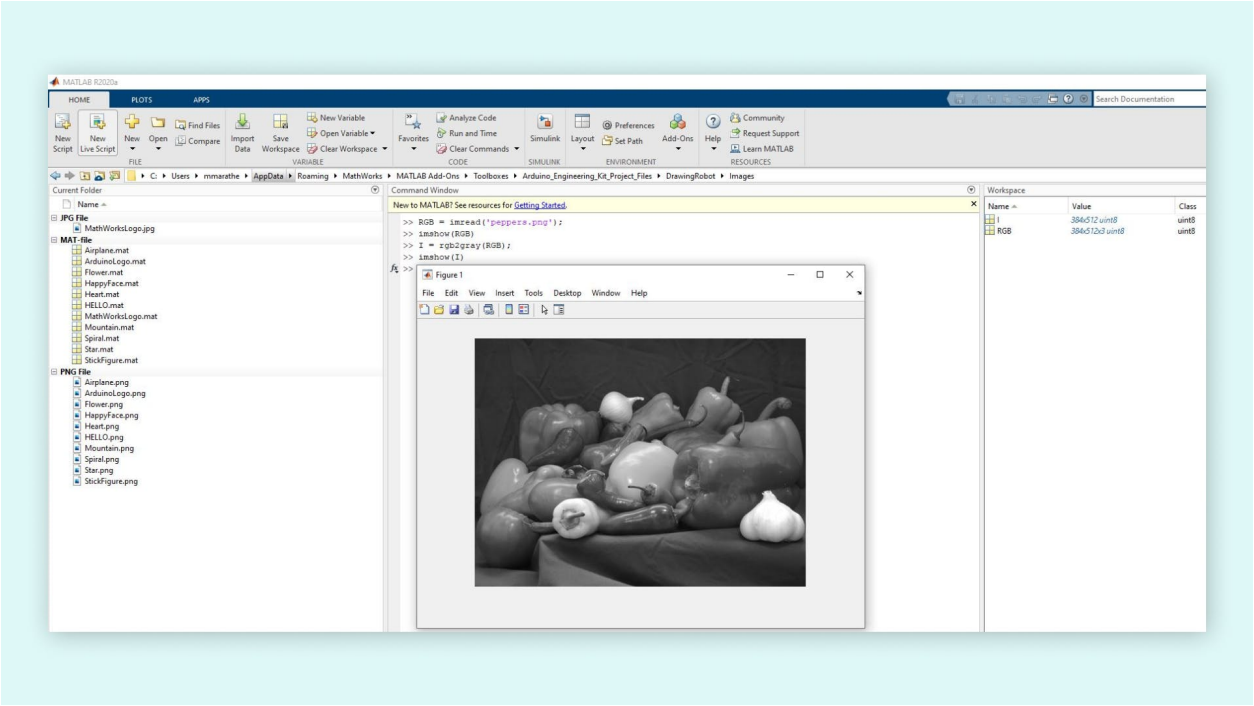
Let's examine an image and convert it to grayscale. Execute the following code at the **MATLAB command prompt** to load and view the 'peppers' image in MATLAB:

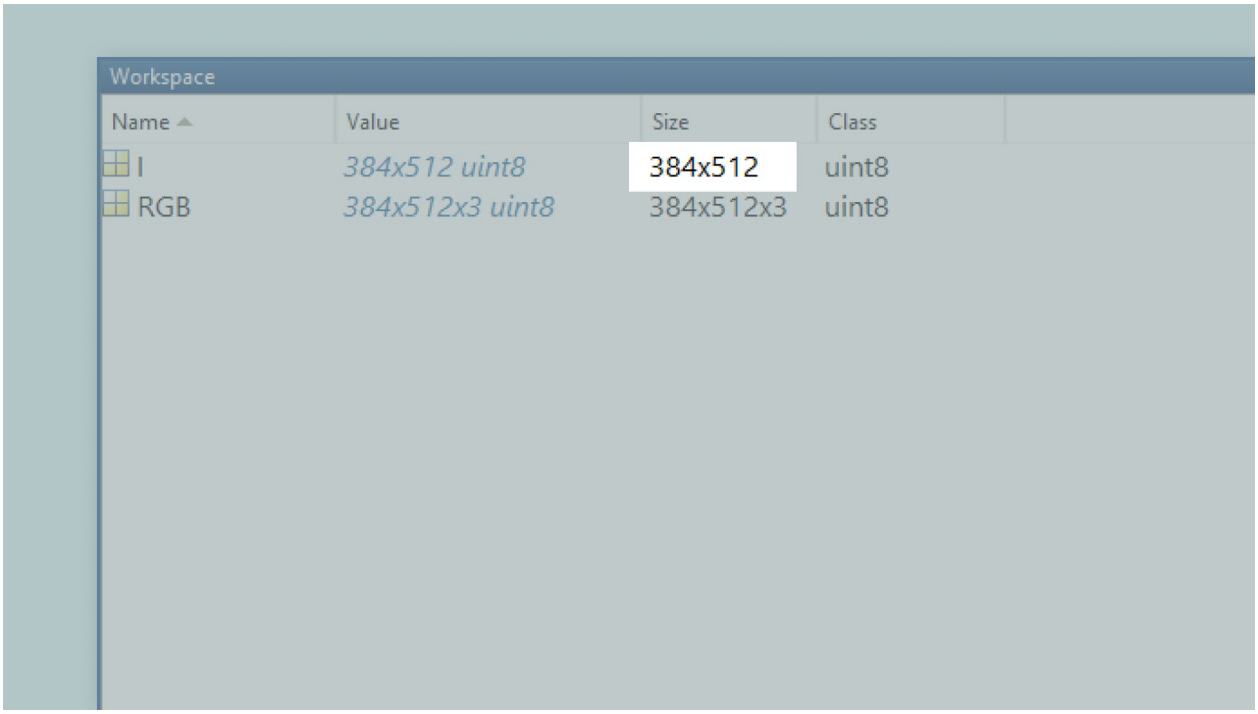
```
>> RGB = imread('peppers.png');
>> imshow(RGB)
```



Now convert the image to grayscale and view it in MATLAB. Also, note the dimensions of the new image in the MATLAB workspace.

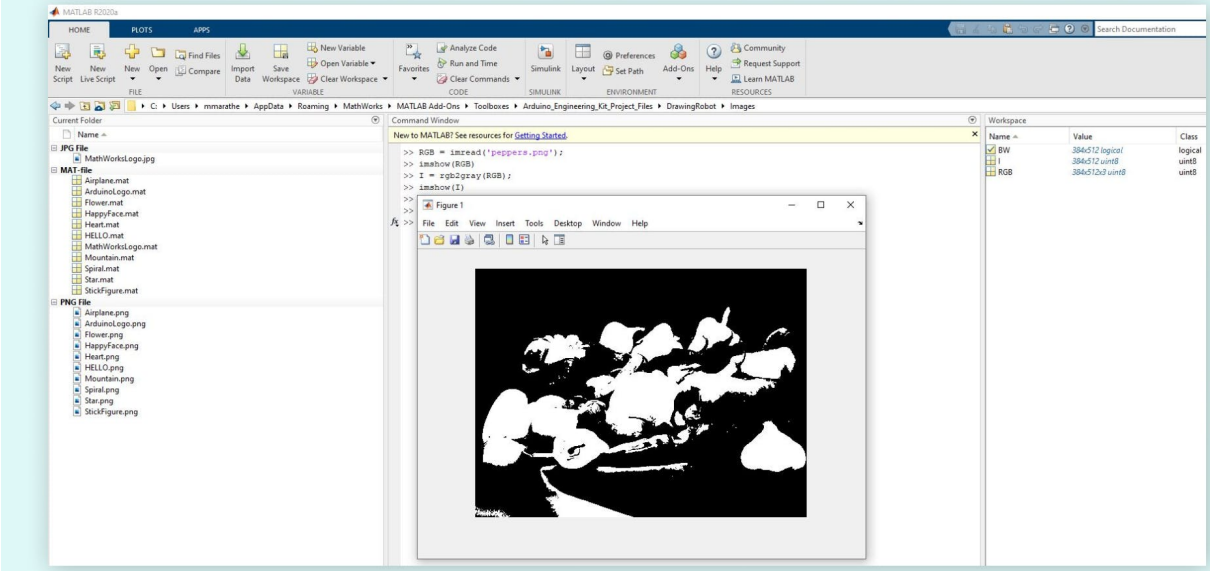
```
>> I = rgb2gray(RGB);
>> imshow(I)
```

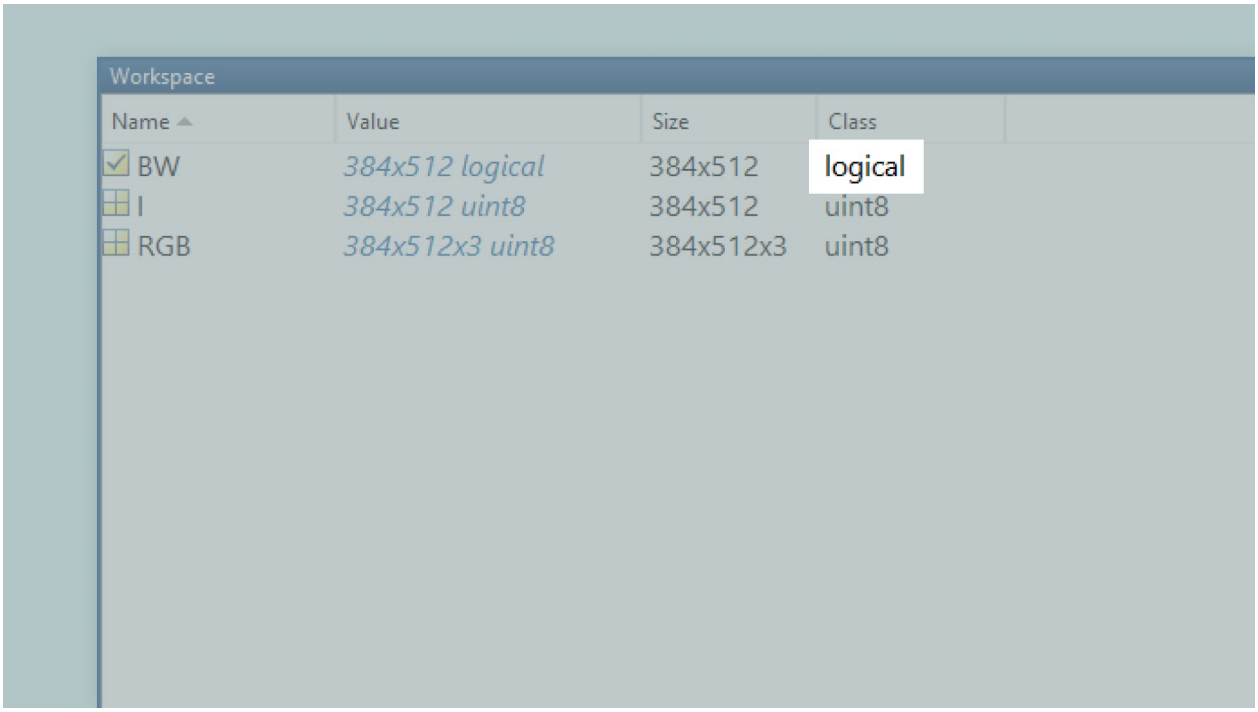




Binary images are images wherein each pixel is either on or off. These are visualized in pure black-and-white, which can be useful for describing specific regions or objects in an image. A grayscale image can be converted to a binary image with the help of the MATLAB function `imbinarize`. This function replaces all values above a certain threshold with 1 and those below the threshold with 0. The threshold can be global or regional. Convert the ‘peppers’ image to binary and view it in MATLAB. Also, note the data type of the resulting binary image.

```
>> BW = imbinarize(I);
>> imshow(BW)
```

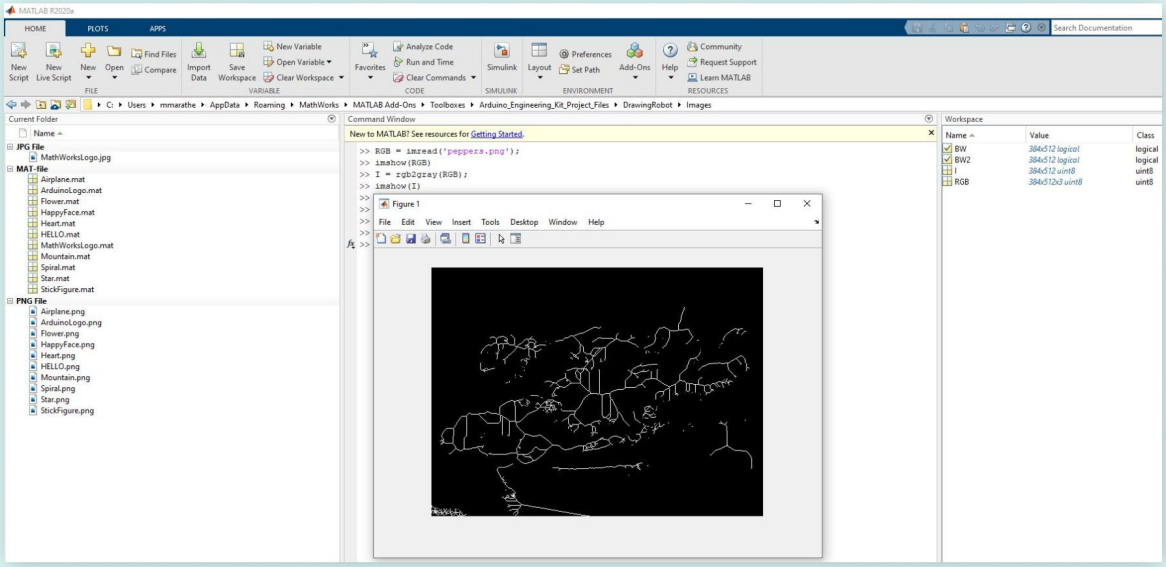




There are many ways in which you can process binary images to extract the features you want. One approach is to use morphological operations such as **dilation** and **erosion**. The most useful of all these for extracting line traces is the **thinning** operation. This operation can thin out objects in the image all the way

down to lines. Try this out on the 'peppers' image to see which line traces represent the thinnest version of the objects shown in the image.

```
>> BW2 = bwmorph(BW,'thin',inf);
>> imshow(BW2)
```



Refer to the following MATLAB documentation pages for more information about these and other image processing functions:

- **Image Type Conversion**
- **Morphological Operations**
- **imbinarize**
- **bwmorph**

In the following sections, we'll use the same techniques to extract line traces from raster images.

Load Existing Image From File

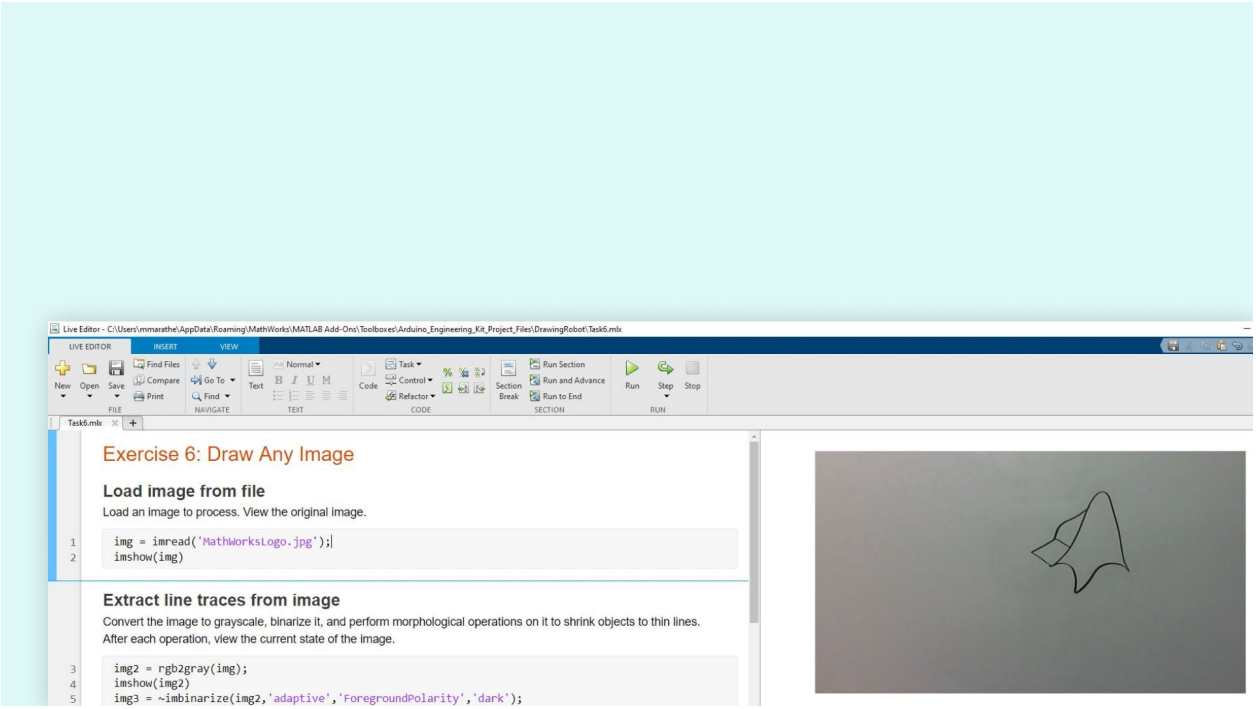
You can use functionalities from the Image Processing Toolbox to convert any image into a form that is drawable by the robot. First, try this on the sample

images provided to you. Later, you can try it on your own photos.

Start by loading an image in MATLAB. We'll use a line drawing of the MathWorks logo. Open the live-script **Task6.mlx** in MATLAB.

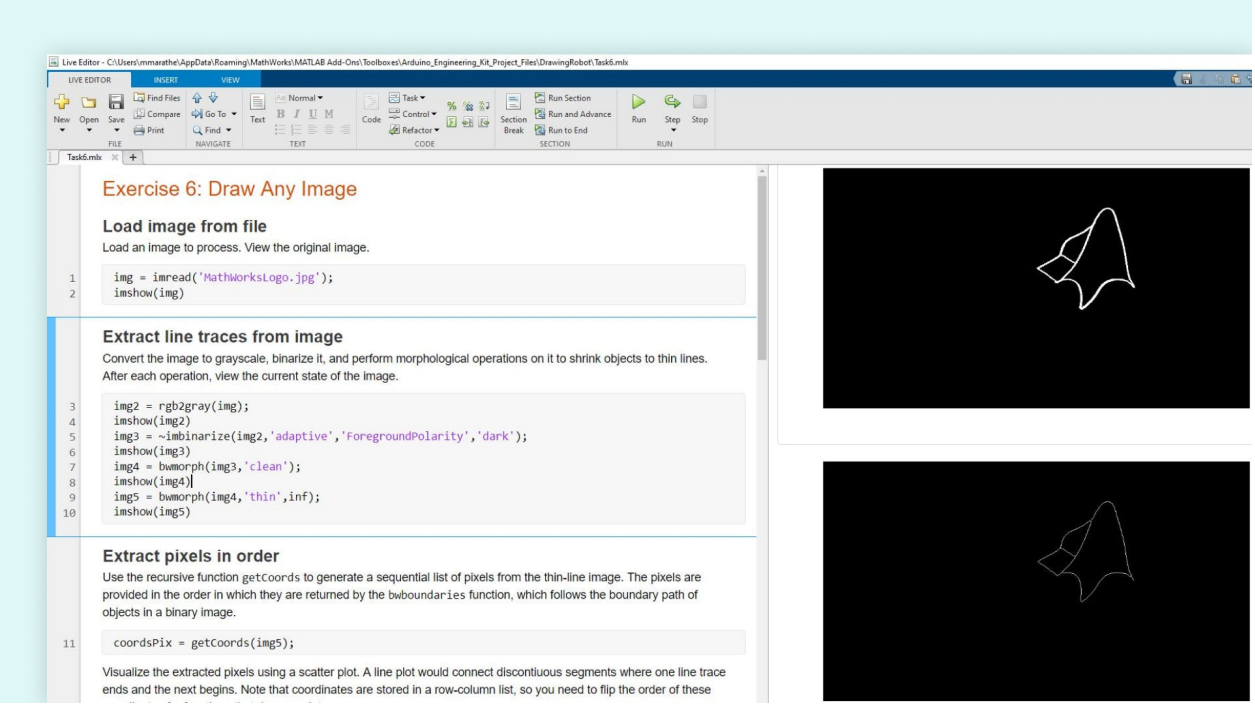
```
>> edit Task6
```

Execute the first section of the live script under the heading **Load image from file**.



Extract Line Traces from Images

Process the image so that it contains only thin traces of lines. Use functions from the Image Processing Toolbox to convert the image to grayscale, convert it to binary black and white, remove isolated pixels, and then thin out objects to lines. Perform these steps by executing the section of code in the live-script **Task6.mlx** under the heading **Extract line traces from image**. The different outputs obtained after executing this section of code shows what the image looks like after each step.



Understand Recursive Functions

So far, you should be familiar with MATLAB functions that take inputs and return outputs. Sometimes another instance of a MATLAB function can be called from within the same function. This is called a **recursive** function. Recursive functions can be useful when you want to solve a problem by iteratively breaking it down into simpler versions of itself. When you get to the simplest or the base case, you return the simplest value. To illustrate how this works in practice, consider how you compute the factorial of a number. This is defined by a product series:

$$n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

Note that this definition contains within it the definition of $(n-1)!$

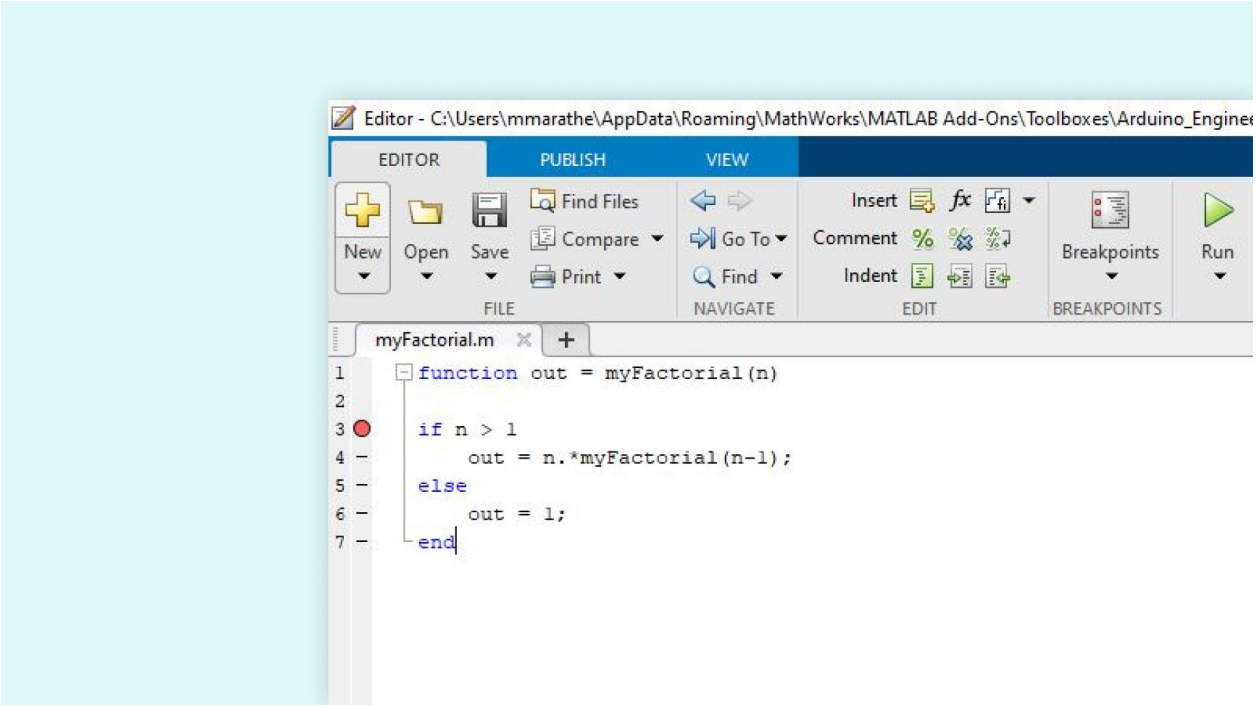
$$n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n$$

Therefore, one possible way to compute $n!$ is to iteratively compute $(n-1)!$ until you reach the base case where n = 1. In MATLAB, a recursive function could be written as follows to perform these steps:

```
function out = myFactorial(n)
if n > 1
out = n.*myFactorial(n-1);
else
```

```
out = 1;
end
```

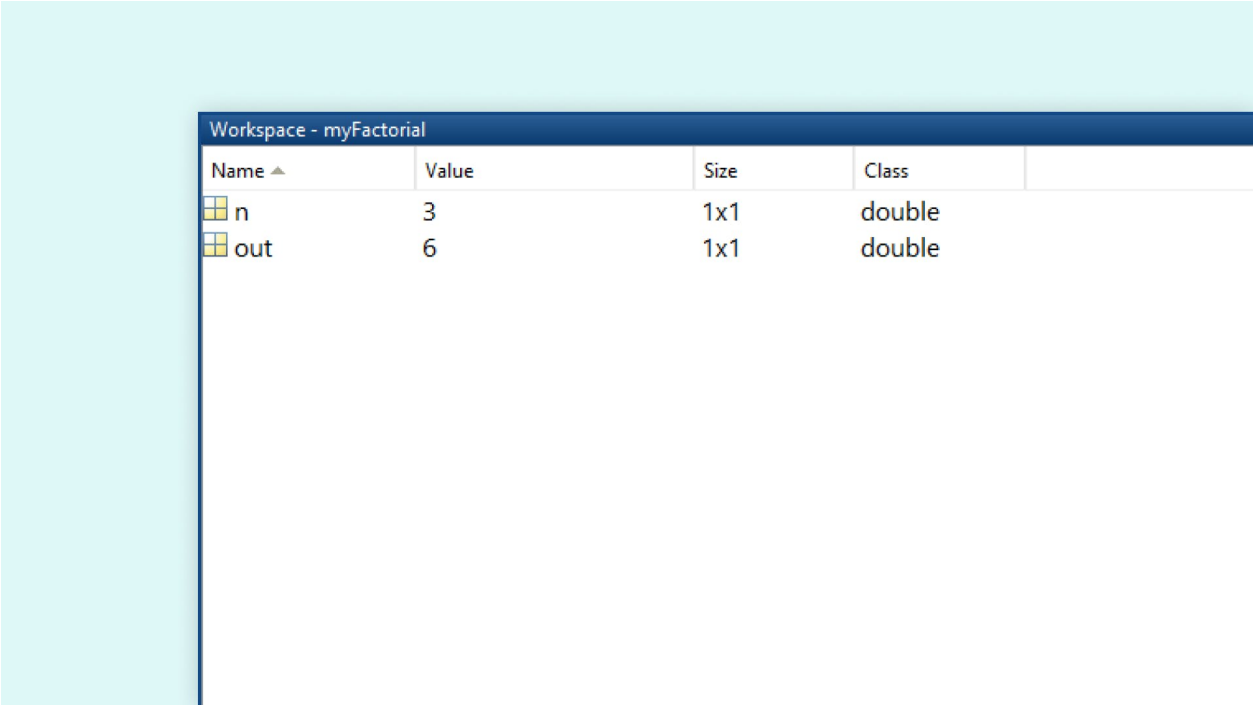
This function multiplies `n` with the factorial of `(n-1)`. In order to compute the factorial of `(n-1)`, MATLAB creates a new instance of the `myFactorial` function. This repeats until the function reaches the computation of factorial 1. Now, instead of creating a new recursive function call, it simply returns 1 and then steps back along the stack until the result is computed and returned for the first call to the `myFactorial` function. To see how this works, implement the above function in MATLAB and add a breakpoint to the first line by clicking on the dash (-), next to the line number.

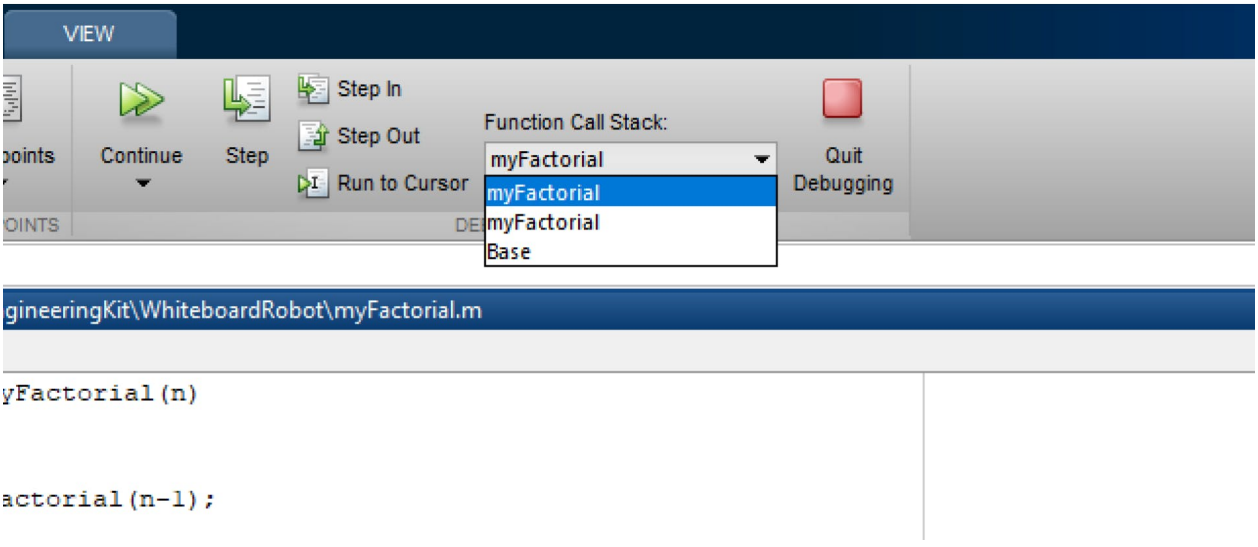


Now call the function from the MATLAB command prompt:

```
>> myFactorial(4)
```

The debugging breakpoint will be hit. Click the **Step** button on the **Editor** toolstrip to move through the code and keep hitting the breakpoint on subsequent instances of the `myFactorial` function that are called. After each step, check the **Workspace** to see the current values of `n` and `out`. You can also check the **Function Call Stack** on the **Editor** toolstrip to see which instance of `myFactorial` is currently running.





Note: This is just an illustration of recursion and is not the most efficient way to compute a factorial. In MATLAB, an easy way to do this is to simply use the prod function on the full sequence 1 to n.

```
>> prod(1:4)
```

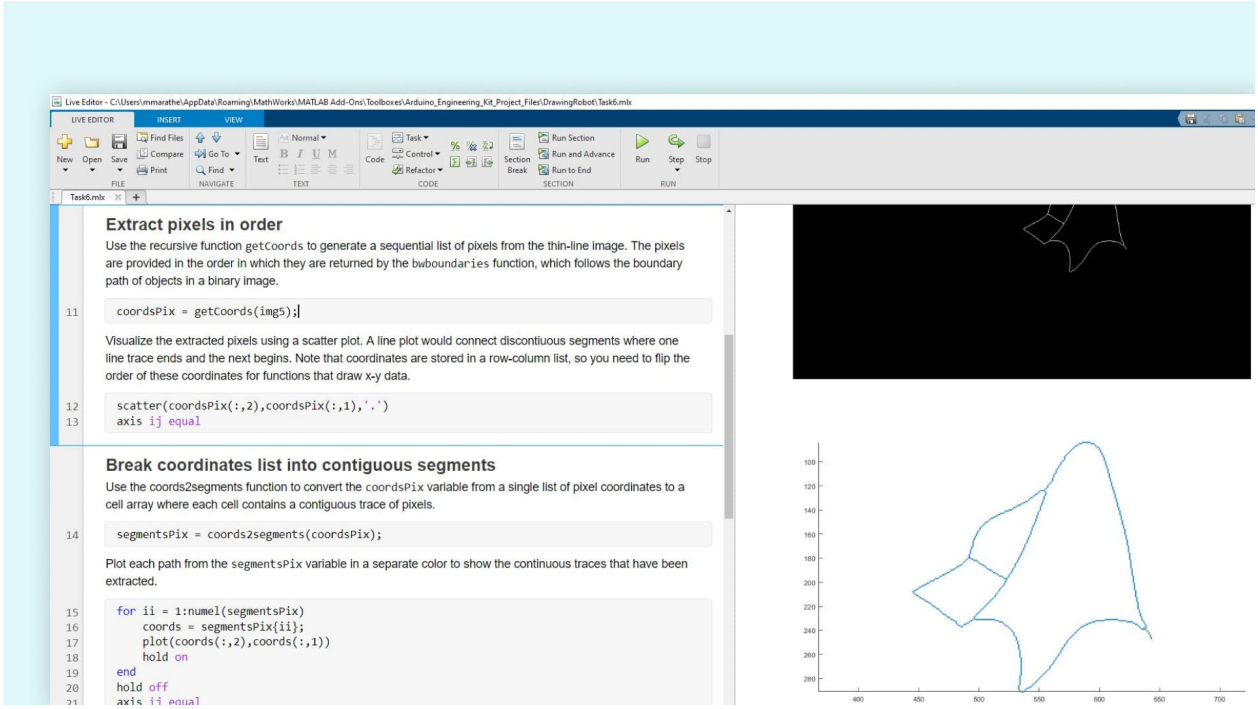
The getCoords() Function line-by-line

The function `getCoords` has been provided to you to help extract traces from an image in the form of pairs of coordinate values that constitute the various lines and curves. It's a recursive function that continues to call itself while there are still pixels available in the binarized image and then concatenates them into a single array. This function, in turn, calls the Image Processing Toolbox function `bwboundaries` recursively, stripping off the outer pixel boundaries from the image each time and concatenating them.

```
function curvePoints = getCoords(shapeImage)

% First, call the function bwboundaries to obtain the boundaries of each
unique region.
[curves,~,N] = bwboundaries(shapeImage);
curves = curves(1:N);

% Ignore hole boundaries
% Get the pixel points from the boundary detectedcurvePoints =
cell2mat(curves);

% Remove all duplicate points from the curve
curvePoints = unique(curvePoints,'rows','stable');

% In the image you're analyzing, remove all the pixels that were
captured by bwboundaries. There may still be pixels remaining in the
image.

% Remove curves from the image
curveInd = sub2ind(size(shapeImage),curvePoints(:,1),curvePoints(:,2));
```

Recursively Extract Pixels in Order

Now, you can call `getCoords` on the image to extract the selected pixels from the image in an order that puts the adjacent pixels next to one another. The output will be an Nx2 array of pixel coordinates that contains the result of each iteration of `getCoords` stacked on top of one another. `N` is the number of pixels identified in the image. You can also easily visualize these pixel coordinates using a scatter plot. Execute the section of code in the live-script Task6.mlx under the heading **Extract pixels in order**.



Understanding the coords2segments() Function

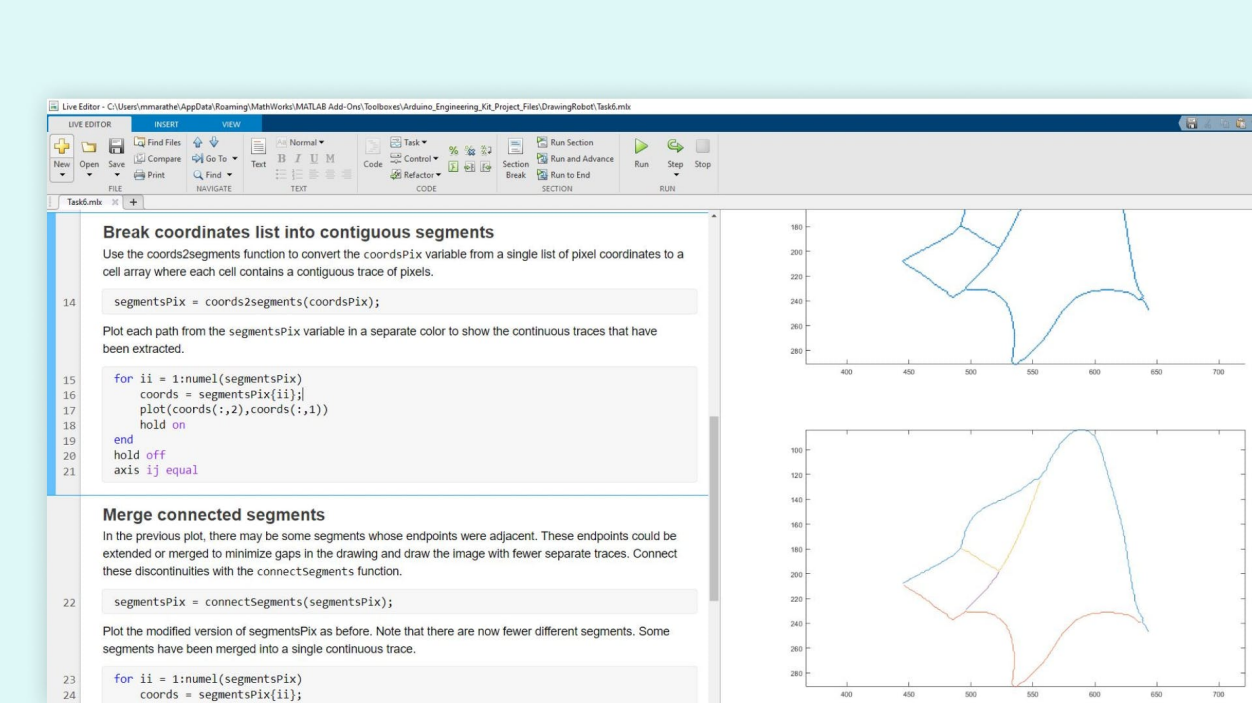
Given a sequence of data pairs obtained with the `getCoords` function, we need to identify the different sets of pixels. This is done by checking each pair of pixels to see if they are adjacent and then splitting them off if they are not. All the resulting segments get stored in a single cell array.

This function looks at every pair of pixels and determines locations where they are not adjacent. It then splits `coordsPix` into sets of contiguous pixels, storing each in a separate cell as elements of the cell array `segmentsPix` .

```
function segmentsPix = coords2segments(coordsPix)

% First, find the locations where consecutive pixels are not adjacent.
Create a breaks variable that is a logical index with value 1 where
there is a break and 0 where pixels are adjacent.
consecutiveDistance = abs(diff(coordsPix));breaks =
any(consecutiveDistance > [1 1],2);

% Use the breaks variable to construct the start and end indices of each
segment. Then loop through the segments and create a cell array
segmentsPix, where each cell contains all the pixel coordinates for that
segment.

% Build cell array of each segment of adjacent pixel coordinates
numSegments = sum(breaks)+1;
segmentsPix = cell(numSegments,1);
breakInds = [0; find(breaks);
size(coordsPix,1)];
for ii = 1:numSegments
segmentsPix{ii} = coordsPix(breakInds(ii)+1:breakInds(ii+1),:);
```

Break Coordinates lists into Continous Segments

Next, you will call `coords2segments` on the array of pixel coordinates extracted in the previous step. This will return a cell array where each element is a list of contiguous pixels. Plot each group of pixels to see which ones are organized together. Execute the section of code in the live-script **Task6.mlx** under the heading **Break coordinates list into contiguous segments**.



Notice how the red and blue traces share a common endpoint at the leftmost end. In the next section, we'll see how to combine these into a single line trace.

Understanding the connectSegments() function

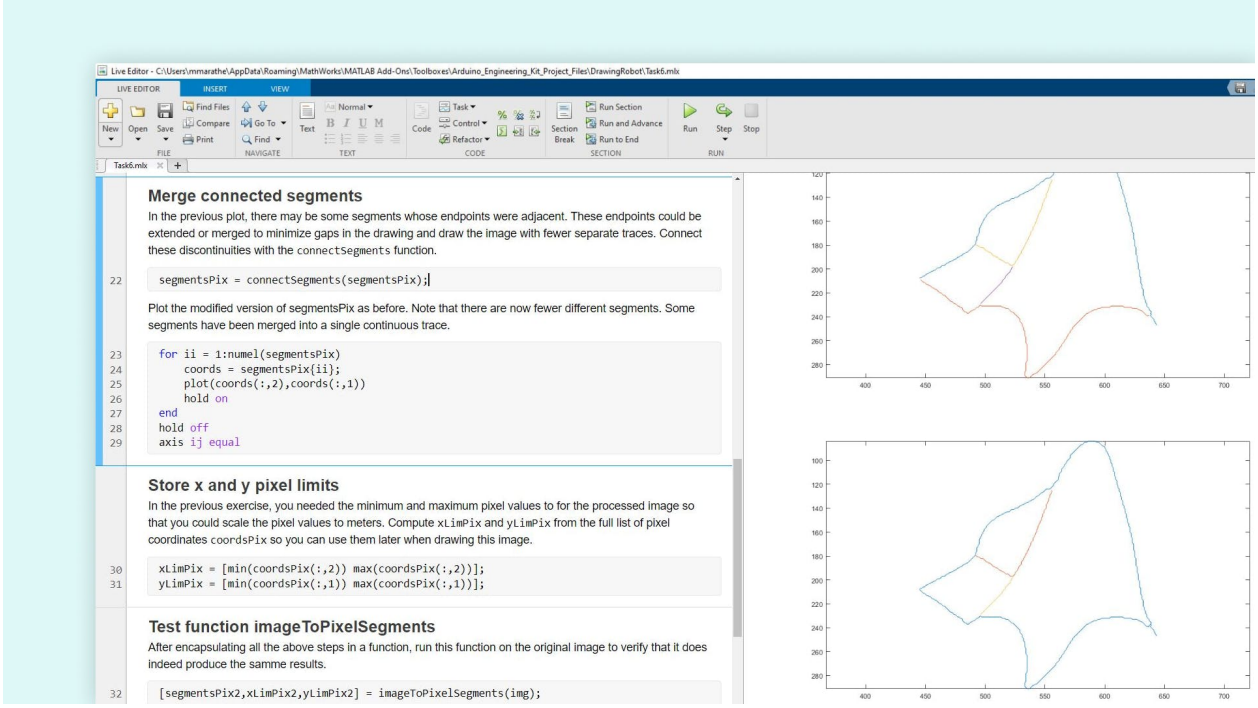
Given the results from the previous function, there might be segments with adjacent endpoints that could be merged with one another. This could reduce the total number of segments in the image. The function `connectSegments` does two things. First, it closes any segments that intersect themselves. Secondly, it merges any segments whose endpoints are adjacent.

```
function segments = connectSegments(segments)
% If a segment intersects itself, such as in the letter "O," it will end
at a point that is adjacent to another pixel in that segment. This will
create a small gap when drawn on the whiteboard. To correct this, add
the final adjacent pixel to the end of that segment so the line is
```

```
closed. This section of code looks at each segment, finds any self-
intersections, and closes them. p1 is the first point in a segment. pN
is the last point in the segment. Add any points near p1 to the
beginning of the segment. Add any points near pN to the end of the
segment.

% Close any segments that self-intersect
for ii = 1:length(segments) points = segments{ii};
  p1 = points(1,:);
  pN = points(end,:);

  % Add any points near p1 to beginning of segment ii
  nearP1 = isadjacent(p1,points(4:end,:));
```

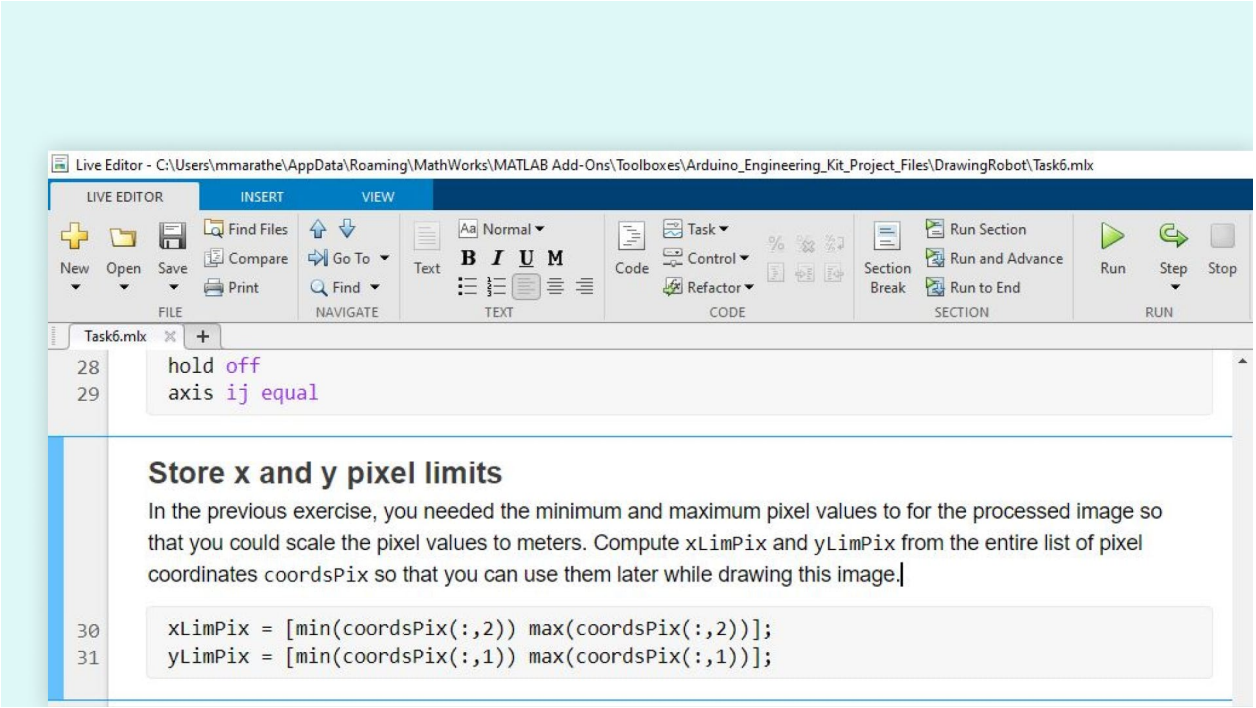
Merge Connected Segments and Get Limits

Now call the `connectSegments` function on the cell array of pixel groups created in the previous step. This will return a new cell array of pixel groups. In this new cell array, some of the groups may be merged if they can be connected to form one path. Each group of pixels in this cell array is a path that will be drawn by the robot, which will lift the marker before drawing the next path. Plot each group of pixels to visualize all the paths that the robot will traverse. Execute the section of code in the live-script **Task6.mlx** under the heading **Merge connected segments**.



Additionally, create new variables `xLimPix` and `yLimPix` as a quick way to store the maximum and minimum pixel positions for use when drawing the image on the whiteboard. Remember, they are needed while estimating the extent to scale

the image to fit on the whiteboard. Execute the section of code in the live-script **Task6.mlx** under the heading **Store x and y pixel limits**.



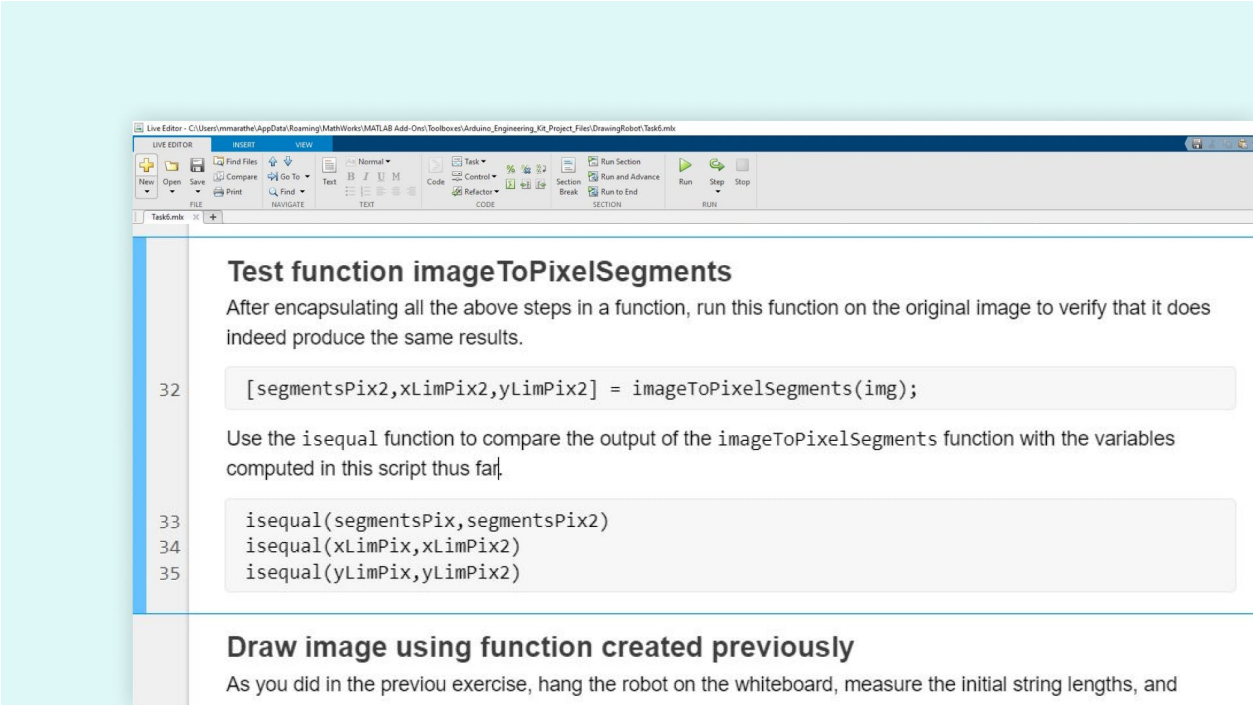
Write a Function To Convert Image To Pixel Segments

The code that you've executed until this section takes an image and generates groups of pixel positions. You can encapsulate all these steps in a single function to easily perform them on any image. The function will return the variables `segmentsPix`, `xLimPix`, and `yLimPix`, that you computed earlier. Create a new MATLAB function and call it `imageToPixelSegments()`. Include the following code in the function and save it as **imageToPixelSegments.m**.

```
function [segmentsPix,xLimPix,yLimPix] = imageToPixelSegments(img)

% Extract line traces from the imageimg2 = ~imbinarize(rgb2gray(img),
'adaptive' , 'ForegroundPolarity' , 'dark' );
img3 = bwmorph(img2, 'clean' );
img4 = bwmorph(img3, 'thin' ,inf);

% Extract pixels in order
coordsPix = getCoords(img4);

% Break coordinates list into contiguous segments
segmentsPix = coords2segments(coordsPix);

% Clean data and merge connected segments
segmentsPix = connectSegments(segmentsPix);
```

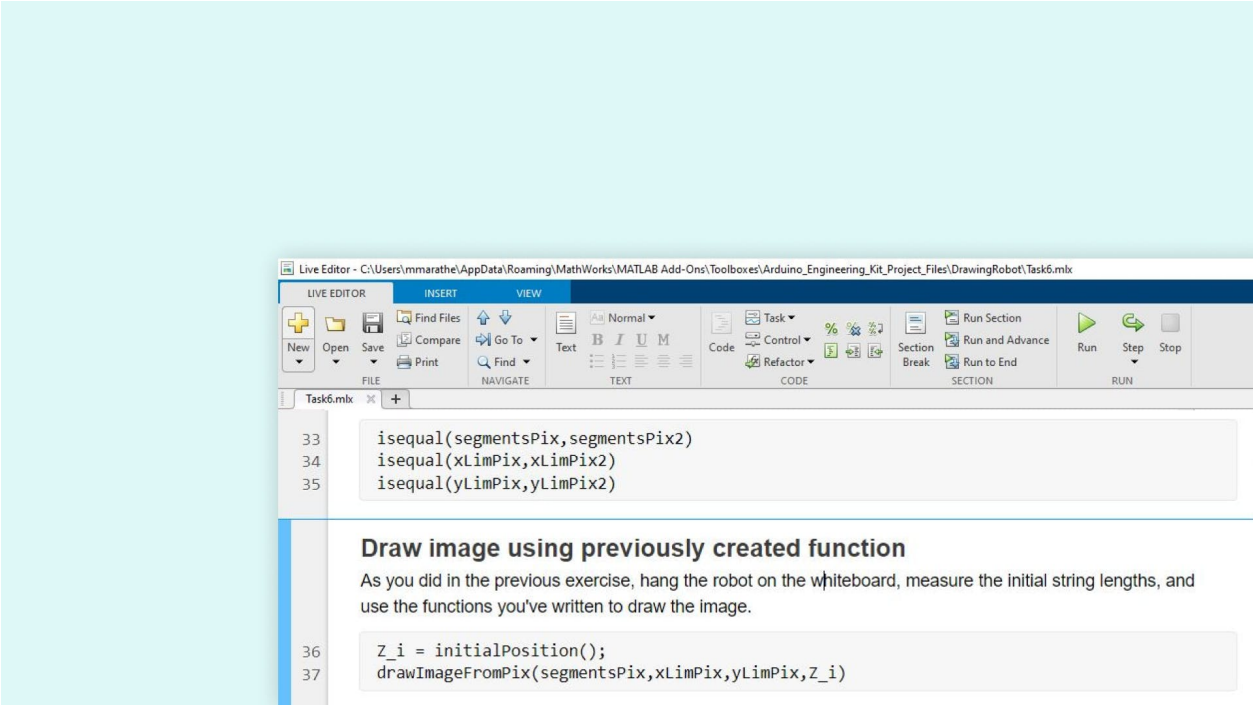
```
% Store x and y pixel
limitsxLimPix = [min(coordsPix(:,2)) max(coordsPix(:,2))];
yLimPix = [min(coordsPix(:,1)) max(coordsPix(:,1))];
```

Now, check that the code in the `imageToPixelSegments` function behaves in the same way as the code you've executed so far. To perform this, execute the function and compare the result to the `segmentsPix`, `xLimPix, and` yLimPix` variables in your workspace. Execute the code in the live-script **Task6.mlx** under the heading **Test function imageToPixelSegments**.



Draw image on whiteboard

In the previous exercise, you wrote a function `drawImageFromPix` that takes a cell array of pixel paths and draws them on the whiteboard using the robot. Now, you can call this function to recreate the MathWorks logo image on the whiteboard. Execute the section of code in the live-script **Task6.mlx** under the heading **Draw image using previously created function.**



Write main script to execute complete workflow

To run an application that involves many steps, it is a common practice to create a main script. This is a script that operates at the highest level and executes an entire workflow from start to finish. We can create a main script to perform all the tasks that you executed in this exercise, relying on the functions you've created thus far.

Create a new empty MATLAB script. Add the following code and save the file as **main6.m**.

```
% Load image from file
img = imread( 'Images/MathWorksLogo.jpg' );

% Convert image to pixel
segments[segmentsPix,xLimPix,yLimPix] = imageToPixelSegments(img);

% Identify initial position on whiteboard
Z_i = initialPosition();

% Draw image
drawImageFromPix(segmentsPix,xLimPix,yLimPix,Z_i)
```

Files

- Task6.mlx
- getCoords.m

- coords2segments.m
- connectSegments.m
- imageToPixelSegments.m
- main6.m

Learn by Doing

Use your understanding of recursive functions to write a new MATLAB function that will recursively compute the nth value of the Fibonacci sequence. Recall that the nth Fibonacci number is the sum of the previous two Fibonacci numbers and the first two numbers are 0 and 1. Your recursive function should have two stopping conditions (for when n = 1 or n = 2). For all other cases, your function should recursively call itself twice.

In this exercise, you are loading an image from a file, processing it, and then drawing it immediately. You may instead want to just do the processing on your images, preview the extracted line traces, and store this pixel data in a MAT-file to load and draw later. Write a function that will do this. When saving the data to a MAT-file, you will need to specify a filename. There are two ways you can specify it. One is to choose the name based on the filename of the image. Use **matlab.lang.makeValidName** to ensure the name is valid. The second option is to prompt the user for a name using the **uiputfile** dialog-box.

In your **main6.m** script, you have hardcoded the name of the image file to process. To make the script more user-friendly and allow you to draw different types of images, you may want to change this so that the user can interactively select which image to draw. Modify the beginning of **main6.m** so that it prompts the user to select a file, rather than defining it in the code. You can use the **uigetfile** dialog for this.

4.7 Capture and Draw Live Images

By now, your drawing robot is pretty capable. It can take any image file and reproduce the image on a whiteboard. This entire workflow is automated in a code. However, there are still some manual steps that involve capturing the image, transferring the file to your computer, and then specifying the file as the one you want to draw. In this exercise, you'll automate these steps as well by using a webcam to capture the image and bring it directly into MATLAB to analyze and draw with the robot.

In this exercise, you will learn to:

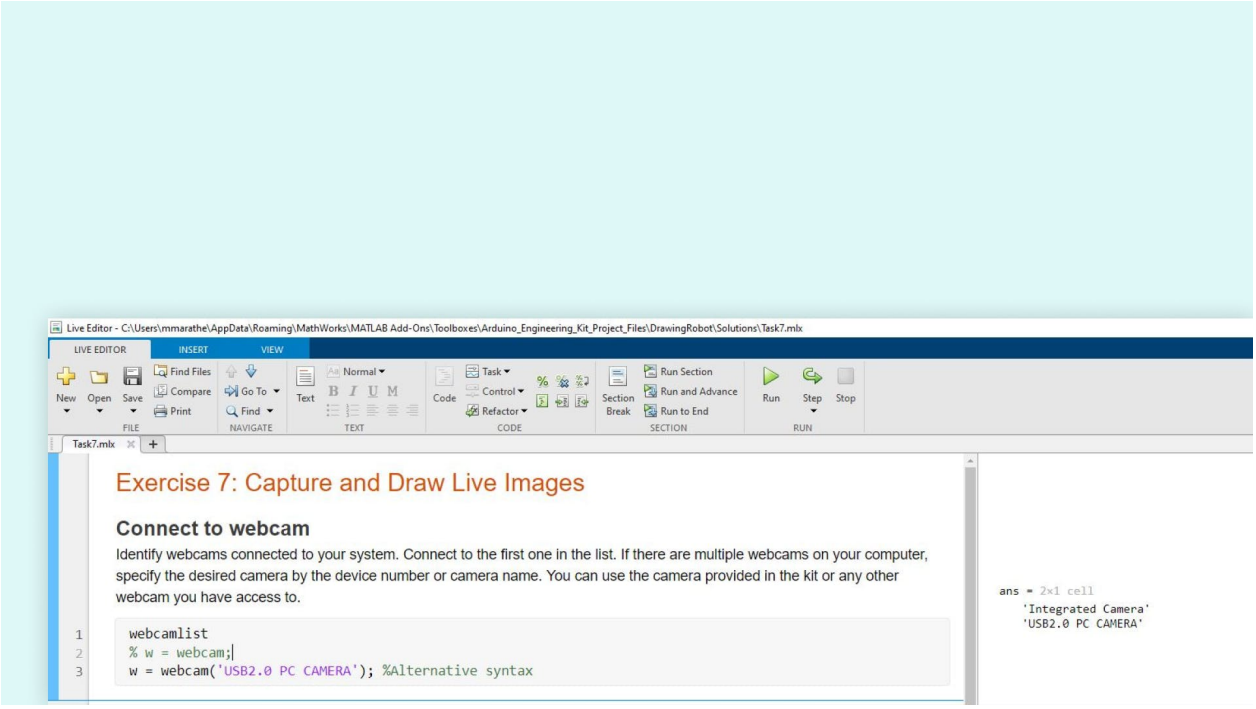
- Connect to a webcam
- Update a main script

Connect to Webcam

In addition to Arduino devices, MATLAB can talk to many other types of hardware. To acquire images directly into MATLAB, you can use a USB webcam. MATLAB's webcam interface can connect to webcams, preview what they see, and take snapshots that bring the current image directly into MATLAB. This functionality requires **MATLAB Support Package for USB Webcams**, which you should have installed while setting up all your MATLAB and Simulink software. If you haven't yet installed the support package, you can do so from the **Add-Ons** option on the **Home** tab inside MATLAB. Locate the USB webcam that came along with the kit and connect it to your computer. Open the live script **Task7.mlx**.

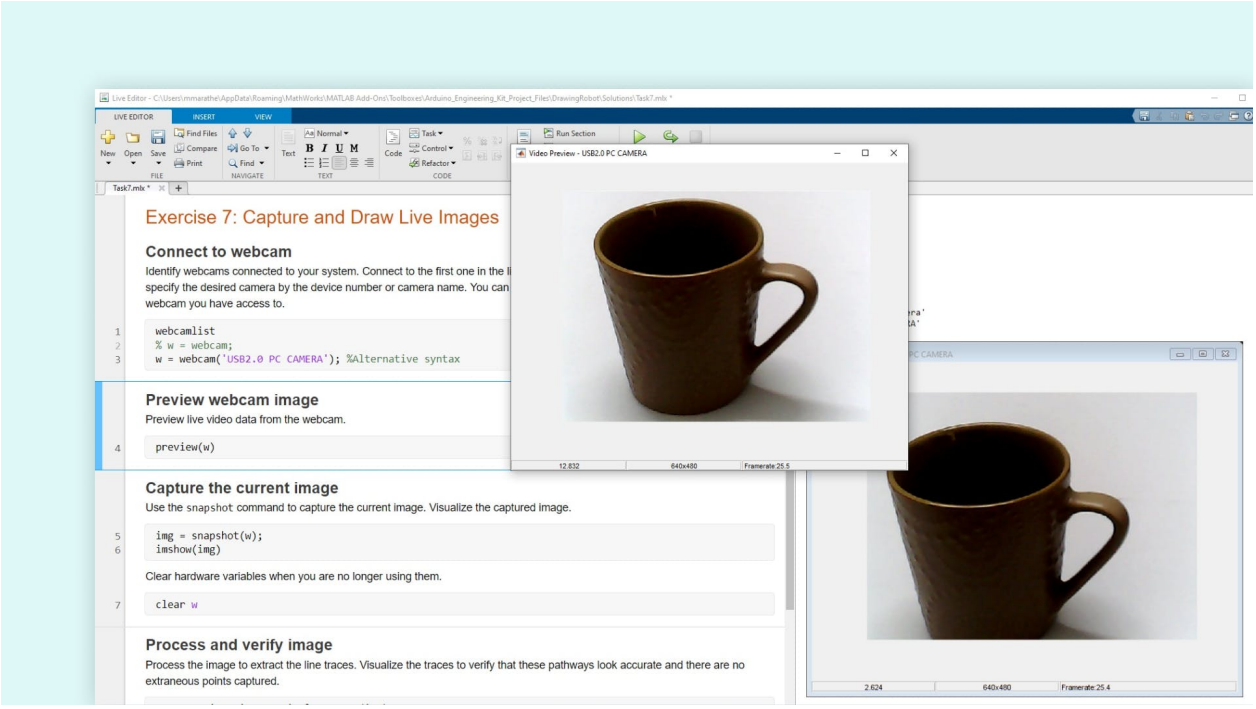
```
>> edit Task7
```

To identify the webcam in MATLAB and connect to it, execute the section of code under the heading **Connect to webcam**.



Preview Webcam Image

Preview the webcam image in MATLAB by executing the section of code under the heading **Preview webcam image**.

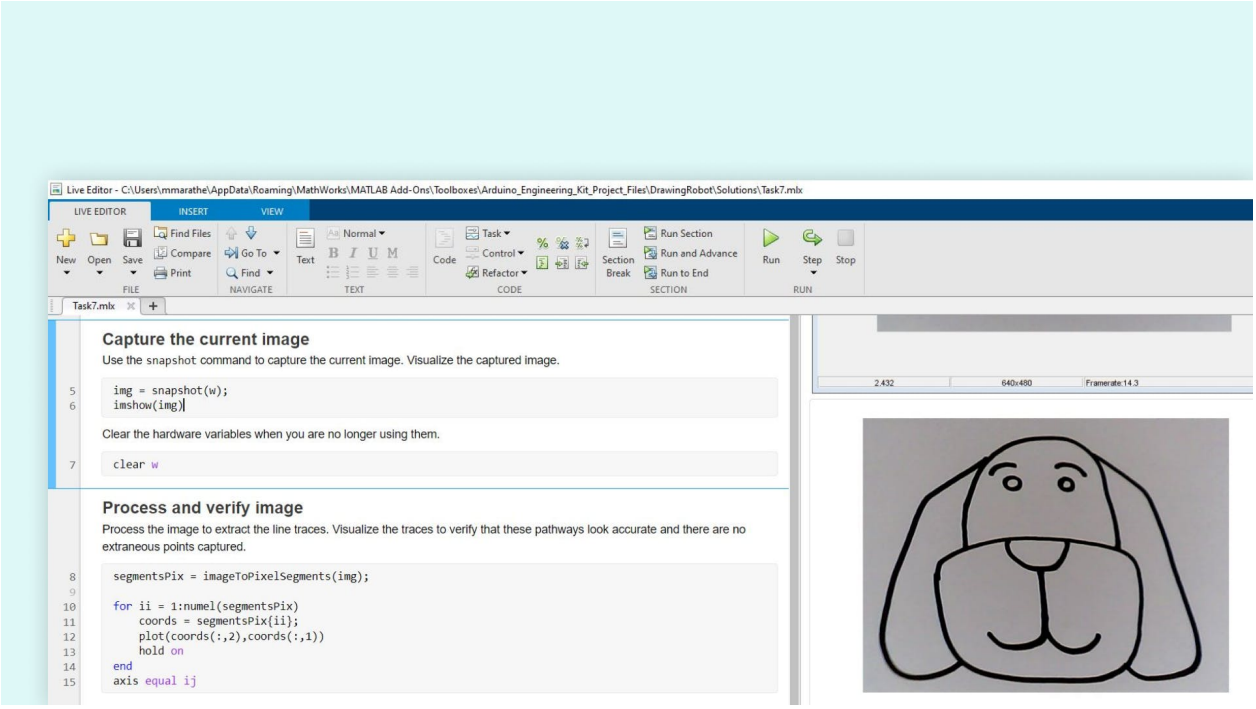


Move the webcam around to make sure it can get a clear view of the images in focus.

Note: If you are using the camera provided in the kit, then you will be able to adjust the focus of the camera manually by rotating the lens-piece on the camera.

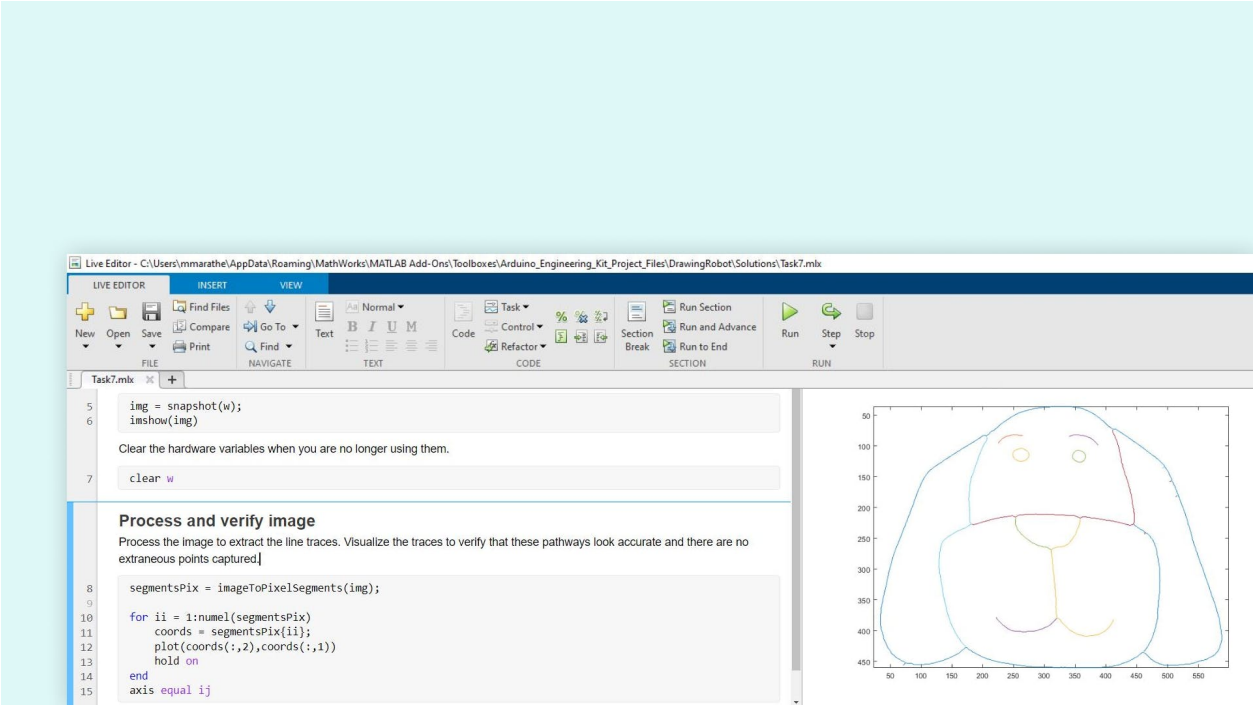
Capture Current Image

Using a black marker, create a simple line drawing on your whiteboard that you'd like your robot to recreate. Aim the camera at the whiteboard and check out the preview window to make sure that the drawing is clear, there is nothing else in the view of the camera, and that there is no glare or uneven lighting on the whiteboard. Alternatively, you can use a pen drawing on a white piece of paper. Capture an image of the drawing by executing the section of code in **Task7.mlx** under the heading **Capture the current image**.



Process and Verify Image

To make sure that your image will be drawn correctly by the robot, run your image processing function `imageToPixelSegments`, and verify that the extracted paths from the image are as expected. Execute the section of code in the live-script **Task7.mlx** under the heading **Process and verify image**. If the processed image does not look correct, capture a new image and try again.



Update Main Script for Live Images

In the previous exercise, you created a main script to run the entire application altogether. The script loaded the desired image from a file. Now, you'll modify that script to obtain the image from a camera. Open the file **main6.m**. Save a new version of the file as **main7.m**. Delete the first line of code from **main6.m**, which loads the image from a file. Replace it with a few lines of code that will connect to the webcam, preview the image, and wait until you strike a key before taking a snapshot of the current image and continue with the rest of the script. The final **main7.m** script should look as follows:

```
% Capture image from webcam
% w = webcam;
w = webcam('USB2.0 PC CAMERA');  % Alternative syntax
preview(w)
pause
img = snapshot(w);
clear w

% Convert image to pixel
segments[segmentsPix,xLimPix,yLimPix] = imageToPixelSegments(img);

% Identify initial position on whiteboard
Z_i = initialPosition();

% Draw image
drawImageFromPix(segmentsPix,xLimPix,yLimPix,Z_i)
```

Execute Main Script to Draw Image Captured Live

Now, try it out with your robot. Execute the following command at the MATLAB command prompt to run your application.

```
>> main7
```

A webcam preview window will open. Aim your camera at a white region of the whiteboard with a simple line drawing on it. Make sure nothing else is in the frame of the image and there is no glare or color variation across the image. Strike any key in MATLAB to continue. The robot will scale the captured image to the full whiteboard and attempt to reproduce it.

Files

- Task7.mlx
- main7.m

Learn by Doing

At the end of this exercise, you should be able to acquire an image from a USB webcam and immediately draw it with your robot. Another possible application for the webcam might be to acquire a whole set of images of whiteboard drawings to reproduce later with the robot. Starting with the code you already have, create a script that can acquire a webcam image, prompt the user for a filename, and save the image with that name. Refer to the functions `uiputfile` and `imwrite` for help with these tasks.

4.8 Lessons Learned

In this chapter, you were introduced to the following topics:

Applying Concepts from Trigonometry to Control The Movements of the Robot

In this project, you learned how to decompose movements through trigonometry to calculate the distances that allow the robot to draw on the whiteboard. You combined trigonometry with equations of motion to move the robot to specific points on your whiteboard. You did it all using MATLAB scripts and the information on the motors from its respective datasheet.

Limitations of the Robot Movement

After knowing how to program the robot movements, you learned about some practical limitations of drawing using the robot and where they exist. You learned why the robot cannot move in certain regions and which areas are preferable to recreate the drawings.

Image Processing

You also learned how to process an image to extract the relevant data necessary to make the drawings on the whiteboard. To achieve this, you learned how to apply some image filters, how to obtain the line traces from the image, how to convert the resulting pixels to meters and then finally how to build segments to create the trajectory that the robot will follow.

4.9 Final Challenge

Throughout this project, you've drawn images on the whiteboard using only one marker. However, your drawing robot can hold two markers. It would be nice if you could draw images that have more than one color. How might you go about doing this?

The simplest approach would be to alternate colors for each line segment. Another approach might be to group the objects in the image based on how close they are to one another and draw each group with the same color. Or, combine two different images and draw each in a different color. Maybe you could identify line traces of different colors within a single image and draw them separately. How would you do that using image processing? Try using the `Color Thresholder` app from **Image Processing Toolbox**. Your final challenge is to come up with an interesting way to create drawings that use multiple marker colors. You can try one of the ideas listed above or come up with your own. Implement the solution in MATLAB, and then draw a multicolor image with your robot.