WEBCAM CONTROLLED ROVER

Build and program a webcam controlled rover to follow paths, move objects with a forklift, and avoid obstacles. You will learn about differential drive robots and how to simulate their behavior, control their position or speed, and perform localization.

YOU WILL LEARN ABOUT:

ROBOT MOVEMENT, DIFFERENTIAL DRIVE, KINEMATICS, PATH FOLLOWING ALGORITHM, WIFI COMMUNICATIONS, IMAGE PROCESSING, SIMULATIONS

5.0 Project Overview

The Webcam Controlled Rover project consists of a programmable robot that can communicate with MATLAB over Wi-Fi to perform operations using an image processing algorithm. The Rover is built using an Arduino Nano 33 IoT, the Arduino Nano Motor Carrier, two DC motors with encoders and a micro-servo motor. On top of the Rover, you will install a colour-coded sticker that will serve as a marker to assist the image processing algorithm that uses a webcam to detect the location and orientation of the robot.

In this project, you will learn to:

- Program the Rover to perform a variety of tasks, such as following paths, moving objects with the forklift, and avoiding obstacles,
- Use differential drive robots work and how to simulate their behavior,
- Control the position or speed of the robot and perform localization,
- Build robots to intelligently travel around and perform all kinds o´

Help

Project Assembly

Before you begin modeling and work on the project, you will need to assemble the Rover by following the instructions in the following video. The assembly of this project will take about **45 mins** approximately.

Note: The default wire length of the DC motor with encoder is necessary for the Drawing Robot project. It is advised not to shorten the wire length for the Webcam Controlled Rover project.

Help

5.1 Control the Rover and Forklift from MATLAB

The Webcam Controlled Rover is a differential drive robot that has separate DC motors controlling each of its wheels. You can control the direction of travel by varying the speed of each wheel without the need for any separate steering mechanism. In this exercise, you will learn how to drive the Rover and operate the forklift using MATLAB.

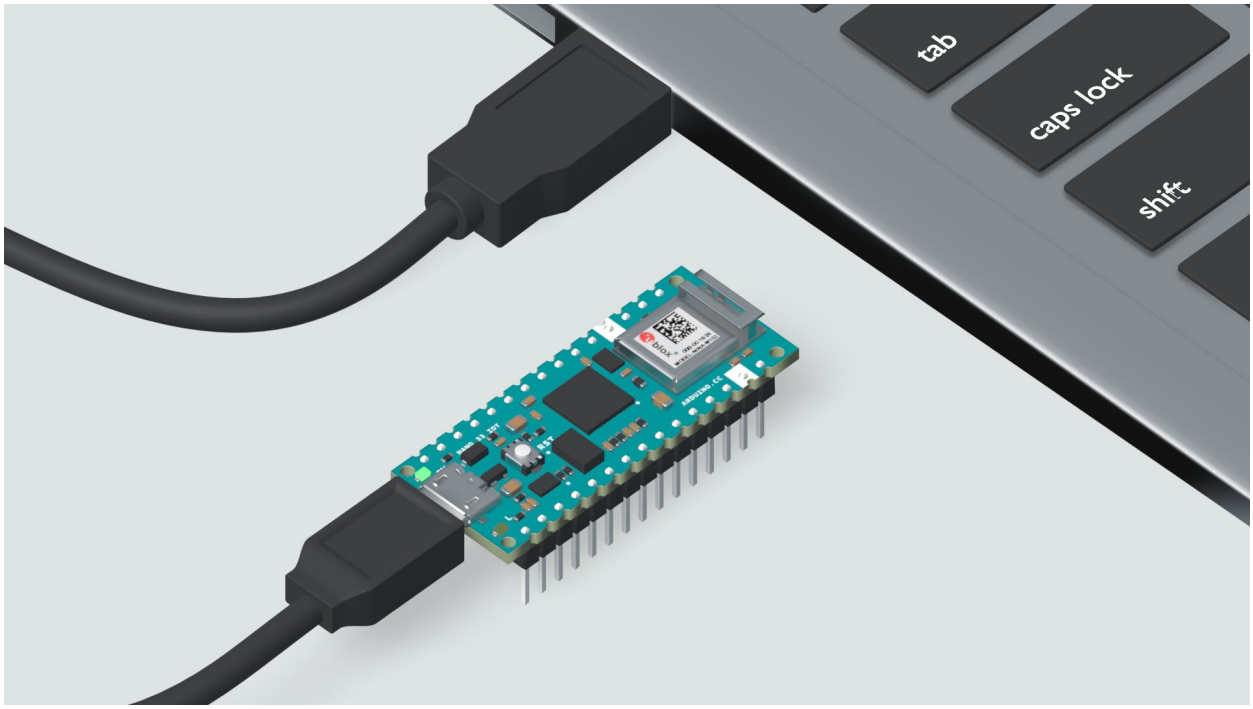
In this exercise, you will learn to:

- Control basic movement of a differential drive robot,
- Connect MATLAB to the Rover,
- Drive the Rover in a straight line,
- Drive the Rover in circles,
- Control the forklift of the Rover.

How to communicate with the Rover

Let's begin by connecting the Arduino Nano 33 IoT to your computer using the USB cable:

Help



To communicate between MATLAB and Arduino, you need to know the port number your Arduino is connected to. You can determine this using the steps described in section 2.2 “MATLAB Getting Started”, in Connecting MATLAB with Arduino. Remember that the port names differ between operating systems; in other words, they are different for Windows (where they are named COM## with the ## being a unique single or two digit numeric identifier), OSX (/dev/tty.usbmodem##), and Linux (/dev/ttyACM##).

Once you know the port number, enter the following commands with the appropriate port number to create (a) an Arduino object with the necessary libraries and (b) a separate object for the Motor Carrier:

```
>> a = arduino('COM##', 'Nano33IoT', 'Libraries', 'MotorCarrier');
>> carrier = motorCarrier(a);
```

Control the Rover From MATLAB

Drive in a straight line

You probably know intuitively that the Rover will travel in a straight lin[illegible] if [illegible] wheels spin at the same speed and in the same direction. However, s[illegible] two motors are oriented exactly opposite to each other on the Rover, they will need to be excited with opposite polarities, in order to move them in the same

Help

direction. This means that if one motor is commanded to move in the clockwise direction, the other motor will have to be commanded to move in the counter-clockwise direction, so that both the motors move the Rover in the same direction. Let's spin the Rover's DC motors at the same speed, but opposite polarities and observe whether it goes in a straight line as expected. Enter the following commands to connect the DC motors to MATLAB (note that the orientation perspective is observing from the back of the rover towards the front):

```
>> dcmLeft = dcmotor(carrier,'M2');
>> dcmRight = dcmotor(carrier,'M1');
```

Now specify that you want the DC motors to spin at 25% of the maximum speed:

```
>> dcmLeft.Speed = 0.25;
>> dcmRight.Speed = -0.25;
```

Once you've specified how fast the motors must spin, the next step is to spin them! Enter the following commands to run the motors for 5 seconds at this speed:

```
>> start(dcmLeft)
>> start(dcmRight)
>> pause(5)
>> stop(dcmLeft)
>> stop(dcmRight)
```

Measure how far the Rover moved when using these commands. Since we don't have feedback of the current position, there is no way to ensure that the Rover will either move completely straight, or that it will move the same distance each time the program is executed, given that there might be slight differences in behavior between the motors. In addition to this, as time goes by, the batteries will have less and less charge and the motors will behave differently.

Drive in a circle

Help

If you were asked how to drive the Rover in a circle, your answer might be a) to spin one wheel but not the other, or b) spin both wheels with the same speed

but in opposite directions.

Both answers are correct, though the way the Rover will move will differ. Let's analyze the first case:

Spin one wheel but not the other. Try the following commands with the Rover:

```
>> dcmLeft.Speed = 0;
>> dcmRight.Speed = 0.25;
>> start(dcmLeft)
>> start(dcmRight)
>> pause(5)
>> stop(dcmLeft)
>> stop(dcmRight)
```

You should observe that the Rover spins in a circle around its left wheel.

The other case required you to spin both wheels with the same speed but opposite directions. Now as discussed earlier, if we provide same polarities to both the motors, they will move in opposite directions due to their opposite alignment with respect to one another and vice versa. Let's try spinning the wheels with equal speeds by running the following code in MATLAB:

```
>> dcmLeft.Speed = 0.25;
>> dcmRight.Speed = -0.25;
>> start(dcmLeft)
>> start(dcmRight)
>> pause(5)
>> stop(dcmLeft)
>> stop(dcmRight)
```

You should observe that the Rover spins in a circle about its centre.

Control the Forklift of the Rover

The Rover's forklift is controlled by a standard servo motor connected to the Arduino Nano Motor Carrier board; this means you can control the angle at which the motor will turn. Typically, standard servos can turn between 0 and 180 degrees, but there might be other types.

Help

Note: The Arduino Engineering Kit Rev2 only provides standard servo motors.

In this section, first, you will move the servo to the **Up** position. This corresponds to the forklift (attached to the servo motor) being in the vertical position (90 degrees counter-clockwise from the horizontal plane). This is the highest achievable lift position of the forklift mechanism. If you have assembled your Rover properly based on the steps explained in the assembly video, then the **Up** position will correspond to 180 degrees on the servo motor.

The function **writePosition** accepts an input between 0 and 1 where 0 corresponds to 0 degrees and 1 corresponds to 180 degrees on the servo motor. You will use the command `writePosition(s,1)` to move the servo in the **Up** position. Similarly, the lowest position on the servo will be 0 degrees (90 degrees clockwise from the horizontal plane). However due to the geometrical limitations of the forklift mechanism, its not possible to achieve 0 degrees with the Rover. Hence the lowest achievable position will be higher than this value, and for the Rover, it turns out to be around 80 degrees on the servo (10 degrees clockwise from the horizontal plane). This corresponds to the **Down** position of the forklift mechanism, thus providing a range of 100 degrees for movement between the **Up** and **Down** positions (180 degrees **Up** position to 80 degrees **Down** Position on the servo). Enter the following commands to connect to the servo and move the forklift between the **Up** and **Down** positions:

```
>> s = servo(carrier,3);
>> writePosition(s,1)    % Up Position (180 degrees on the servo)
>> pause(5)
>> writePosition(s,0.45) % Down Position (80 degrees on the servo)
```

Review

- We have seen how to drive the Rover in a straight line and in circles.
- We have seen how to control the forklift, which will be used in later exercises to lift the target.

Help

Learn By Doing

- Drive the Rover in a circle of a different radius by sending different speeds to both wheels.
- Drive the Rover in a sequence: move forward, then turn left 90 degrees in place, and then move forward again.

Help

5.2 Simulating Rover Kinematics and Open Loop Control

In section 5.1, you learned how to make the Rover drive in a straight line and in circles. However, to move the Rover along more complex paths you need to understand the underlying kinematic equations that relate the speed of each wheel to the Rover velocity and direction of travel. In this lesson, you will learn the basic kinematic equations behind differential drive robots and how Simulink can help you visualize and simulate the Rover's movement for various inputs. You will then apply what you learned through these simulations to control the Rover in the real world.

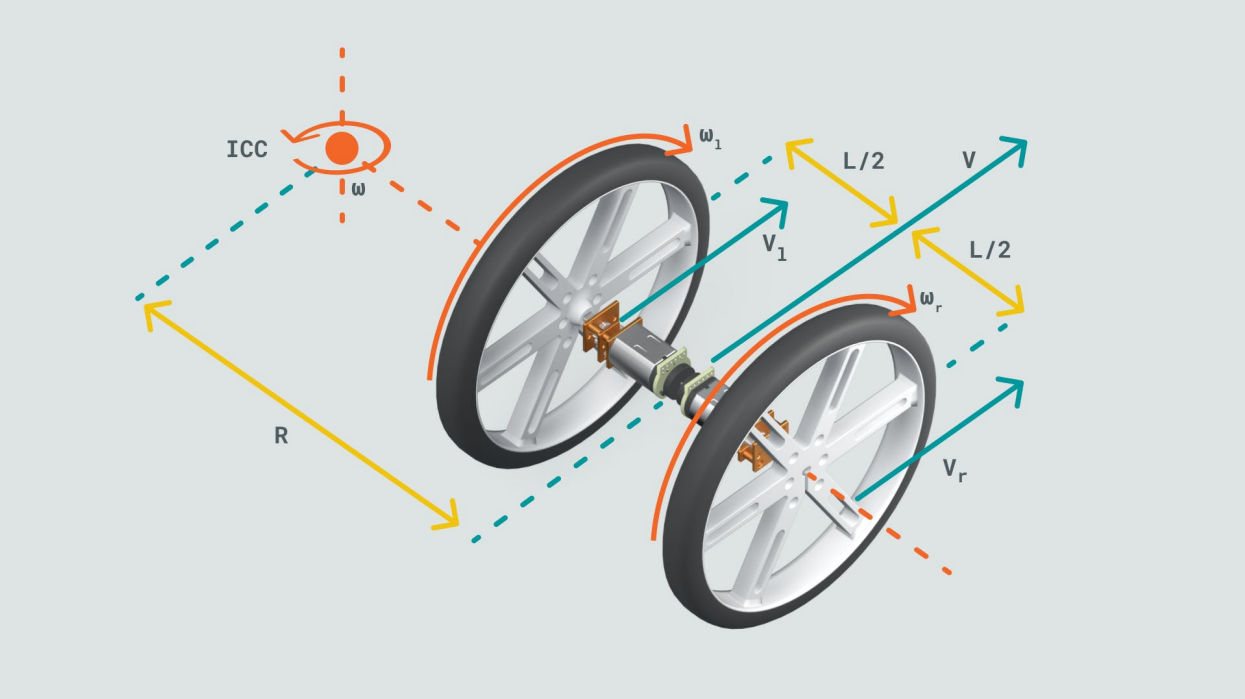
In this exercise, you will learn to:

- Understand kinematic equations of the Rover,
- Simulate the Rover using kinematic equations,
- Model and simulate Rover movement using Simulink,
- Perform open-loop control of the Rover,
- Use Simulink models to control the actual Rover.

Kinematics of the Rover

The following diagram shows the basic kinematics of the Rover (or any differential drive robot):

Help



The diagram introduces a few important parameters associated with differential drive robots, including radius of rotation (R), rate of rotation (ω), instantaneous center of curvature (ICC), wheel velocities (v_l, v_r), distance between the wheels (L) and the forward velocity (v). If you want to learn more about the concepts presented in this diagram, please refer to **this page**.

Let's begin by focusing on the rate of rotation and forward velocity.

Rate of rotation: $\omega = \frac{v_r - v_l}{L}$

Forward velocity: $v = \frac{v_r + v_l}{2}$

Wheel velocities (v_l , v_r) can be calculated using the wheel radius (r) and rotational speeds (ω_l, ω_r):

$v_l = \omega_l \cdot r$

$v_r = \omega_r \cdot r$

Plugging these values into the previous equations yields:

$\omega = \frac{r \cdot (\omega_r - \omega_l)}{L}$

Help

$v = \frac{r \cdot (\omega_r + \omega_l)}{2}$

Rearranging these equations:

$\omega_r + \omega_l = \frac{2v}{r}$

$\omega_r - \omega_l = \frac{L \cdot \omega}{r}$

Solving for ω_r and ω_l:

$\omega_r = \frac{v + (\frac{L}{2})\omega}{r}$

$\omega_l = \frac{v - (\frac{L}{2})\omega}{r}$

In matrix notation:

$$\begin{bmatrix} \omega_l \\ \omega_r \end{bmatrix} = \frac{1}{r} \cdot \begin{bmatrix} 1 & \frac{-L}{2} \\ 1 & \frac{L}{2} \end{bmatrix} \cdot \begin{bmatrix} v \\ \omega \end{bmatrix}$$

For our Rover, L = 8.5 cm and r = 4.5 cm. Using these values and the above equations, let's calculate how fast we need to spin each wheel to move the Rover at a desired forward velocity and rate of rotation. Let's look at two simple scenarios:

1) Drive the Rover straight forward at 10 cm/s (v = 10 cm/s, ω = 0 deg/s)

ω_r = ω_l = 127.3 deg/s (includes unit conversions).

2) Spin the Rover at a rate of 72 deg/s with zero forward velocity (v = 0 cm/s, ω = 72 deg/s)

ω_r = 68 deg/s, ω_l = -68 deg/s (includes unit conversions).

However, since the left and right motors are oriented exactly opposite to each other in our Rover, ω_r = -127.3 deg/s in the first case and ω_r = -68 deg/s in the second case.

Implementing Equations in Simulink

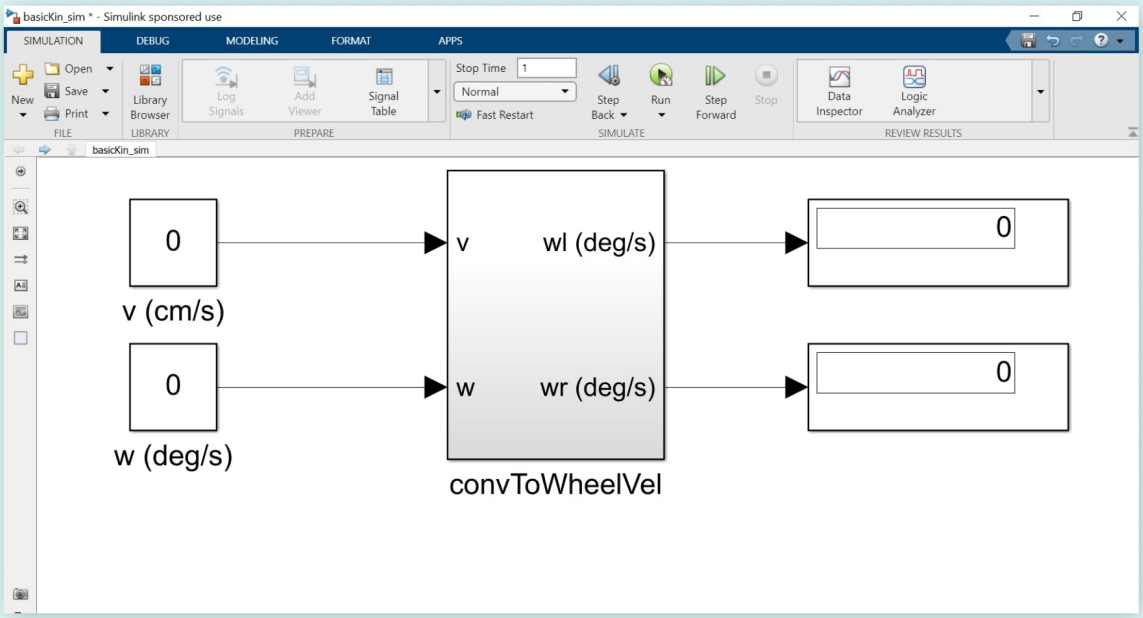
Help

The goal of this example is to guide you into a Simulink implementatio differential drive kinematics, that can then be used to drive the Rover. We have prepared a model in which a series of blocks are connected to each other. Given

a desired linear speed (v) and rate of rotation (ω) for the whole Rover, the Simulink model will perform the matrix operations shown above to compute the corresponding angular speeds for each one of the motors (ω_r and ω_l). Open the model by executing the following command in MATLAB:

```
>> basicKin_sim
```

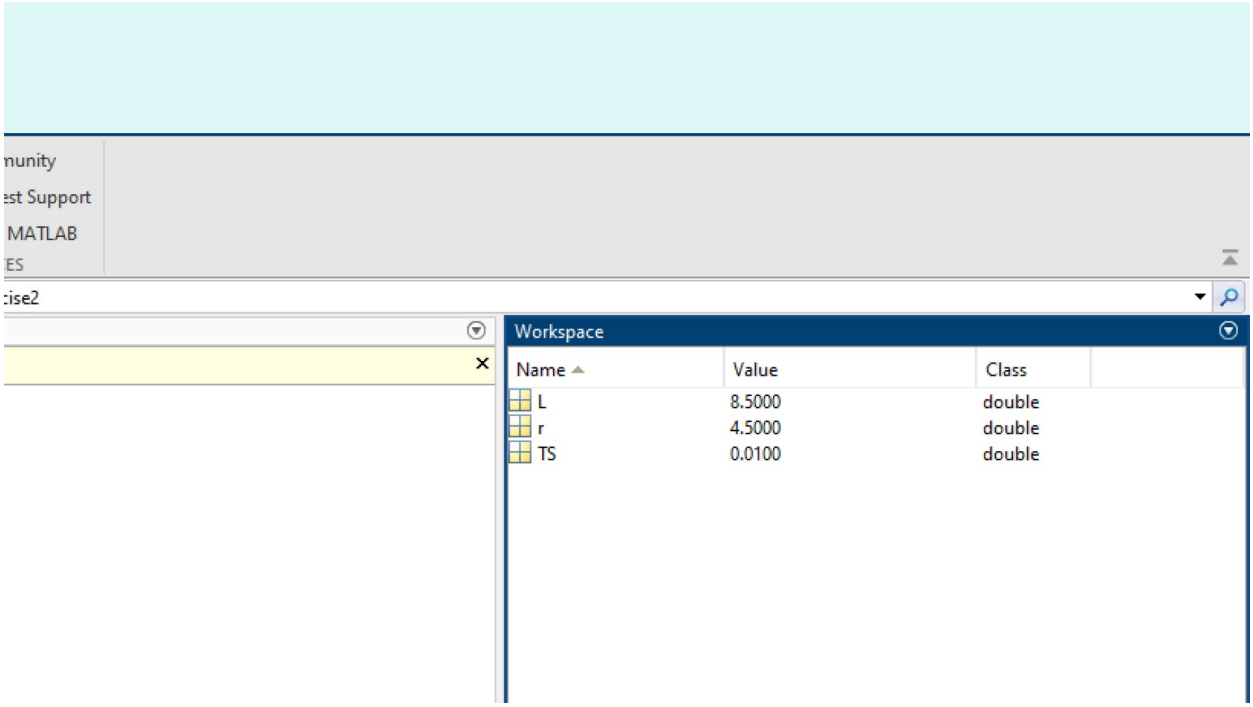
Note: Starting in MATLAB R2021a, the project files have been integrated within the MATLAB & Simulink Support Package for Arduino. If you have MATLAB R2021a or later, please make sure that you follow the instructions mentioned on **this** page to add the project files to the MATLAB path.



The blocks on the left are constant blocks where you can specify the desired linear velocity and rate of rotation. **convToWheelVel** is itself a collection of blocks that work together to generate the output from the input values. The display blocks on the right will show the resulting angular speed for the left (ω_l) and right (ω_r) wheels.

But before viewing the **convToWheelVel** subsystem, note that `L` , `r` were defined in the **Workspace**.

Help

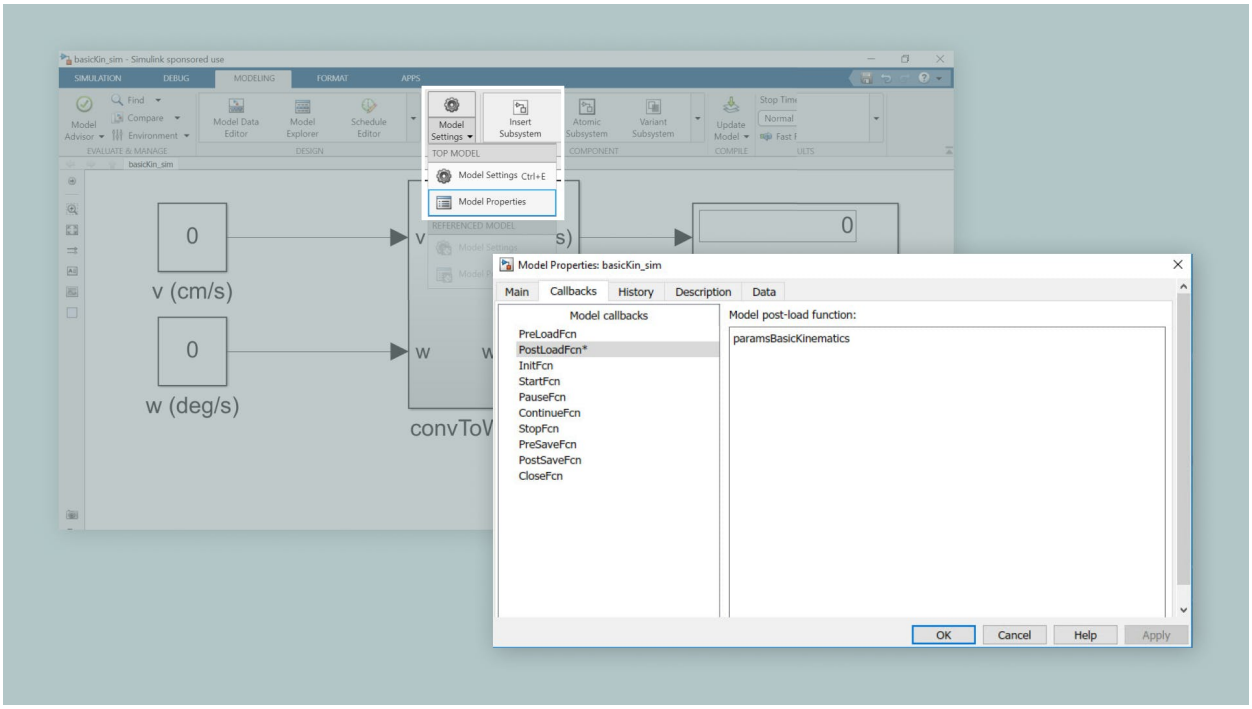


Simulink has the concept of model callbacks that lets you call MATLAB scripts or functions at different times. This model uses the callback **PostLoadFcn** to call a MATLAB script right after loading the Simulink model into the memory. The script being called here is `paramsBasicKinematics`, and these variables (`L`, `r`, and `TS`) are defined as constant values in this script.

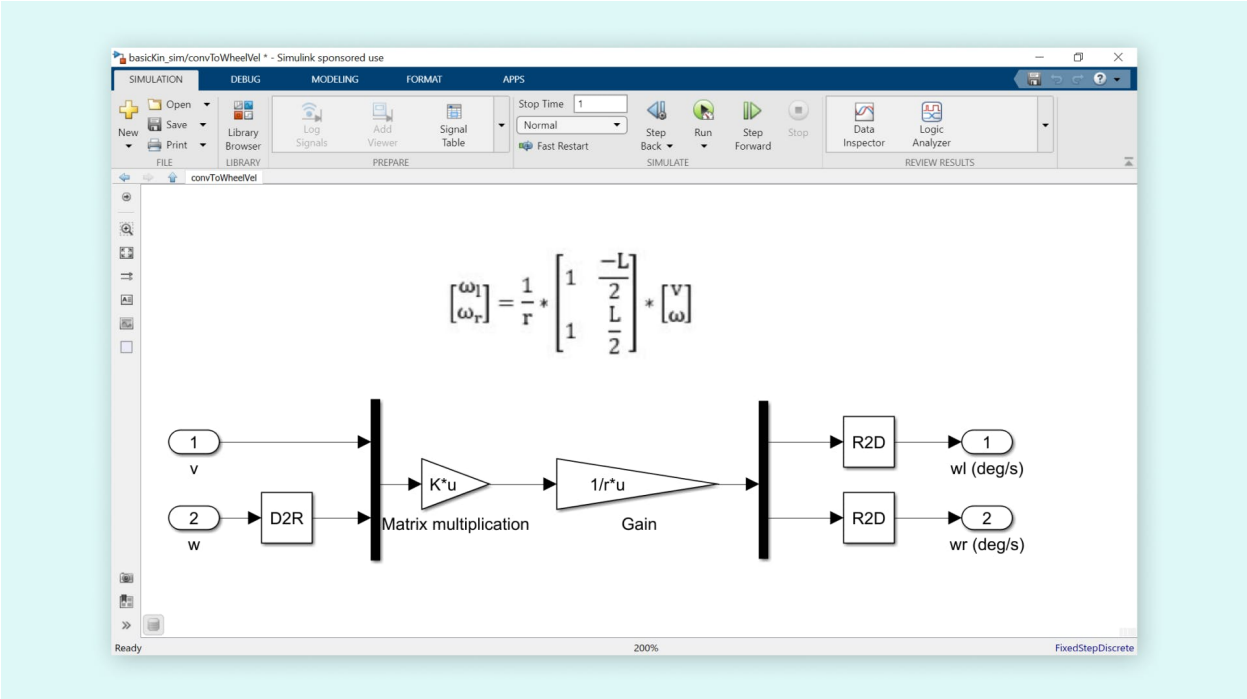
`L` and `r` are the distance between the two wheels (8.5 cm) and wheel radius (4.5 cm) of our Rover respecitvely. `TS` is the sample time of the model. Sample time is a parameter that dictates when the outputs of the Simulink model should be produced, and if appropriate, when to update blocks during the simulation.

You can view the model callbacks by selecting **Modelling > Model Settings > Model Properties** from the Simulink toolbar. Click **Callbacks** in the Model Properties window and select **PostLoadFcn*** as shown below.

Help



Alternatively, you could also right-click anywhere in the Simulink model canvas and select **Model Properties**. Now let's open the **convToWheelVel** subsystem by double-clicking it.

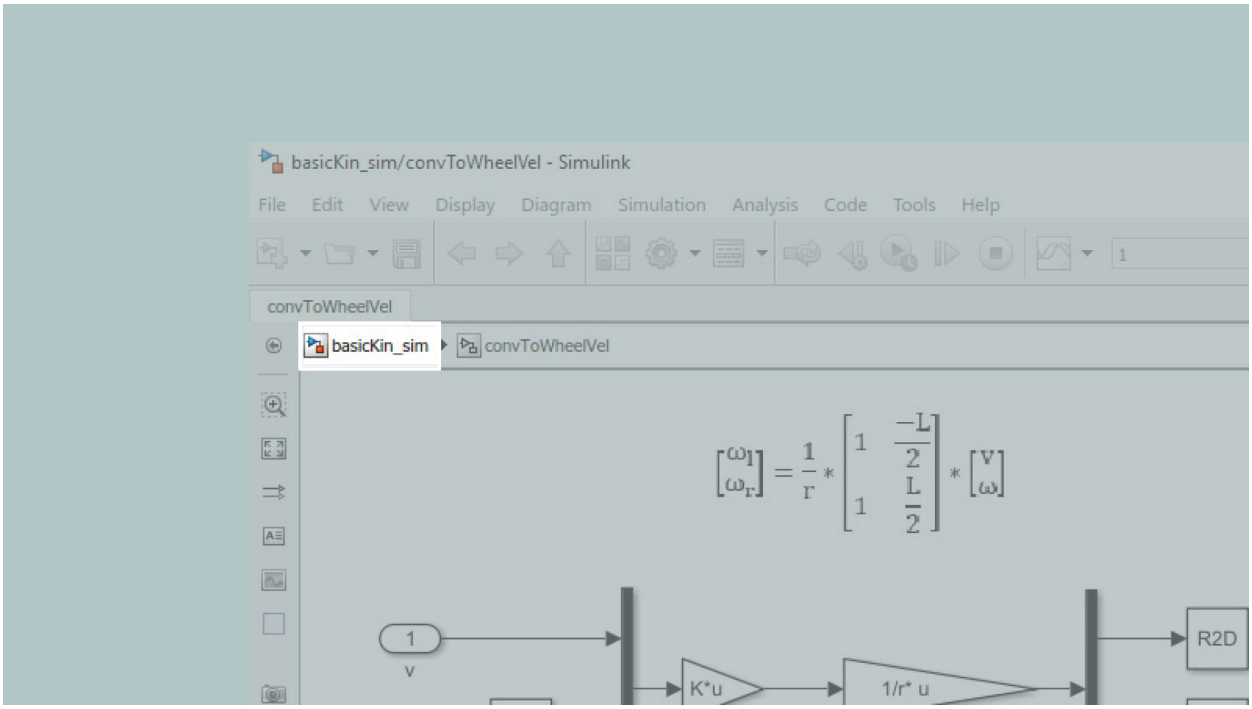


Here you can see the **Inport** blocks (`v` and `ω`) on the left side and the **Outport** blocks (`wl(deg/s`) and `wr(deg/s`)) on the right. The D2R block ad Help inport `w` is converting degrees to radians, while the R2D blocks are performing the inverse operation before sending the resulting data to the outports. The

thick black lines are **Mux** and **Demux** blocks. They allow us to arrange the data into vectors and separate them back to scalars, as needed. This allows us to perform a matrix multiplication with the first **Gain block** (`K*u`) on the vector and the second **Gain block** will multiply the components of the resulting vector with `1/r` .

This subsystem calculates (ω_l) and (ω_r) for a given desired velocity and rate of rotation. The equations implemented in this subsystem are displayed within the model for reference. This is a feature in Simulink that allows you to include documentation within the models themselves, which makes it easier to share your work with others.

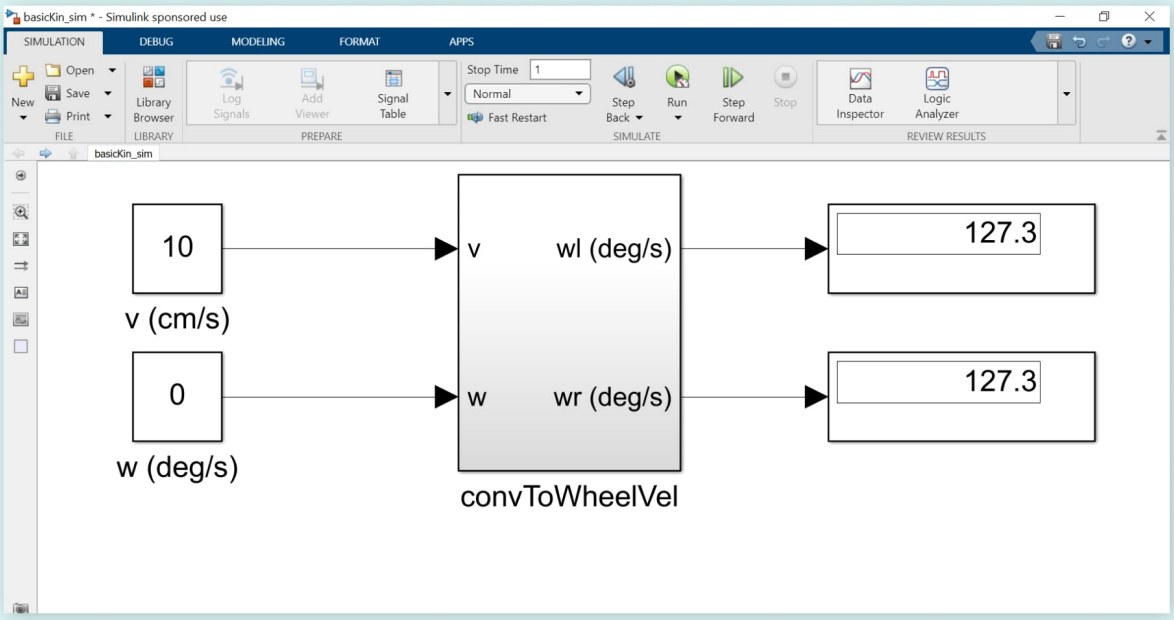
Let's navigate back to the top-level model by clicking `basicKin_sim` as shown below.



As an example, try `v` = 10 cm/s, `w` = 0 deg/s and then click the `Run` icon as from the **Simulation** tab. Simulink will then perform the computation and update the display once every `0.01` seconds as mentioned by sample time `TS` . On the display, you should see results consistent with the values calculated earlier, in this case:

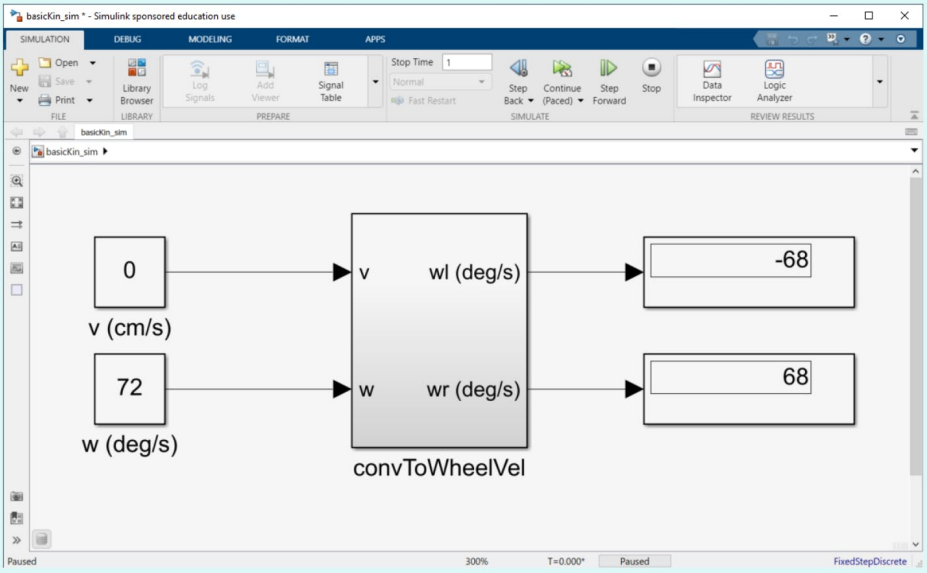
ω_l = ω_r = 127.3 deg/s

Help



Try once more with `v` = 0 cm/s and `w` = 72 deg/sec and run the model again. The results are consistent again, and in this case:

ω_l = -68 deg/s, ω_r = 68 deg/s



Simulating the Rover's Motion

Help

In the previous section, Simulink was used to implement basic kinematic equations of the Rover, but the model did not consider the location of the Rover in real-world coordinates. The location (x, y) and the heading (θ) of the Rover in an X-Y coordinate system can be described using the following equations:

$$\theta(t) = \int \omega(t) \cdot dt$$

$$x(t) = \int v(t) \cdot \cos\theta \cdot dt$$

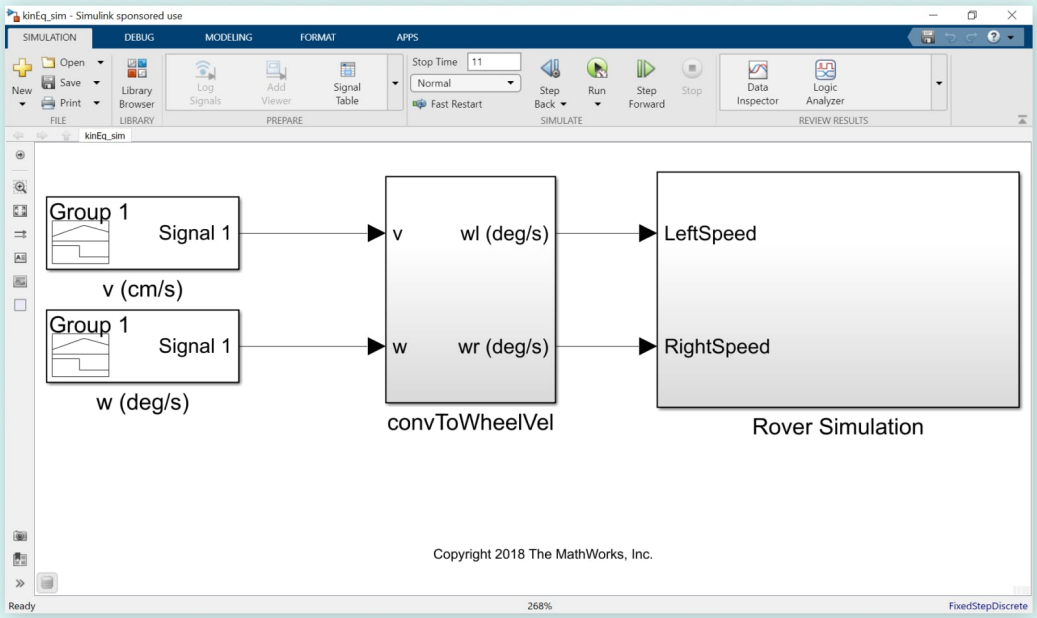
$$y(t) = \int v(t) \cdot \sin\theta \cdot dt$$

Let's see how to use Simulink to implement these equations and simulate the Rover's movement over time. Start by opening the **kinEq_sim model** from MATLAB:

```
>> kinEq_sim
```

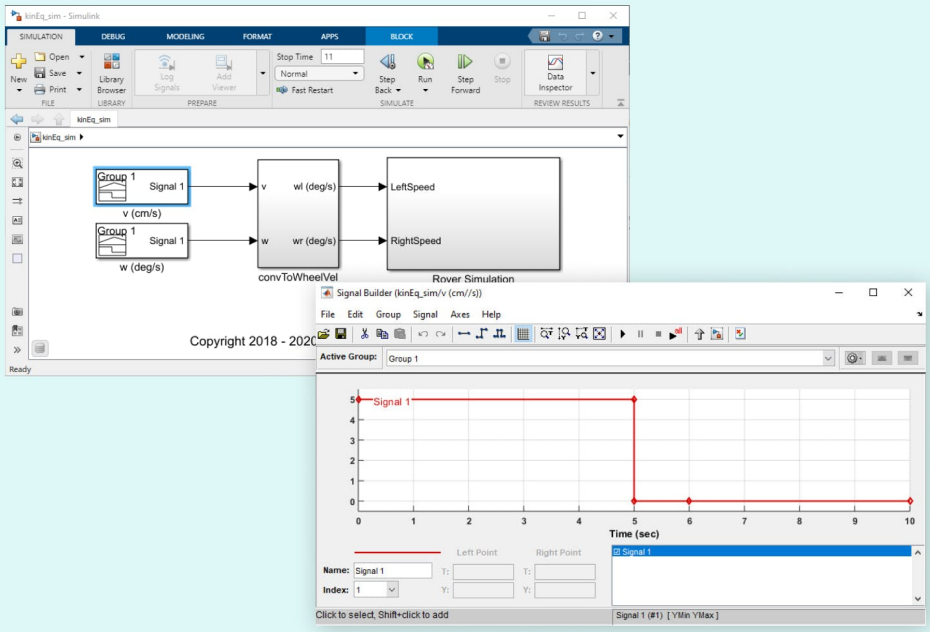
In this model, you see that instead of simple constant inputs that were used in the previous model, there are now **Signal builder** blocks labeled `v (cm/s)` and `w (deg/s)`. These provide signals that change over time to the **convToWheelVel** subsystem. Instead of simply displaying the results, they will be passed on as inputs to the **Rover Simulation** subsystem, which mathematically models the Rover's behavior and creates a visual simulation of the Rover moving in a `100 cm x 100 cm` arena.

Help



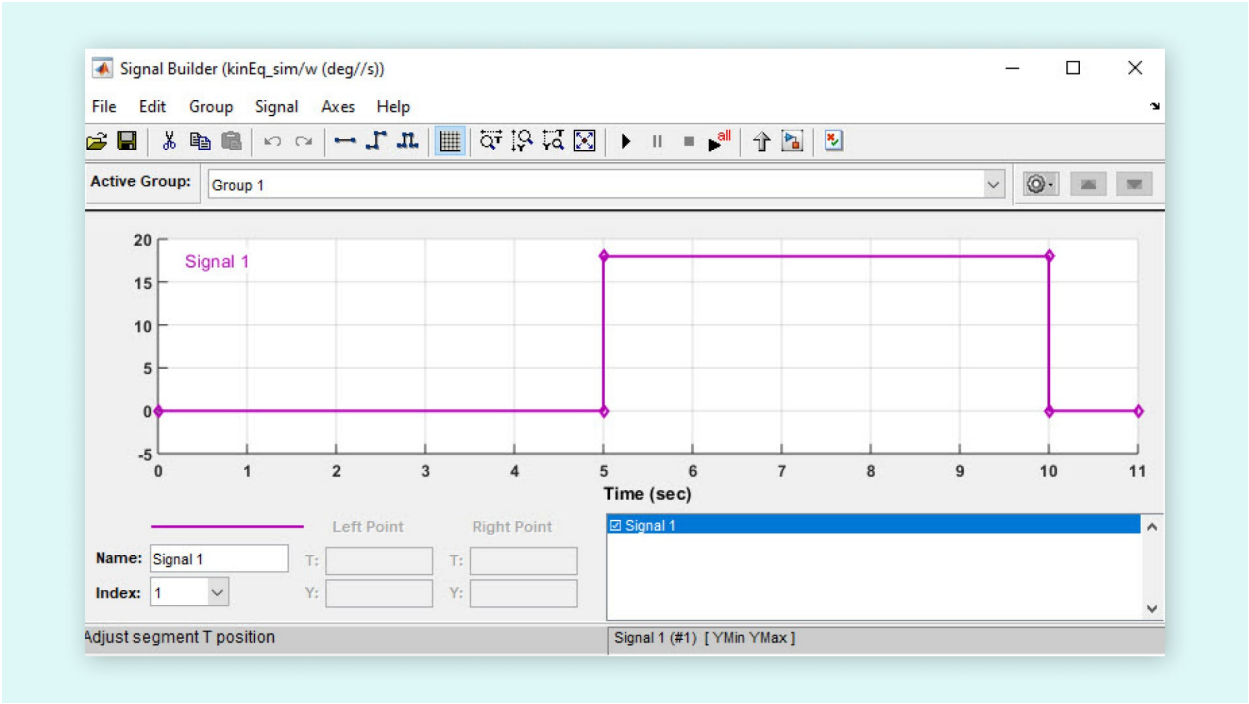
Note: The **convToWheelVel** subsystem is the same subsystem from the previous model.

Double-click on the `v (cm/s)` block to see what the signal looks like.



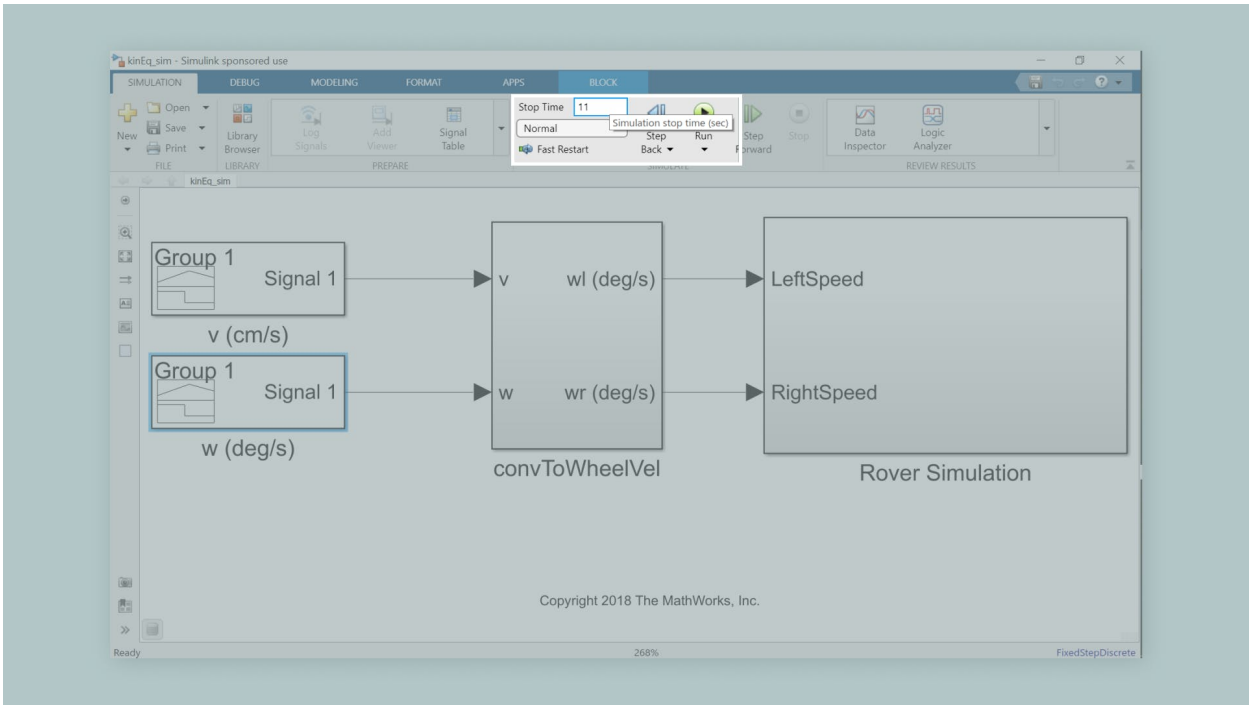
The signal looks like a step input that has a magnitude of `10 cm/s` for five seconds and then drops down to `0 cm/s`.

Help



The signal for `w (deg/s)` is a rectangular signal that has a magnitude of `72 deg/s` between 5 to 10 seconds and zero at all other times.

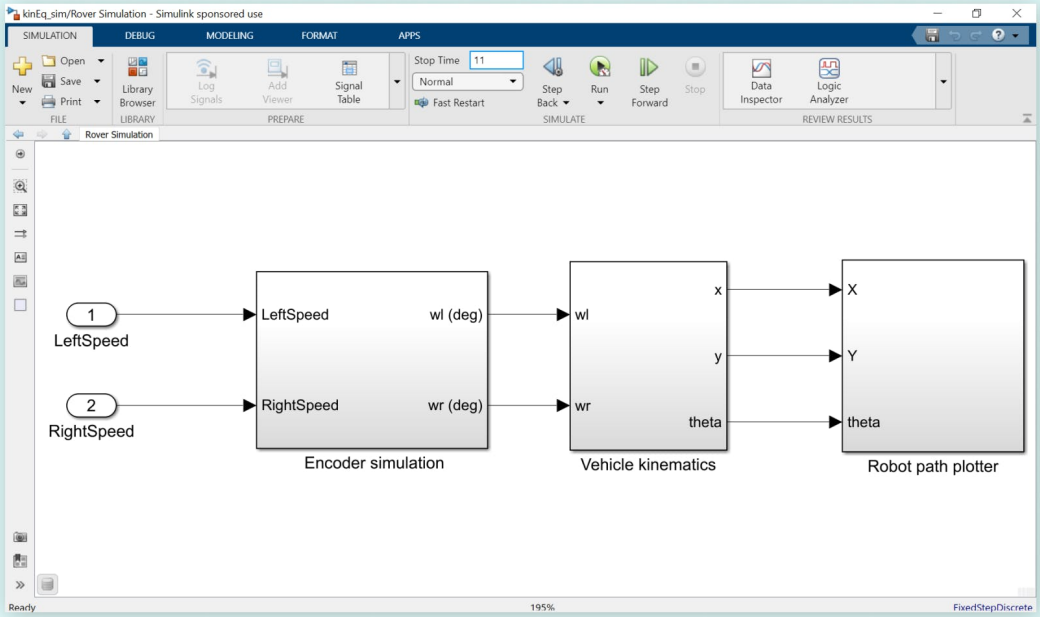
Since the input signals are both zero after 10 seconds, the simulation stop time is set to `11` seconds as shown below.



Help

Open the **Rover Simulation** subsystem. There are two inports to bring ... the angular speed of each wheel, and there are three subsystems here: the **Encoder**

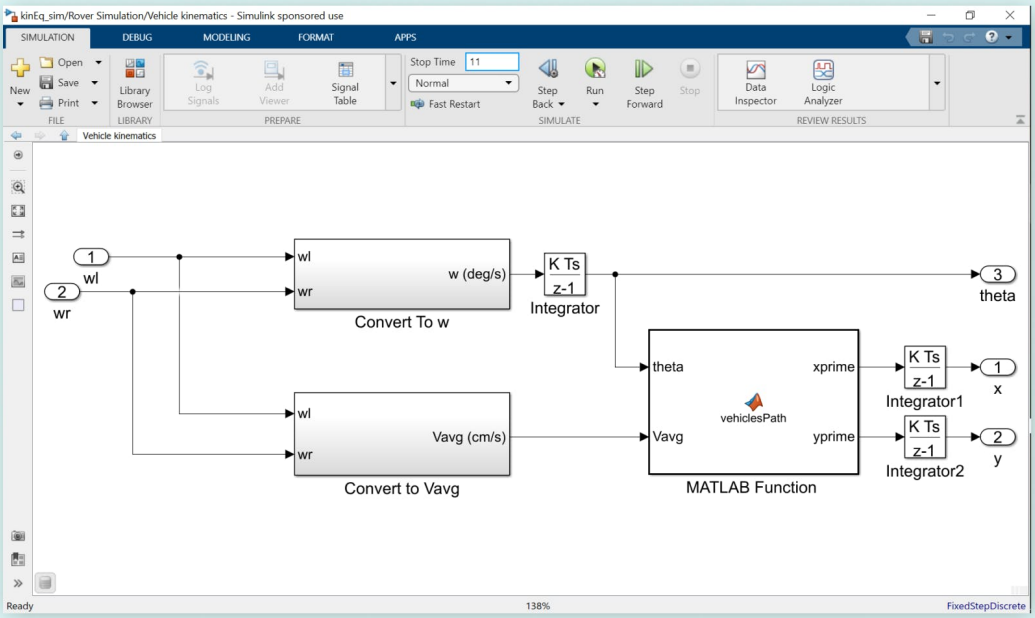
simulation, the **Vehicle kinematics**, and the **Robot path plotter** models.



The **Encoder simulation** subsystem converts the desired wheel rotational speeds from deg/s to degrees rotated by the wheel, to mimic the encoder on your Rover. To learn more about the actual encoder used and its behavior, refer to section 3.1.

The **Vehicle kinematics** subsystem implements the equations for $\theta(t)$, $x(t)$, and $y(t)$, introduced earlier. Open this subsystem. At this point, you know enough about Simulink to find more information about each one of the blocks. Spend as much time as you need to understand how the equations translate into the following block diagrams. Essentially, you will see that the outports are the result of integration operations, and they represent the orientation of the Rover along with its `x` and `y` coordinates.

Help

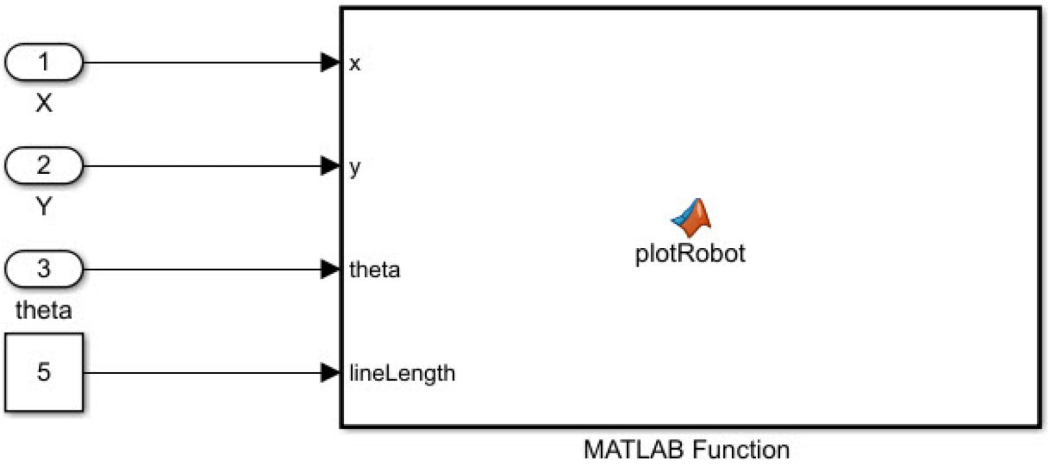


Explore the **vehiclesPath** MATLAB Function block to see how the equations are implemented here. A MATLAB function block allows you to integrate MATLAB code within Simulink models.

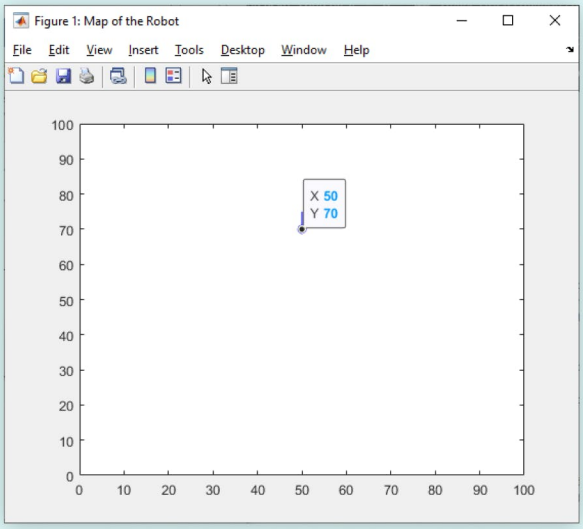
Navigate back to the **Rover Simulation** subsystem and open **Robot path plotter** subsystem.

You will see how there is a constant block along with the other inports that is sending the value `5` to the MATLAB Function block. This is used to draw the robot's trajectory. Open the **plotRobot** MATLAB Function block to see how to visualize the robot's path. Observe how the rover's position, in terms of its `x`, `y`, and `θ` coordinates change with time.

Help



Once you are familiar with how the model works, click **Run**.



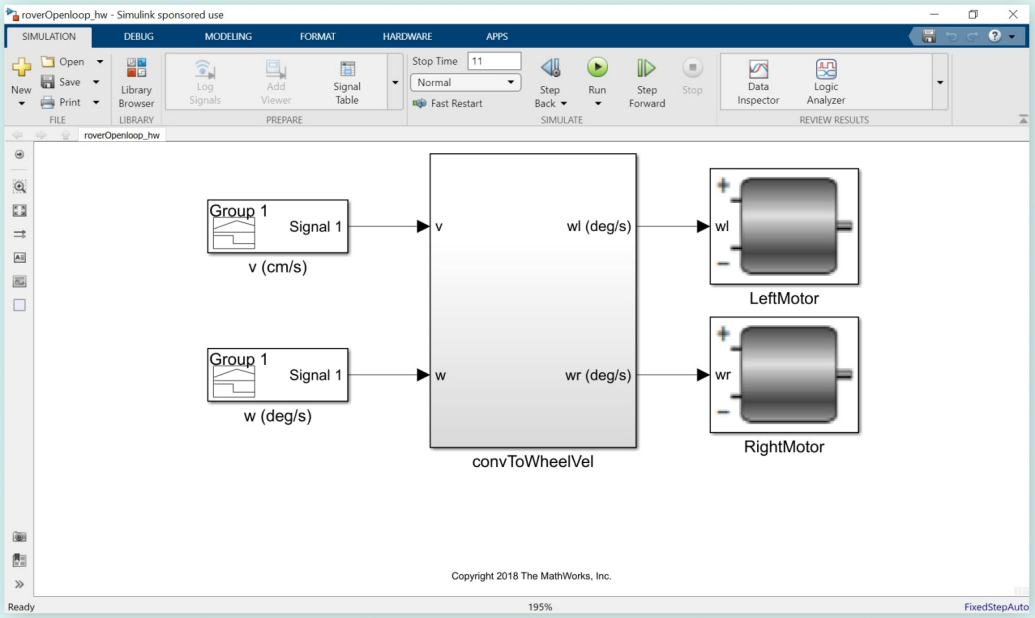
You will see the rover move around the 100 cm x 100 cm arena for 11 seconds. The simulation is based on a starting position of `StartX = 50 cm` , `StartY = 20 cm` and `StartTheta = 90 deg` . These variables were defined earlier in the **Workspace** through the model callback **PostLoadFcn** just as you Help previous model. Overall, the Rover moved 50 cm (10 cm/s x 5 sec) in the `y`

direction and then turned 360 degrees (72 deg/sec x 5 sec) based on the **Signal Builder** inputs.

Driving the Rover using Open-Loop Control

Once you have tried out the simulation, it's time to make the Rover do the same manoeuvre in the real world. We have prepared a Simulink model for you to quickly see how this could work. Open the **roverOpenloop_hw** model in MATLAB:

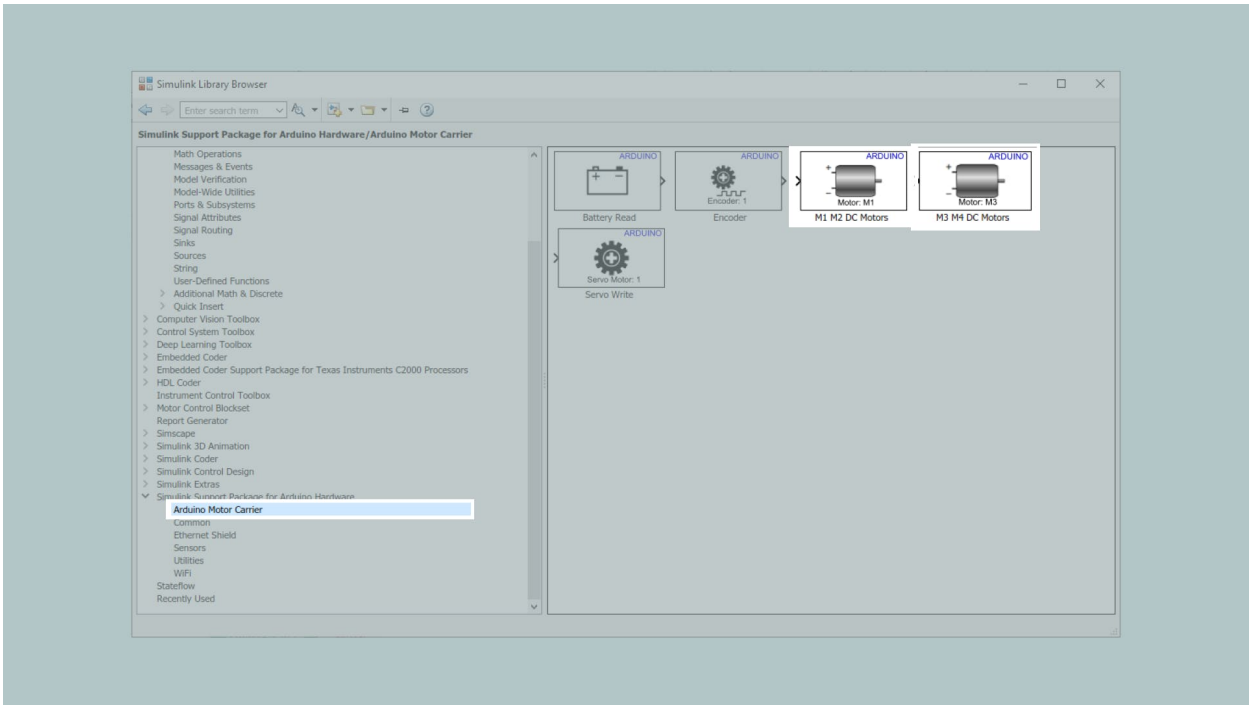
```
>> roverOpenloop_hw
```



The input signals and the **convToWheelVel** subsystem are the same as the previous model, but the rotational speeds are sent directly to the actual motors that are presented as blocks inside the model.

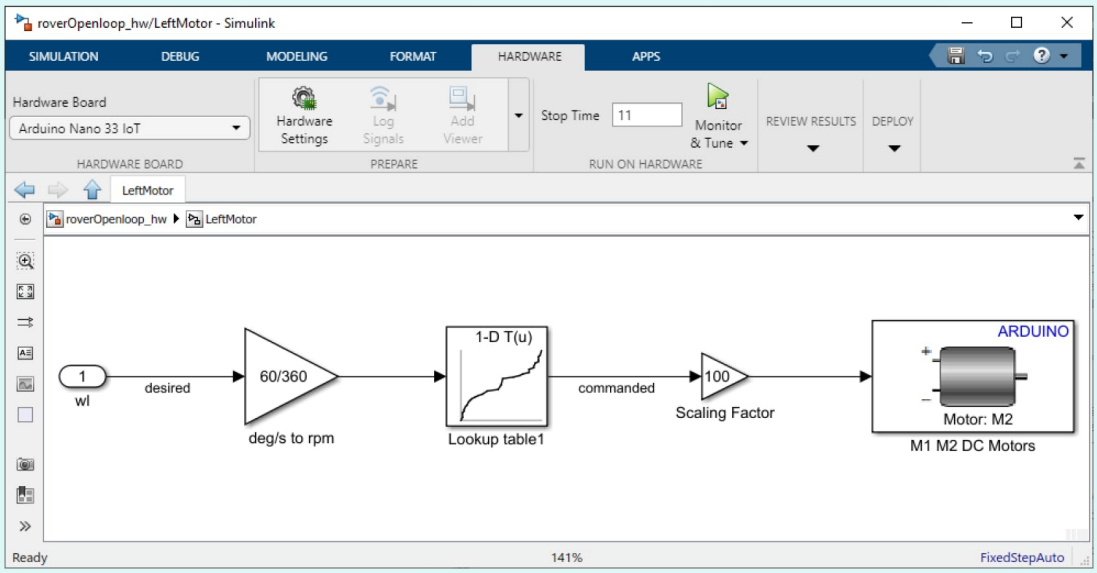
The **LeftMotor** and **RightMotor** blocks come from the library **Simulink Support Package for Arduino Hardware** as shown below.

Help



As discussed in section 3.6 “Characterizing a DC Gear Motor in MATLAB”, the input to the Motor blocks should be a PWM signal, not a commanded rotational speed. So, inside the **LeftMotor** and **RightMotor** subsystems, the commanded rotational speeds are converted to PWM using new motor characterization data. The reason behind the need for new characterization data is that the motor characterization in section 3.6 **Characterizing a DC Gear Motor in MATLAB** was performed when the Rover's wheels were spinning freely in the air. The new motor characterization was performed with the Rover moving on the floor and the battery having a full charge. The provided data (`motorResponse.mat`) for the Rover shows the relationship between PWM value and the commanded motor speed (rpm - revolutions per minute). This is represented by the **Lookup table** block within the **LeftMotor** and **RightMotor** subsystems.

Help

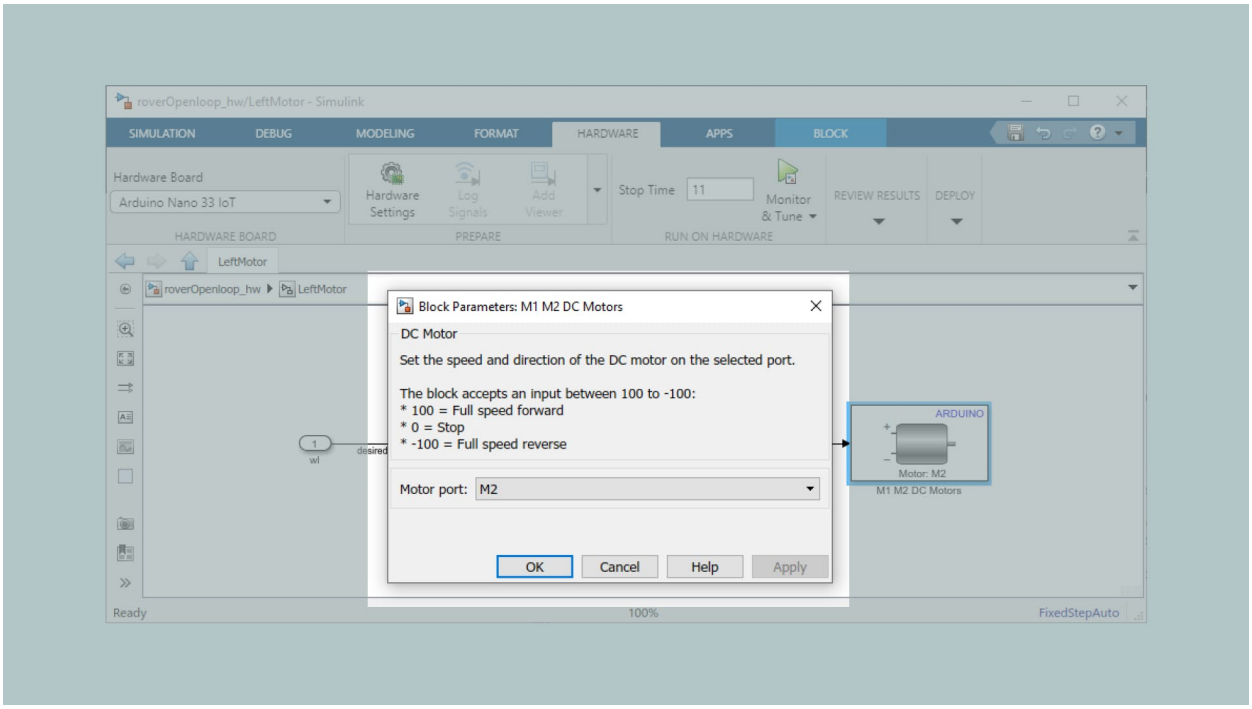


The characteristic curve obtained from the live-script **characterizeMotorScript.mlx** under section 3.6 populates the PWM data against the measured rpm of the output motor shaft. Hence the rotation rates `wl` and `wr` need to be converted to rpm values by inserting a **Gain** block with a value equal to 60 sec/360 degrees. Additionally, another **Gain** block (gain = 100) needs to be added just before the **M1 M2 DC Motors** block since it accepts values between -100 to 100, and the output of the **Lookup table** ranges from -1 to 1. For the right motor, the gain value is `-100` instead of `100`, since the left and right motors are oriented exactly opposite to each other, as discussed earlier.

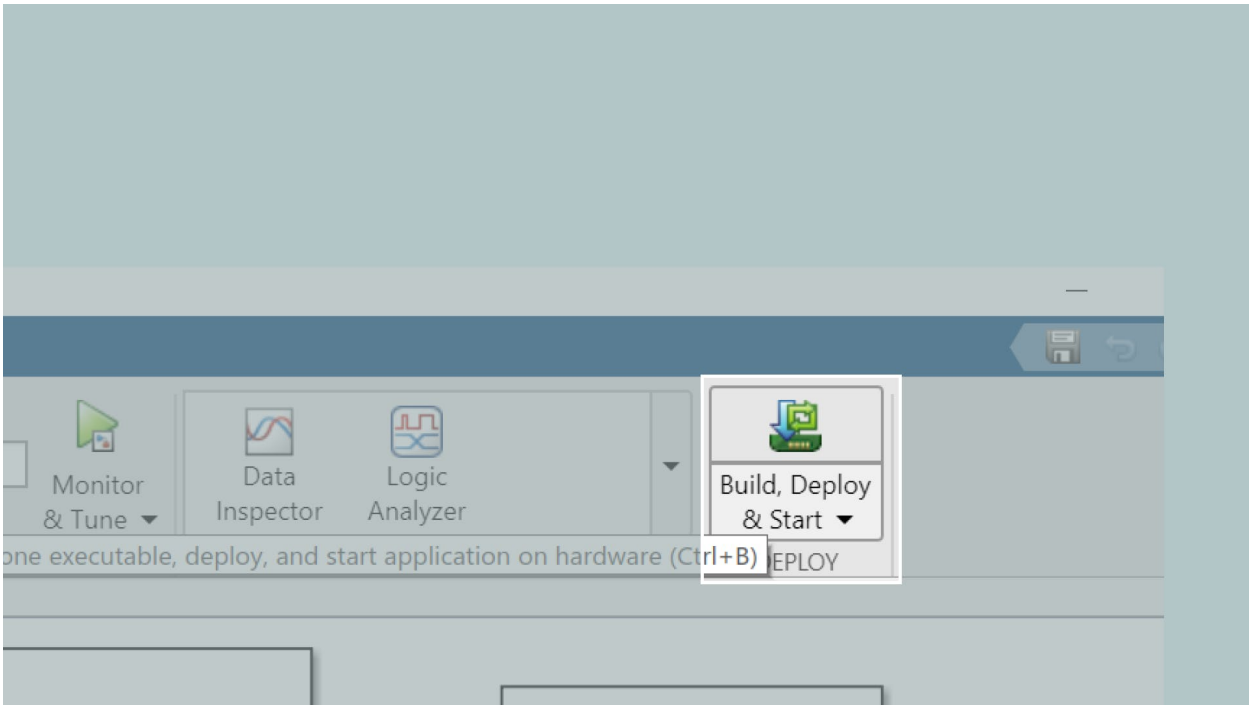
You can also derive the motor characteristic curve for your Rover locally, instead of using the provided data (`motorResponse.mat`), by placing it on the ground and executing the steps from the live-script **characterizeMotorScript.mlx** provided in the **Exercise2** folder. You will need to run the live-script twice for both the motors, save the responses with different variable names and then use them in the **Lookup tables** within the **roverOpenLoop_hw** model, as discussed above.

Check the port numbers specified by the **DC motor** blocks in the **LeftMotor** and **RightMotor** subsystems to ensure that the connections for the right motors of your Rover match the ones specified in these blocks. The image below shows the **M1 M2 DC motors** block inside the **LeftMotor** subsystem.

Help



Press the **Build, Deploy & Start** button in the model from the **Hardware** toolstrip. This will generate the code equivalent of your model and upload it to the Arduino Nano 33 IoT board, which will ensure that the Rover can run the code on the board even when it is detached from the computer.



Disconnect the USB cable from the Rover and place the Rover on the some other surface where it has enough space to move around. The Rover is fragile, and it could get damaged if it falls from your desk.

Help

Power on the Rover (using the battery) and measure the distance it covers on the floor. Your goal is to check whether the real experimental conditions replicate what you simulated in the model earlier.

Recall that in simulation the Rover moved 50 cm in the y-direction and then turned 360 degrees. Did the real-world Rover do the same?

You're likely to see that it did not. In general, it's very difficult to accurately control your system using open-loop control unless you have a very accurate model of your motor behavior and operating environment. In our case, the motor characterization was not rigorous and a simple relationship for PWM versus motor speed was used. That said, even if the characterization were rigorous there would be issues if you want to operate the Rover on a surface different from the surface used for motor characterization, or if the battery is not operating at the same voltage, or if the Rover was placed on an inclined plane, etc. Another possibility could be that the desired inputs lie in the dead-zone of the motor characteristic curve. In other words, the desired `v cm/sec` and `w deg/sec` values, when converted to individual motor rpm requirements, correspond to PWM values that are lower than the minimum PWM value needed to actuate the motors.

Need For Closed-Loop Control

In the next exercise, you will see how closed-loop control will help address the limitations observed with open-loop control and enable much more accurate control of the Rover position.

Review

- We saw how to use kinematic equations and inverse kinematics to simulate the Rover movement.
- We saw how to translate that understanding to control the Rover in the real world.
- We saw that mathematical models need to be tweaked to better accommodate real-life conditions.

Help

Files

- basicKin_sim.slx
- kinEq_sim.slx
- roverOpenloop_hw.slx
- roverMotors_hw.slx
- Data.xlsx

Learn By Doing

Try updating input signals in the **Signal Builder** block using various combinations of `v` and `w`. Can you make the Rover move in a circle, or follow a specified sequence of movement (for example, forward 50 cm in the y-direction, turn 90 degrees, forward 30 cm in the x-direction)?

If your Rover is not moving, try increasing the values of `v` and `w`. Use the live-script **characterizeMotorScript.mlx** to derive your own motor characteristics curve and find out the dead-zone. With help of the kinematics equations described in this section and the dead-zone band, deduce the minimum `v` and `w` required for the open loop control to work. Update the **Signal Builder** blocks with these new inputs. Is the Rover now able to reproduce the desired response?

Help

5.3 Closed-loop Control of Rover Position

In this exercise, Simulink will be used to implement a PI controller that controls the distance moved by the rover. You will see how to read the encoder data from the rover's wheels and use this data as feedback to your controller. You will also see how to tune PI values of the controller to optimize performance.

In this exercise, you will learn to:

- Understand the basics of a PI controller design and, by extension, PID controllers,
- Read encoder data,
- Tune parameters of a controller in Simulink,
- Implement a controller on the rover.

Basics of a PID Controller

A proportional integral-derivative-controller (PID) is a control-loop feedback mechanism used in systems requiring continuously modulated control. PID controllers operate on the error term, the difference between actual and desired behaviour, and apply a correction to your system based on the Proportional, Integral, and Derivative terms associated with the error. This has the effect of making corrections for past error (I), present error (P) and future error (D). The correction is denoted by the equation:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int e(t) \cdot dt + K_d \cdot \frac{\mathrm{d}}{\mathrm{dt}} e(t)$$

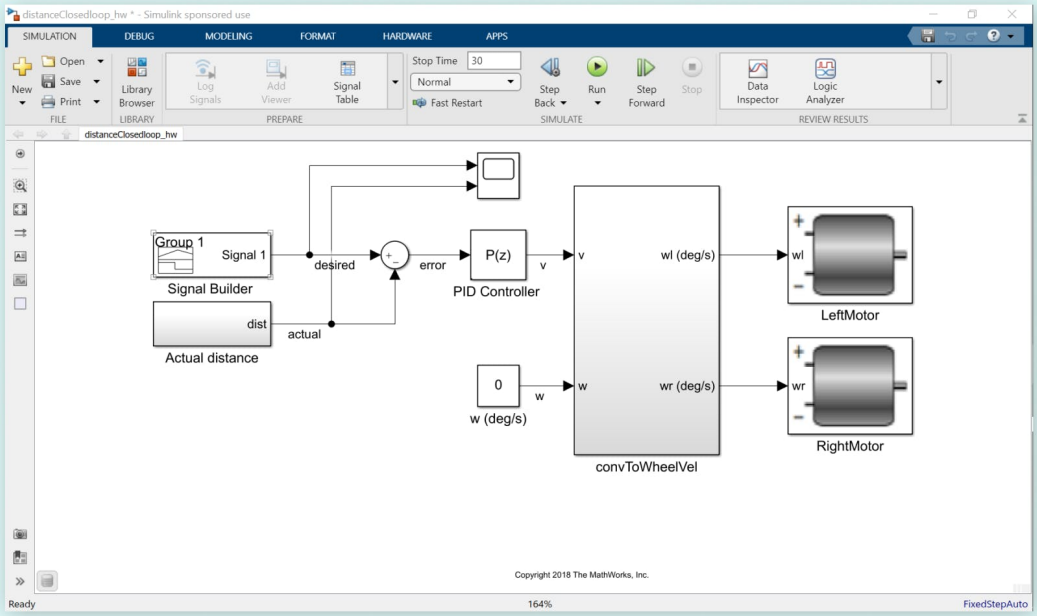
Where K_p (in Simulink's PID Controller block this is represented by the letter P), K_i (I), and K_d (D) are constants coefficients that define the system. This exercise will introduce some concepts associated with PID control in the conte mobile rover project. Refer to **this link** for additional information.

Help

Control Rover Position using a PID Controller

In the previous section, the limitations of using open-loop strategy to control the rover's position were mentioned. Let's see how closed-loop PID control can address these limitations. Start by opening the `distanceClosedloop_hw` model:

```
>> distanceClosedloop_hw
```



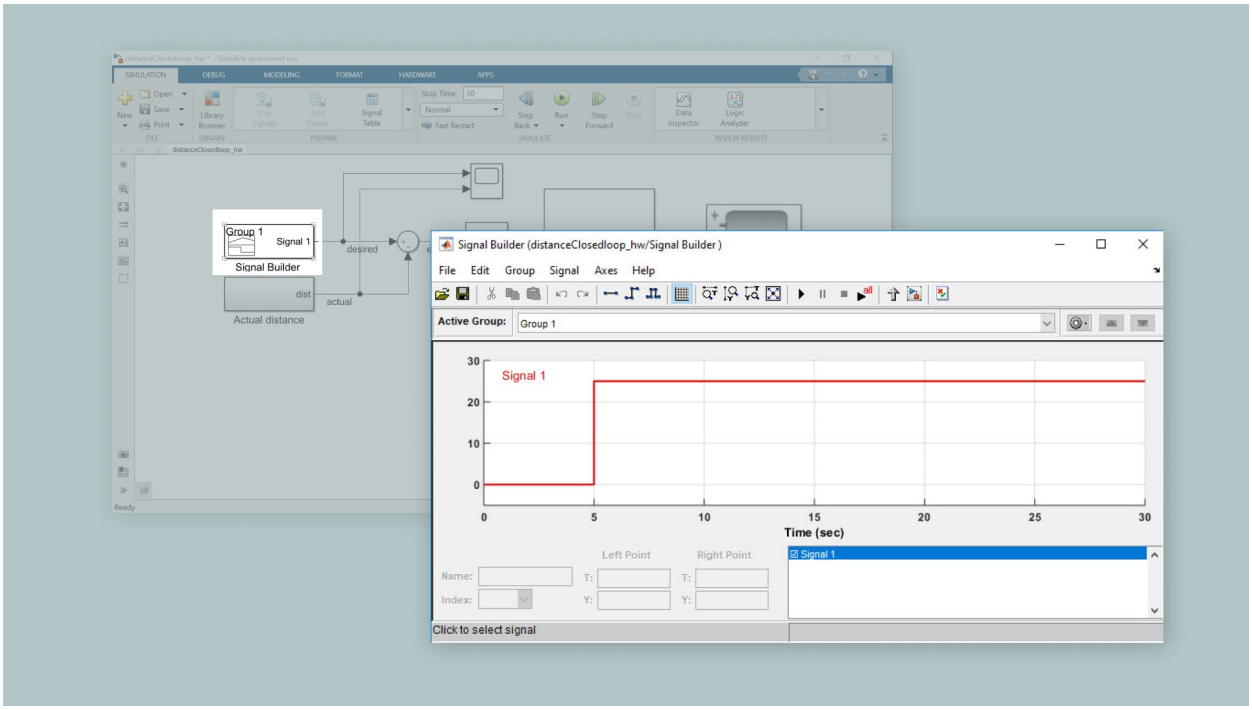
The right side of this model is the same as the **roverOpenloop_hw** model from the previous exercise. The difference is that the velocity input to the **convToWheelVel** subsystem is now based on the error term between the rover's actual and desired distance. The challenge will be to determine the actual distance by means of a sensing device on the rover. In this case, the rotary encoders can be used for the same, as you will see later.

To simplify the controller design at the early stages, let's consider the rover moving in a straight line. In this case, the commanded rate of rotation for the rover is zero. In the next section, you will see how to control the rover's angle of rotation.

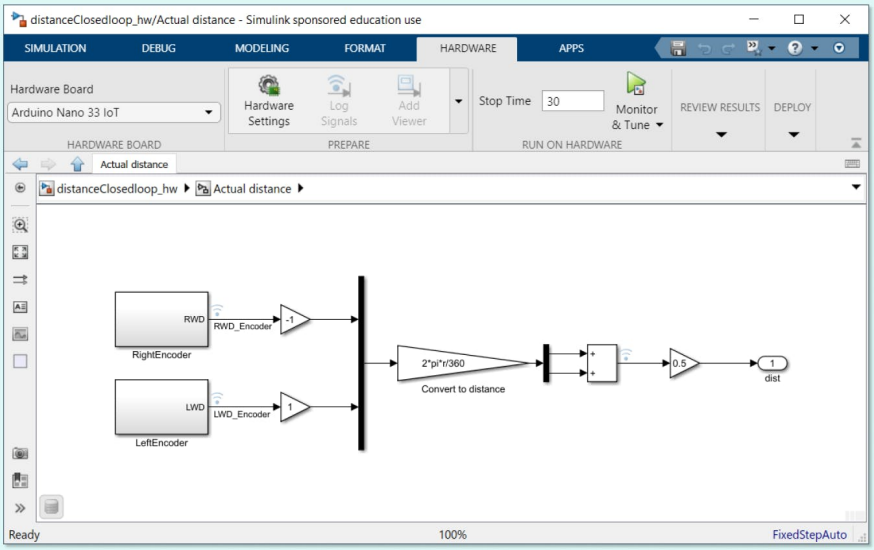
Help

The desired distance is specified using the **Signal Builder** block. Double-click and open this block. Note how the specified step input of `25 cm` occurs at `5`

seconds . The following image shows the time diagram for the signal.



As mentioned earlier, the sensor that will determine the distance travelled by the rover, is the encoder. The measured distance from the encoder is read into the model in the subsystem labelled **Actual distance**. Open this subsystem and observe the different blocks that compose it.



Help

There are two subsystems: **RightEncoder** and **LeftEncoder**. The outputs from these subsystems will be arranged into a vector before being multiplied by a constant to convert the encoder readings to distance. The constant is based on the system's characteristics, namely the wheel's radius. The result will be computed into an average and sent to the `dist` outport.

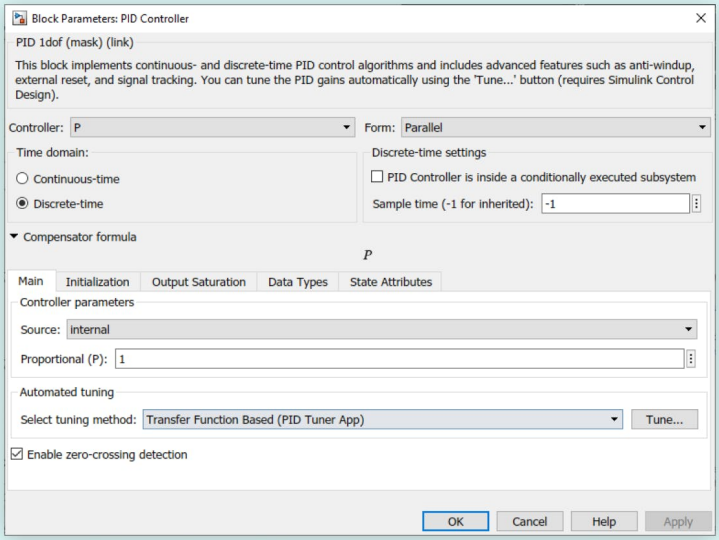
The **RightEncoder** and **LeftEncoder** blocks read count data from the wheel's encoders and convert that to degrees. To convert encoder degrees to the distance moved by the wheels, the following equations are used:

$$dr = \frac{2\pi r}{360} \cdot \mathrm{RWD}\ dl = \frac{2\pi r}{360} \cdot \mathrm{LWD}$$

Here `dr` and `dl` denote the distance moved by the right and left wheels respectively and `r` is the wheel radius.

Navigate back to the top-level `distanceClosedloop_hw` model and notice how the error is calculated as the difference between desired and actual distance. This is used as the input to the PID Controller.

Open the `PID Controller` block and note that by default a simple Proportional controller is used, since the Integral and Derivative terms (`Ki` and `Kd`) are both zero.



Help

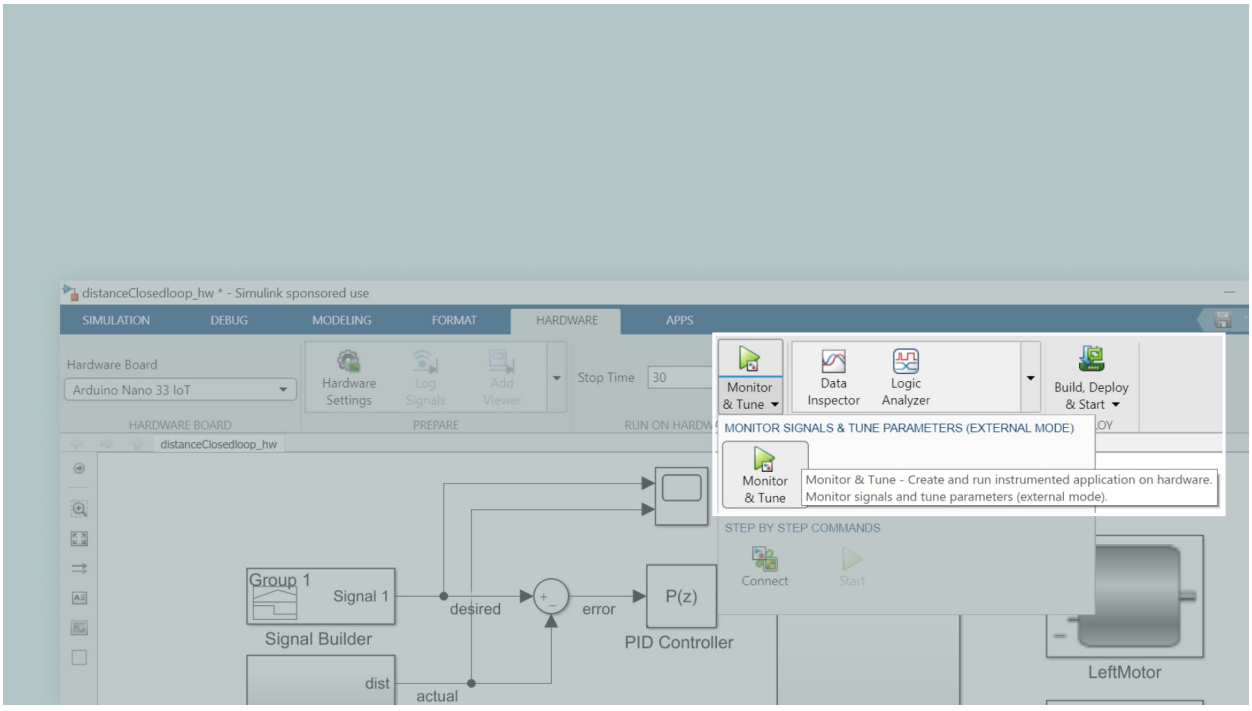
You are going to test this on the Rover using Simulink's **External mode (Monitor & Tune)** option. In external mode, Simulink builds an executable from the model and downloads it to the Arduino hardware. The connection with Simulink is maintained as the executable runs on the hardware, so you can tune parameters and monitor signals of interest using Scope blocks. Refer to **3.1 "DC Motors"** for more information about External mode.

Make sure that the `Stop Time` (**Simulink toolbar > HARDWARE > RUN ON HARDWARE**) is set to `30` seconds.

For convenience and to help you learn some of the concepts behind PID control, start by putting the rover on top of the target or some other platform that allows its wheels to spin freely. Your rover must be connected to the computer by the USB cable, which means it cannot move far. Turn **ON** the rover battery as that is where the motors are getting their power from.

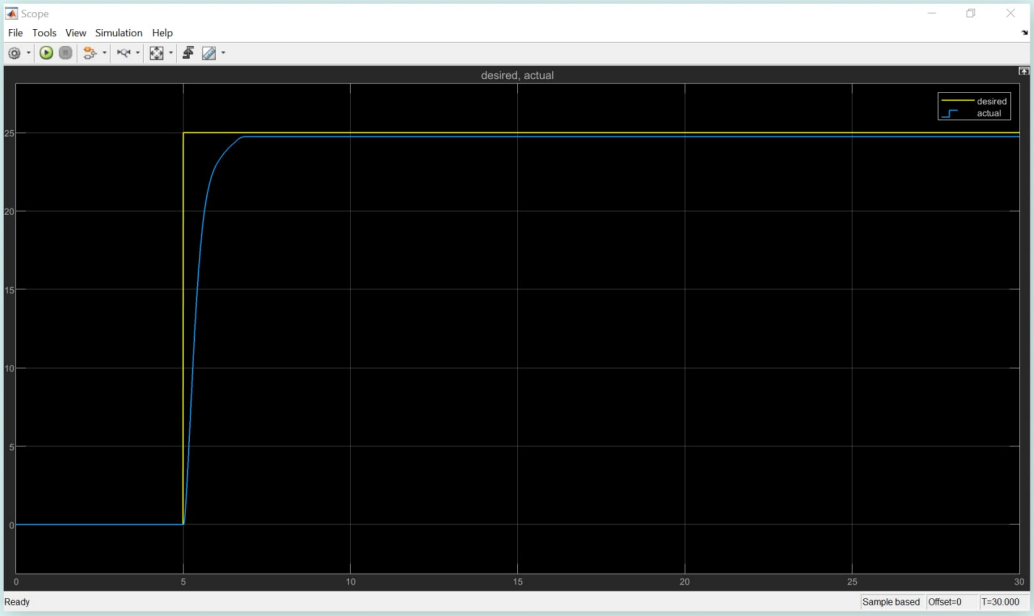
Help

Click on **Monitor & Tune** (HARDWARE > RUN ON HARDWARE). After downloading the code to the hardware, the Scope block will open automatically, which will help visualize the desired and actual positions as the executable runs.

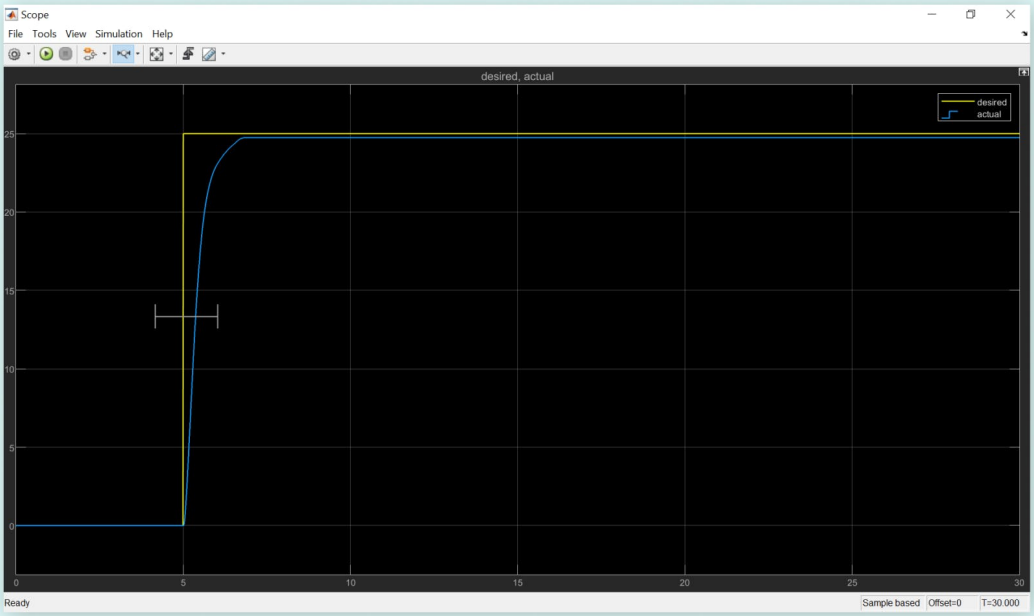


The result of the scope shows, in this case, the desired distance in yellow and the actual distance in blue. You can see that there is a small discrepa between them. PID controllers try to compensate for these errors.

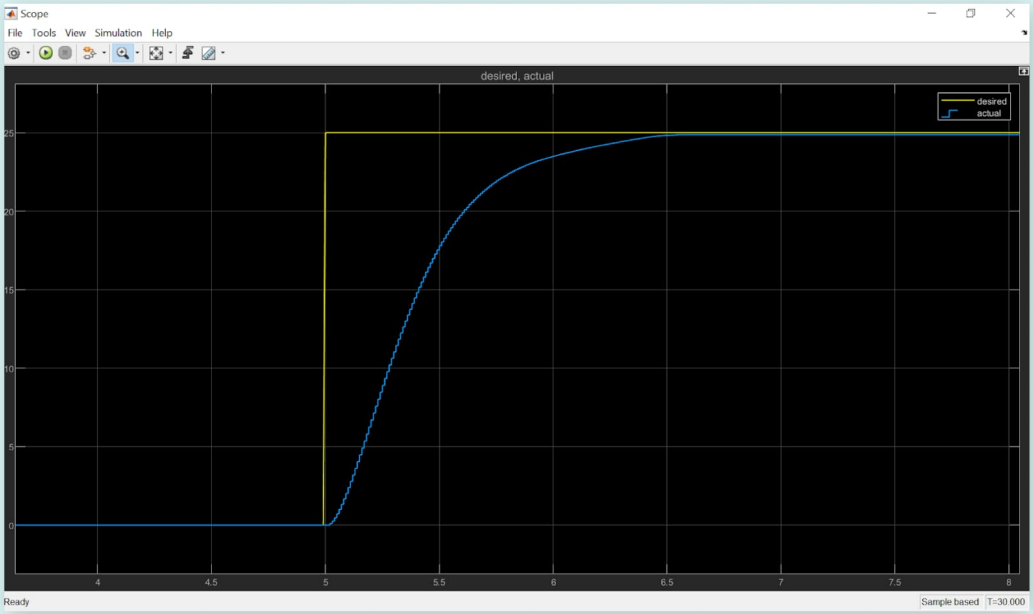
Help



Zoom in on the x-axis to get a closer look at the step response. To do so, click the magnifying glass on the toolbar and select the area you want to zoom in by clicking and dragging on the graph as shown below.



Help



The time it takes to reach 90% of the desired value is called **Rise Time**. For `P = 1` the rise time is approximately `1.5 seconds`. Given the specific characteristics of the rover, a slower rise time is preferred since you may use the rover in small areas and generally don't want the rover moving too fast as it might lift the rover off the ground. This is something you can learn by experimenting with the rover yourself.

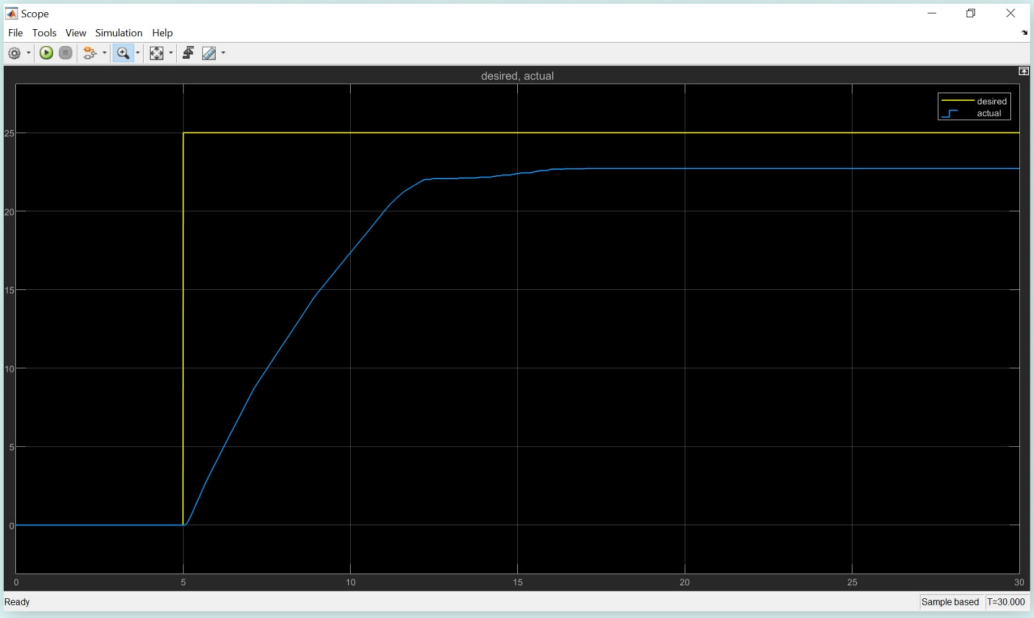
The following reference table depicts how increasing the gains `P` (K_p), `I` (K_i), and `D` (K_d) affect key characteristics of the system response:

Help

Increase gains to get desired response	Rise time	Overshoot	Steady-state error
P	Decrease	Increase	Decrease
I	Decrease	Increase	Decrease
D	Minimal Change	Decrease	No change

Please note that neither `overshoot` nor `steady-state error` have been introduced yet in the system response. They will be discussed in the following paragraphs.

Based on the previous table, to increase the rise time you should decrease the P value. Change the P value in the `Controller` block to 0.01, select **OK** and then click on **Monitor & Tune**.



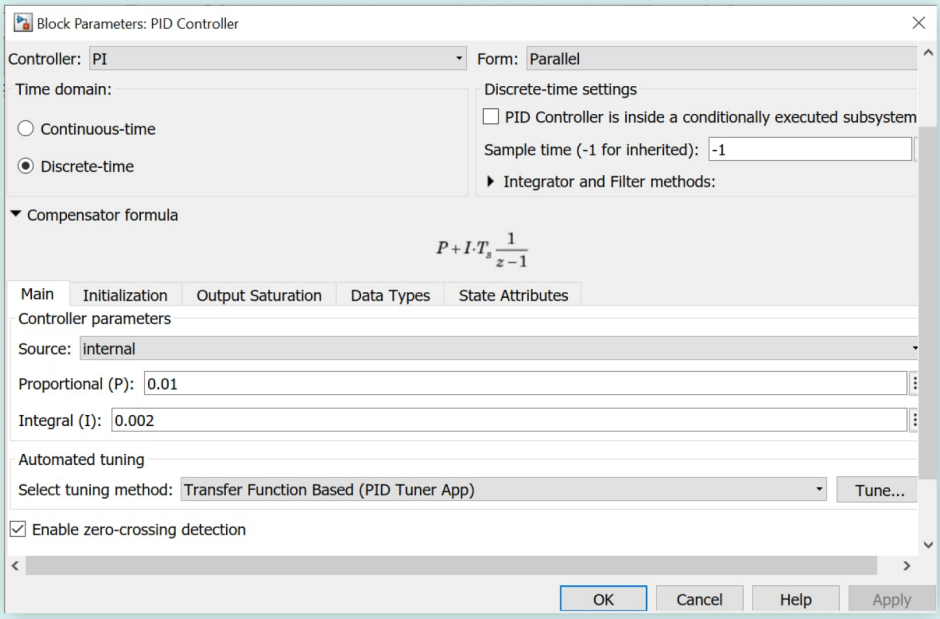
This increased the rise time to 11 seconds while also significantly increased the steady-state error, which is defined as the difference between desired and actual values of a variable once the system has reached an equilibrium.

Help

Note: Equilibrium is also known as steady state.

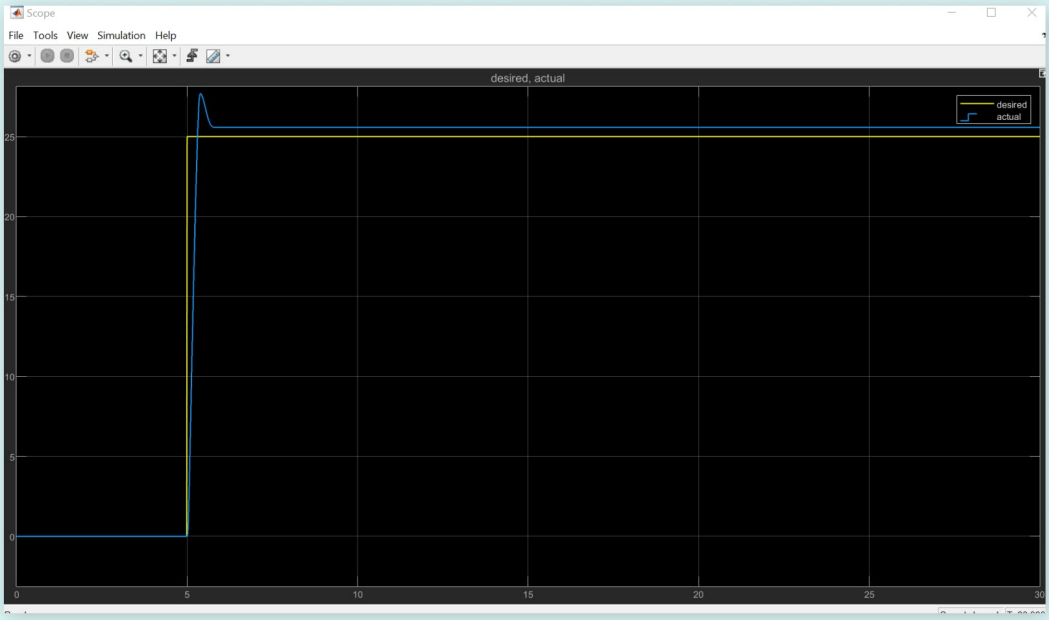
Note: In some of our tests, we noticed that one of the motors was not spinning freely and this caused the steady-state error to be greater than 5. If this happens, you can use the learnings from above to have a different P gain that would give a better steady-state error.

Based on the table, adding an **Integral** term to the controller will help decrease the steady-state error. Open the PID controller block, select **PI** in the PID Controller drop-down list and enter an integral gain value (`I`) of 0.002. It is a good rule of thumb to always make sure the `P` value is higher than the `I` value.



Click Apply, and then click on **Monitor & Tune**. Your scope should look like this.

Help



Introducing the `I` gain reduced steady-state error. However, as you can see here, it introduced yet another effect, called the **Overshoot**. This overshoot is measured to be roughly 6 cm. This is undesirable in our case, as the rover would go past the target location and then oscillate as it attempts to correct its position.

Referring to the table, to eliminate overshoot you can increase the `P` and `I` values.

Try increasing the `P` and `I` values using trial and error until you achieve a good balance of **Rise Time**, **Steady-State Error** and **Overshoot** in your model.

Now, disconnect the USB cable connected to the PC and move your rover to the floor to check how the controller performs. Connect the battery and measure how far it moves. Does it move the expected 25 cm?

You're likely to find that it does move the 25 cm, but you need to adjust the PID values to account for the new operating conditions (i.e., wheels are not free-spinning when placed on the floor). The exact values will depend on the surface you use, but in several tests, on a relatively hard, carpeted surface it was found that `P = 8.0` and `I = 0.002` provided a good starting point. In some tests these values gave an `overshoot` and `steady-state error` both cm.

Help

In the next exercise, you'll see how to eliminate `overshoot` and `steady-state error` using logic-based programming and state transitions.

> **Note:** In some of our tests, the rover never moved in a straight line and this is because of the difference in motors. You will learn how to address this issue in the next exercise.

Control the Rotation angle of the Rover Using a PI Controller

This section complements the previous one that was looking at the rover moving over a straight line for a certain distance in a predetermined amount of time. In this case, you will focus on getting a similar level of control on the rover's rotational movement.

Earlier in this exercise, you saw the equations to convert encoder degrees to the distance travelled:

$$dr = \frac{2\pi r}{360} \cdot \mathrm{RWD}\ dl = \frac{2\pi r}{360} \cdot \mathrm{LWD}$$

The equation to obtain the angle moved by the Rover is:

$$\theta = \frac{dr - dl}{L}$$

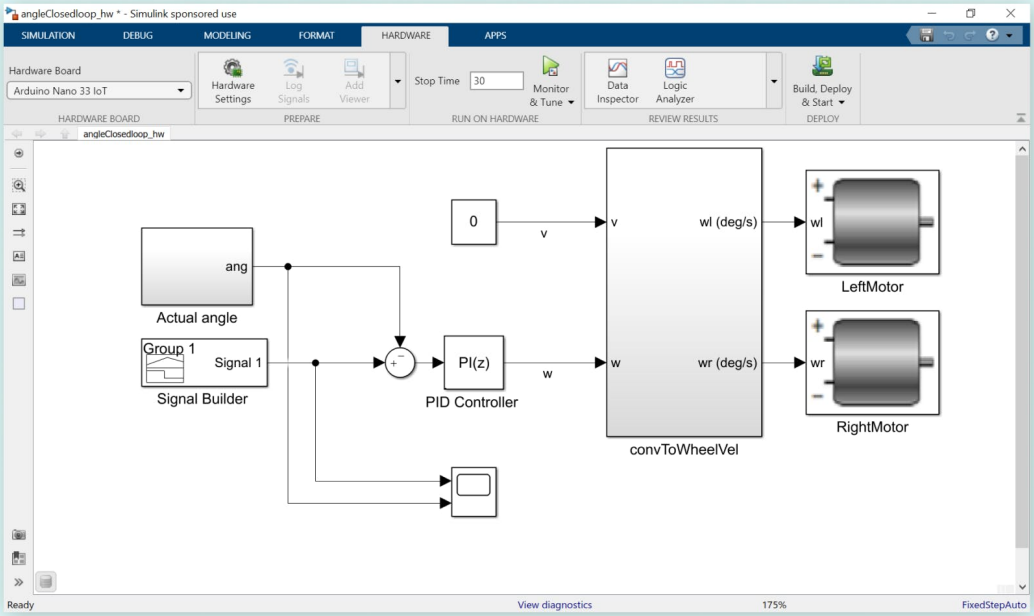
To design the PI controller for the rotation angle of the rover, you can use the same approach that was used for controlling the distance moved by the rover. Open the **angleClosedloop_hw** model:

```
>> angleClosedloop_hw
```

You can see the analogy between this model and the previous one. In this case, the linear velocity is kept constant (zero) while the rate of rotation is being controlled with the `PID Controller` block. This time instead of having distance as the input, it is the rover's angle that is considered; this data is obtained from the encoder sensor using the equation above. If you would like to

Help

understand more about how the data is being captured, open the **Actual angle** subsystem and explore.



Open the **PID Controller** block and note the PI values. These values `P = 3` and `I = 1` were a good starting point during the tests we performed when designing the experiment. In our case, the rover was moving on a relatively hard, carpeted surface. When deployed, the rover should rotate in a circle about itself and attempt to reach 90 degrees.

Monitor & Tune the model with your rover on the floor and tune the PI values as needed until you're satisfied with the performance within the circumstances of your experimental setup, just as you did in the previous section.

Review/Summary

- We saw how to design PID controllers to control the position and angle of the rover.
- We saw how to use Monitor & Tune (External mode) to tune the P, I and D gains by trial and error in Simulink and monitor the results using the Scope block.

Help

Files

- distanceClosedloop_hw.slx
- angleClosedloop_hw.slx

Learn by Doing

Implement a PID controller and understand how the differential term affects your controller performance on different surfaces.

Help

5.4 Program Rover to follow Path Instructions

Now that you've learned how to implement a closed-loop PID controller on the rover, let's see how to make the rover follow a sequence of path instructions. Simulink and Stateflow will be used to instruct the rover to move forward a specified distance and turn a specified amount. We will also see how state logic can eliminate undesired effects like **steady state error** and **overshoot** of your controller.

In this exercise, you will learn to:

- Use states to program your rover to follow a sequence of path instructions,
- Use state logic to eliminate steady-state error and overshoot of your PI controller,
- Understand State logic.

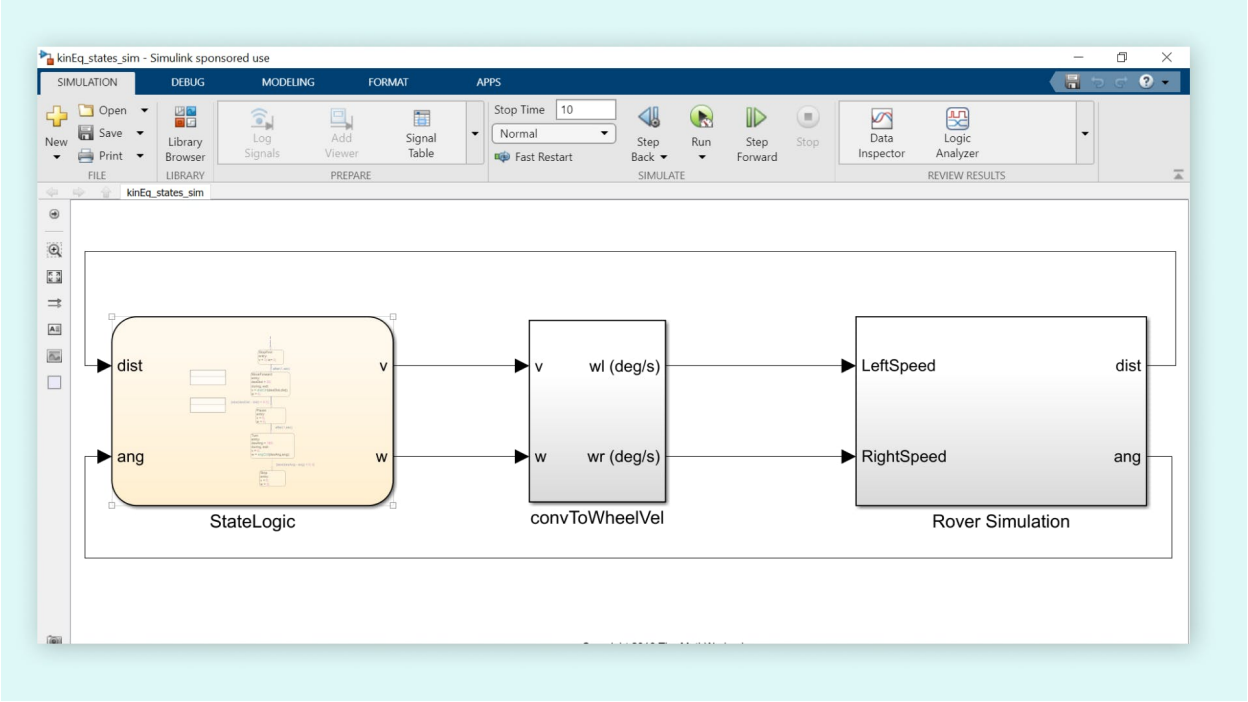
Simulate the Rover Following a Sequence of Path Instructions

For you to start experimenting, a model including the state logic needed to command the rover is provided. Start by opening the model `kinEq_states_sim.slx` :

```
>> kinEq_states_sim
```

This model will execute a simulation of the movement of the rover. It has the **convToWheelVel** and **Rover Simulation** subsystems and the StateLogic chart. The outputs of the Rover simulation, distance, and angle are fed to the state logic in the model, creating a feedback loop.

Help



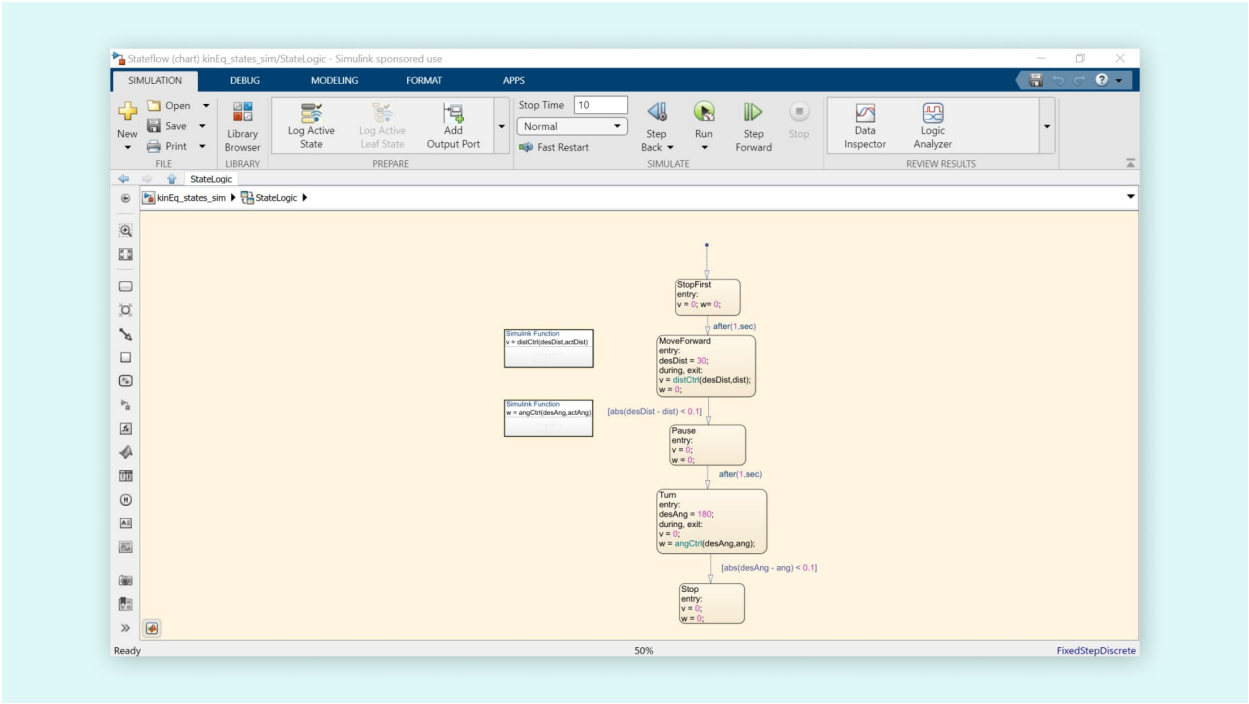
This model uses the same **convToWheelVel** subsystem from previous exercises to calculate wheel speeds based on desired rover velocity and rate of rotation, which will be provided by the **StateLogic** chart.

The **Rover Simulation** subsystem has one modification from the previous exercises. It functionally is still the same, but now it also outputs values for rover distance and angle.

These values (distance and angle) are fed into the **Stateflow Chart** represented by the **StateLogic chart**, which calculates the rover's velocity and rotation rate.

Before opening the chart, you should be introduced to Stateflow. Stateflow is a software package that extends MATLAB and Simulink with tools for modeling logic and event-driven systems. This example presents key aspects of Stateflow in the context of the rover project. If you want to explore Stateflow further, you can view the **Stateflow overview video** and other **related videos**.

Open the **StateLogic chart** by double-clicking it. You will see a different type of block diagram where the different steps to be executed on the rover are displayed inside rounded boxes. These are called **states** and are displayed sequentially. They have a **transition condition** that is shown by an arrow at the exit of each box. These different events can be either temporal in nat Help exit the block after a certain time) or boolean statements (e.g., comparing different variables against a constant).



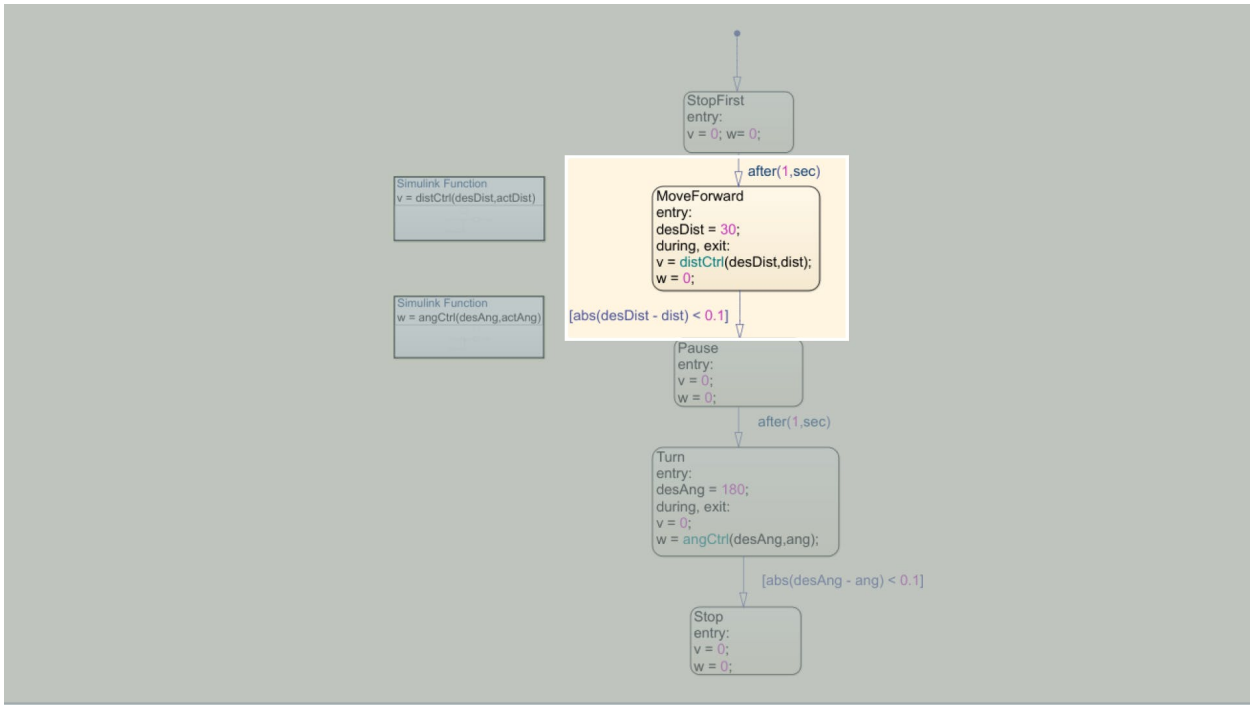
This Stateflow chart has **five states** that each correspond to an operating mode of the rover. When **active**, the contents of the state are executed sequentially.

The rover is in the **StopFirst** state when the simulation begins. In this state, the rover velocity and rate of rotation are set to zero, so the rover is not moving.

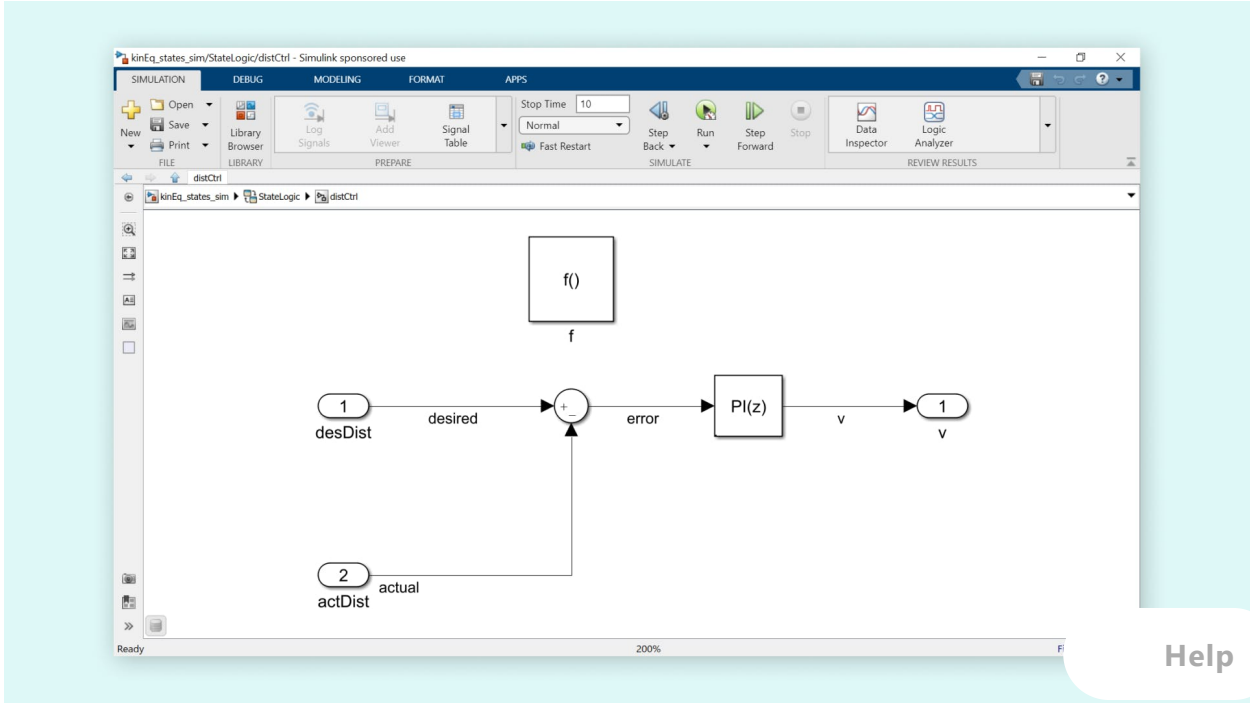
This state becomes inactive when the **transition condition** is met, in this case `after(1, sec)`. The transition condition is tested at every time step of the simulation denoted by the sample time.

One second into the simulation, the rover transitions into the **MoveForward** state where it moves forward a specified distance. The **entry action** is executed immediately upon entering this state, and here the `desDist` is set to **30 cm**. The entry condition can be thought of as "setup" code for each state; the entry code runs before anything else in the chart as soon as the state becomes active.

Help



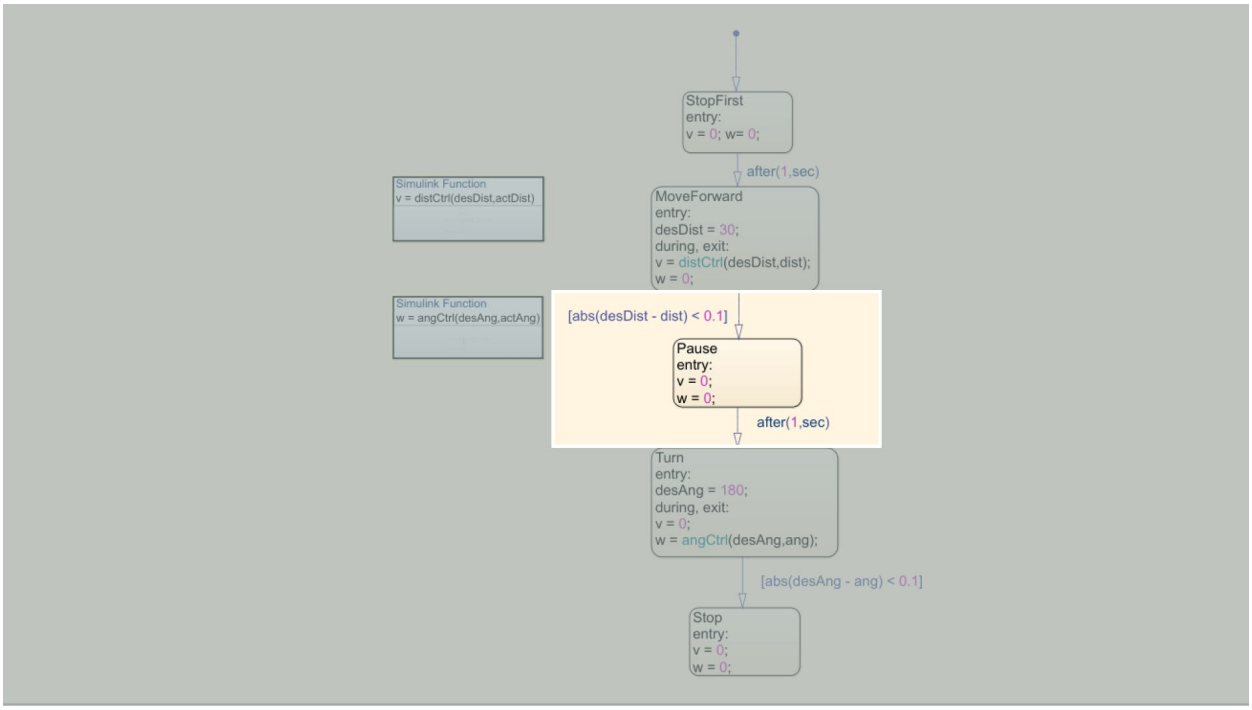
After the **entry** phase and for as long as this state is active, the commands in the **during** action are executed; the code for during can be thought of as the body of a "for" loop. Then, as the state becomes inactive, the exit commands are executed. The Simulink Function `distCtrl` is called, which implements the PI controller that was designed in the previous exercise. A Simulink Function is a function that is defined using Simulink blocks and it can be called at every time step from within Stateflow charts or within other Simulink models.



To transition out of the **MoveForward** state, the next transition condition must be met: `abs(desDist - dist) < 0.1` .

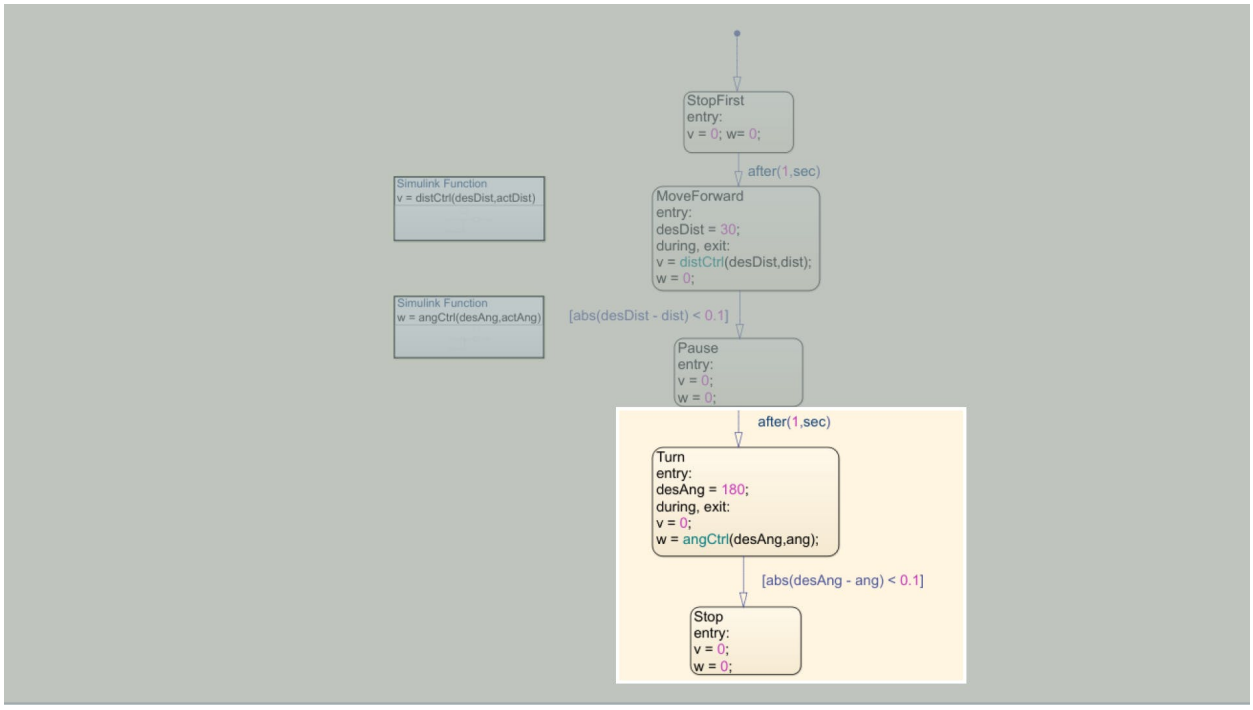
This specifies that when the rover is within 0.1 cm of the desired distance, it should transition to the next state. Note that if your PID controller produces **overshoot** in the rover's distance response, it will be eliminated since the rover transitions to the next state the first time it comes within 0.1 cm of the desired distance, and any oscillatory behaviour will be eliminated.

After moving forward **30 cm**, the rover's next state is **Pause**. The rover is stationary in this state, but only for **1** second when the next transition condition is met.

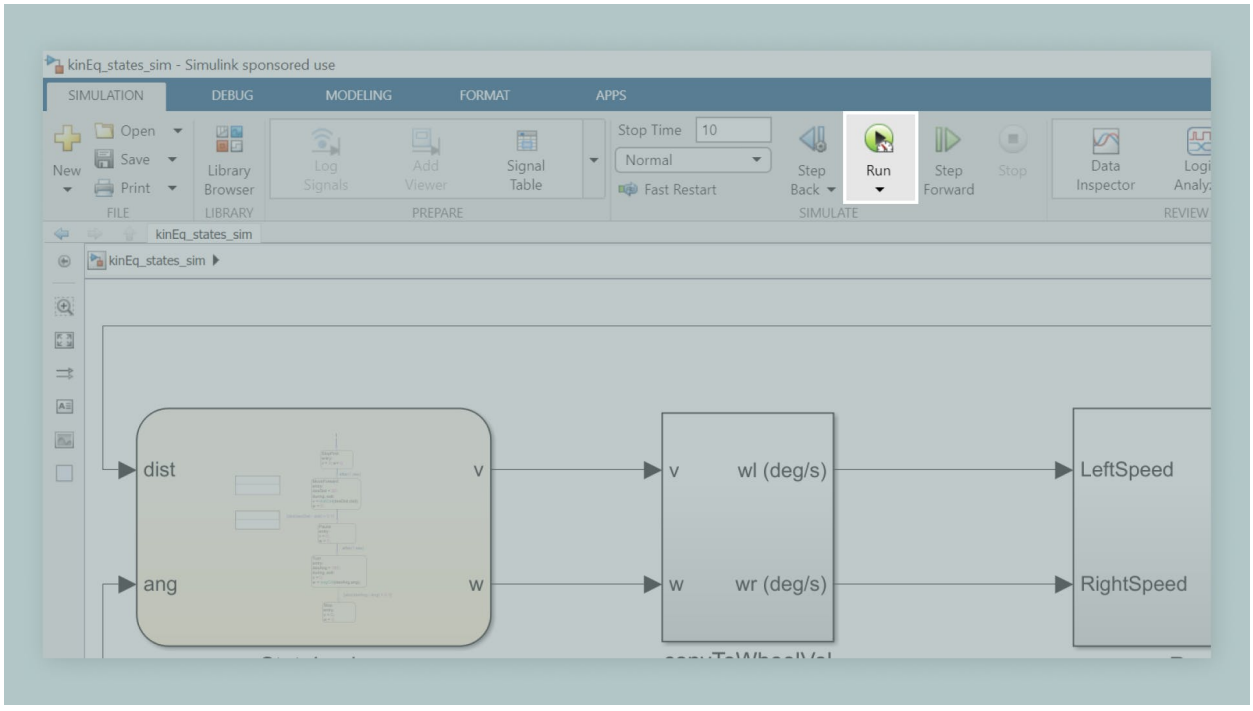


After pausing, the rover enters the **Turn** state. This state is like the **MoveForward** state except instead of moving forward **30 cm**, the rover is instructed to reach 180 degrees. This state uses the Simulink Function `angCtrl` , which implements the PI controller that was designed in the `angleClosedloop_hw` model. Once the rover angle is within 0.1 degrees of the desired angle, the rover transitions to the **Stop** state.

Help

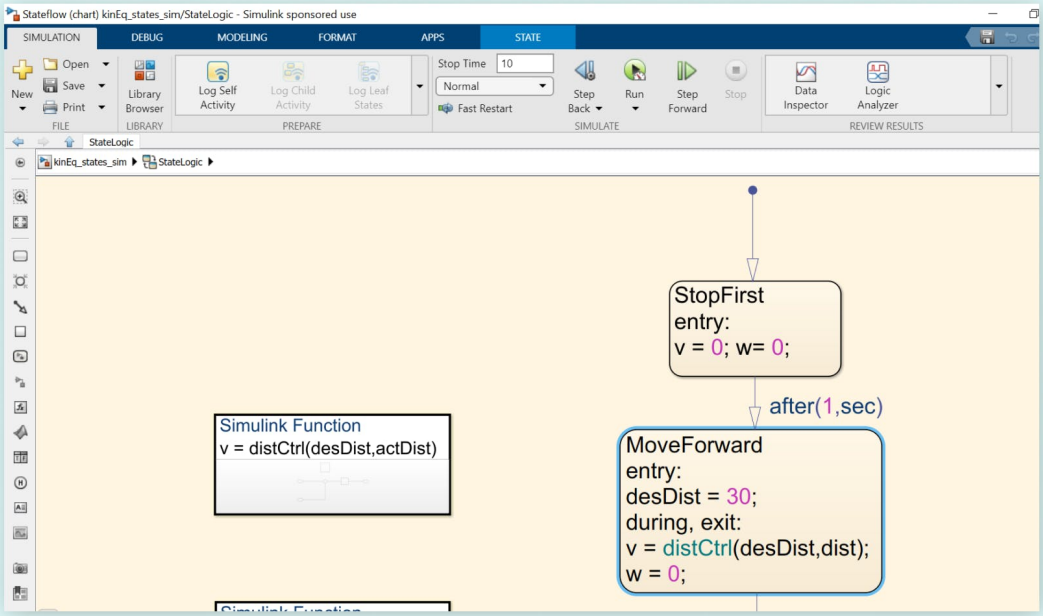


Run a simulation of the model to see how the rover progresses through the states. Do so by clicking the **Run** button on the toolbar from within the **StateLogic** chart, as shown in the following image.



When the simulation begins, you'll see the different states highlighted as they become active.

Help



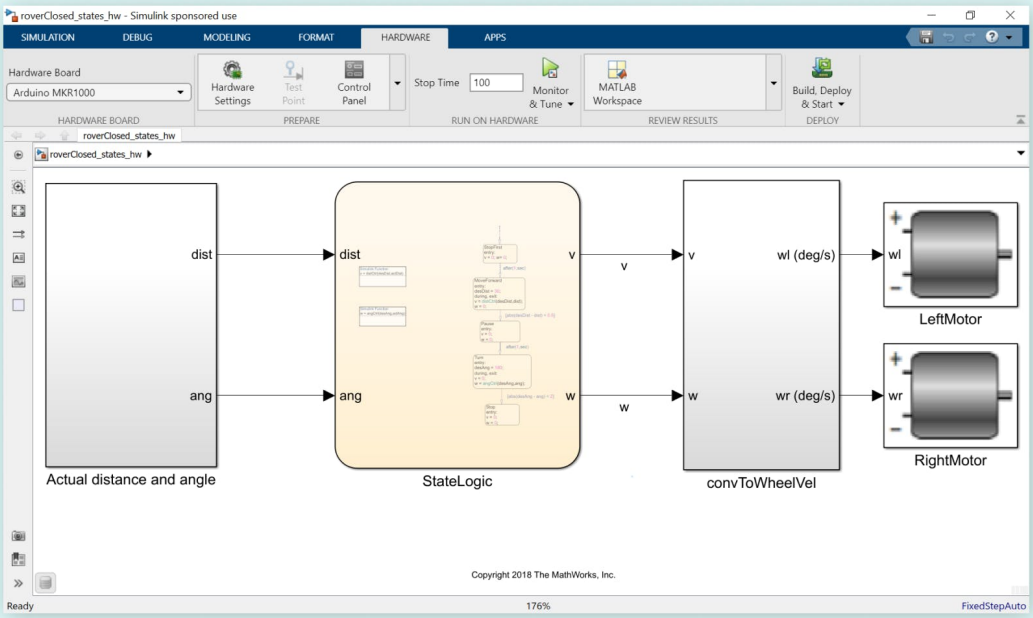
Program the Actual Rover to Follow Path Instructions

Now that you've simulated the rover's behavior to follow the sequence of instructions, let's deploy the application to the actual rover. Connect your rover to the PC using the USB cable and Open `roverClosed_states_hw.slx` model by typing the following:

```
>> roverClosed_states_hw
```

This time, the model has the motors on the right side, and the angle and distance information is fed into the Stateflow chart on the left side of the diagram.

Help



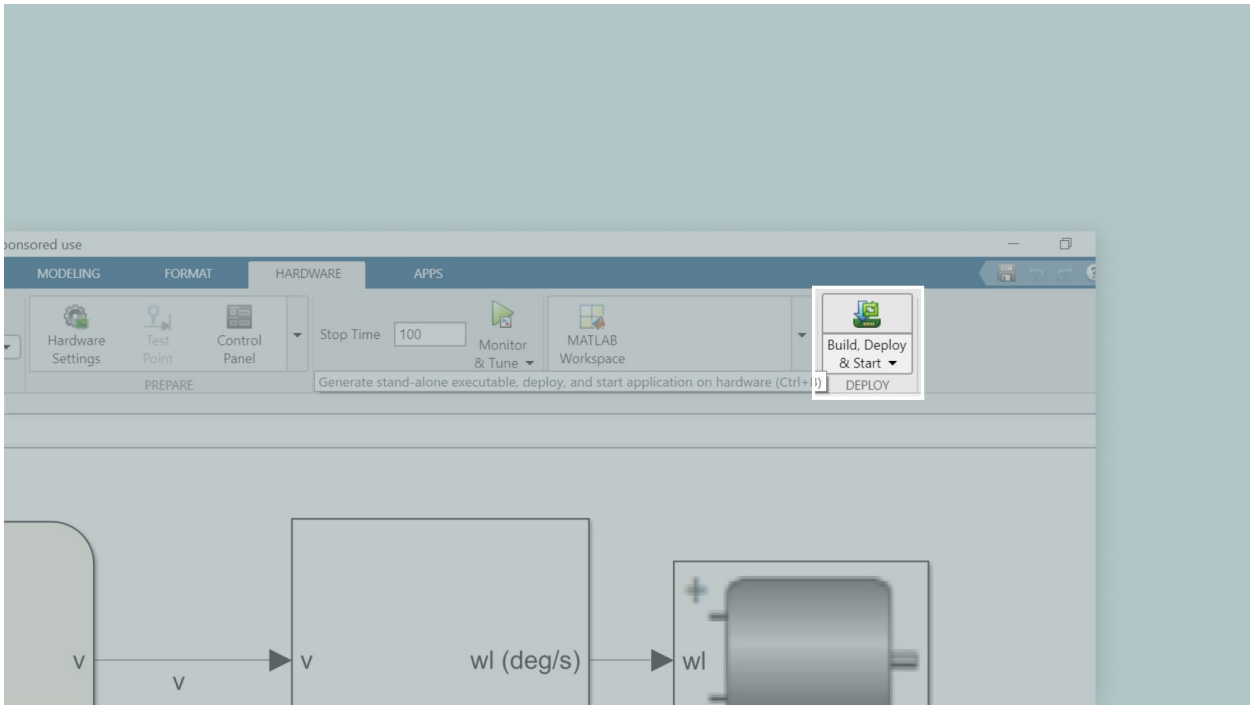
This model is like the simulation model but with a couple of key differences:

- Input to the **StateLogic** chart is now the rover's actual distance and angle, as measured by the encoders.
- The rotational speeds (`wl` , `wr`) calculated in the **convToWheelVel** subsystem are the input to the actual motors.

Before running this model on the rover, adjust the PI controller gains inside **StateLogic** to match your designs from the previous exercise.

Click the **Build, Deploy & Start** button to download the code to the Arduino Nano 33 IoT.

Help



Once the build process completes, place the rover on the floor and power it on using the batteries. Observe how closely your rover follows the sequence specified in the **StateLogic** chart.

If your rover is not moving in a straight line or turning in place during those respective states, it could be because the operating conditions have changed from when you did the motor characterization experiments. You could try any of the following:

- Charging your battery back to the original voltage
- Letting the motors cool off for a while
- Enabling both controllers during rover motion

We have provided a pre-built model that shows how to enable both controllers. To open this model type `roverClosedBoth_states_hw.slx` in **MATLAB Command Window**. In this model, explore the `MoveForward` and `Turn` states inside the StateLogic chart to understand the implementation. Remember, you can also use external mode to debug any issue that you are running into.

Help

Review

- We have seen how to use states to make your rover follow a path sequence.
- We have seen how to use states to disable your controller, thereby eliminating the overshoot and steady-state error of the controller.

Files

- kinEqs*states*sim.slx
- roverClosed*states*hw.slx
- roverClosedBoths*states*hw.slx

Learn by Doing

Expand the path instructions specified in the **StateLogic** chart and see how the rover performs.

Help

5.5 Localization to Calculate Rover Position

When working with Rovers, it's often important to know their location as they move around. This exercise shows how to use a webcam with a localization algorithm to calculate the Rover's location and heading. To ensure the Rover remains in the webcam's field of view, the Rover's movement will be constrained within a designated "Arena".

Start by building the arena and then perform some calibration steps to account for conditions specific to your setup (e.g., lighting, or location of the webcam relative to the arena). Then you will learn how the localization algorithm works and use it to calculate the Rover location and heading within the arena. This will also be used to locate the target that the Rover will move with its forklift.

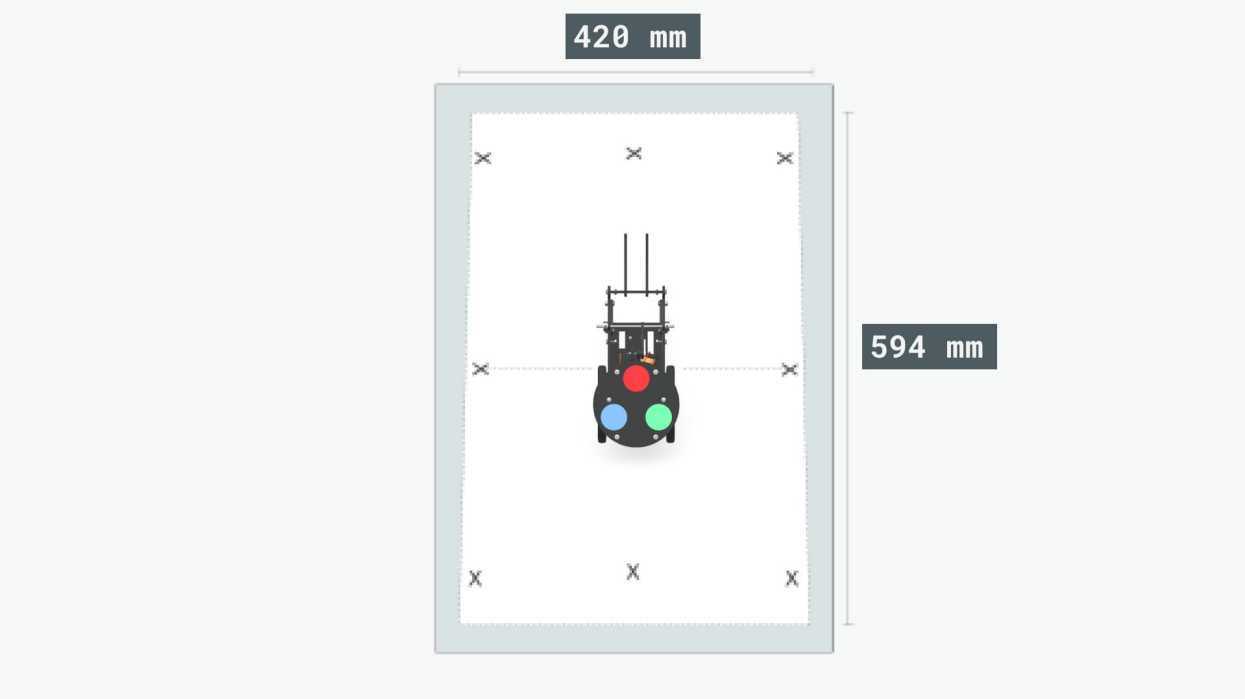
In this exercise, you will learn to:

- Set up and calibrate the Rover's operating environment,
- Use localization to calculate the Rover's position and orientation.

Building an Arena for the Rover

The following image shows an example arena for the Rover made by using some white paper and a black marker. In the picture, you can also see the Rover. This can give you an idea of proportions and about how big or small your arena can be.

Help



The main requirement for your arena is that it must have a white background. You can construct it using white poster board or by stapling together several sheets of paper.

The size of the arena is flexible, but it must fit within the field of view of your webcam. The localization algorithm requires that the webcam be placed at least 100 cm (approximately 3.5 feet) above the arena.

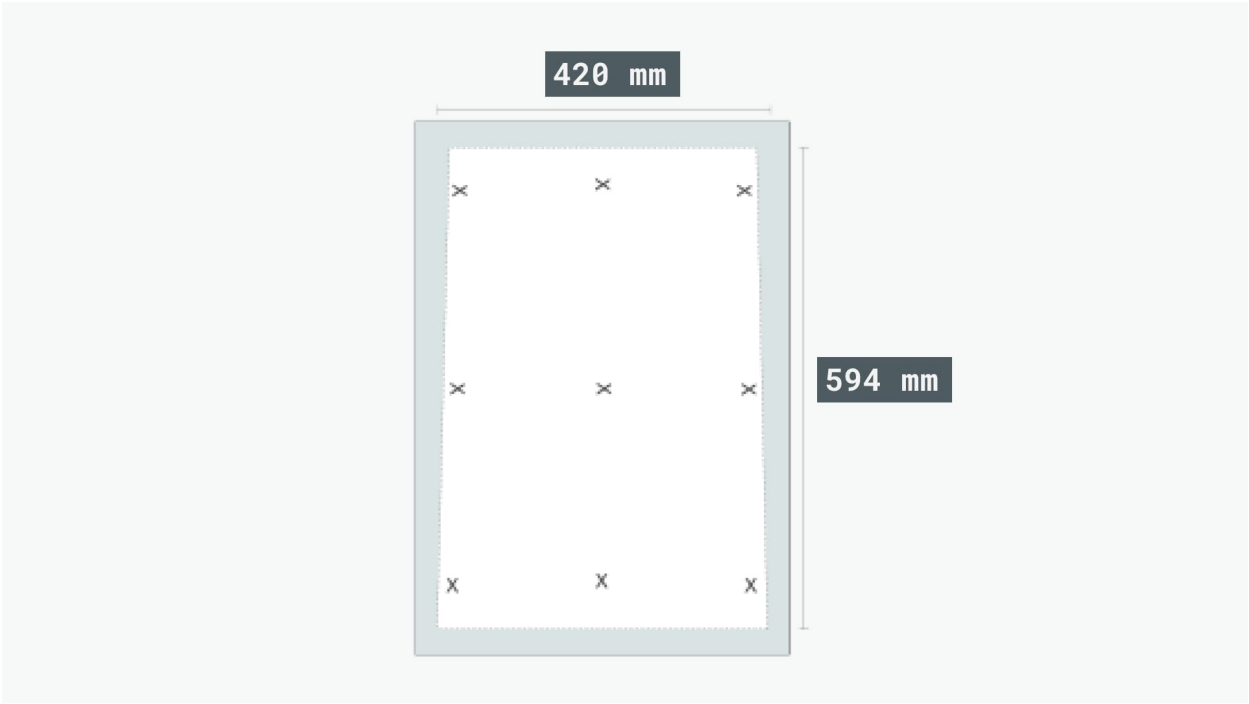
Set up your webcam over the area in which you plan to build your arena and view the video feed in MATLAB by executing the following commands:

```
>> cam = webcam('USB2.0 PC CAMERA');
>> preview(cam)
```

Note: You will need MATLAB Support Package for USB Webcam installed to run these commands. Refer to section 1.3 Installing MATLAB, Simulink & Add-Ons.

Adjust the location of the webcam and the size and placement of the it fits within the field of view as shown in the image below.

Help

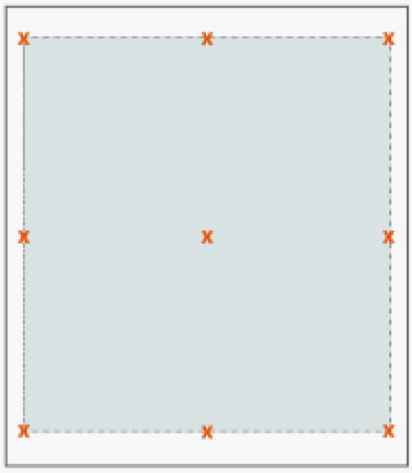


You can accommodate larger arena sizes as you move the webcam farther away, but at a certain point, the accuracy of the localization algorithm will be compromised. From previous tests, it was found that an arena-size of 75 cm x 75 cm located within 450 cm (approximately 15 feet) of the webcam works well.

When you build your arena, ensure that the white background extends beyond the actual size of the arena. Based on our experience, roughly a 5 cm padding on all sides is needed for the image processing algorithm. This will ensure that there is some buffer space between the floor and the arena.

Mark the corners of the arena, centers of all edges and the overall center point with an X that is visible from at least 100 cm away, as you saw in the picture. A black marker was used for this operation.

Help



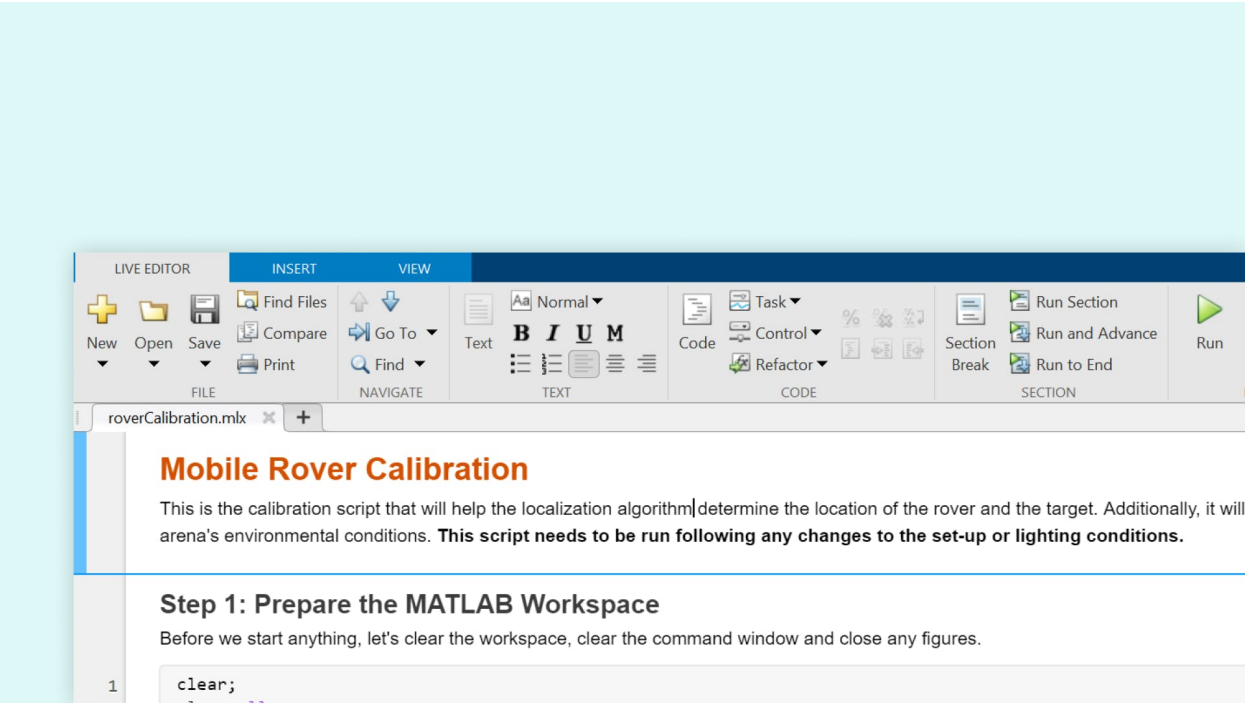
Confirm once again that the webcam is at least 100 cm above the arena and the entire arena is within the field of view. The next step is calibrating the image processing algorithm.

Calibrate the Arena and Environment

The `roverCalibration` script has five steps to perform the calibration needed by the localization algorithm. Open the script and run through the steps.

```
>> edit roverCalibration
```

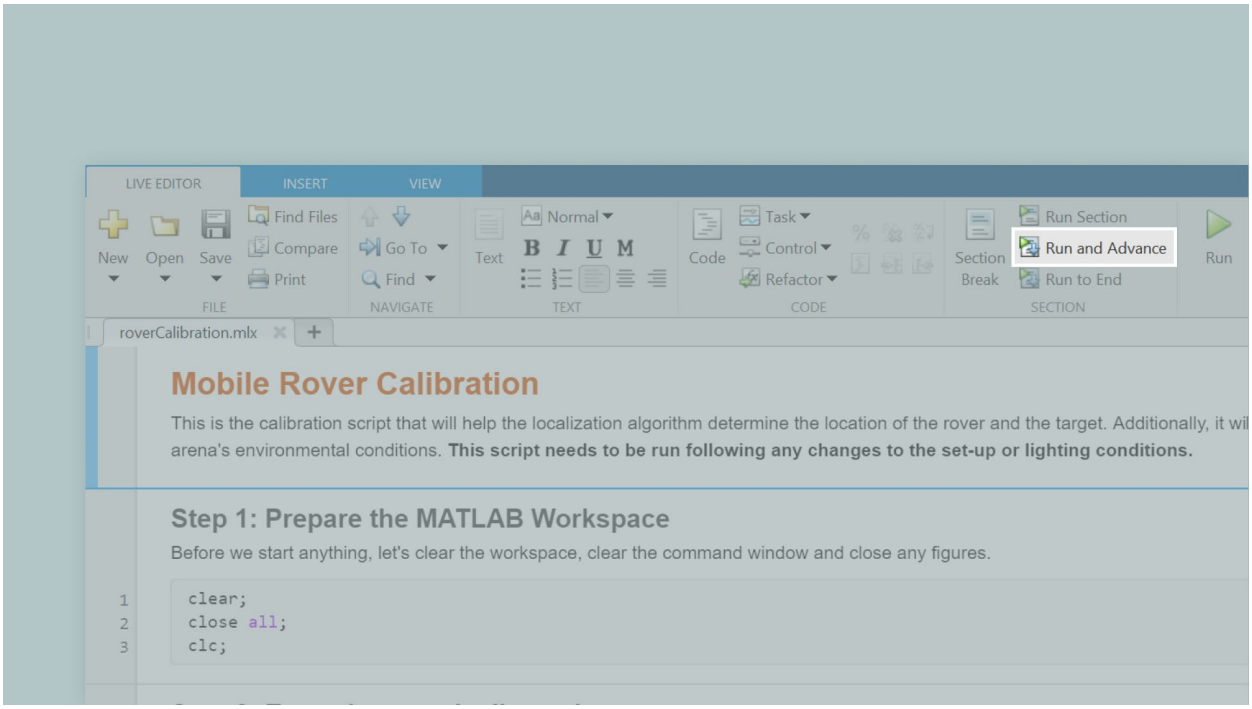
Help



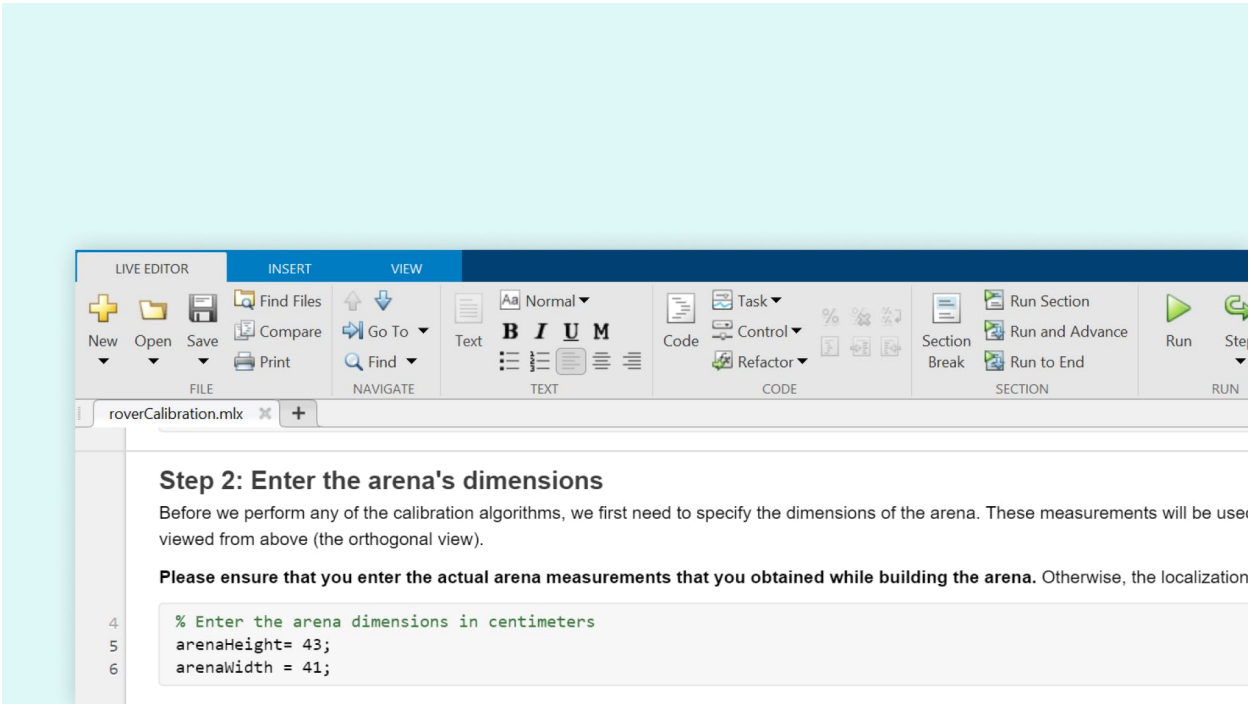
Note: The calibration process needs to be executed every time there are changes to the webcam/arena set-up, including lighting conditions. It is a time-consuming task, so it is recommended to wait until you are comfortable with your setup before running this program. Here is a step-by-step explanation of how the calibration process is performed.

The script's section called "Step 1: Prepare the MATLAB Workspace" helps clear any existing variables and close any open figures. Click **Run and Advance** to execute this code and move to the next section.

Help



For the calibration to work, the algorithm needs to know the arena dimensions. Enter your arena's height and width in "Step 2: Enter the arena's dimensions" as shown below. Units must be in centimetres.

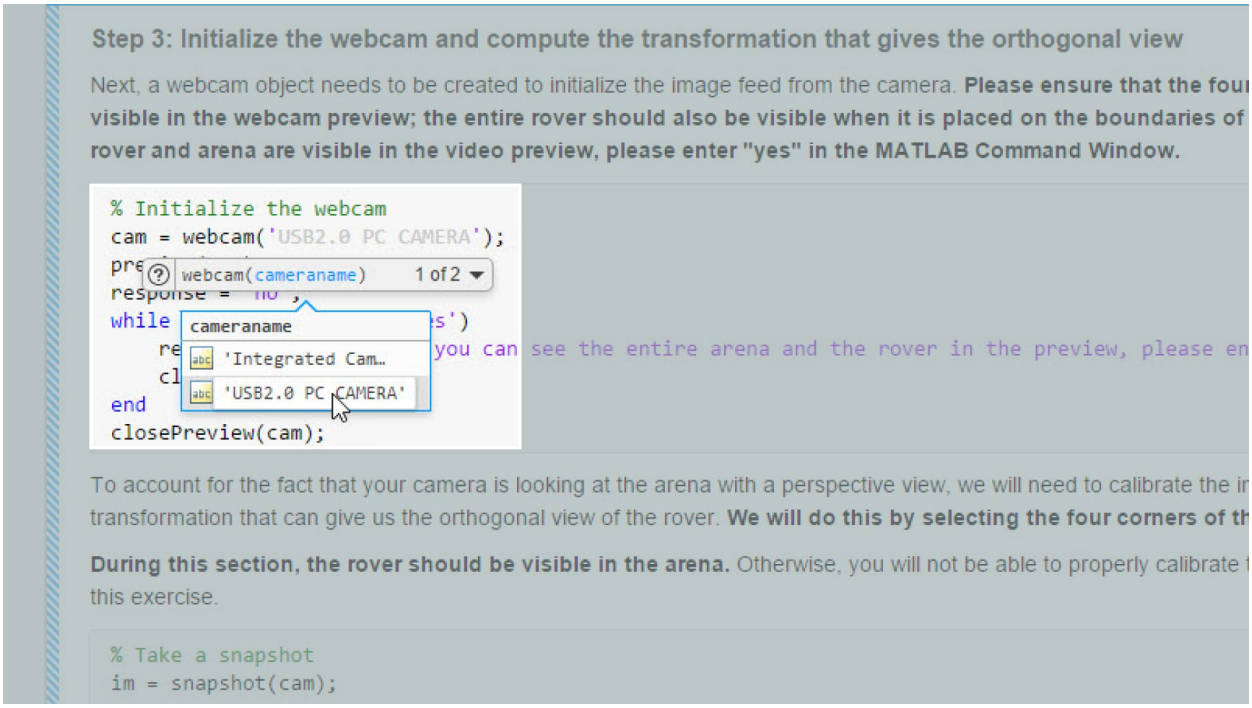


After entering the dimensions, click **Run and Advance**.

Help

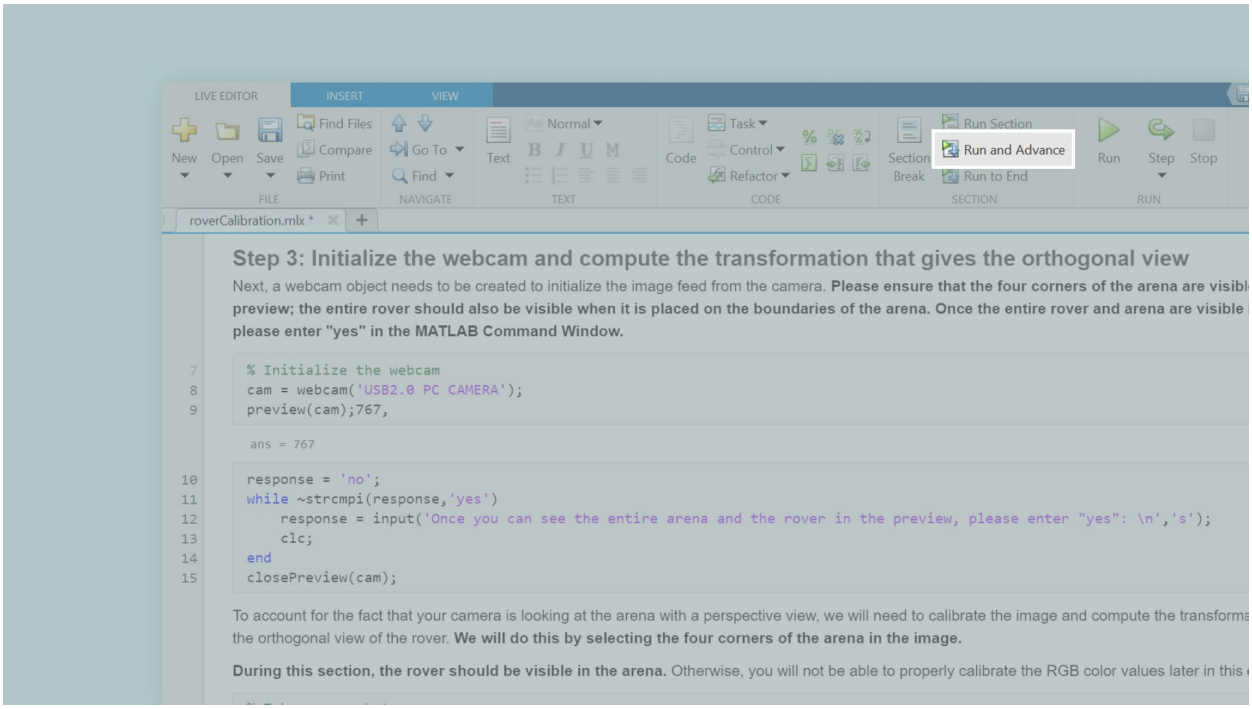
In "Step 3: Initialize the webcam and compute the transformation that gives the orthogonal view," the webcam will be initialized. Later, the transformation that

gives the orthogonal view of the Rover will be computed. To select your webcam, you can use the automatic suggestion feature inside the Live Script as shown below.

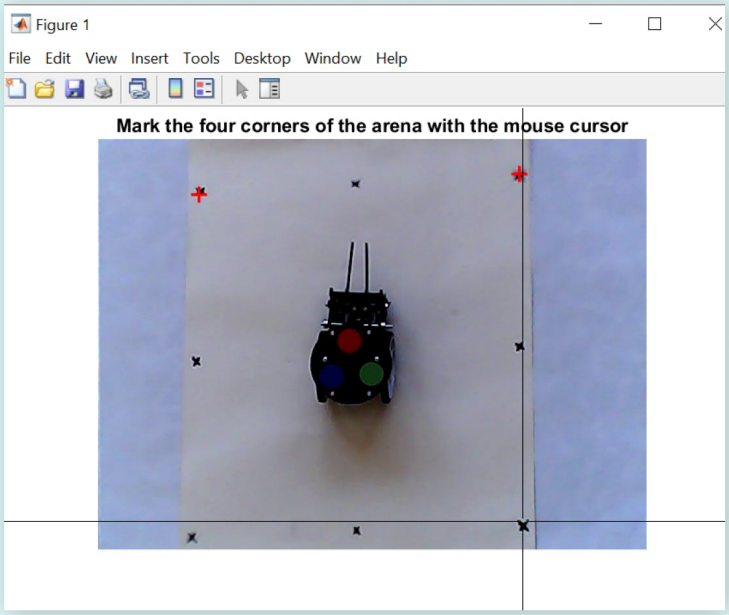


Now let's look at the transformation portion of this step. Typically, it is hard to place the webcam directly over the arena, and instead, the camera is placed where it takes images from some downward angle. Image processing can be used to translate the image to a corrected version of itself, which will make it easier for our localization algorithm to calculate the position. Execute this section by clicking the **Run and Advance** button to see what effect this has on the image. Follow the instructions that appear in the MATLAB Command Window.

Help



The webcam image will be displayed; to compute the transformation of the image, click the four corners of the arena in the image.



Important! Ensure that the Rover is in the arena. If it is not, you need to execute this section of the code once more with the Rover placed in the arena.

Help

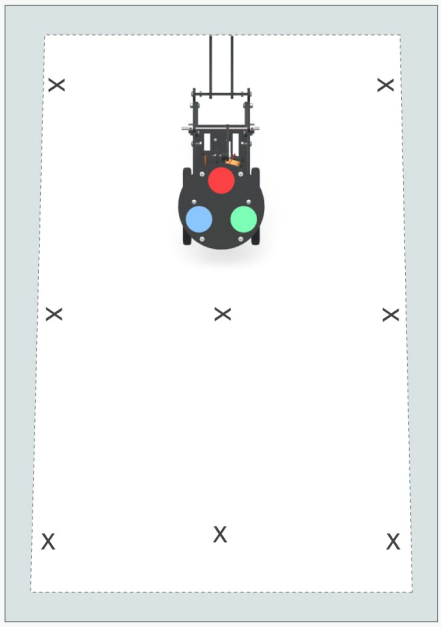
If you would like to look at the transformed image, execute the following command in MATLAB:

```
>> imshow(orthoImg)
```

Note: It is common for orthogonally rotated images in this project to have missing portions of image. These are the black portions at the bottom of the image above.

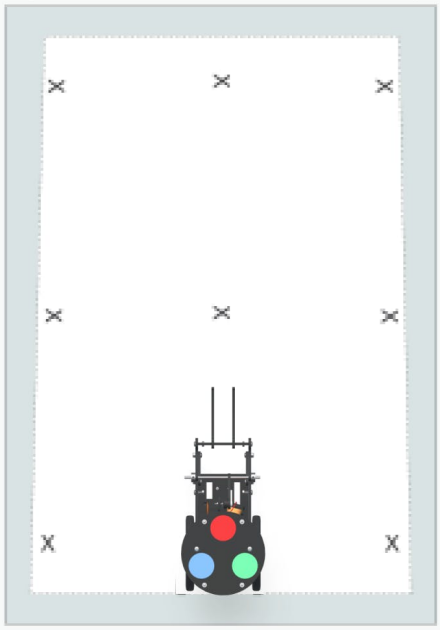
Because the webcam takes pictures from an angle, the Rover's height is altered differently for various distances. Step 4 has been split into "Step 4a: Compute the location offset for the Rover" and "Step 4b: Compute the location offset for the target". In Step 4a, you will capture images of the Rover on the four edges of the arena (**TOP**, **BOTTOM**, **LEFT** and **RIGHT** as shown in the image series below) to account for this.

TOP

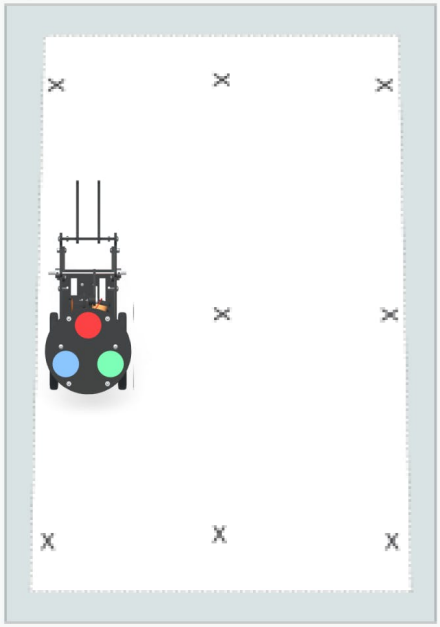


Help

BOTTOM

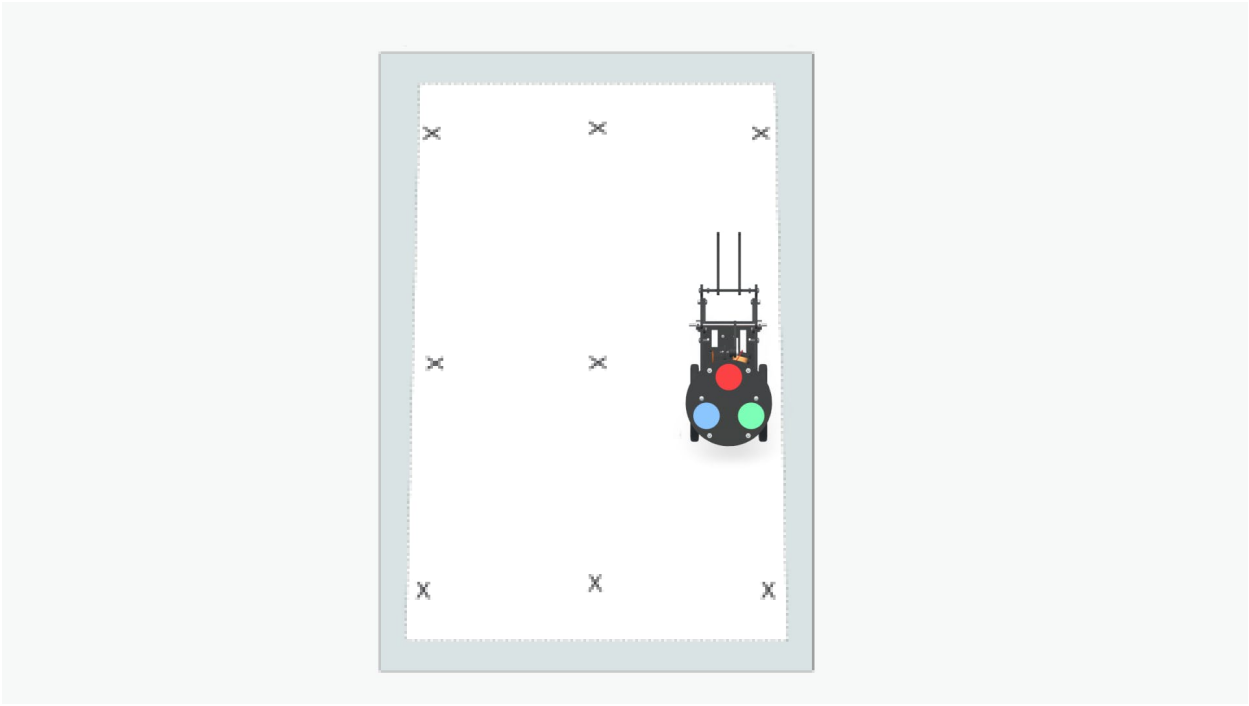


LEFT

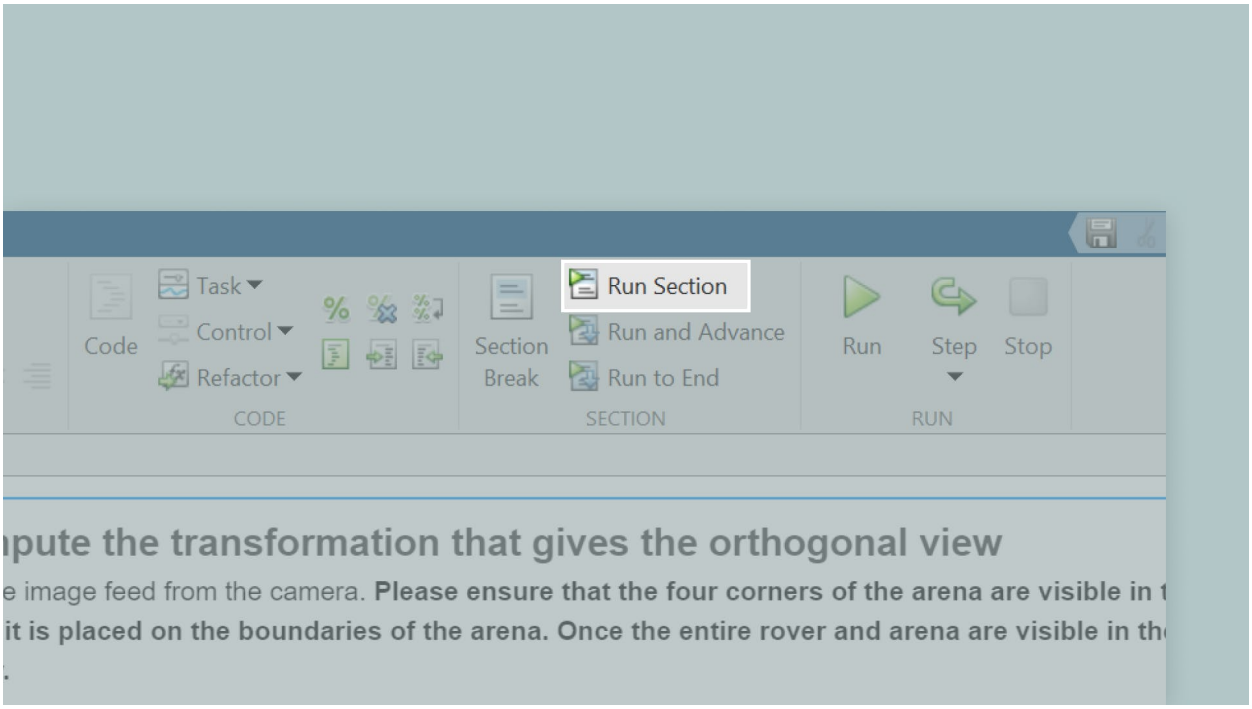


Help

RIGHT

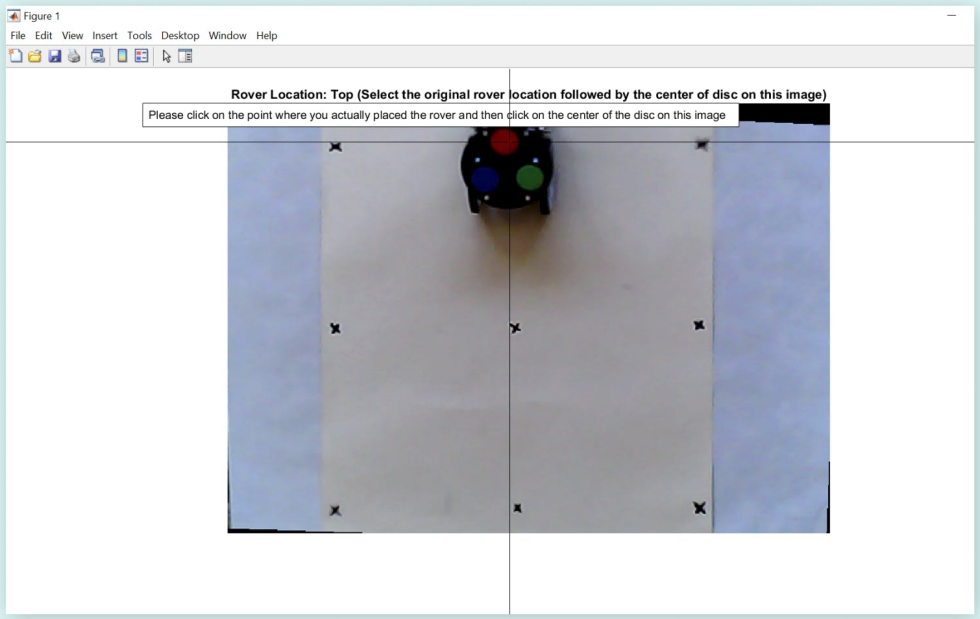


Run the first section of Step 4a of the live script by clicking **Run Section**, then place the Rover as instructed in the prompts displayed in MATLAB.



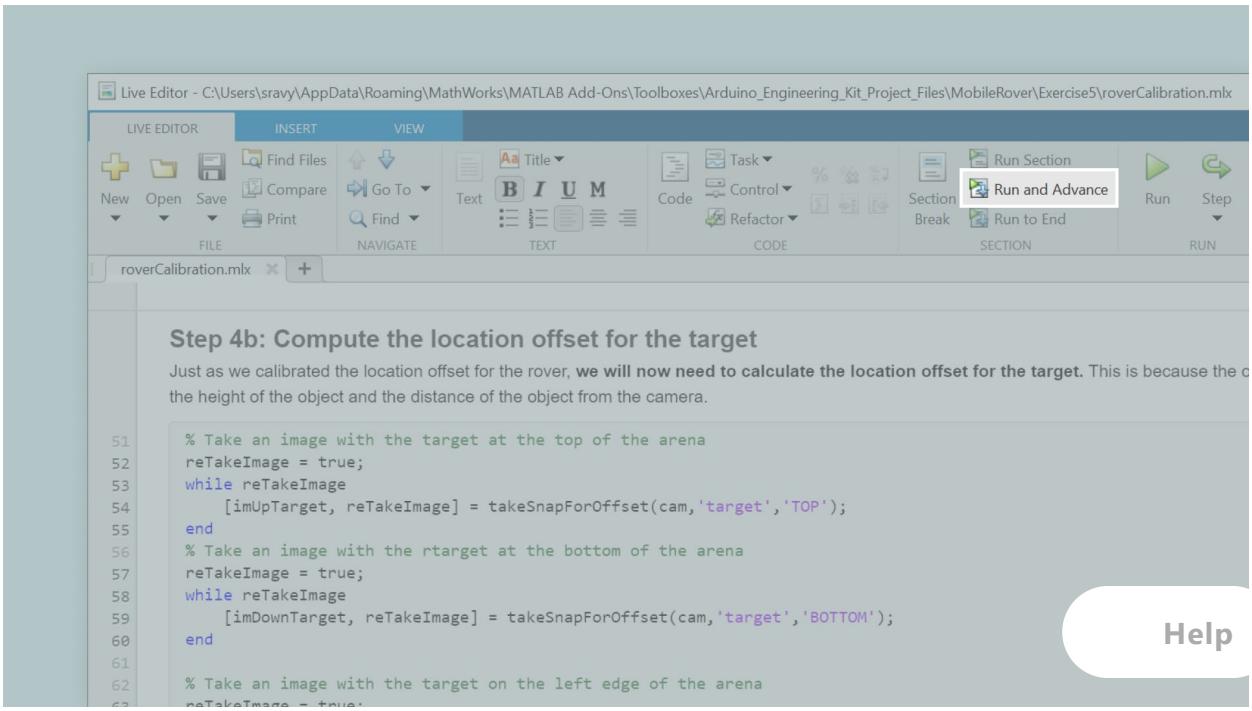
In the next section of code, the `calibrateLocationOffset` function operates on the images that you just took. The image that pops up sl Help clear visualization of how the height is translated differently for different depths

from the camera. Click the points specified in the pop-up figures (centre of the Rover and the centre of the disc) to compute the offsets.



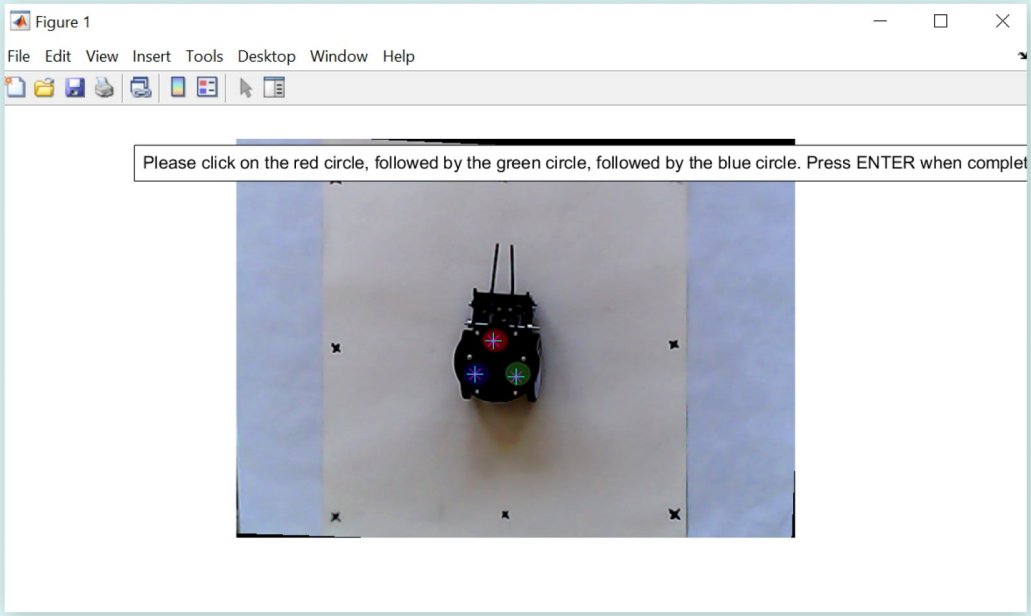
Now let's click the section entitled "Compute the offsets of the Rover" and then click **Run and Advance** to perform all the necessary tasks.

Next, let's repeat these tasks for the target in Step 4b. Again, click the **Run and Advance** button and follow the prompts displayed in MATLAB.



Run the next section titled "Compute the location offset for the target", where you will place the Rover in different locations as per instructions on MATLAB **Command Window** and then click the locations where the target was placed followed by the centre of the target on the image. The localization results depend heavily on these steps and can cause issues if the calibration is not done accurately. A sample image is offered in that section of the live script.

In "Step 5: Calibrate the colour threshold of the disc", you will see how to account for the lighting. Click this section of the code to select it and click **Run section** of code. On the image that appears, ensure that you click the circles in this order: Red first, Green next, and then Blue.



Note: Once you have clicked the circles, you should press the ENTER key to confirm your selections.

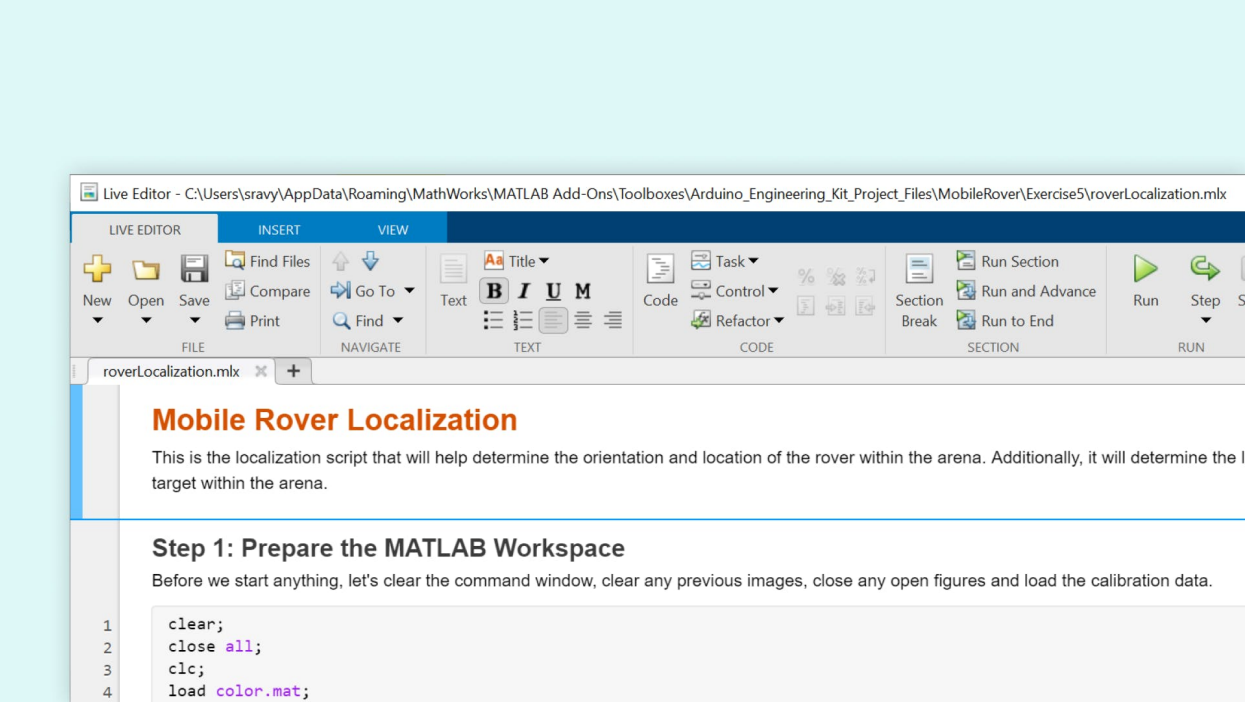
With that, you have now calibrated the parameters to get an orthogonal view of the Rover, corrected for any location offsets within that view and accounted for how lighting can affect the RGB values for colours. All the calibration data is saved and will be reused during the execution of the localization scri

Help

Localization of the Rover and the Target

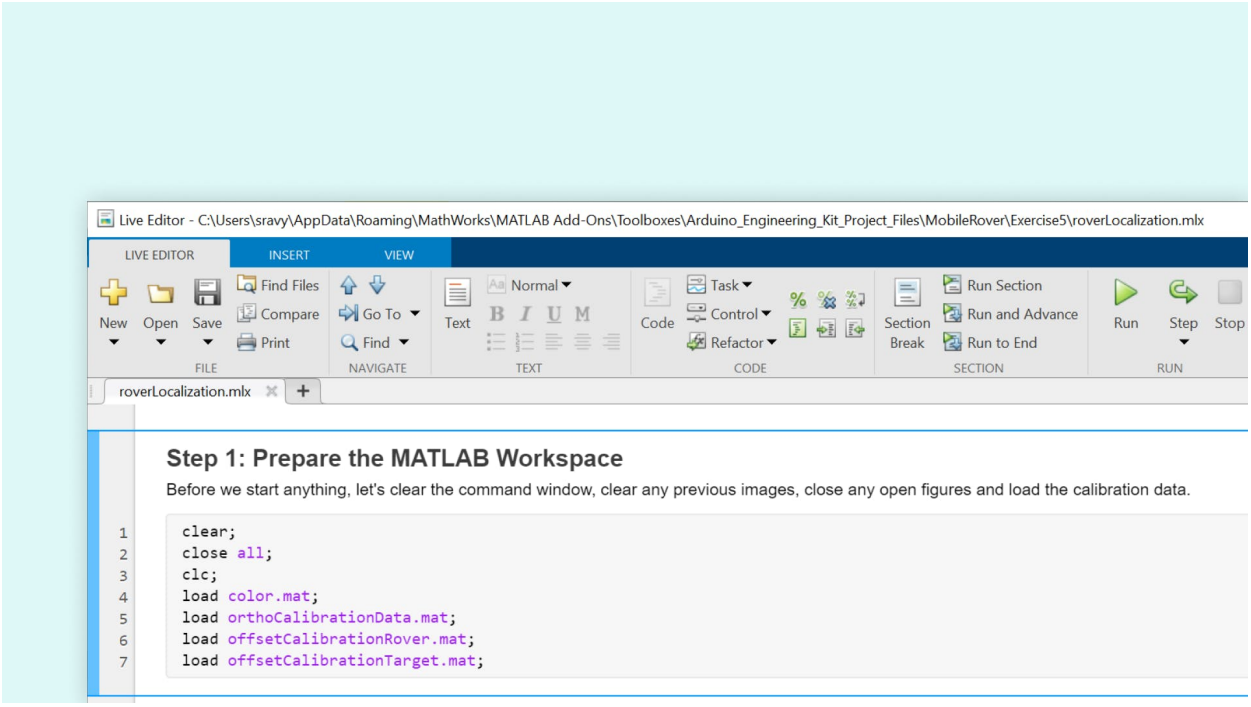
Now that you've completed the calibration, let's see how to use the localization algorithm to calculate the position and heading of the Rover within the arena in five simple steps. Open the live script `roverLocalization.mlx` :

```
>> edit roverLocalization
```

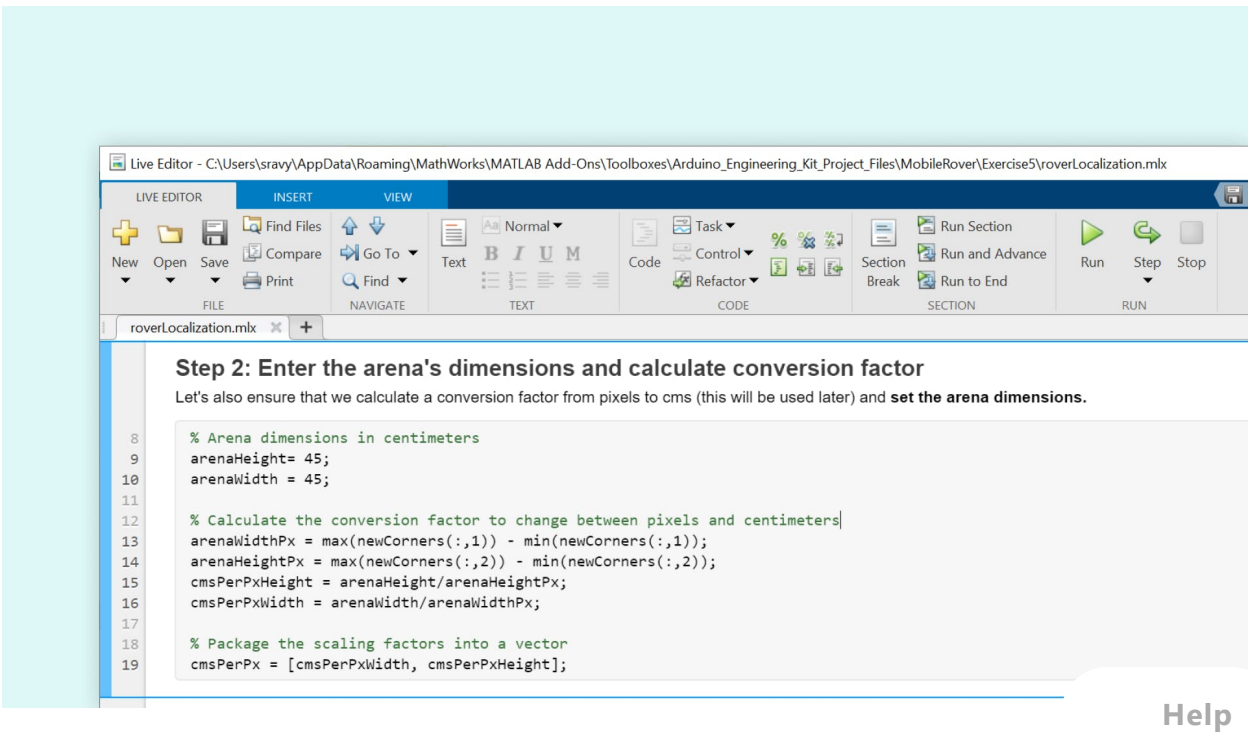


Let's run "Step 1: Prepare the MATLAB Workspace" of this live script to clean up the **Workspace** and load the Calibration data generated earlier.

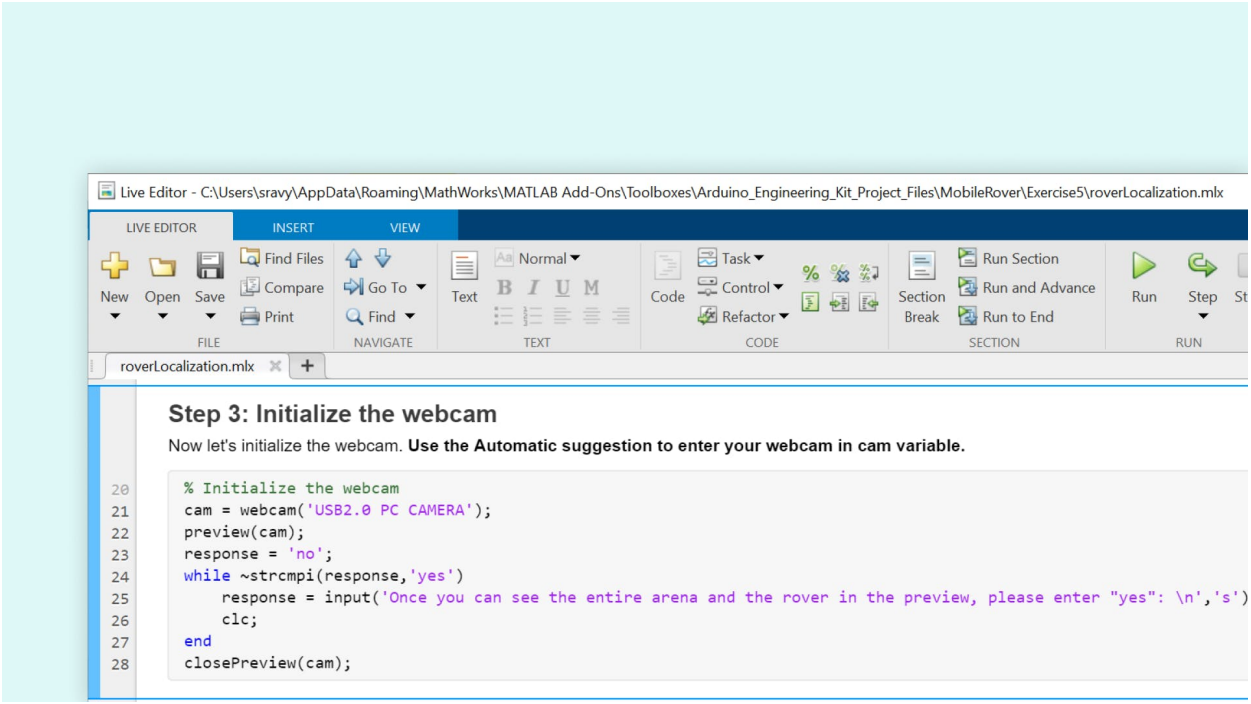
Help



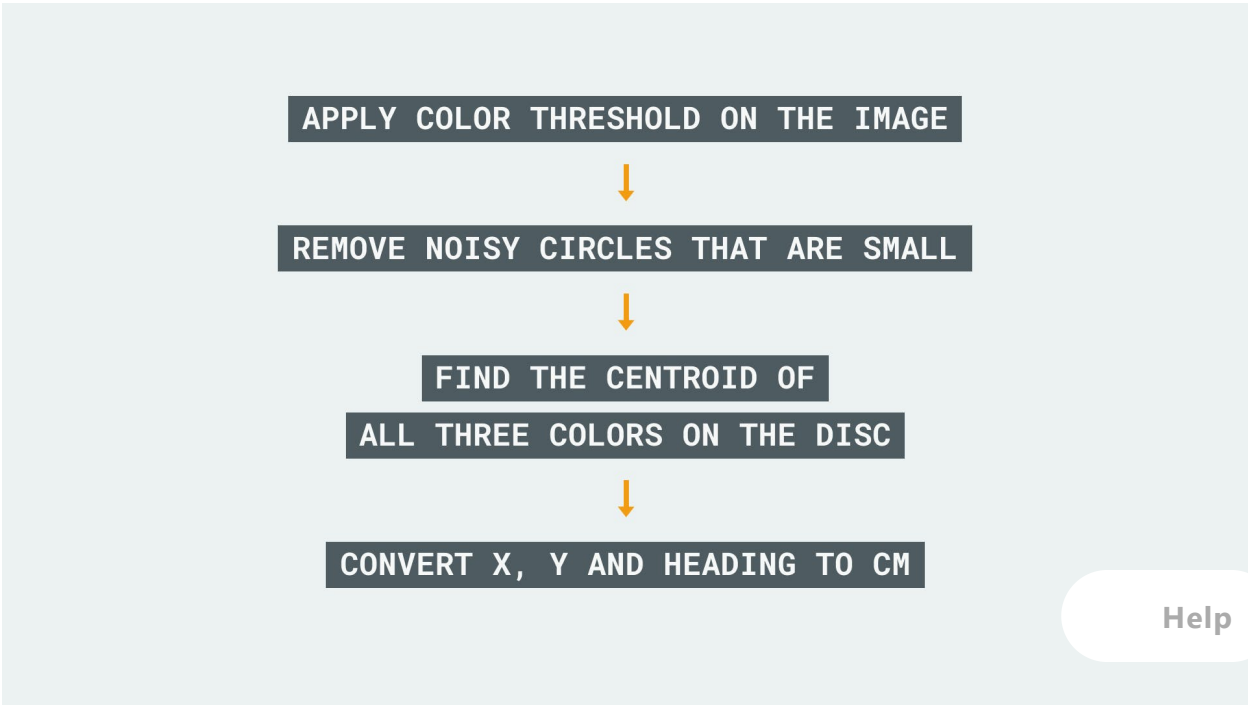
Enter your arena's dimensions in "Step 2: Enter the arena's dimension and calculate conversion factor" and run this section of code. The conversion factor calculated in this section relates the real-world dimensions of the arena in centimetres to the number of pixels within the image. This is achieved by dividing the physical arena measurements with the corresponding number of pixels from the image.



In "Step 3: Initialize the webcam," select the appropriate webcam using the automatic suggestion feature that was introduced in the previous section and run this section of code to initialize the webcam.



The following section, named "Step 4a: Get the location and heading of the Rover", uses the calibration data to determine the location and heading of the Rover within the image. Shown below is a schematic that provides an overview of the localization algorithm itself.



Line-By-Line Explanation: roverPosAng Function

To obtain the Rover position and heading from the orthogonal image, we created the `roverPosAng` function.

This function calculates Rover position `(x,y)` and heading. `x` and `y` are returned in pixel coordinates.

```
function [x,y,heading] = roverPosAng(IMG,P)
```

Create a disk-shaped structuring element using the `strel` function. A structuring element is a key component of morphological operations. It represents a binary valued neighbourhood of pixels, in which the true pixels are included in the morphological computation, and the false pixels are not. After creating the structuring element, apply the RGB threshold based on the calibration data in vector P.

```
% Create a disk object (will be used to clean up the image) and apply
the RGB threshold values for the image
delta = 20;
s = strel('disk',3);
threshR = P(1,:) + delta.*[-1 1 1];
threshG = P(2,:) + delta.*[1 -1 1];
threshB = P(3,:) + delta.*[1 1 -1];

% Threshold each channel of the image
Rin = IMG(:,:,1) > threshR(1) & IMG(:,:,2) > threshR(2) & IMG(:,:,3)
> threshR(3);
Gin = IMG(:,:,1) > threshG(1) & IMG(:,:,2) > threshG(2) & IMG(:,:,3)
> threshG(3);
Bin = IMG(:,:,1) > threshB(1) & IMG(:,:,2) > threshB(2) & IMG(:,:,3)
> threshB(3);
```

Use the `strel` object `s` to remove noise from the thresholded images. Any disk with a radius less than 3 pixels is considered noise.

```
% Remove RGB points that do not have a radius of 3 pixels
Rf = imopen(Rin,s);
R = imclose(Rf,s);
Gf = imopen(Gin,s);
G = imclose(Gf,s);
Bf = imopen(Bin,s);
B = imclose(Bf,s);
```

Help

Find the centroids of the three coloured circles on the marker and use that to determine the centre of the disk.

```
% Get the red centroid
sR = regionprops(R,'centroid');
cR = sR.Centroid;

% Get the green centroid
sG = regionprops(G,'centroid');
cG = sG.Centroid;

% Get the blue centroid
sB = regionprops(B,'centroid');
cB = sB.Centroid;

% Determine where the centre of the Rover's coloured wheel is located
cPlate = (cR + cG + cB)/3;
```

Calculate the heading of the Rover based on the red marker that aligns with the forward direction of the Rover. Use the concept of quadrants since `atand` is used.

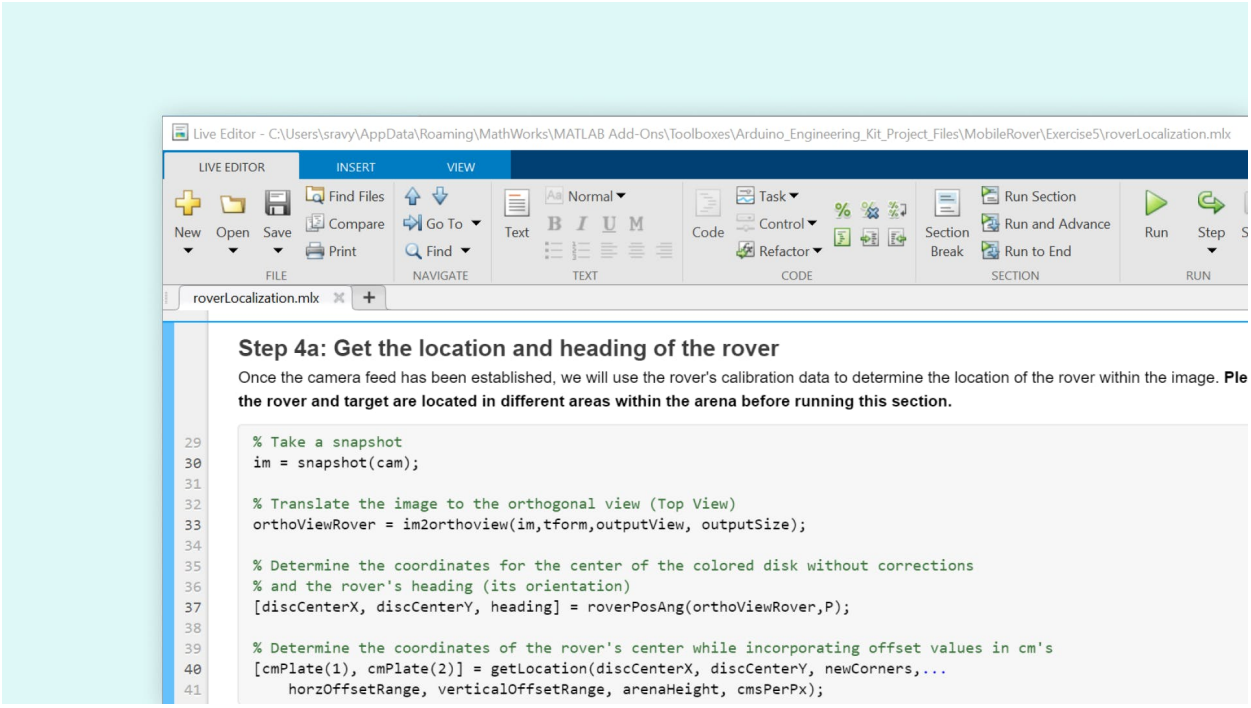
```
%% Heading calculation (Getting the orientation of the Rover)

thetaRC = atand((cPlate(2) - cR(2)) / (cPlate(1) - cR(1)));

% Add 180 offset if red marker lies to the left of center, given the
% conventions noted earlier.
if cR(1) >= cPlate(1) && cR(2) > cPlate(2)
 heading = 180 - abs(thetaRC);
elseif cR(1) > cPlate(1) && cR(2) > cPlate(2)
 heading = abs(thetaRC);
elseif cR(1) >= cPlate(1) && cR(2) > cPlate(2)
 heading = 180+ abs(thetaRC);
elseif cR(1) > cPlate(1) && cR(2) > cPlate(2)
 heading = 360 - abs(thetaRC);
end
```

Output `x`, `y`, and `heading` variables

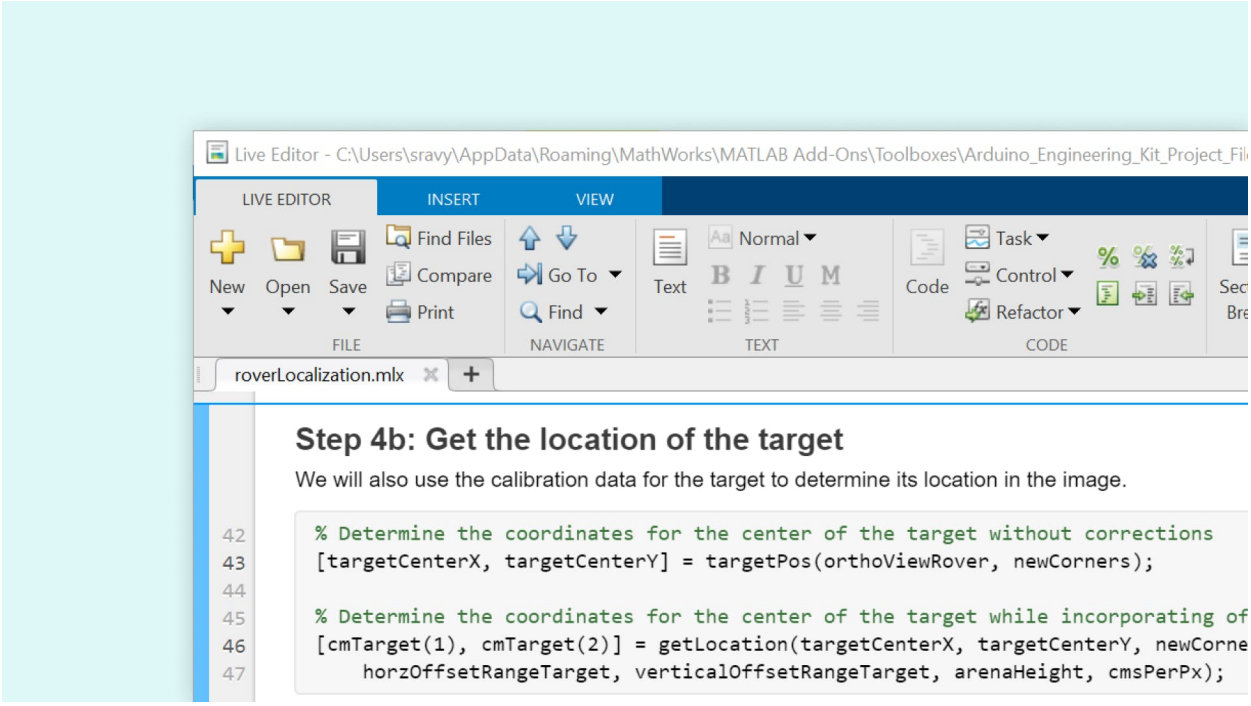
```
% Output the heading and (x,y) coordinates of the Rover
x = cPlate(1);
y = cPlate(2);

if heading > 360
 heading = heading - 360;
end
```

Help

For more details, refer to `roverPosAng` and `getLocation` functions. Let's **Run and Advance**.



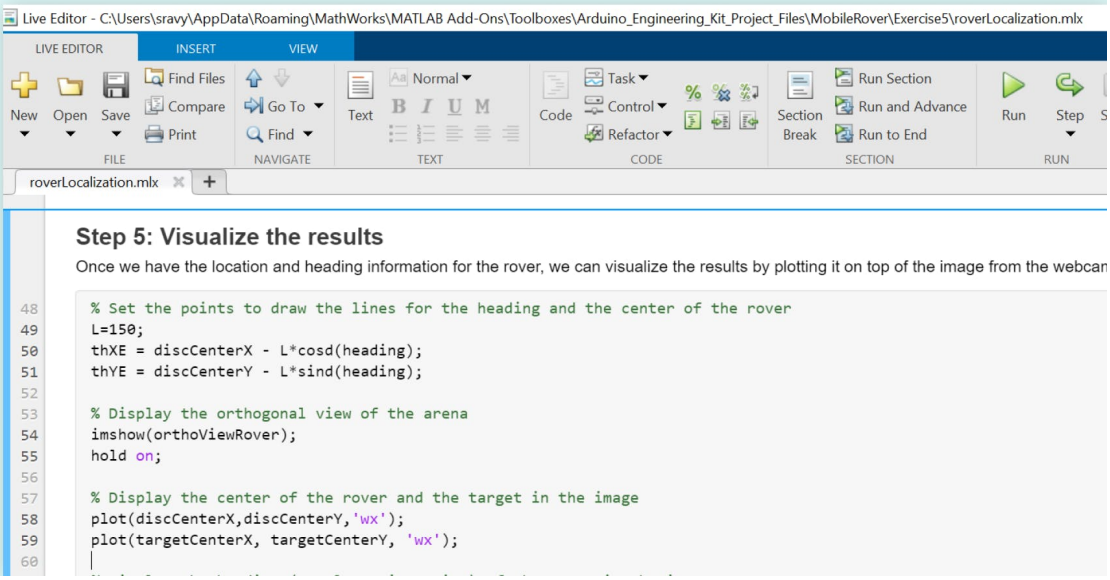
In "Step 4b: Get the location of the target", the location of the target is calculated using the calibration data. Let's run this section of code.



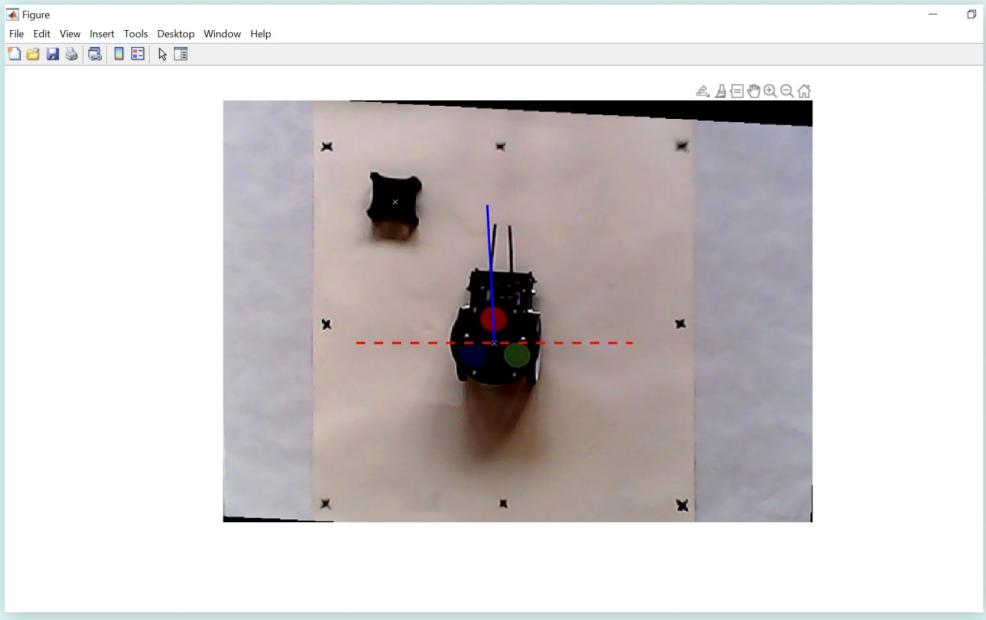
Help

Now that you have the location and heading information of the Rover and target, run "Step 5: Visualize Results" to visualize the results superimposed on

top of the actual image.



When running this code, you should see an image displaying the whole scene with the Rover, the target, and the arena:



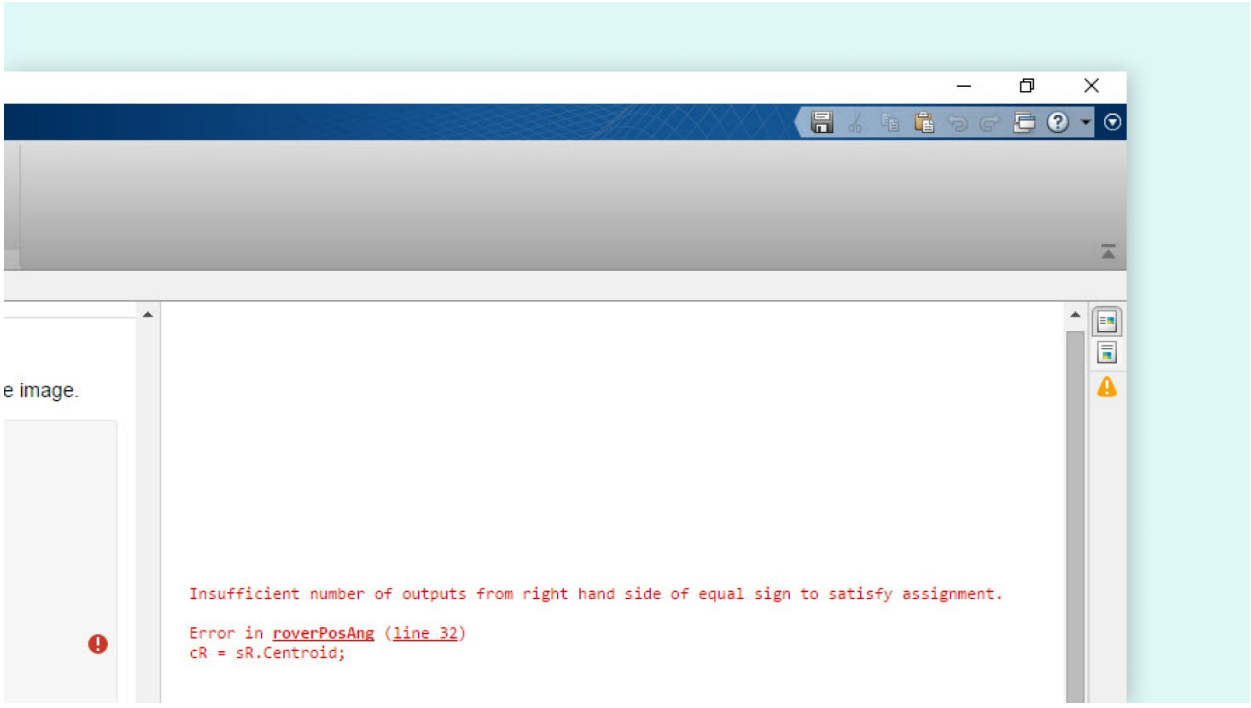
Troubleshooting Tips

Help

When calibrating the RGB values in `roverCalibration`, if you select the points in the correct order and the algorithm says that the order is incorrect, try re-running the live script in different lighting conditions. If the image in MATLAB appears to be oversaturated or extremely bright/dark, it can impact the RGB colour calibration.

Use the following documentation link to find more information about how to adjust the properties of the webcam object and improve the quality of the image: **https://www.mathworks.com/help/supportpkg/usbwebcams/ug/set-properties-for-webcam-acquisition.html**

If you receive an error message like the one below while running `roverLocalization`, the algorithm was not able to find a centroid for the Rover.



To address this, you could:

- Re-run the last section of the calibration algorithm to get new RGB threshold values. Poor lighting conditions or sudden changes in lighting can impact how the algorithm performs.
- Adjust the size of the `strel` object on line 5 of the `roverPosA` function.

Help

If you receive a similar error message from the `targetPos` function, try:

- Adjusting the constants used to threshold the image on line 18 of `targetPos`. Certain lighting conditions can cause an acceptable threshold value for black objects to be higher or lower than what is present.
- Adjusting the size of the `strel` object on line 5 of the `targetPos` function.

If the coordinates of the Rover or target are not accurate in `roverLocalization`, ensure that the `arenaHeight` and `arenaWidth` values you entered are accurate. If the dimensions do not match, you could get the correct coordinates in pixels but incorrect coordinates in centimetres.

Review

- We saw how to set up the arena and calibrate for different parameters.
- We saw how to perform localization to calculate the Rover position and orientation.

Files

- roverCalibration.mlx
- roverLocalization.mlx

Learn By Doing

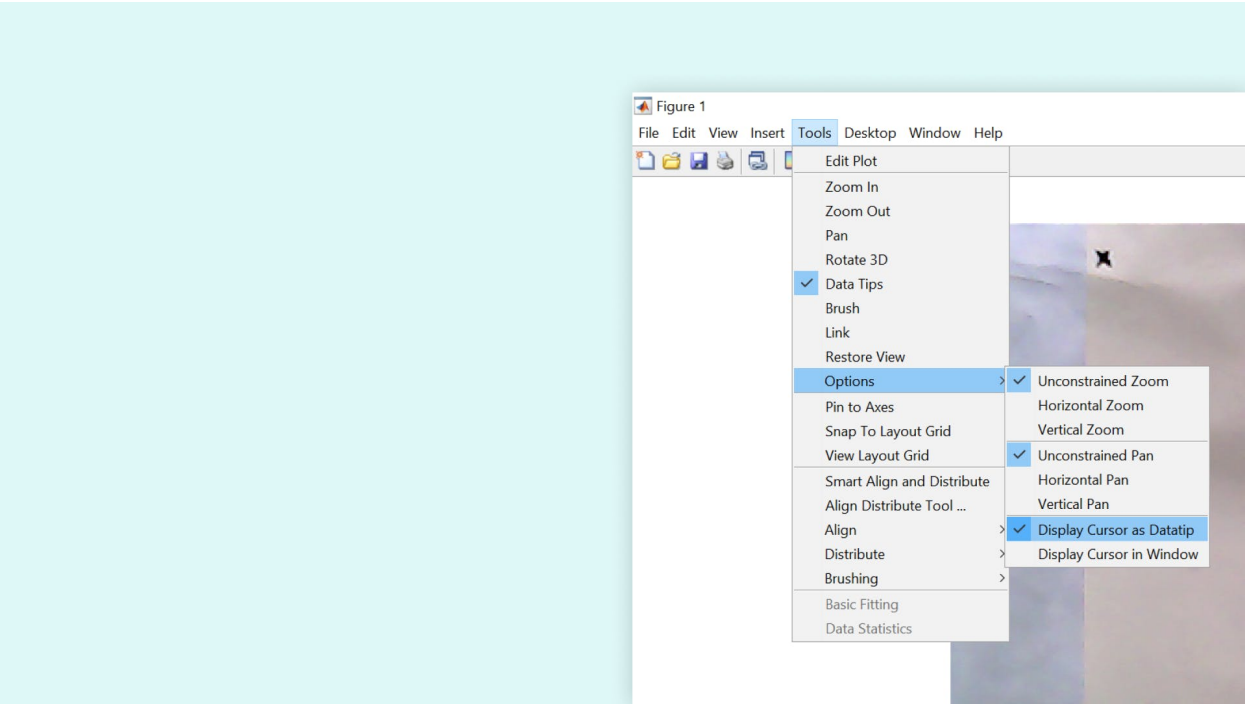
Run `roverCalibration` and observe how the location offset values affect the accuracy of the localization algorithm. Try picking points that are slightly off from the desired points.

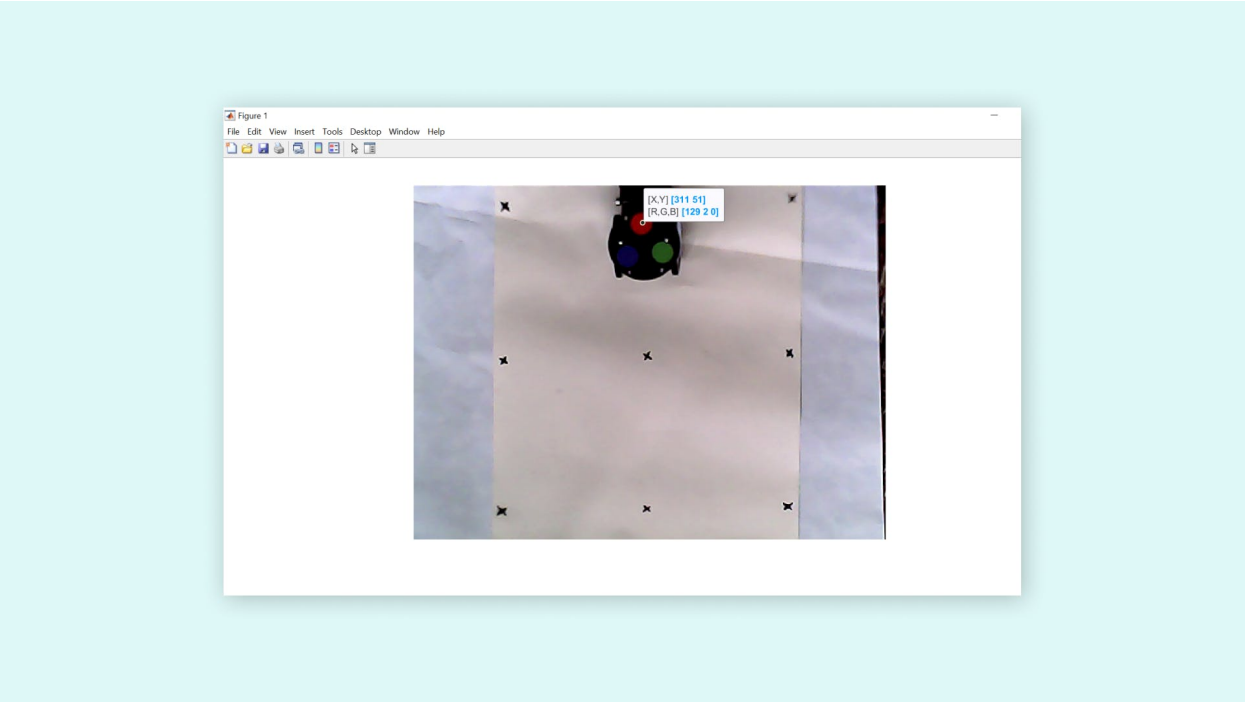
Try changing the size of the `strel` object used by the targetPos and `roverPosAng` function. To understand how the `strel` function helps in our algorithms, type the following command in MATLAB:

Help

```
>> doc strel
```

To improve accuracy, the size of this object may need to change depending on your setup. The data cursor makes measuring the pixel size of different objects in the image very easy as it provides the exact value of the pixel. You can enable the data cursor by clicking the button shown in the screenshot below.

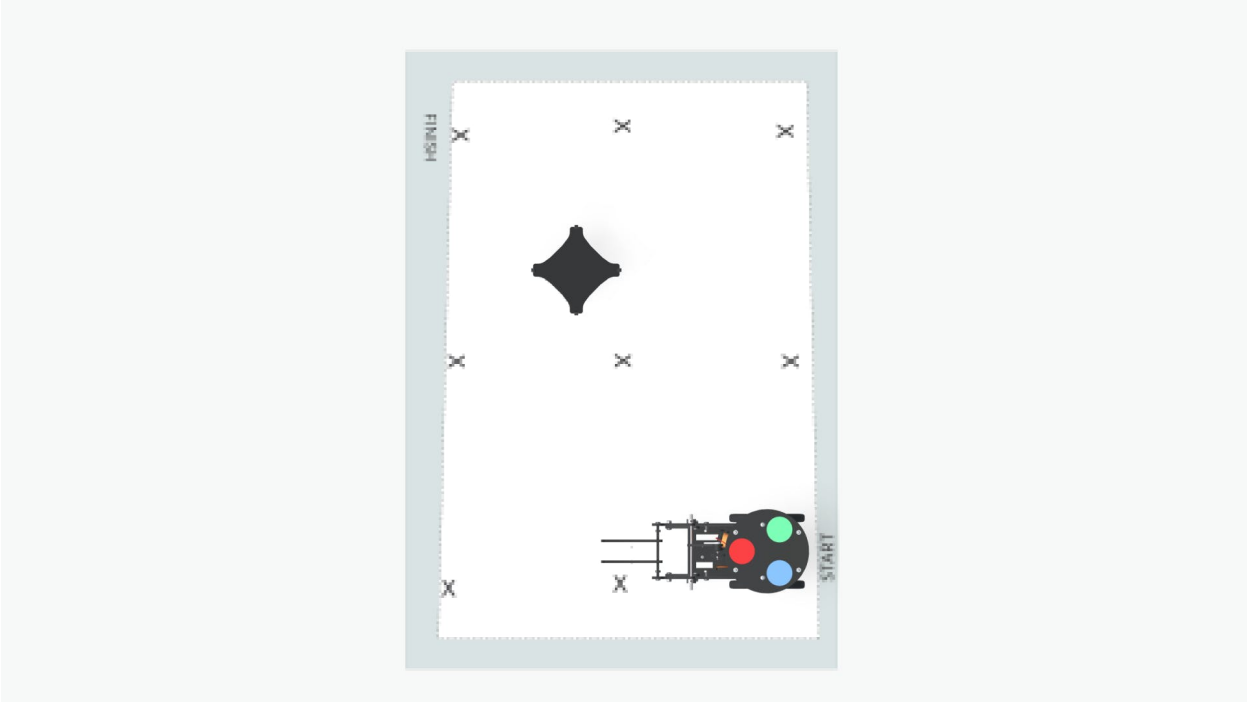




Help

5.6 Navigate the Arena and Move the Target

In the last exercise, you used a localization algorithm to calculate the position and orientation of the rover and the location of the target. You will incorporate this information in a Simulink model and program the rover to go from its starting point to the target's location and finally to the endpoint location.



You will also learn how to control the forklift from Simulink so that you can pick up the target and drop it off at the endpoint. After completing this exercise, you'll be able to program the rover to travel between any number of waypoints.

In this exercise, you will learn to:

- Use state logic to plan the path of the rover between any number of waypoints,
- Use the forklift to pick up the target and drop it off at the endpoint.

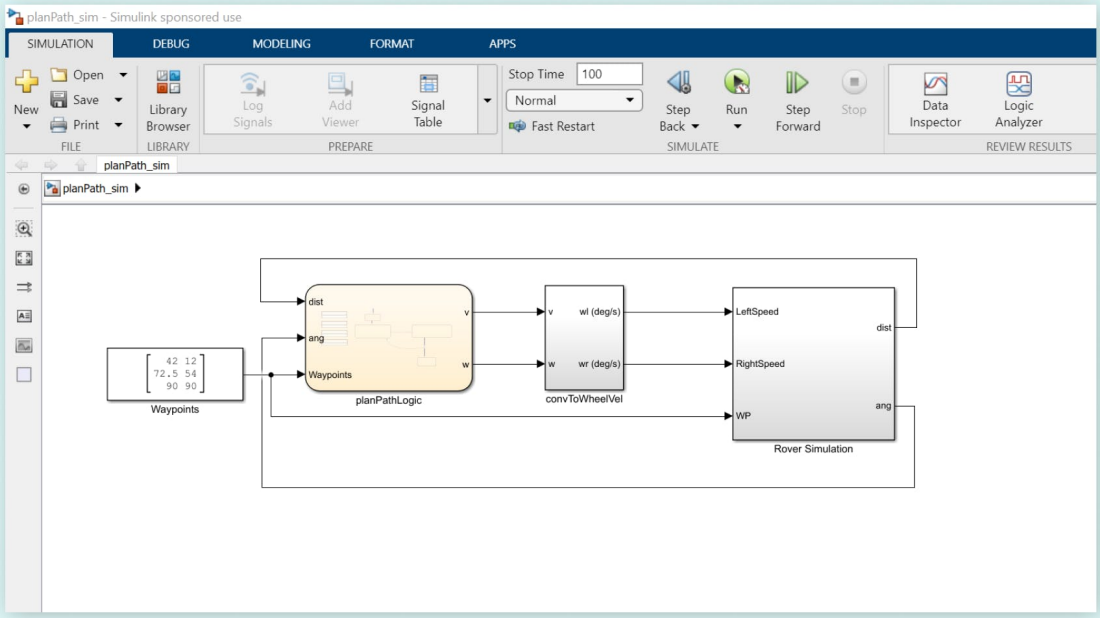
Help

In section 5 you learned how to use the webcam with a localization algorithm to get the initial locations of the rover and target in your arena. Let's see how to

incorporate this information into Simulink and program the rover to navigate between specified waypoints. Let's start by opening `planPath_sim.slx`.

```
>> planPath_sim
```

The model that simulates the movement of the rover and uses a **Constant** block containing the waypoints should now be displayed in Simulink. The block is called **Waypoints** as seen in the model; these are pairs of coordinates that will be used by the rover to calculate its path. The **Waypoints** block feeds the Stateflow chart named **planPathLogic**, in addition to the distance and angle, to describe the state machine commanding the rover. The **RoverSimulation** subsystem has also been modified to take the waypoints as input.

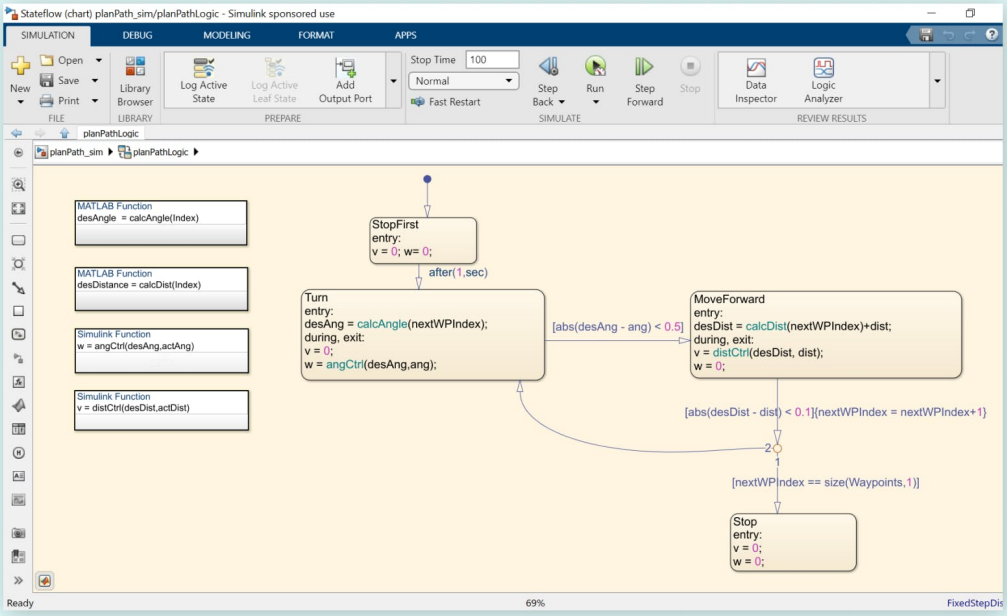


This model uses the same **convToWheelVel** subsystem from previous exercises to calculate wheel speeds based on rover velocity and rate of rotation.

The **RoverSimulation** subsystem is like previous exercises, except now it takes one additional input: a matrix providing the rover's starting location in the first row, the object's location in the second row, and the endpoint location in the third row (x, y). The subsystem uses this information to plot the waypo
animation it creates.

Help

These three locations are also used as input to the **planPathLogic** chart, which handles the path planning for the rover. Let's open **planPathLogic** to see how it works.



The chart has four states corresponding to the operating modes of the rover: **StopFirst**, **Turn**, **MoveForward**, and **Stop**.

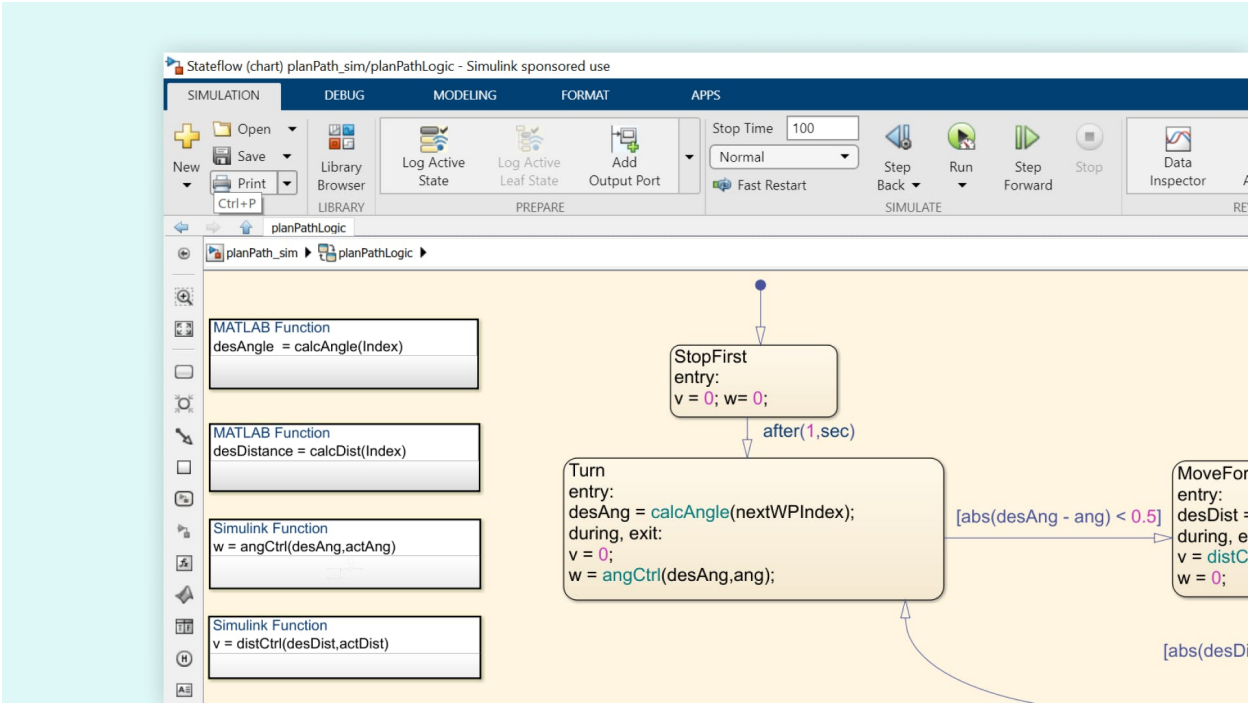
When the simulation begins, the rover is in the **StopFirst** state. The rover is in this state for **1** second and does not move during this time.

After the **StopFirst** state, the rover enters the **Turn** state. Here it rotates the appropriate amount until it's facing the next waypoint. Then, the rover enters the **MoveForward** state where it moves forward the appropriate distance to reach this waypoint.

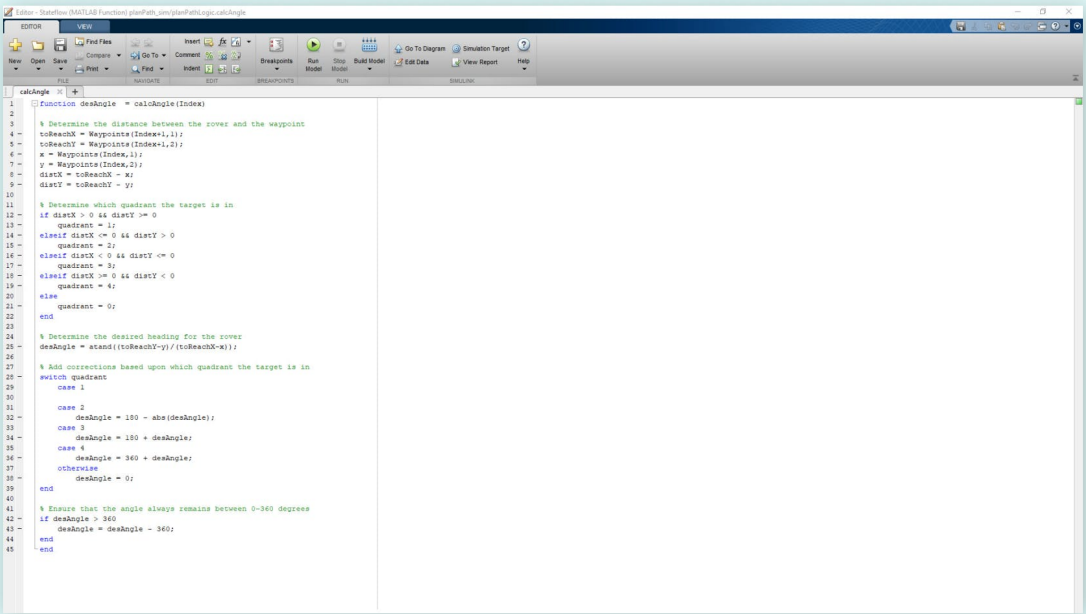
> **Note:** These states Turn and MoveForward each call a MATLAB function upon entry.

Turn calls `calcAngle` to calculate how much the rover should rotate based on the current location and the location of the next waypoint.

Help



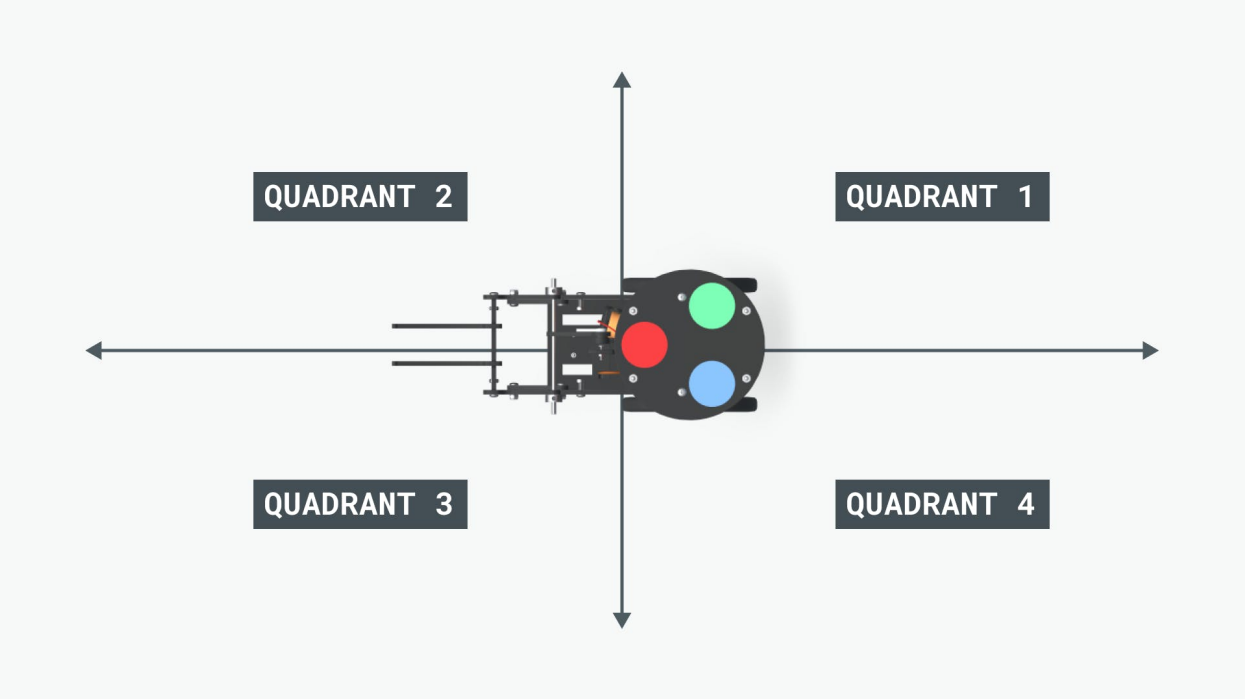
Open `calcAngle` and explore the code to understand how it works. By determining the distances between the rover and waypoints, the desired heading can be calculated using trigonometry. Note how the quadrant of the next waypoint relative to the rover's current location is identified. This is necessary when using the inverse tangent functions (`atand`).



```
function desAngle  = calcAngle(Index)

% Determine the distance between the rover and the waypoint
toReachX = Waypoints(Index+1,1);
toReachY = Waypoints(Index+1,2);
x = Waypoints(Index,1);
y = Waypoints(Index,2);
distX = toReachX - x;
distY = toReachY - y;

% Determine which quadrant the target is in
if distX > 0 && distY >= 0
    quadrant = 1;
elseif distX <= 0 && distY > 0
    quadrant = 2;
elseif distX < 0 && distY <= 0
    quadrant = 3;
elseif distX >= 0 && distY < 0
    quadrant = 4;
else
    quadrant = 0;
end

% Determine the desired heading for the rover
desAngle = atand((toReachY-y)/(toReachX-x));

% Add corrections based upon which quadrant the target is in
switch quadrant
    case 1

    case 2
        desAngle = 180 - abs(desAngle);
    case 3
        desAngle = 180 + desAngle;
    case 4
        desAngle = 360 + desAngle;
    otherwise
        desAngle = 0;
end

% Ensure that the angle always remains between 0-360 degrees
if desAngle > 360
    desAngle = desAngle - 360;
end
end
```

Help

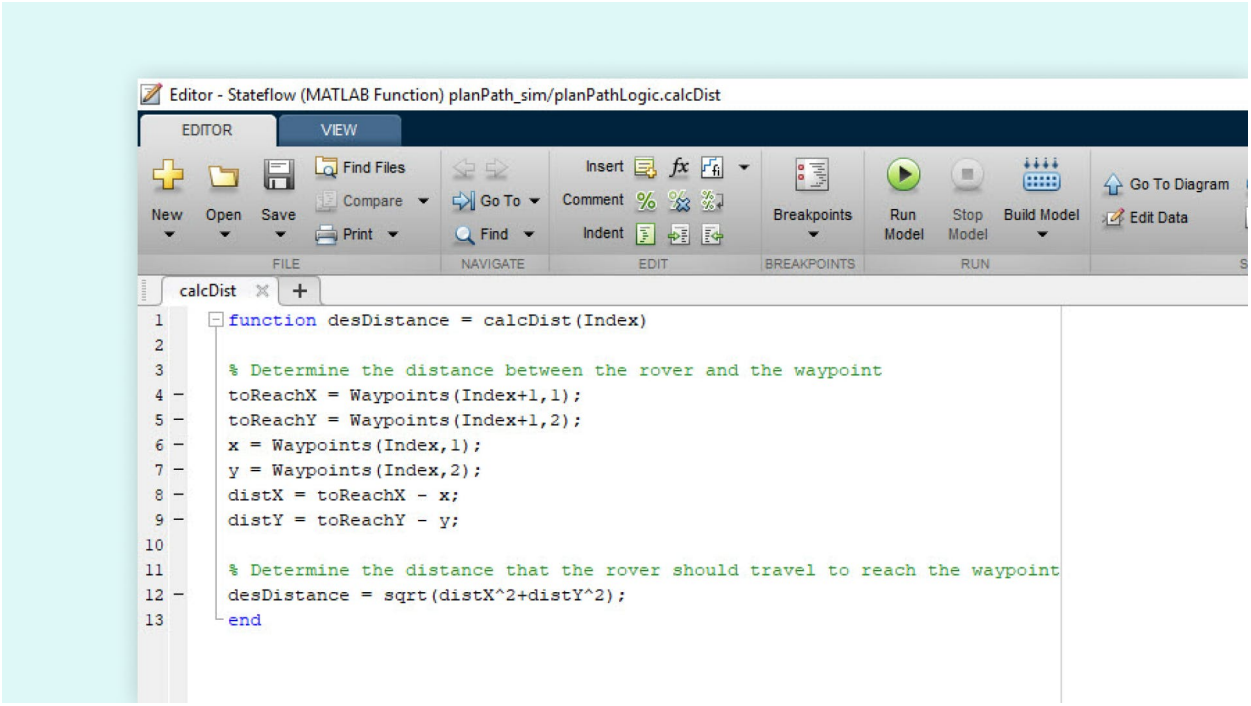
To better understand quadrants in the context of the rover, please refer to the image below.



After calculating how much to rotate, the `angCtrl` Simulink function is called to rotate the rover, exactly like section 4. This uses a PI controller to rotate the rover until the transition condition is met (i.e., the rover is within 0.5 degrees of the specified angle).

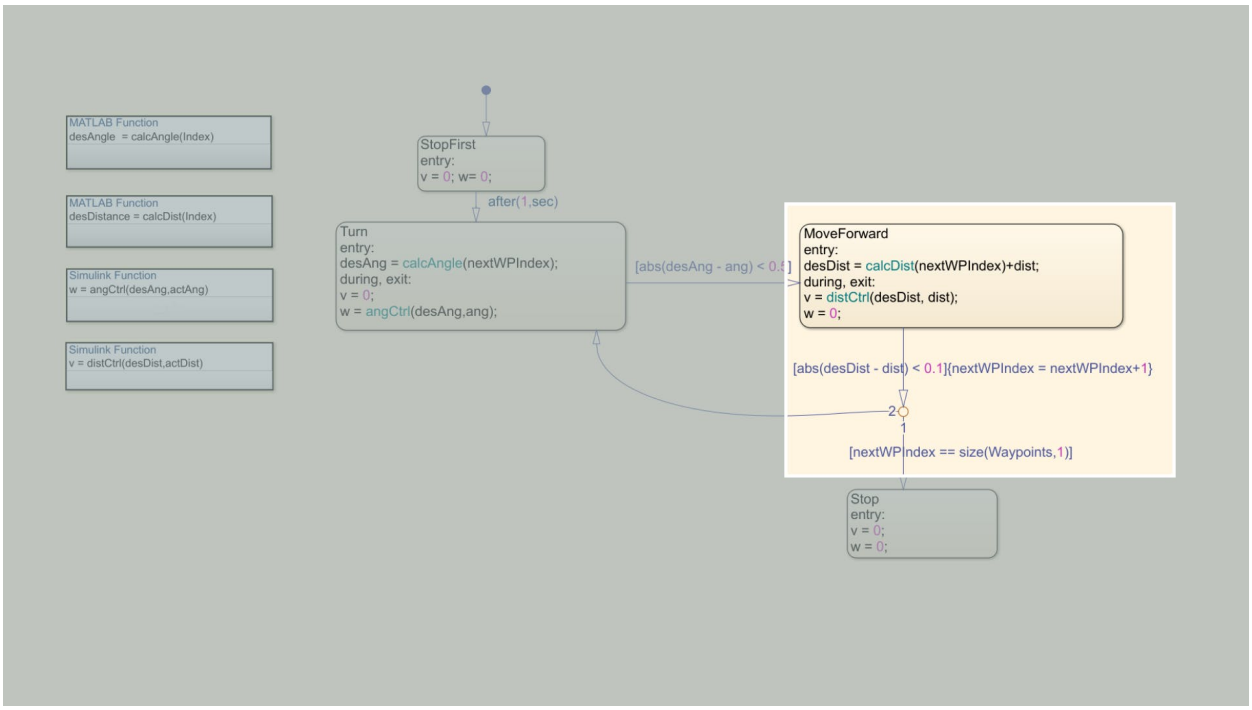
The rover should now be facing toward the next waypoint and will enter the **MoveForward** state. In this state, the rover moves forward the appropriate distance to reach the waypoint. Open the MATLAB function `calcDistance` where this distance is calculated using the Pythagorean Theorem.

Help



After calculating how far the rover should move, the `distCtrl` Simulink function is called to move the rover to within 0.1 cm of the specified distance.

Once this transition condition is met, the rover exits the **MoveForward** state and then it reaches a **junction** in Stateflow.



Help

Junctions are used when you need to decide between two different states, in this case **Stop** or **Turn**. The `nextWPIndex` variable is incremented each time

the rover exits the **MoveForward** state. Once the index value equals the number of waypoints, the sequence is complete, and the rover enters the **Stop** state. Until then, the rover reverts to the **Turn** state and the process starts again for the next waypoint.

For this example, consider the simple sequence where the rover moves from its initial location to the target and then from the target to the final endpoint. However, the StateLogic chart is designed to be generic so that it can plan the rover's path for any arbitrary number of waypoints.

Check the **Workspace** and you will notice that `L`, `r`, `TS`, `StartX`, `StartY` and `StartTheta` are created.

However, other variables are needed to run different simulations, like the origin coordinates and initial orientation for the rover.

Define the rest of the variables that are needed, namely the rover coordinates `StartX` and `StartY` and the heading by typing the following in the Command Window.

```
>> StartX = 42;
>> StartY = 12;
>> StartTheta = 90;
```

Recollect, the **RoverSimulation** subsystem uses these variables to plot the rover location in the animation.

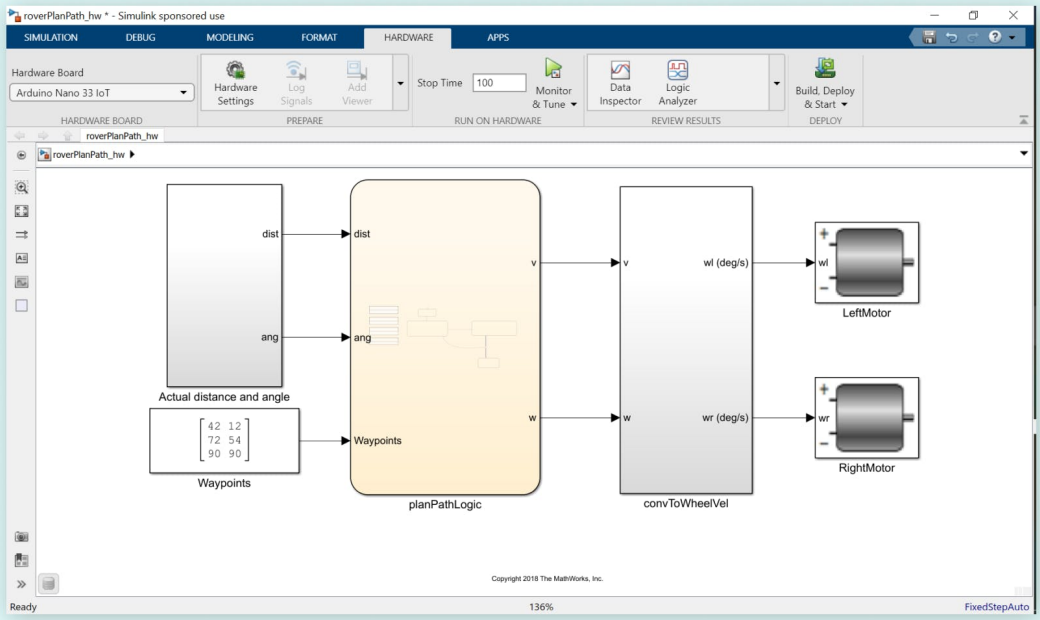
Click **Run** to see how the rover behaves in simulations. You should be able to see both the animation of the rover and the Stateflow chart at the same time.

Deploy the Path-Following Algorithm to the Rover

Now that your path-following algorithm works in simulation, let's deploy the application on the actual rover. Start by opening the model `roverPlanPath_hw.slx`.

Help

```
>> roverPlanPath_hw
```



This model uses the same path-following algorithm as the simulation model but the inputs to the model are now the rover's actual distance and angle as measured by the encoders. Also, the rotational speeds (`wl` , `wr`) calculated in the **convToWheelVel** subsystem are input to the actual motors.

Update the **Waypoints** block so that the first row represents the initial rover position, the second row is the target location, and the third row is the desired endpoint location. Use the real-world results that you got in section 5. If you have changed the location of the rover and/or target, rerun `roverLocalization.mlx` .

For example - if you got the `cmPlate` values to be `[10 10]` and `cmTarget` values to be `[30 40]` and if you want the rover to end at `[50 50]` , then you will enter `[10 10; 30 40; 50 50]` as your **Waypoints** matrix.

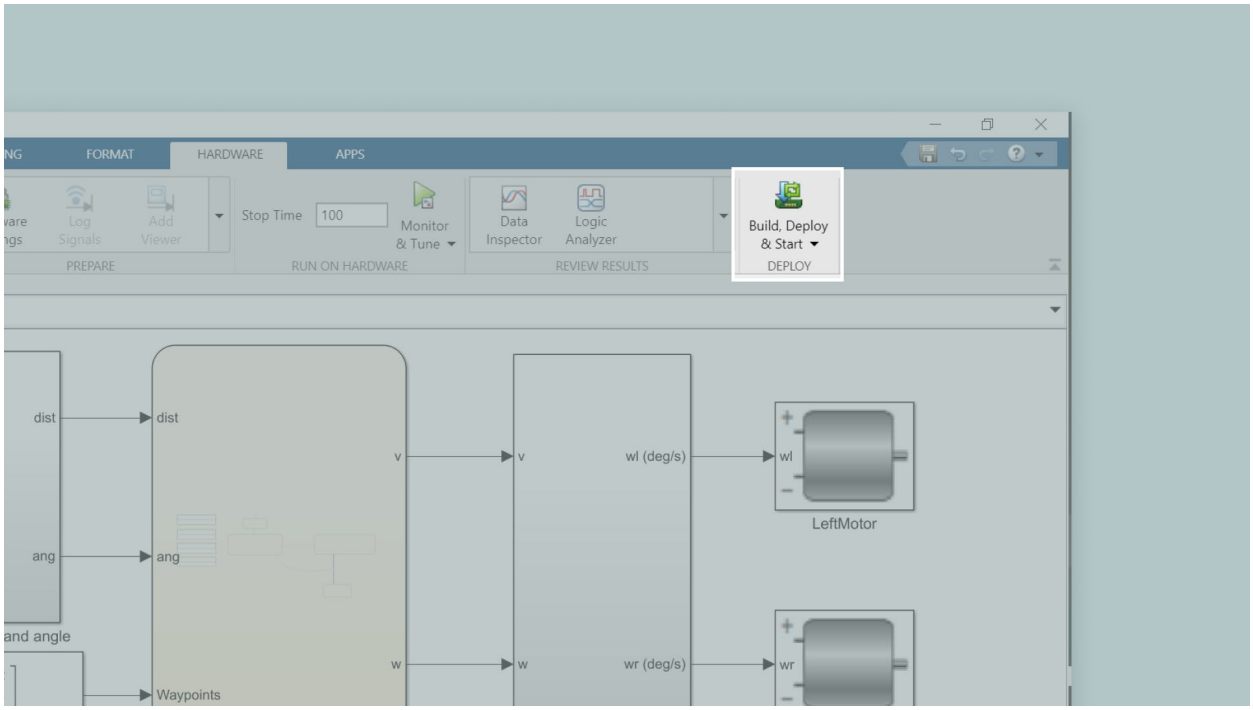
Define the rover location variables to match the heading and x, y location of your rover from the localization results by typing the following:

```
>> StartTheta = heading;
>> StartX = cmPlate(1);
>> StartY = cmPlate(2);
```

Help

Ensure that the battery is powered **OFF** and connect the rover to the computer using the USB cable and click the **Build, Deploy & Start** button to download the

code to the Arduino.



Once the build process completes, remove the USB cable. Before powering your rover **ON**, remember to remove the target and mark that spot on your arena, as your first experiment will be to simply get the rover to move from its original location to the location of the target.

Place the rover back on the arena and power it ON. Observe how closely your rover navigates to the target and the final endpoint location. Troubleshoot your experiment until the results are satisfactory. Remember if the rover does not follow the path accurately, you can enable both controllers for the rover motion as you did with the `roverClosedBoth_states_hw` model in section 4.

Use the Rover's Forklift to Move the Object

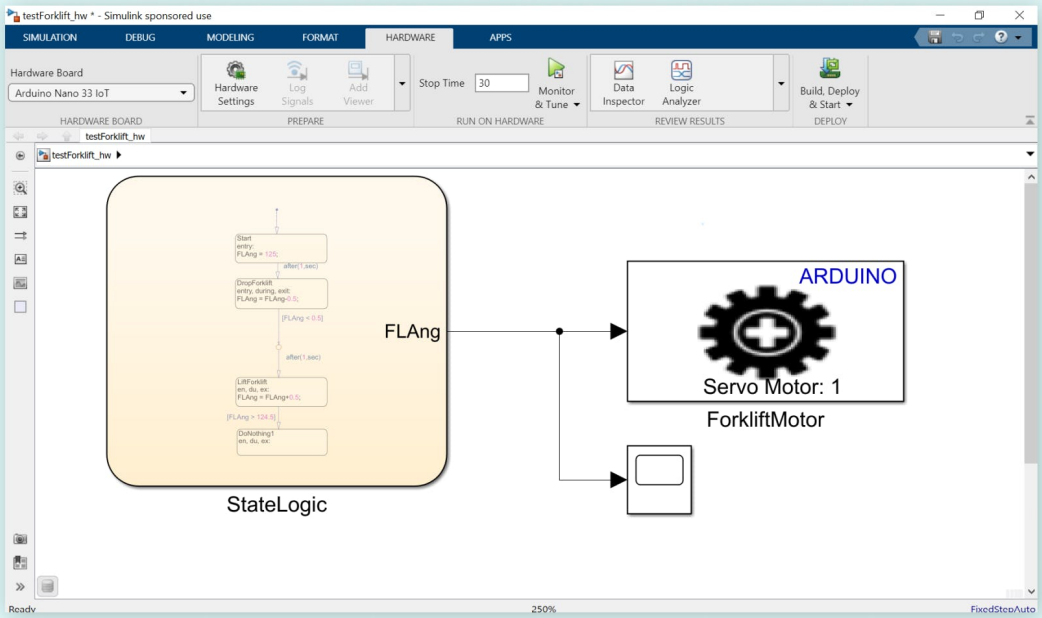
Now that you've learned how to navigate the rover between waypoints, let's see how to operate the forklift so that it can eventually be used to pick up and move the target.

The first step is to learn how to control the forklift from Simulink. For this, we have prepared a model called **testForklift_hw**. Open this model by ty[illegible] following:

Help

```
>> testForklift_hw
```

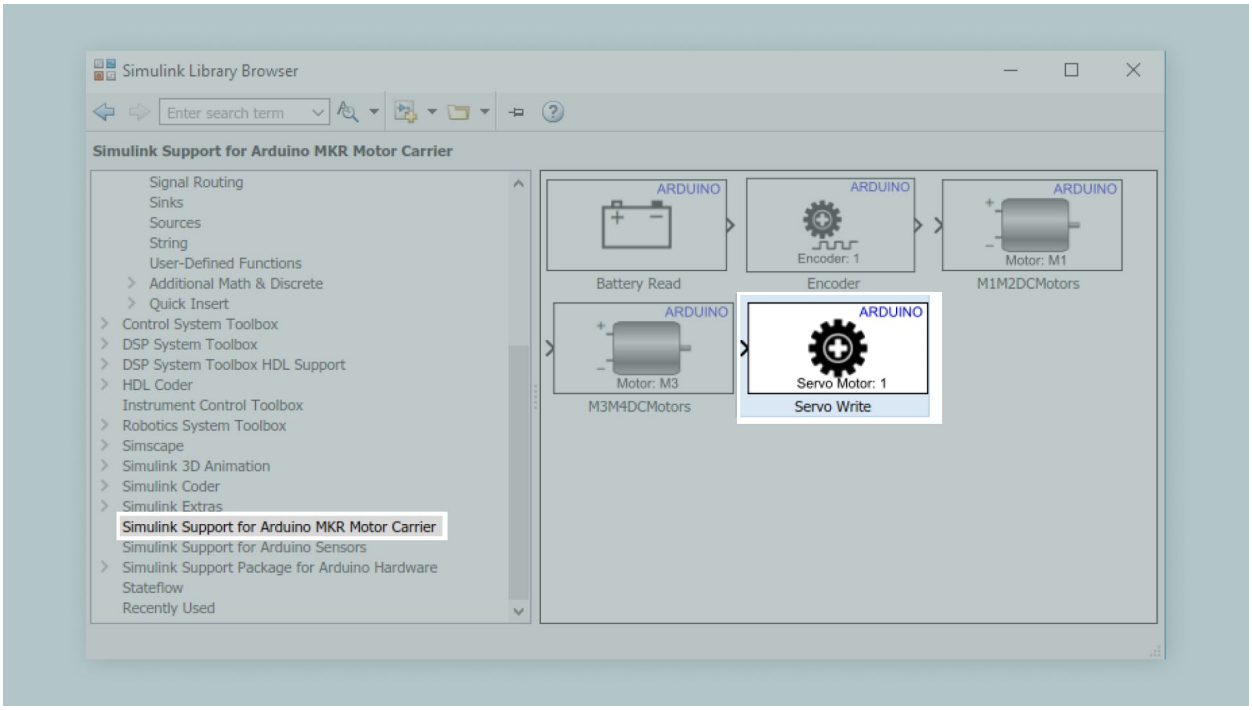
In the model, you will see a Stateflow chart and a **ServoMotor** block. The idea is that the forklift will be raised and lowered by controlling the servo motor via a logic diagram and then similar techniques will be incorporated into the model to command the movement of the rover.



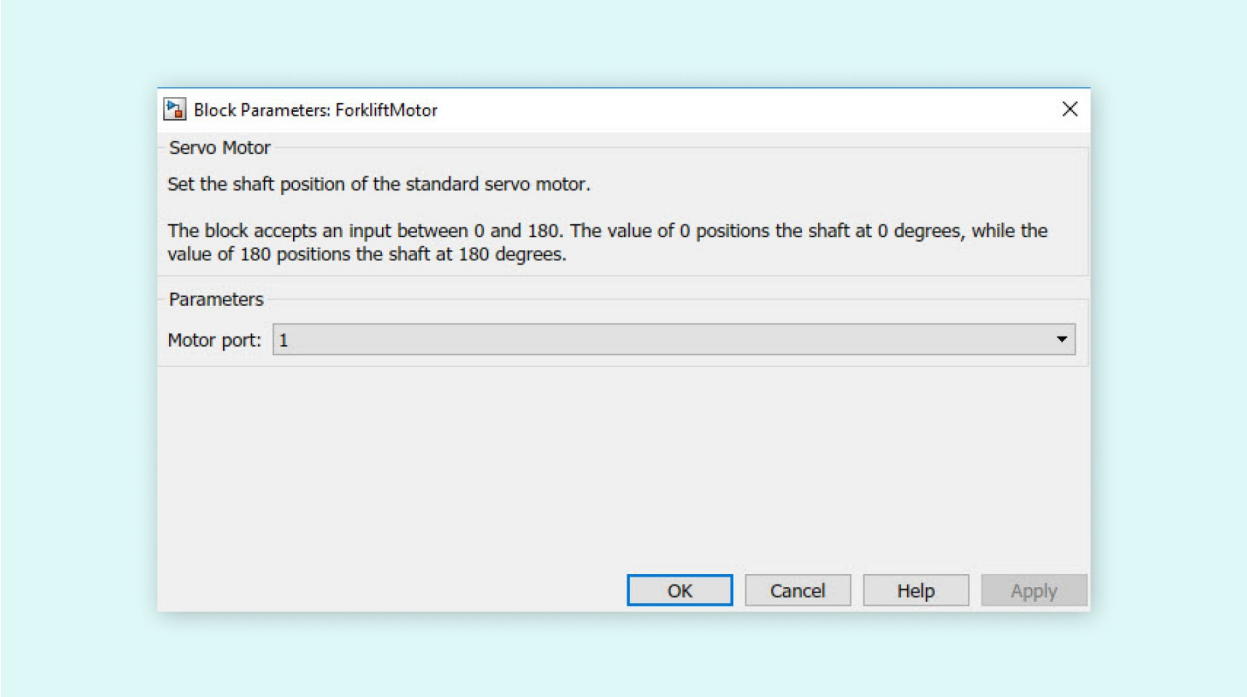
This model uses a Stateflow chart to define the logic for raising and lowering the forklift. The forklift is controlled by a standard servo motor whose angle ranges from 0 to 125 degrees. The chart outputs the angle of the servo, where 0 degrees is the **DOWN** position and 125 degrees is the **UP** position.

The Servo Motor block is from the library **"Simulink Support for Arduino Nano Motor Carrier"**

Help

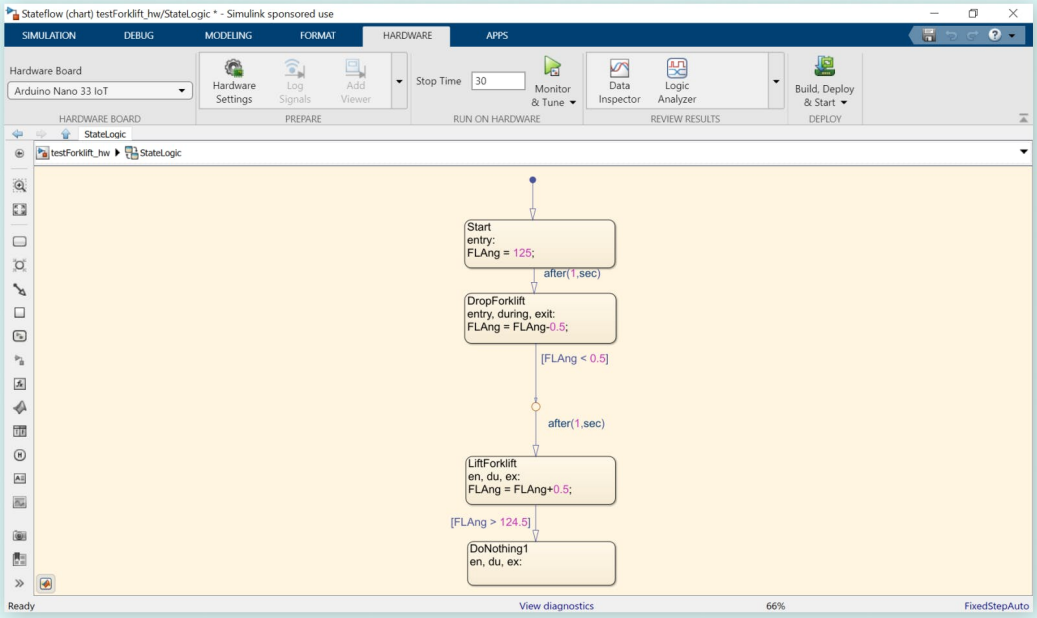


Select the port number based on where the servo motor is connected to the Arduino Nano Motor Carrier. Given the suggested assembly instructions, this should be port number 1.



Let's take a closer look at the StateLogic chart that moves the forklift from **UP** to **DOWN** and back to **UP**.

Help



This chart includes two main states, **DropForkLift** and **LiftForkLift** , and a few intermediate states where no action is taken.

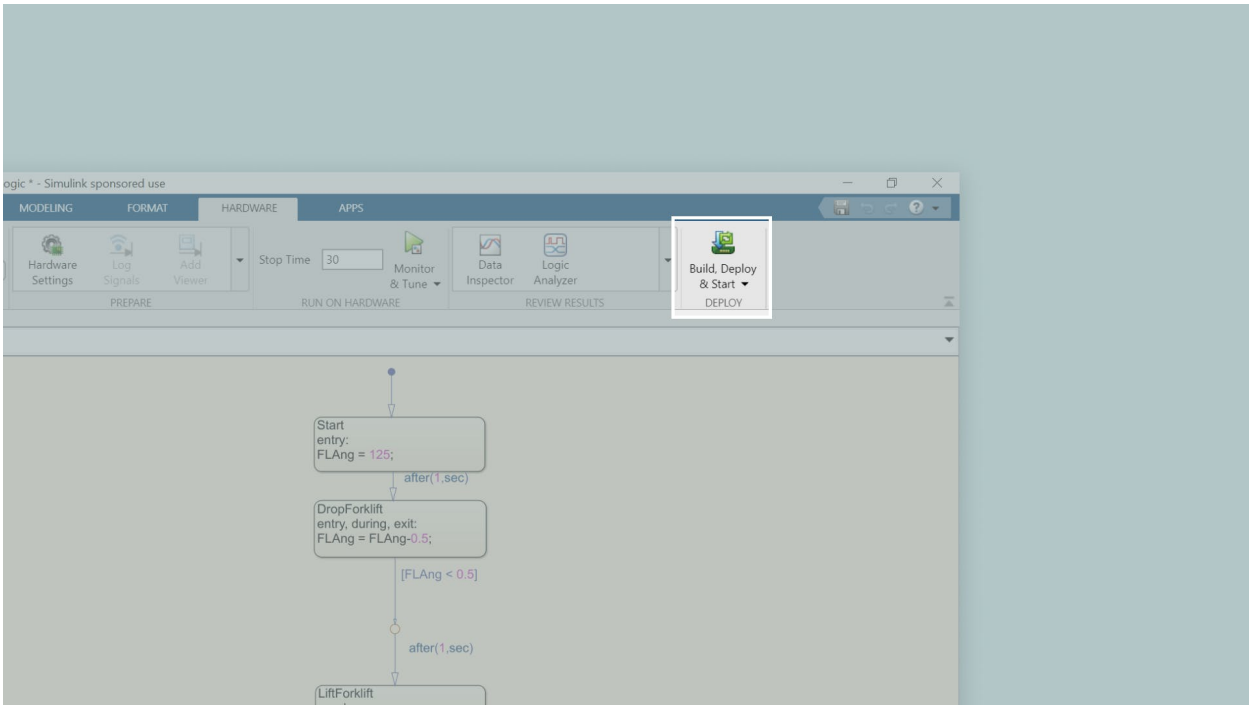
In the **DropForkLift** state, the servo angle is reduced by 0.5 degrees at every timestep until it reaches the **DOWN** position, which stands for 0 degrees.

In the **LiftForklift** state, the servo angle is increased by 0.5 at every timestep until it reaches the **UP** position or 125 degrees.

Before deploying this application to the rover, ensure that the forklift is in the **UP** position as shown in the image below. This is the recommended position for general situations where the rover is navigating around the arena.

Help

Click the **Build, Deploy & Start** button to download the code to the rover.



Power the rover on to see how the forklift performs. The rover is not going to move since the model is only controlling the servo motor and not affecting the DC motors that steer the wheels.

Help

The next step is to incorporate the forklift action into our earlier task of navigating between waypoints. In other words, you need to merge the forklift

chart with the path planning chart.

To simplify this task, a model is provided that can do this operation for you. Open the model `roverPlanPath_Forklift_hw.slx` and explore the updated Stateflow logic. **Build, Deploy & Start** the updated model to your rover and see how it performs on the actual arena. Remember to place the rover and the target back at the same initial starting location. If not, remember to rerun the `roverLocalization` live script and update the **Waypoints** block in this model.

Review

- We saw how the rover can plan its own path from any starting location all the way to an endpoint.
- We saw how to lift the target using the forklift and include this with the path planning algorithm.

Files

- planPath_sim.slx
- roverPlanPath_hw.slx
- testForklift_hw.slx
- roverPlanPath*Forklift*hw.slx

Learn By Doing

Try out different combinations of rover starting positions and object locations and see how accurately your rover navigates to the requested waypoints. If the accuracy is not satisfactory, try adjusting your PI controller gains.

Try adding additional waypoints and see how accurately your rover navigates between them.

Help

5.7 Control the Rover over Wi-Fi

In the previous exercise, you saw how to apply a localization algorithm to calculate the initial location and orientation of the rover, and how to incorporate this information into the path-planning algorithm. In general, it's also desirable to perform localization at regular intervals as the rover moves around, to supplement the location information calculated from encoder data.

This exercise shows how to configure your Simulink model to receive data from MATLAB over Wi-Fi. It also shows how to send data from MATLAB to your rover based on the rover's current position calculated via localization. This would potentially allow you to have a rover for which waypoints can change over time given the localization information obtained from the camera. In other words, you would not have to reprogram the rover every time the initial conditions change, as it will use the camera to feed the algorithm in real-time.

In this exercise, you will learn to:

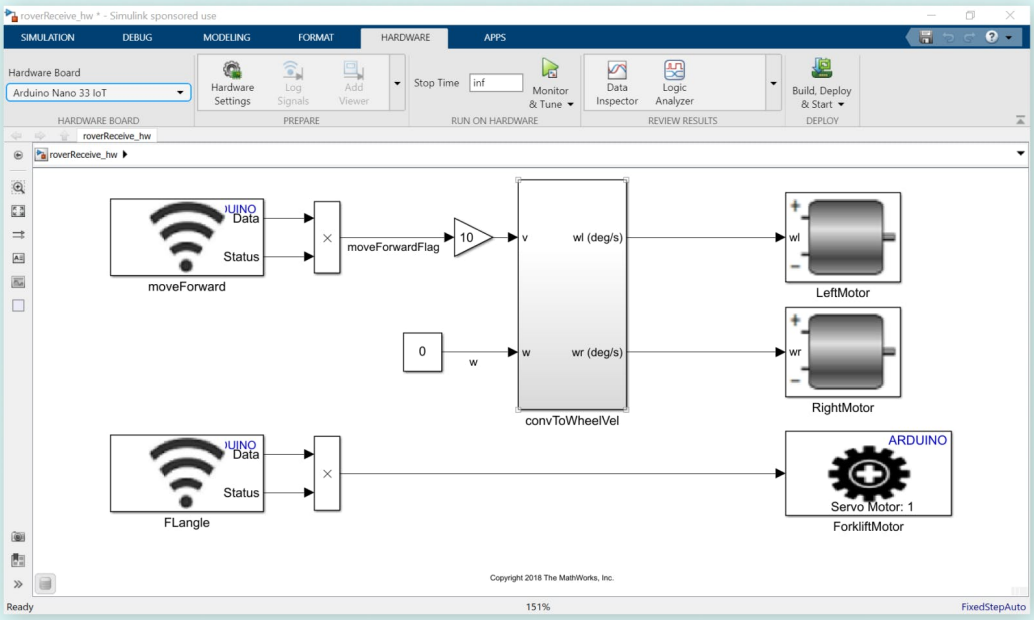
- Send instructions from MATLAB to the rover via Wi-Fi,
- Establish Wi-Fi communication between rover and MATLAB,
- Use localization to calculate rover position as it moves around.

Recieve Wi-Fi Data in Your Simulink Model

Let's begin by opening `roverReceive_hw.slx` :

```
>> roverReceive_hw
```

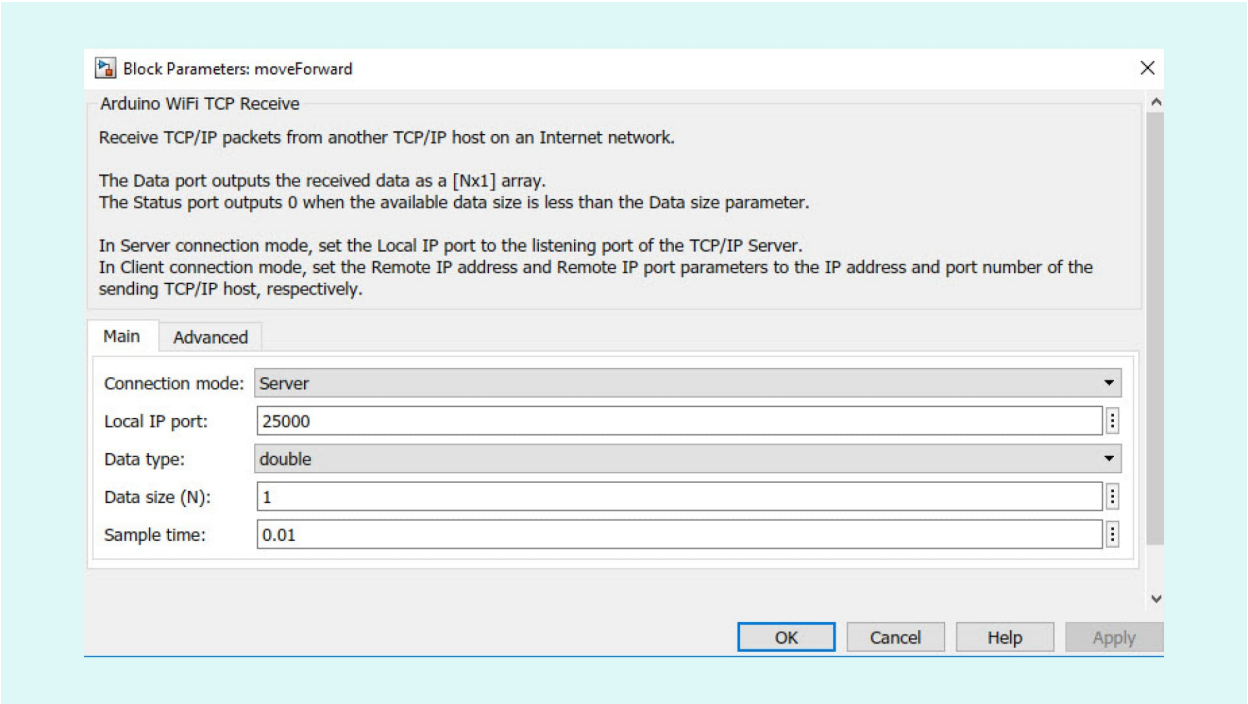
Help



This is a simple model that demonstrates how you can send instructions to the rover over Wi-Fi. It uses subsystems and blocks you're familiar with for calculating wheel velocities and actuating motors.

However, what's new is, this model uses data coming in through Wi-Fi to instruct the rover on whether to move forward or stop and whether to move the forklift or not. The blocks **moveForward** and **FLangle** control the wireless communication to and from the rover. Open the block labelled **moveForward** to see how this is configured.

Help



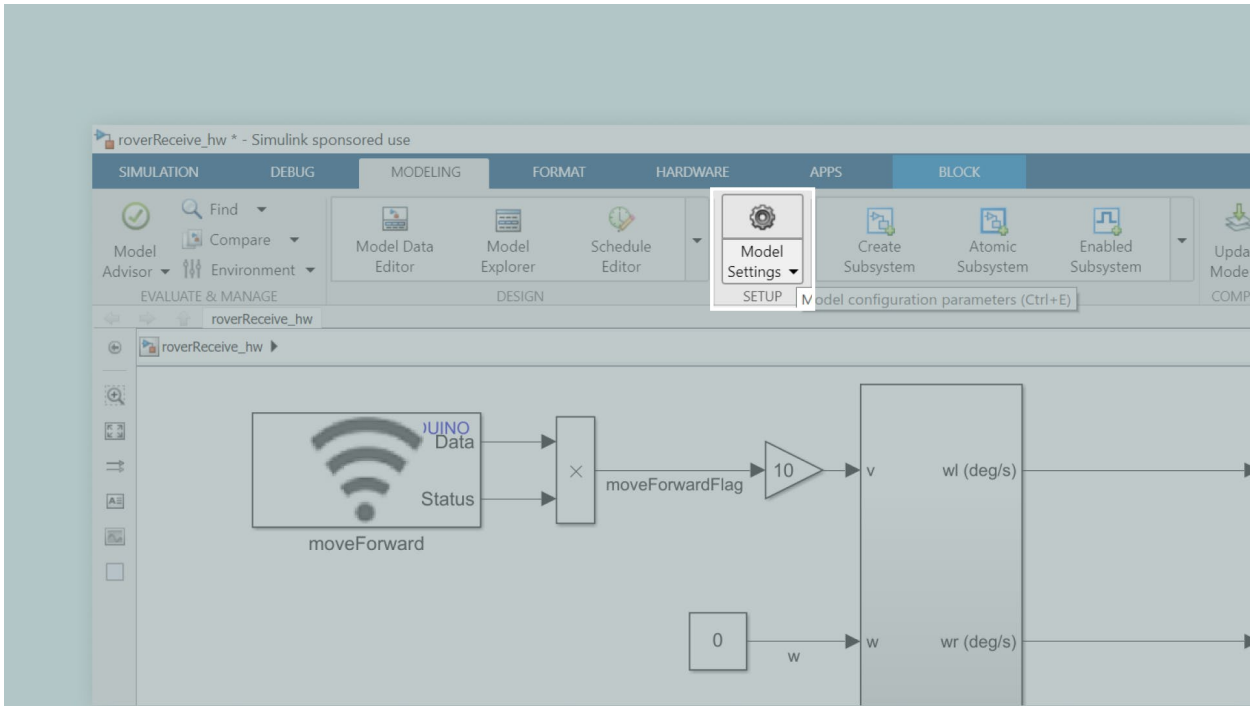
This block allows you to input data of type specified by **Data type** (in this case double) and size specified by **Data size (N)** at the **Local IP port**. Data is read at 0.01 second intervals as specified by the **Sample time**. The **Local IP port** value used here is the default. Each block is set at a different **Local IP Port** to differentiate the data streams from each other.

Data sent to this block (**moveForward**) will be either 0 or 1. When the input is 1 the rover moves forward at 10 cm/s, and it does not move when the input is 0.

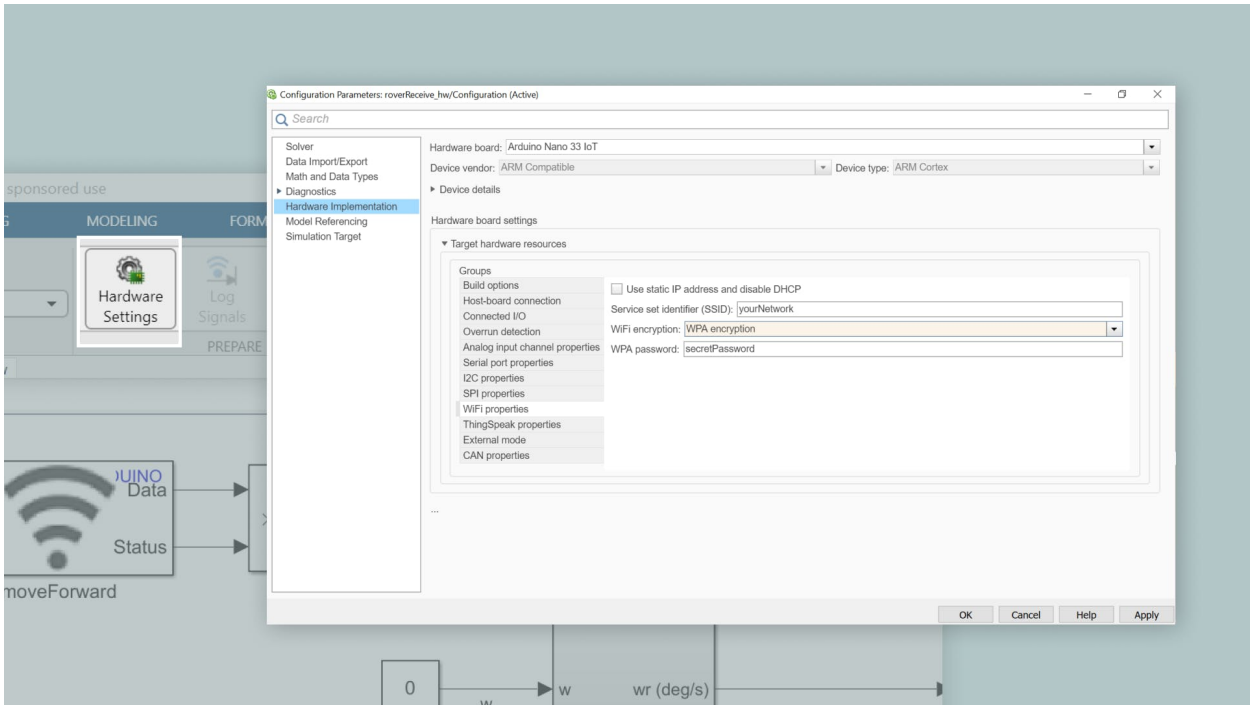
The other Wi-Fi receive block labelled **FLangle** controls the forklift to move from the **UP** position to **DOWN**.

There's one final Wi-Fi configuration step before you deploy this model to the rover. You need to configure the Wi-Fi properties of this model so that the rover gets an IP address that can be used to communicate. To do so, Open **Modelling** and click the **gear** icon.

Help



This opens the **Configuration Parameters** dialog. To configure your model, Open **Hardware Implementation > Hardware board settings > Target hardware resources > WiFi properties**.



Enter the SSID and WPA password of the wireless network. These are the same as those you would use to connect your laptop or smartphone to the network. This implies that, for this to work, you need to have a functioning Wi-Fi network operating in the space where you are experimenting. At this point, it is

Help

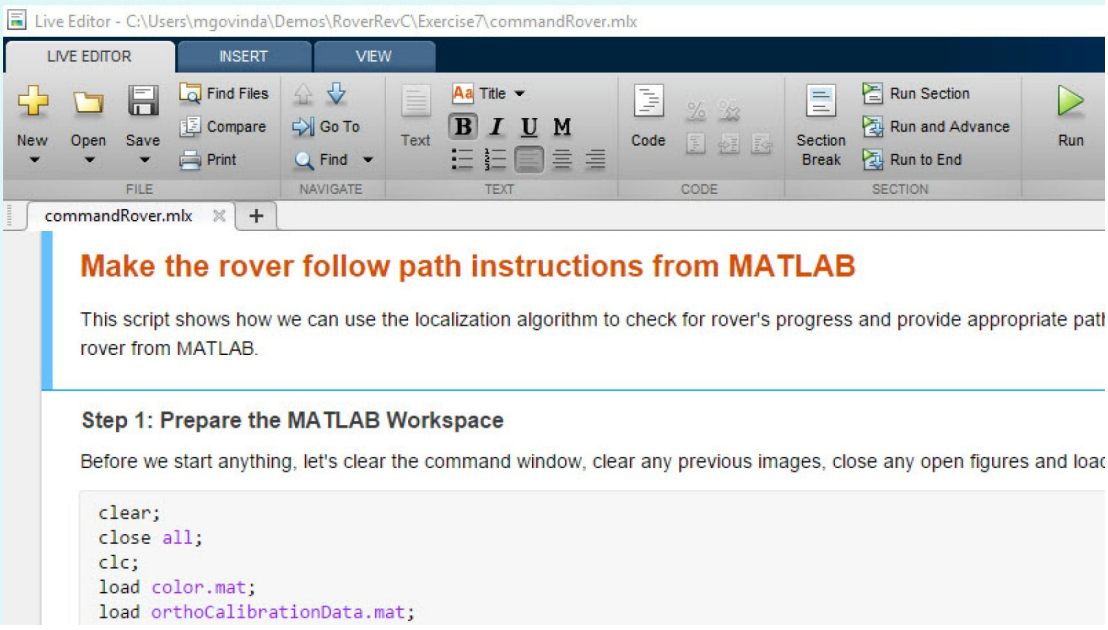
not possible to use the access point functionality of the Arduino Nano 33 IoT in your kit to create a dedicated network.

The Simulink model is now set up so that the rover can receive commands over Wi-Fi from MATLAB. Later on we will upload this code to the Rover and obtain the IP address. First, let's transition to MATLAB to see how to send commands over Wi-Fi.

Send Commands from MATLAB to the Rover over Wi-Fi

Start by opening `commandRover.mlx` :

```
>> edit commandRover
```



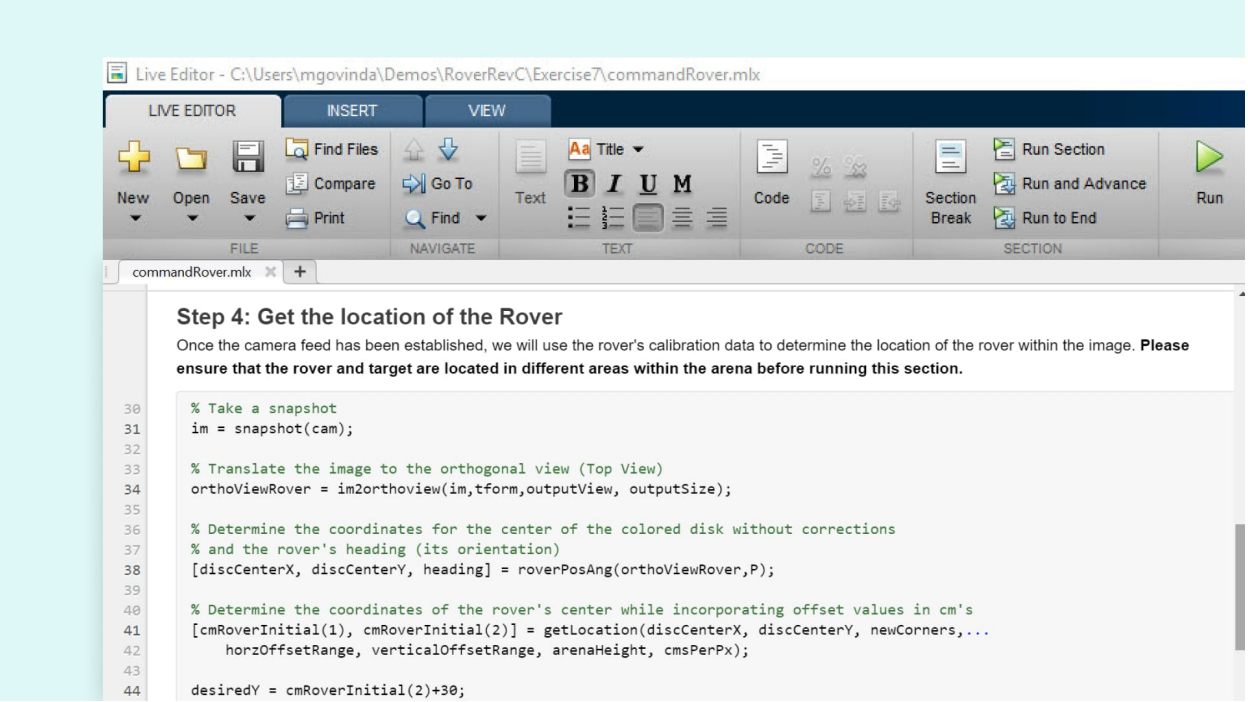
Let's review the code to see how it works. Several of the code sections should only be run after deploying the Simulink application to the rover, for now, let's simply understand what the code is doing and then it can be run later in the exercise.

Help

The first three steps perform the necessary setup for the localization algorithm, including loading calibration data, entering arena information, and initializing

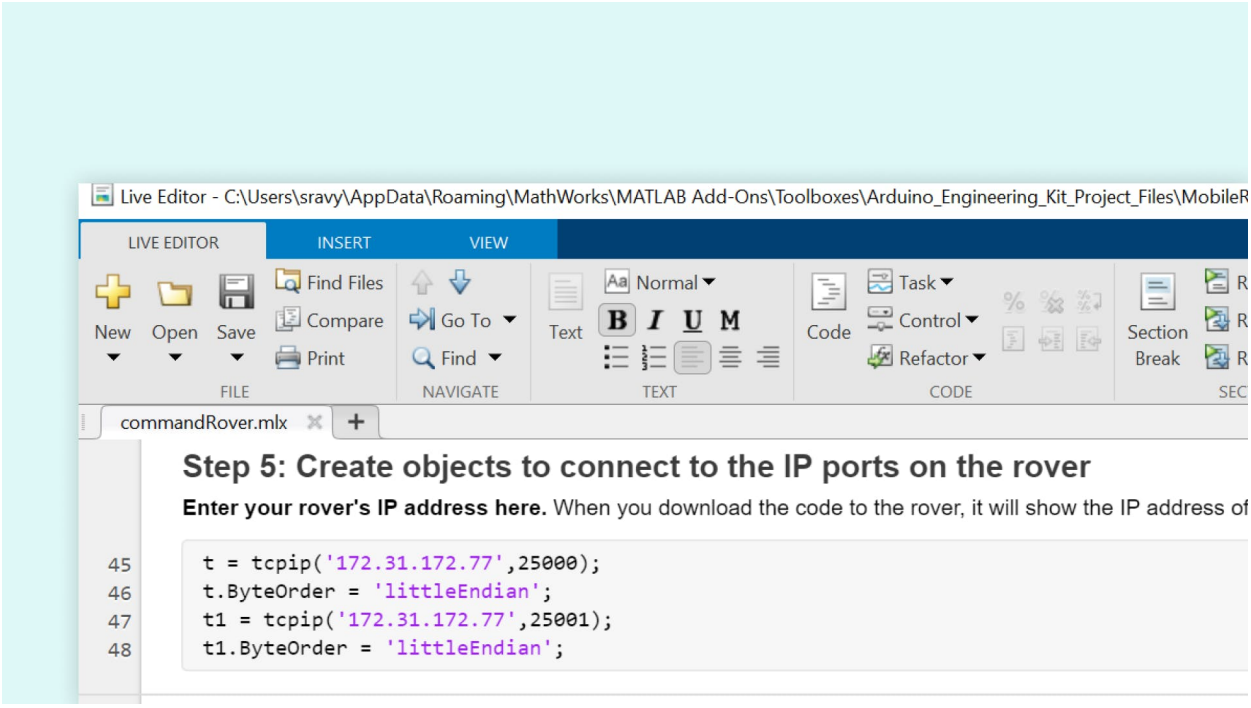
the webcam. You can refer to the previous exercises if you need to get more information about how they work, as we explained them in depth.

In "Step 4: Get the location of the Rover," the localization algorithm is called to calculate the location and heading of the rover and store the results in `cmRoverInitial`. The desired y-location of the rover is specified to be the initial location plus 30 cm.



Now that you've established the initial rover position and desired final position, you can set up the Wi-Fi communication to the rover. Begin by using the `tcpip` function to create objects that represent the connection between MATLAB and the IP ports corresponding to the **moveForward** and **FLangle** blocks in `roverReceive_hw.slx`.

Help



Note: Once these objects are created, you can send data to the specified IP ports using `fwrite`.

The section labelled "Step 6: Move forward and drop the forklift" is where the data is being sent from MATLAB to the deployed Simulink application over Wi-Fi. This step starts by defining a variable `moveFlag` that is used as input to the moveForward block. When `moveFlag` equals 1, the rover moves forward in the y-direction at 10 cm/s, and when it equals 0 the rover doesn't move.

`moveFlag` is initially set to 1 and conditional logic is used to instruct the rover to continue moving forward until the rover's y-location (as calculated by the localization algorithm, which runs once per second) is within 1 cm of the desired y-location.

Once the criterion is met, two things happen:

- `moveFlag` is set to 0 so that the rover stops moving forward.
- A `for` loop is used to move the forklift from the **UP** position (125 degrees) to the **DOWN** position (0 degrees).

Help

```
if abs(desiredY - cmRoverCurrent(2)) < 1
    % Code to drop the forklift
    fopen(t1);
    for angle = 180:-1:0
        fwrite(t1,angle,'double');
    end
    fclose(t1);
    moveFlag = 0;
else
    % Do nothing
end
```

In summary, the MATLAB code uses Wi-Fi communication to instruct the rover to move forward to the desired y-location and then drop the forklift.

Controlling the Rover from MATLAB over Wi-Fi

Now that you've learned about the Simulink models and MATLAB code, let's start controlling the rover from MATLAB.

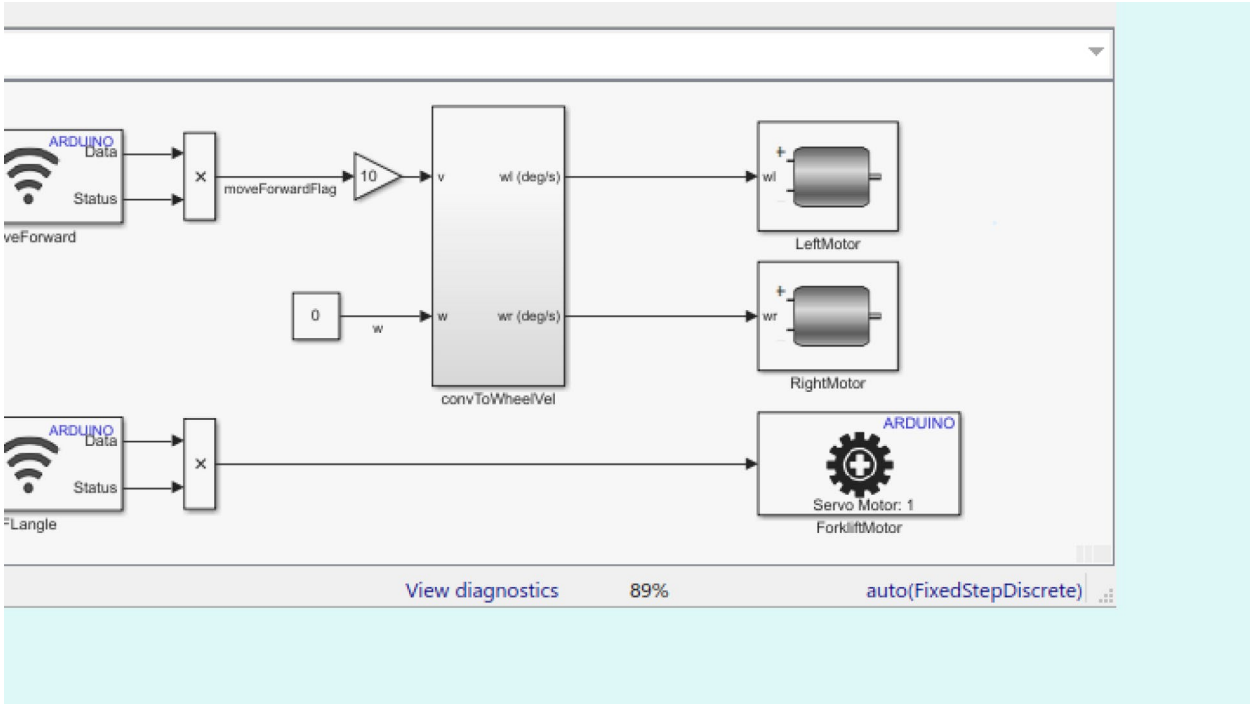
Start by deciding on a place to put the rover on the arena. This must be a location where it's free to move forward at least 30 cm in the y-direction.

> **Note:** As usual, if any aspects of your setup have changed (e.g., webcam or arena location, lighting), you need to recalibrate.

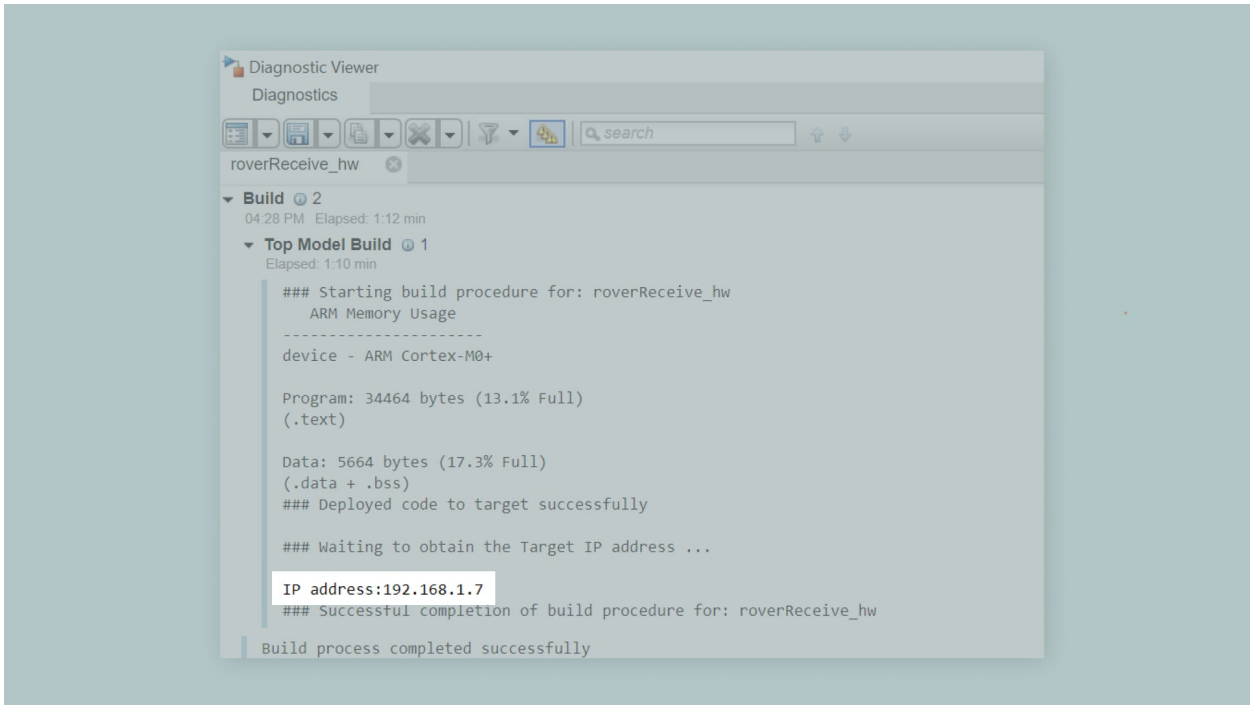
If you haven't already, deploy `roverReceive_hw.slx` to the rover. Make sure that the Rover is connected via USB. Click on **Build, Deploy & Start** to upload the Simulink code to the Rover. You can now disconnect the USB cable.

When the code deployment is complete, click View Diagnostics to get the rover's IP address.

Help



In the **Diagnostics Viewer** that opens, note the IP address of the rover.

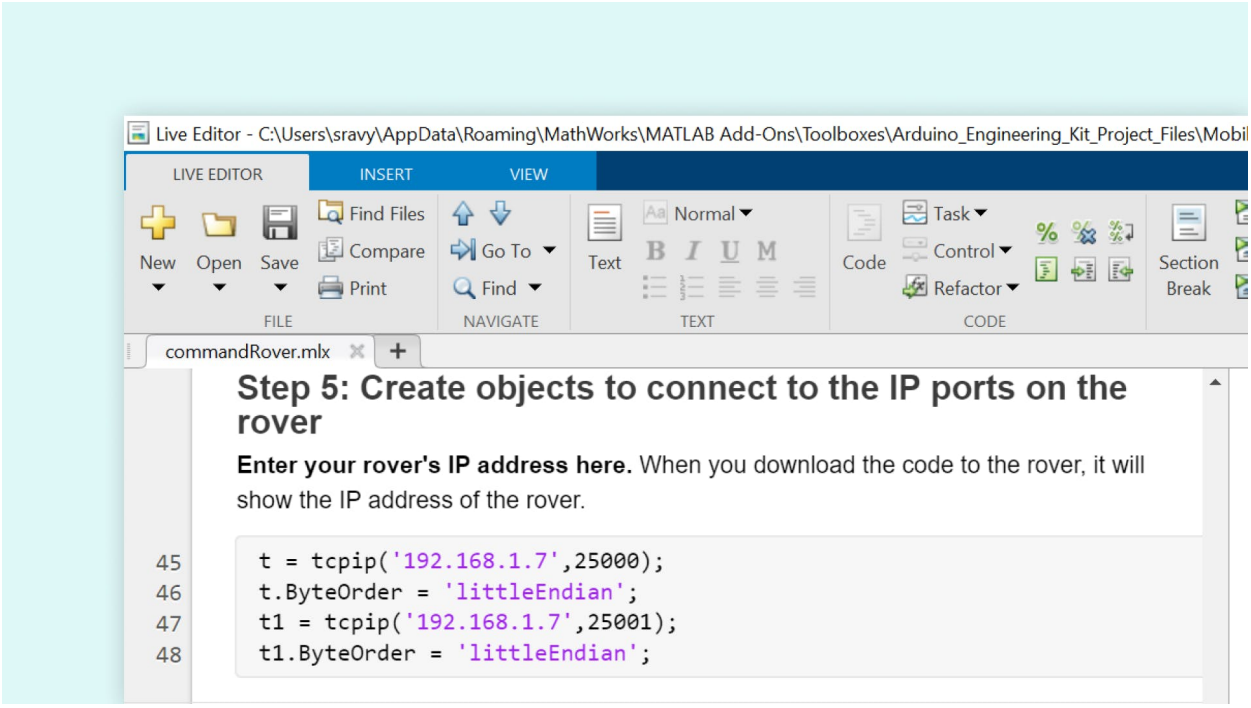


Open `commandRover.mlx` and run the first four steps. Remember to follow any instructions within the Live Script.

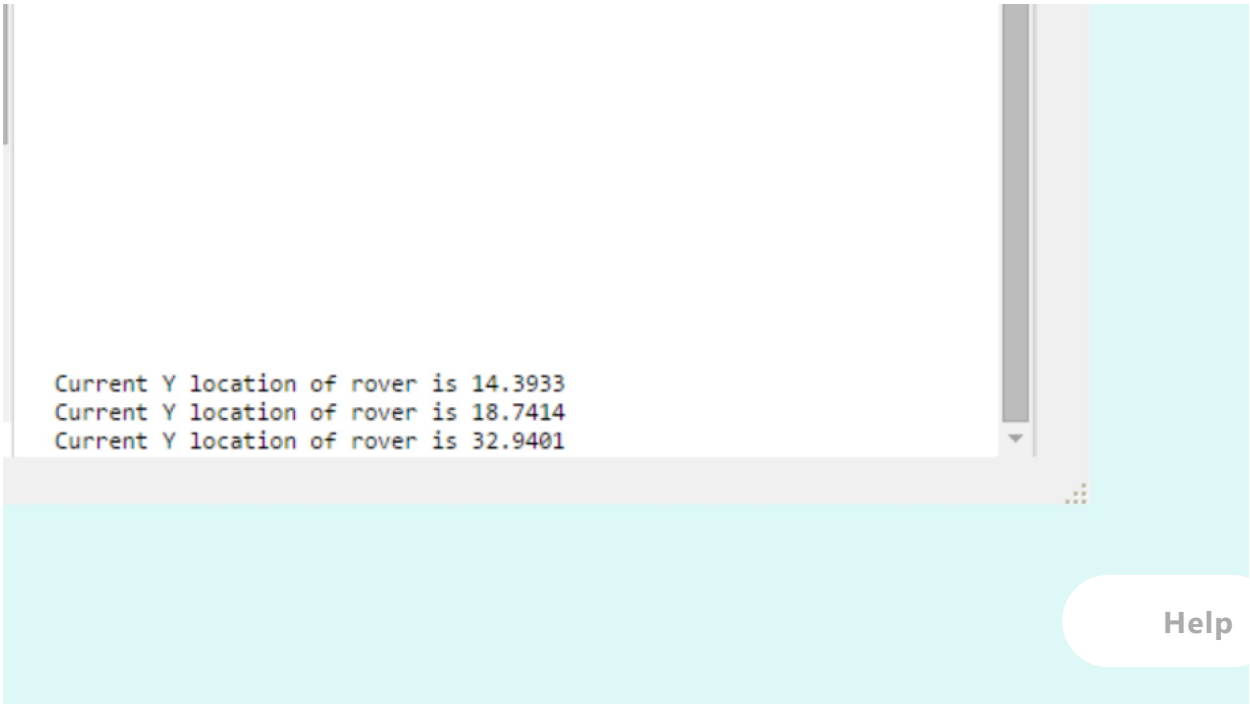
Enter your rover's IP address in "Step 5: Create objects to connect to on the rover" as the first input argument for `tcpip` in `t` and `t1`. The second argument (port number) remains unchanged. Click **Run and Advance**.

Help

Note: The port numbers of 25000 and 25001 corresponds to the `moveForward` and `FLangle` blocks respectively.



You will notice that the rover moves for approximately 1 second and stops. MATLAB calls the localization algorithm to calculate the rover's current y-location and displays it in the Output Window.



Once the rover has moved 30 cm in the y-direction from wherever it started, the forklift will drop. At this point, the application is complete.

Review

We saw how to communicate from MATLAB to the Rover and how to receive instructions over Wi-Fi on the rover.

Files

- roverReceive_hw.slx
- commandRover.mlx

Learn By Doing

Add another Wi-Fi block to your Simulink model and update your MATLAB code accordingly so that you can command the rover to turn when needed.

Help

5.8 Lessons Learned

In this chapter you were introduced to a series of topics:

Differential Drive Math

The control of a differential drive robot. You had to physically build a simple two-wheeled rover carrying a mechanism (forklift) to lift and drop objects.

Open-Loop vd Closed-Loop PID Controller

To get the rover to move, you learned about two possible strategies called open-loop and closed-loop. After discarding the open-loop controller that was just using information from the model of the motors under non-optimal conditions, you learned about how to implement a closed-loop controller using a PID controller using Simulink. While this strategy was a lot better than the open-loop controller, it still presented two undesired side effects: steady-state error and overshoot.

State Machines to Compensate Error

Once again, there is a way to avoid those two effects, in this case by using a state machine implemented with Stateflow, yet another toolbox within the MATLAB suite. Stateflow was also useful to create a simple program to control the forklift on the rover, which enables creating more challenging examples that include both moving the rover around, looking for an object at a desired location, and carrying it to yet another location. However, the initial conditions of the system were still unknown: where was the rover located at the time of starting our MATLAB Live Script?

Image Processing To Help Locate Objects in Real Time

This final question is solved using a camera and an image processing algorithm that will look for the rover's RGB marker, which will help us to find the rover with pretty high accuracy on an arena of a predetermined size. We saw that this requires a complex calibration procedure that needs to be performe Help the conditions change.

Wireless Communication

Finally, using wireless communication, it is possible to have the rover in the same Wi-Fi network as the computer running MATLAB, which allows the rover to directly talk to Simulink and in that way open the possibility to a dynamic configuration mechanism.

Help

5.9 Final Challenge

Given all the learned lessons in this chapter, we are now ready to formulate a challenge including all the components of this module.

Control the Rover using the Localisation Algorithm

Throughout the previous exercises, you saw how the rover can plan its path from a starting location to the target and finally to a specified endpoint location. One caveat with this approach is that there is no feedback on rover position other than the encoder data. If the wheels start slipping or the encoder data is noisy, you'll have problems.

It would be a good practice to have multiple sensors providing feedback on rover position to mitigate these potential issues. As you saw in section 7, you can send signals from desktop MATLAB to a deployed Simulink model using Wi-Fi blocks. For the final challenge, implement a path-planning algorithm on the rover that uses the localization algorithm from section 5 to provide additional feedback.

This is your final challenge. Go create an awesome Webcam controlled rover!