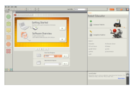
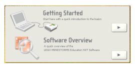
The Tribot Programming Environment and the Tribot's Sensors
Tutorial 1

In this tutorial you will familiarize yourself with the programming environment and build some small programs for the Tribot. These programs are designed to facilitate experimentation with and calibration of the sensors and actuators used by the Tribot. Recall that one of the challenges in robotics is to define and specify a set of assumptions about the environment in which your robot operates. These assumptions must include the capabilities and tolerances of the robot's sensors and actuators. For example, it would be difficult to program a robot to pick up an egg without knowing how much force is exerted by the actuators—too little, and the egg will fall; too much, and the egg will be crushed.

1 The Tribot Programming Environment





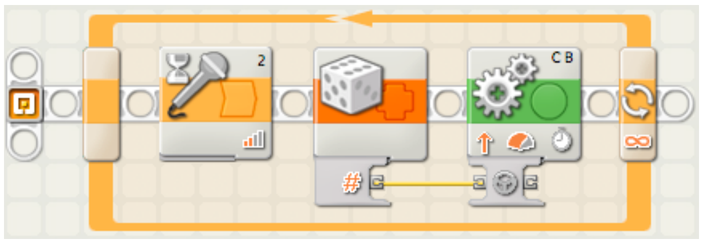
1. Log in into the computer and run the *Mindstorms Edu NXT* software by clicking on this icon . The software will take a few seconds to load; you will then see the application window.
2. View the "Software Overview" by clicking on the corresponding button in the central panel. This will identify the key panels of the the software development environment, including:

   Robot Educator: contains instructions for building and programming the robots
   Block Palette: used to the control the robots
   Block Configuration: used to configure the blocks
   Controller: used to download your programs to the robot
   NXT Window: used to view the status of you robot
   Help and Navigation: launches a browser with information on the software environment

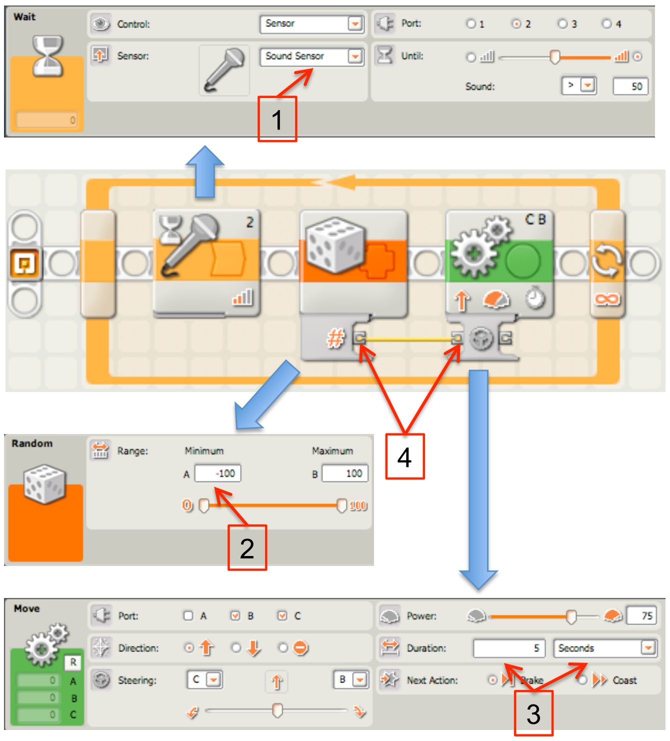
3. View the "Getting Started" by clicking on the corresponding button in the central panel. This will go over the process for building, downloading, and running programs on your robot.

2 The Sound Sensor

1. Start a new program called "SoundTest"
2. Construct the following program:



(a) Switch to the “Complete Palette” on the bottom left.
(b) Select the Loop block from the “Common” Category
(c) Select the Wait block from the “Common” Category and place it inside the loop.
(d) Select the Random block from the “Data” Category and place it inside the loop after the Wait block.
(e) Select the Move block from the “Common” Category and place it inside the loop after the Random block.
(f) Configure each of the three blocks in the loop in the following manner.



Wait
Control: Sensor
Sensor: Sound Sensor
Port: 1 2 3 4
Until:
Sound: > 50

Random
Range: Minimum A -100 Maximum B 100

Move
Port: A B C
Direction:
Steering: C B
Power: 75
Duration: 5 Seconds
Next Action: Brake Coast

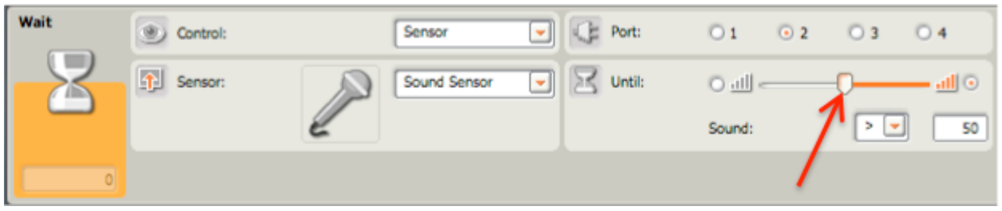
1 Click on the Wait block (in the loop) and change the sensor type from *Touch Sensor* to *Sound Sensor*.

2 Click on the Random block and change the Minimum (A) value of the block from 0 to -100.

3 Click on the Move block and change the Duration to 5 seconds.

4 Open the connections drawer of the Move block by clicking on the slit at the bottom of the block. Connect the Random block output to Move block’s steering input. Click on drawer again to close most of it.

3. Download the program into your Tribot and give it a run.
4. Adjust the sensitivity of the sensor by using the slider in the Until section of the Wait block’s configuration panel. Try running your program with various sensitivities.



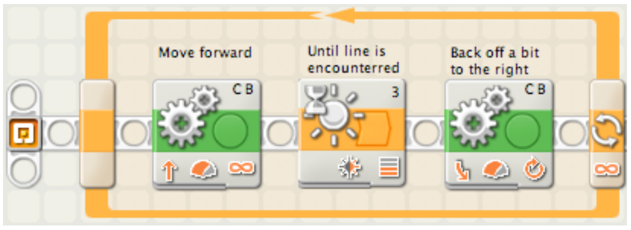
Wait
Control: Sensor
Sensor: Sound Sensor
Port: 1 2 3 4
Until:
Sound: > 50

5. **Questions for Section 2:**
   (a) What does this program do?
   (b) What are the optimal settings for voice command activation?

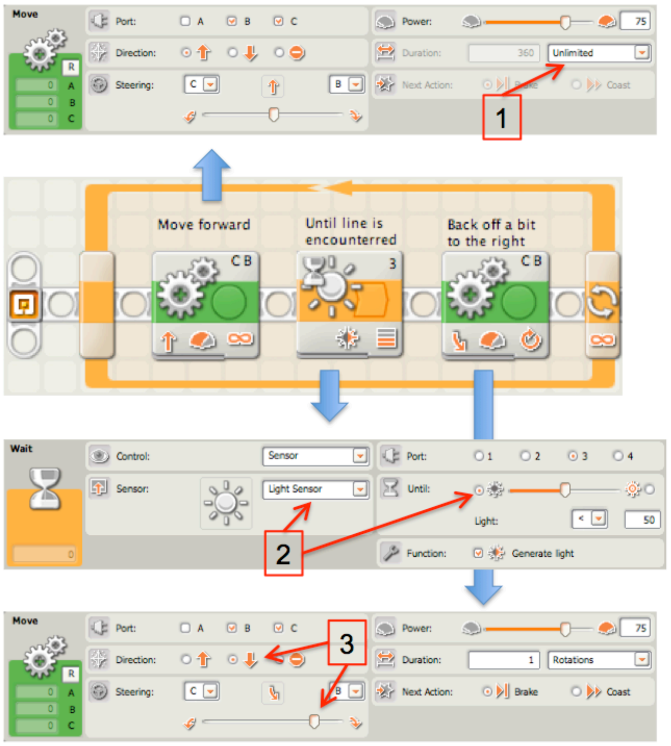
Record your results in a log book. You will need them later.
Note: remember there are other groups around you, so don’t be too loud.

3 The Light Sensor

1. Start a new program called "LightTest"
2. Construct the following program:



   (a) Switch to the "Complete Palette" on the bottom left.
   (b) Select the Loop block from the "Common" Category
   (c) Select the Move block from the "Common" Category and place it inside the loop.
   (d) Select the Wait block from the "Common" Category and place it inside the loop after the Move block.
   (e) Select the Move block from the "Common" Category and place it inside the loop after the Wait block.
   (f) Configure each of the three blocks in the loop in the following manner.



Move
Port: ☐ A ☑ B ☑ C
Direction:
Steering: C, B
Power: 75
Duration: 360 Unlimited
Next Action: Brake, Coast

Wait
Control: Sensor
Sensor: Light Sensor
Port: 1, 2, [✓ 3], 4
Until:
Light: < 50
Function: ☑ Generate light

Move
Port: ☐ A ☑ B ☑ C
Direction:
Steering: C, B
Power: 75
Duration: 1 Rotations
Next Action: [✓ Brake] Coast

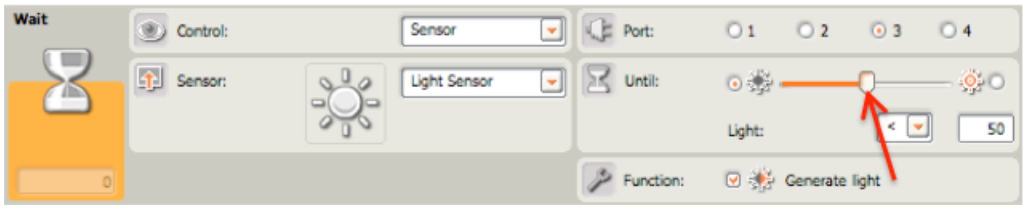
$\boxed{1}$ Click on the Move block and change the Duration to unlimited.

$\boxed{2}$ Click on the Wait block (in the loop) and set the sensor type to *Light Sensor*. Change the Until setting from light to dark.

$\boxed{3}$ Click on the Move block and change the direction to reverse. Also, move the steering slider to the right to get the robot to back off and change direction simultaneously.

3. Use the provided bristol board and black electrical tape to create a ring. Use the black tape to demarcate the ring boundary.

4. Download the program into your Tribot, place the Tribot in the center of the ring, and give it a run. Does the program do what you expected it to do?
5. Adjust the sensitivity of the sensor by using the slider in the Until section of the Wait block's configuration panel. Try running your program with various sensitivities.



Wait

Control: Sensor

Sensor: Light Sensor

Port: 1 2 [✓ 3] 4

Until:

Light: < 50

Function: [✓] Generate light

6. **Questions for Section 3:**
   (a) What does this program do?
   (b) What are optimal settings for the sensor to detect the black line?
   (c) If the lighting conditions change, will the optimal settings be affected?
   (d) If the speed of the motors change, will the optimal settings be affected? Why or why not? Experiment by adjusting the Power setting of the first Move block in the loop.
   (e) What assumptions must be made making when using the light sensor?

   Record your results in a log book. You will need them later.

4 The Ultrasonic Sensor

1. Start a new program called "UltrasonicTest"
2. Build a simple program that drives the robot forward until it detects an object in front of it. The robot should make a left turn to avoid the object and continue moving. Just like in the preceding program, you will need to use a Loop block, a Move block, a Wait block, and another Move block.
3. Run the program with various sensitivities for the ultrasonic sensor.
4. **Questions for Section 4:**
   (a) What are optimal settings for the sensor to trigger when an object is too close? (An object is too close if the Tribot cannot avoid it by making a sharp (90+ degrees) turn.
   (b) Suppose the speed of the motors change, are these settings still optimal? Why or Why not?

   Record your results in a log book. You will need them later.

5 If You have Time

Try writing some of the programs that are included in the Common Palette of the Robot Educator. In particular try: 08. Drive in Square, 12. Detect Sound, 14. Detect Distance, 16. Detect Dark Line, and 19. Sensor Bumper. If you have more time, try others as well. Become comfortable with using, configuring and combining various blocks.

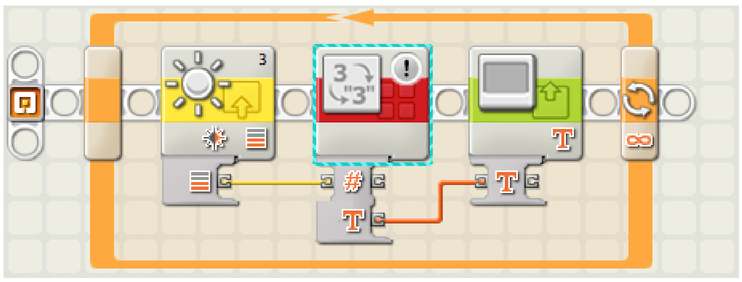
Characterizing Sensors
Tutorial 2

In this tutorial you will empirically evaluate the Tribot's sensors. You will be provided with program templates that you will use to measure the accuracy of the sensors. This is called "characterizing" the sensors, i.e., determining the sensors' characteristics. This tutorial builds on the previous one by having you empirically evaluate the sensors you will be using for this module.

1 Characterizing the Light Sensor

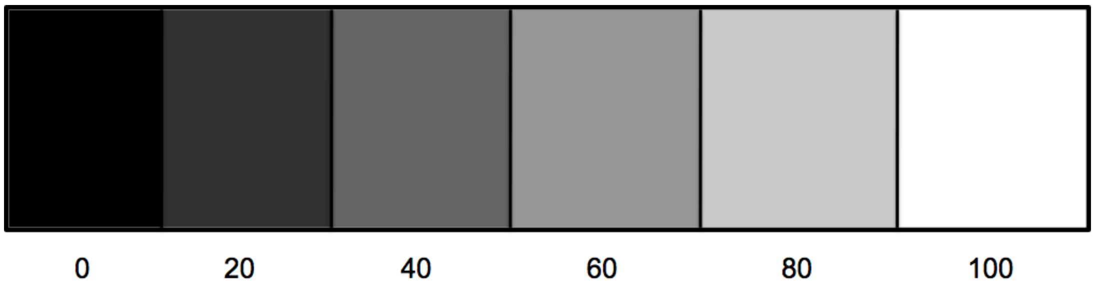
In this section, you will sample the return of the light sensor for the various gray-scale bands illustrated below.

1. Download and open the program "SensorProbe" which can be found on the course website at: `http://www.cs.dal.ca/~prof1106/robotics.html`. It should look like this:



   Note that the first block in the loop is a Light Sensor block.
2. Save this program as "LightProbe" and download it to the Tribot.
3. Download the program into your Tribot and give it a run.
4. Create a table with two columns: one column labeled "Grayscale Value" and the other labeled "Light Sensor Reading". **This is to be included with your lab report (the lab sheet that has several additional questions for you to answer).**
5. While the program is running, for each band in the gray-scale below:

   (a) Place the robot so that its light sensor is directly over the band.
   (b) Record the gray-scale value and the light sensor reading in your table.



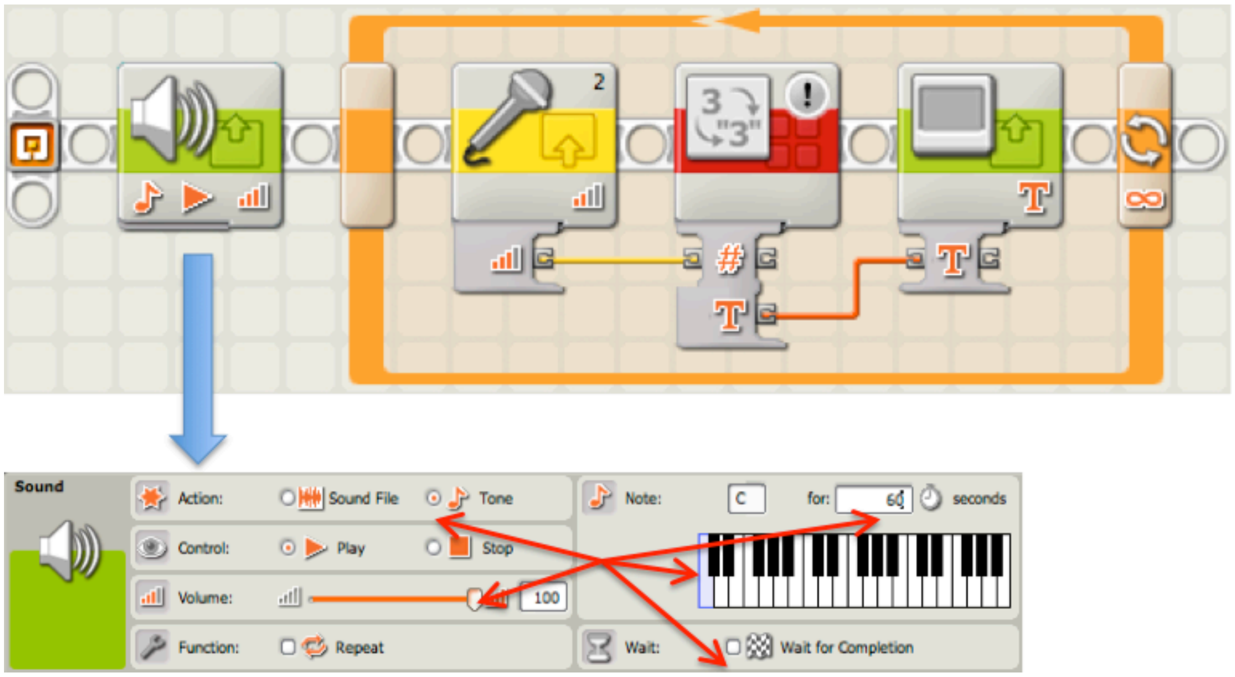
6. Plot a graph of "Grayscale Value" on the x-axis and "Light Sensor Reading" on y-axis. **This is to be included with your lab report as well.**
7. Stick a piece of electrical tape onto your wooden desk. Read the light sensor when:
   (a) the robot is sensing the desk
   (b) the robot is sensing the tape, and
   (c) the robot is sensing the edge of tape/desk.

   Include your observations as part of the report.
8. **Questions for Section 1:**
   (a) Try moving the robot slowly from sensing the desk to sensing the tape. Is the transition in sensor values abrupt or gradual?
   (b) What is an appropriate value to distinguish the tape from the desk? (i.e. if greater than this value, then it is desk and if less than this value then it is tape)
   (c) Your graph should be a straight line. What is its slope? What are the possible sources of error or noise in the "Grayscale Value vs. Light Sensor Reading" graph?
9. **Things to think about:**
   (a) Would the lighting conditions affect the readings? For example, would shadows or additional overhead lights change your measurements?

2 Characterizing the Sound Sensor

In this section, you will record sound sensor values for different distances away from the sound source and at two different frequencies. In Physics, a sound wave loses pressure at a rate of $\frac{1}{distance}$, also called the Inverse Law. We will use this to characterize the sound sensor. The Tribot's brick will serve as a sound source, emitted from its left-front edge (where the slits are located). Measurements will be made as the sound sensor is moved to various distances away from the sound source.

1. Open the program "SensorProbe" which you downloaded in the previous section.
2. Replace the Light Sensor block (the first block in the loop) with a Sound Sensor block and connect its output connector to the Number-to-Text converter input .
3. Add a sound generation block before the main loop.
4. Set it to play the lowest tone (a "C"), maximize the volume, uncheck the Wait for Completion box, and set the duration to 60 seconds.
5. Save this program as "SoundProbe" and download it to the Tribot.



Sound

Action: Sound File [✓ Tone]

Control: [✓ Play] Stop

Volume: 100

Function: ☐ Repeat

Note: C for: 60 seconds

Wait: ☐ Wait for Completion

6. Create two tables, each with three columns labeled "Distance", "Sound Sensor Reading", "Computed Pressure" (explained later in this tutorial). This will be included with your lab report (the lab sheet that has several additional questions for you to answer).
7. Dislodge the sound sensor from the Tribot and turn it around so that it faces the sound source.
8. Run the program. For each distance: 1cm, 2cm, 4cm, 6cm, 8cm, and 10cm.
   (a) Use a ruler to help you manually fix the distance between the sound source and the sensor to the specified distance.
   (b) Hold the sensor in place and read the value from the display.
   (c) Record the distance and sensor value in one of your tables.
9. Change the sound block to emit the highest tone (a "B").
10. Repeat step 8 and record results in the other table.
11. It is possible to compute what the sound pressure values *should* be using the readings at 1cm and the inverse square law. The computed values are of the form $\frac{s_0}{\ell}$ where s_0 is the sound level measured at 1cm and ℓ is the distance (in cm) from the sound source. For example, if the sound level at 1cm was 96 and you wanted to know what it would be at 16cm, then you would plug in 96 and 16 into the formula, $\frac{96}{16} = 6$. In the third column of each table, provide these computations.
12. Draw two line graphs, one for each table. The x-axis should be the distance, and the y-axis the sensor reading and computed values. Include both the measured and computed values. These graphs should be included with your lab report (the lab sheet that has several additional questions for you to answer).
13. Reattach the sound sensor to the Tribot when finished.
14. **Questions for section 2:**

    (a) What are the possible reasons for differences (errors) between the computed values and the actual values?
    (b) How does the frequency (i.e. high note or low note) affect the recorded values?

15. Things to think about:
    (a) Computing the values assumes that the measurement at 1cm is accurate. Is this a good assumption? Suppose the 1cm measurement was too low due to ambient noise, how would this affect the graph of computed values?
    (b) Is maintaining the sound sensor at the same angle to the sound source important? Why or why not?

3 Characterizing Ultrasonic Sensor

In this section, you will use the ultrasonic sensor to measure distance to a flat plane.

1. Open the program "SensorProbe" which you downloaded in the previous section.
2. Replace the Light Sensor block (the first block in the loop) with a Ultrasonic Sensor block and connect its output connector to the Number-to-Text converter input.
3. Set the sensor to display distance in centimeters.
4. Save this program as "UltrasonicProbe" and download it to the Tribot.
5. Create two tables, each with two columns labeled "True Distance" and "Ultrasonic Sensor Reading". This will be included with your lab report (the lab sheet that has several additional questions for you to answer).
6. Run the program.
7. For each distance: 10cm, 20cm, 30, 40cm, 50, 60cm, 70, 80cm, 90, and 100cm
    (a) Use a ruler to help you manually fix the distance between the sensor and a flat object, such as a text book, directly facing it.
    (b) Read the value from the display.
    (c) Record the distance and sensor value in one of your tables.
8. Get together with another group and repeat step 7, but use a facing Tribot instead of a flat surface. The distances should be measured between the ultrasonic sensors of the two Tribots, which should be facing each other. Record results in the second table.
9. Plot results from both tables in a single graph and label them clearly. These will be included in your lab report (the lab sheet that has several additional questions for you to answer).
10. In your graph, also draw a line connecting coordinates (1cm, 1cm) (100cm, 100cm). This line represents the true distance, i.e., what would be considered a perfect reading.
11. **Question for section 3:**
    The Tribot's returned values are what it believes the distance is in centimeters. Do the measured face-on distances match up closely with the true distance? Why might it deviate?
12. Things to think about:
    (a) Suppose that instead of using a textbook, you used your hand or a piece of clothing, would this change the distance measured? Why?

Characterizing Drive Actuators
Tutorial 3

In this tutorial you will empirically evaluate the Tribot's actuators. For example, suppose that you wanted your robot to make a 90 degree right turn. For each degree that the robot turns, its left wheel must rotate forward and its right wheel must rotate backward for a specific duration (time, degrees, or rotations in NXT). Without knowing the duration needed to turn the robot 1 degree, it is not possible to program the robot to make a 90 degree turn. In fact, as we shall see, the duration itself may also change, depending on the speed of the turn, and its weight!

Also, you will evaluate movement accuracy. Initially you would expect that the robot will move or turn by an exact amount and that the robot should behave identically for a rerun of the same program. However, this may not be the case, and knowing the accuracy of the robot's movement will be critical when deciding how much to rely on the expected position of the robot in a particular task.

1 Drive Actuators and Distance

1. Start a new program called "DistanceTest"
2. Build a simple program that drives the robot forward 25cm. This is an easy program, except for the fact that you do not know the duration (time, degrees, or rotations in NXT) at which the motors must be powered to move the Tribot 25cm.
3. Run the program with various durations (use degrees) until you have settled on the correct duration.
4. **Questions for Section 1:**
   (a) For top (100), medium (50), and slow (20) speeds:
      i. What is the approximate duration required to travel 25 cm in the forward direction?
      ii. Compute the approximate duration required to travel 1 cm in the forward direction.
      iii. Using the approximate duration for 25 cm forward, run the forward movement 10 times, carefully setting the robot in the same initial position and marking its end point each time. What is the difference between the shortest and longest distances covered?
      iv. Repeat the last three steps, but driving the robot in the backward direction.
      v. Is the duration for going forward and backward the same for the same distance? If not, why not?
      vi. Change the Next Action setting of the motors from *Break* to *Coast*. How does this change the distance traveled by the robot?
   (b) Why is the forward duration required to travel 25 cm different for different speeds?

(c) Is the range (shortest to longest) of distances traveled different for different speeds? Which range is the largest? Which range is the smallest?
(d) Did the robot stay mostly centered or did it consistently curve to one side or the other? If it curved, how might you correct for this?
(e) Create a program that moves forward 25cm and then backward 25cm for a speed/power of 50. Put this into a loop that runs 10 times. Mark the initial position of the robot before and after the program is run. How far away did the robot deviate from its initial position?
(f) If you increase the number of forward and backward movements, will the robot be closer or further from its initial starting point?
(g) Based on your findings in the preceding questions, what are the two ways you can improve positional accuracy?

2 Drive Actuators and Speed

The speed programmed into a motor block is not guaranteed to work as you might first expect. That is, if you double the programmed power, you are not guaranteed to get an actual two-fold increase in speed. In this section we will evaluate the speed at which the robot moves versus the power setting.

1. Place start and finish lines separated by 50cm. For powers of 10, 15, 20, 30, 40, 50, 60, 70, 80, 90 and 100, how long does it take to travel 50cm? Download the program "Displaytimer-value.rbt", which will record the time that passes between activating the program and until the robot detects black tape.
2. Using $velocity = \frac{distance}{time}$, calculate the speed for each power level. Plot these on a graph with the x-ordinate being power and the y-ordinate being speed.
3. **Questions for Section 2:**
   (a) Is the relationship between the power and speed linear? That is, if you add the speed at 20 and speed at 40, do you get the speed at 60?
   (b) If your initial power was 40, what power does your graph predict would give double the speed?
   (c) Using your program from before, what is the slowest speed your robot can run and still move?
   (d) If you set the robot's power adapter on top of the top motor (to add weight to your robot) does this slowest speed still move the robot forward? If not, what is the new slowest speed?

3 Drive Actuators and Turning

In this section we will investigate the accuracy of turning using the drive actuators.

1. Create a new program called "TurningTest".
2. Build a program that rotates the robot 360 degrees (a sharp right turn) at medium (50) speed.
3. **Questions for Section 3:**
   (a) What is the duration that completes a full circle (360 degrees) in the clockwise direction?

(b) Add to your program so that after the desired rotation is complete, the robot drives forward 10 cm. Being careful to start your robot in the same orientation, run this test 10 times (clockwise direction) and mark the position of the light sensor for each. What is the distance between the two furthest points? In most programs, turning or rotating is often followed by movement. As turning accuracy is reduced what should happen to positional accuracy?

(c) Repeat the last two steps for the counter-clockwise condition.

(d) Increase the speed to 100. Run the program 2 or 3 times. Does the increase in speed seem to change the required turning duration?

(e) Based on your experiments in the previous section, predict whether the range of final positions will increase or decrease for this speed. Then, for the clockwise direction only, test your prediction by running the test 10 times and measuring the range. Was your prediction correct? If not, why not?

(f) Is the robot off-center after a rotation? If so, by how much?

4 If You have Time

Based on all you have learned about your robot, make a program that:

1. Moves forward and turns right,
2. Repeats last step 4 times to complete a square,
3. Rotates 180 degrees, and
4. Moves in a square again, retracing the steps of the previous square
5. **Questions for Section 4:**
   (a) How far was the robot (in cm) from its initial position?
   (b) With some calibration, can you improve your accuracy? If so, what is the closest you can get?

Even with careful calibration, it is difficult to get perfect accuracy. So that raises the question... "How can we get robots to do what we want if they won't do what we ask?" We'll take a look at this in the upcoming labs.

Modeling the Real World
Tutorial 4

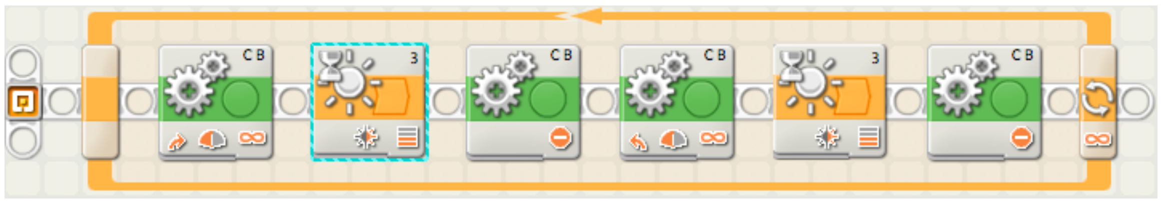
In this tutorial you will investigate methods for modeling the robot's behaviour and environment. You will begin by writing a small program that will cause the Tribot to follow a thick black line (electrical tape). You will then experiment with the program, extend it in various ways, and derive a general approach for modeling robot behaviour in terms of states and transitions.

Robot behaviour in general can become very complex, particularly as the complexity of the robot's environment and the complexity of the tasks it must perform increases. Programming complex behaviour can quickly become overwhelming without some ability to manage this complexity. One approach is to model a robot's behaviour in terms of states and transitions. A state refers to a situation where a unique set of assumptions hold. For example, consider the SoundTest program from the previous tutorial. Initially a Tribot may be in a "Wait for sound" state in which the Tribot does not move and waits for a sound to be made. Once it hears a sound, it transitions to the "Move randomly" state in which it moves for five seconds. After the five seconds are up, it transitions back to the "Wait for sound" state. In another example, consider the LightTest program from the previous tutorial. In this program the Tribot first enters the "Move forward" state, in which it moves forward until it encounters a black line. When the line is encountered, it transitions into the "Back-off" state, in which it backs to the right, and then transitions back to the "Move forward" state.

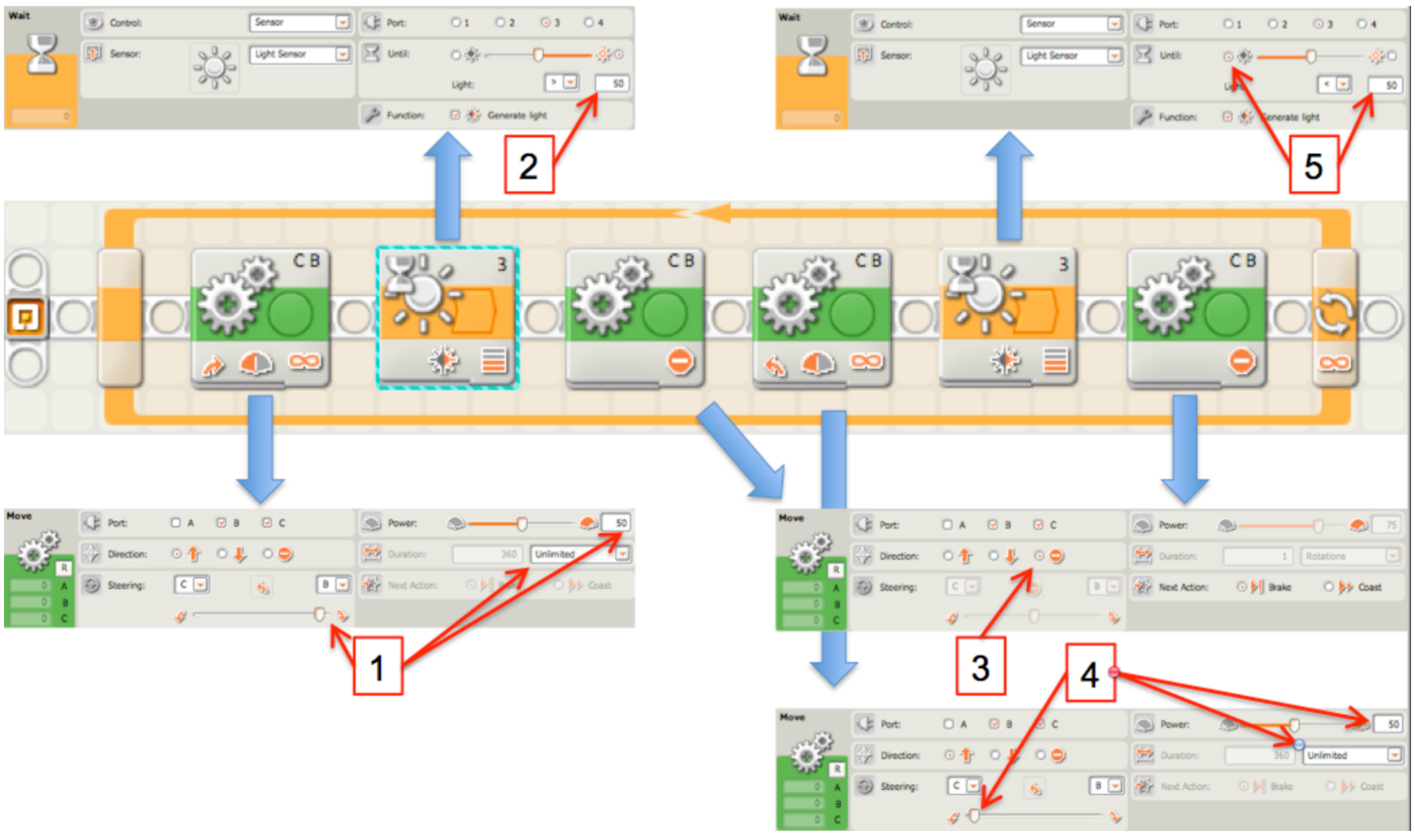
As we shall see, there are also problems with this approach. The number of states can quickly become very large and the number of transitions even greater. Furthermore, the state-transition approach may not be appropriate for certain tasks. In this tutorial you will investigate the state transition approach, as well as situations in which it is and is not appropriate.

1 A Simple Line-Following Program

1. Start a new program called "FollowLine"
2. Construct the following program:



3. Configure each of the five blocks in the loop in the following manner.



$\boxed{1}$ Click on the first Move block and set Power to *50*, the Duration to *unlimited*, and the Steering to almost completely right. You can adjust the Power and the Steering later.

$\boxed{2}$ Click on the first Wait block (in the loop) and change the sensor type from *Touch Sensor* to *Light Sensor*. Adjust the sensitivity of the sensor to wait for light (no line) using the observations you made in the previous tutorial.

$\boxed{3}$ Click on the second Move block and change the Direction to *Stop* . Click on the fourth Move block and change the Direction to *Stop* as well.

$\boxed{4}$ Click on the third Move block and set the Power to *50*, the Duration to *unlimited*, and the steering to almost completely left. You can adjust the Power and the Steering later.

$\boxed{5}$ Click on the second Wait block (in the loop) and change the sensor type from *Touch Sensor* to *Light Sensor*. Change the Until to wait for dark instead of light and adjust the sensitivity of the sensor to detect black (line) using the observations you made in the previous tutorial.

4. Using the ring you created in the previous tutorial, test how well the Tribot follows the line.
5. Try adjusting both the Power (speed) at which the motors run and the sharpness of the turn that the Tribot makes.
6. Using an additional bristol board create a more challenging course for the Tribot. Be sure the route is closed and that it has a couple sharp corners. Try running your Tribot with different speed and turn settings.

7. Find the optimal settings for your line-follower.
8. Save the program by clicking on the $\boxed{\text{File}}$ menu and then on $\boxed{\text{Save}}$.
9. Start a new program called “LegoLineFollow”.
10. Create the $\boxed{\text{17. Follow a Line}}$ program in the $\boxed{\text{Common Palette}}$ of the Robot Educator. Try running this program and compare it to your program.
11. Be sure to save this program as well.
12. Answer the remaining questions below.
13. **Questions for Section 1:**
    (a) Briefly explain how this first program works?
    (b) How many states does this program have and what causes the transitions from one state to another (in this case)?
    (c) How does the second program work?
    (d) Does the second program have states? If no, why not? If yes, how many states does it have; what are the states; and what are the transitions?
14. **Things to think about:**
    (a) Based on the first program and the two examples in the introduction, what features (program blocks) tend to represent a program state?
    (b) Does the Tribot always stay on track (with the original settings)?
    (c) Under what conditions will the Tribot go off-track? Why?
    (d) Try removing the second and fourth Move blocks from the program (the ones that stop the Tribot before it changes direction). Run the Tribot over the track again, using the original settings and some of your own settings. Does it perform better or worse? Why?
    (e) Does the second program work better or worse than the first program?

2 Extending the Line-Follower

One problem with our current simple program is that the Tribot will bump into things as it is following the line. Ideally, we would like to make use of the ultrasonic sensor to detect objects in the Tribot’s path and pause the Tribot until the object is moved. Our next goal is to extend the basic line-following program to achieve this requirement.

1. Start with the “FollowLine” program (the first program you created), with the optimal settings that you determined in the previous part. Make a copy of the program by saving it as “FollowLineStop”. Click on the $\boxed{\text{File}}$ menu, then $\boxed{\text{SaveAs}}$, then change the name to “FollowLineStop” and click $\boxed{\text{OK}}$.
2. Now extend “FollowLineStop” so that the Tribot stops if its ultrasonic sensor detects an object less than 10 inches in front of it. Hint: You will very likely need to use Wait blocks and Move blocks (to stop and resume transit), as well as (potentially) Loop blocks and/or Switch (Conditional) blocks.
3. Your first approach could be something basic: Use Wait blocks, before the first and third Move blocks, to wait until the ultrasonic sensor does not register anything closer than 25cm.
4. **Questions for Section 2:**
    (a) How many states does this program have and what are they?
    (b) What are the transitions between the states?

5. **Things to think about:**
   (a) Identify why this approach is not optimal. Think about what would happen if drivers drove cars in this manner.
6. (Optional) A better approach would be to keep checking if there are any objects in the way. However, now you have to check for two different conditions: 1) if the state of the light sensor has changed, and 2) if the state of the ultrasonic sensor has changed. Unfortunately, the simple Wait blocks, which wait for the state of a single sensor to change are not sufficient, so you may need to use a Loop block to create your own Wait-like mechanism. Note: this is a challenging problem. Here is a couple things to think about:
   - What are the states in this program?
   - What are the transitions between the states?

3 Extending the Lego Line-Follower

Interestingly, the structure of your program has a significant effect on how easy or hard it is to extend it. As an example, we will now try extending the Lego line-following program to also stop if there are any objects in front that are closer than 10 inches.

1. Start with the “LegoFollowLine” program (the second program you created) and make a copy of the program by saving it as “LegoFollowLineStop”.
2. Now extend “LegoFollowLineStop” so that the Tribot stops if its ultrasonic sensor detects an object less than 25cm in front of it. Hint: You will only need one Switch (Conditional) block to check if an object is too close, one Move block to stop the Tribot, and one Wait block to wait until the object has been moved.
3. These blocks will need to be used at the beginning of the main loop right before the other Switch block in the loop.
4. **Questions for Section 3:**
   (a) How many states does this program have and what are they?
   (b) What are the transitions between the states?
   (c) Was this extension easier than the one for the first version of the program? Why?

4 If You have Time

If you have any additional time try adjusting the settings so that the Tribot can follow the line even faster. Try writing your own line following program and see if you can create a better line-following program.

Dealing with an Imperfect World
Tutorial 5

In this tutorial you will investigate methods for dealing with the harsh reality that the world is not perfect, that sensors fail, and that almost nothing goes exactly as planned. You will begin with a line following program that is almost identical to the one we developed in the last tutorial. We will then “mess” with the program, and then investigate how to compensate for the “messing”.

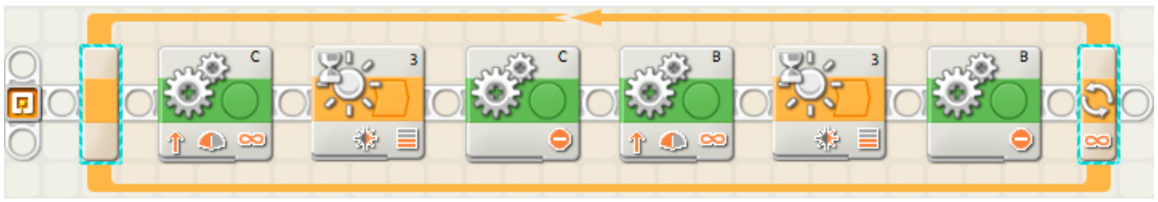
To deal with the imperfections of the real world, we must do two things. First, we must identify when something has gone wrong, i.e., the failure mode. For example, you identify that you have missed the turn-off because you have driven for too long or that the furnace is broken because it’s suddenly very cold in the house. Note, in most cases, the failure modes are explicitly enumerated and hence form part of the underlying assumptions in the system. In many cases, systems fail because the designers have not taken all failure modes into account, i.e., their assumptions are either incorrect or incomplete.

Once the identification of failure modes is accomplished, recovery mechanisms must be designed to deal with the failure modes. In some cases, the only solution is to shut the system down and wait for a repair-person. However, if the system itself is in an inconvenient location, say Mars, this is not a workable solution. For example, in the case of missing the turn-off, we find a way to reverse directions and return to the turn-off. In the case of a furnace failure we call the landlord (and hope for the best). Of course, if a failure occurs in the recovery mechanism, we may need a recovery mechanism for the recovery mechanism—especially if you are the landlord. In the language of states and transitions, a failure mode occurs when the system enters a state it’s not supposed to be in. The goal of recovery is to return the system to a state that it is supposed to be in.

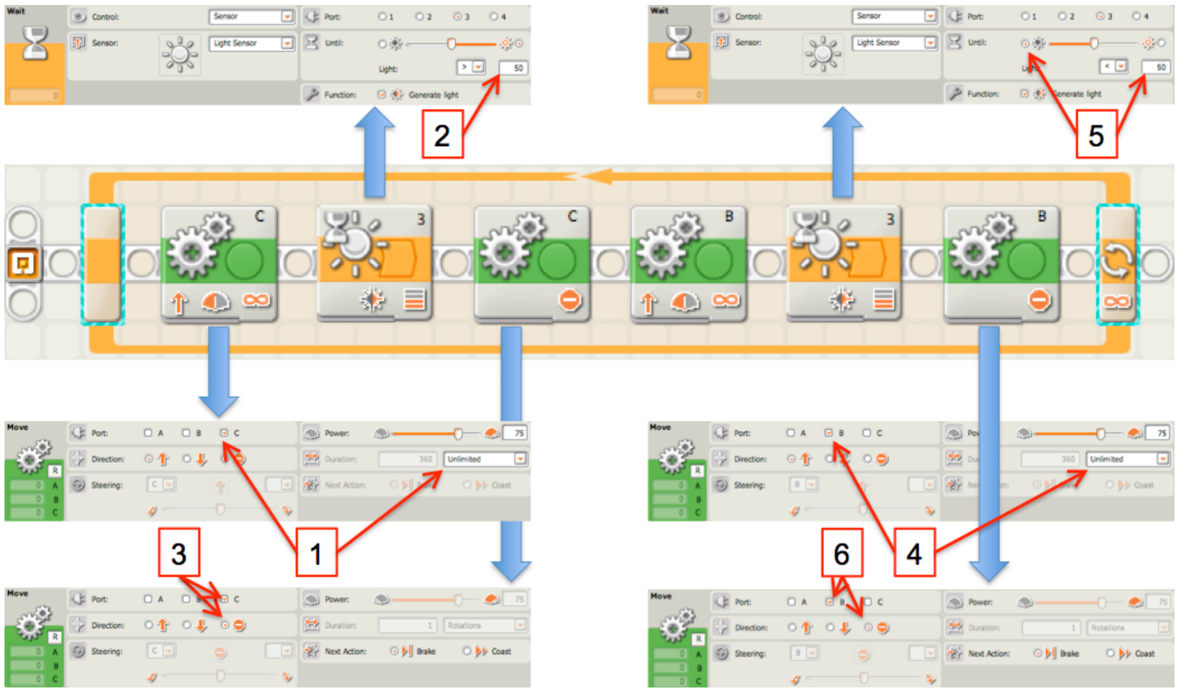
In this tutorial you will investigate how to deal with the most common type of failures, those of sensors and actuators.

1 Another Simple Line-Following Program

1. Start a new program called “FollowPreset” and construct the following program:



2. Configure each of the five blocks in the loop in the following manner.



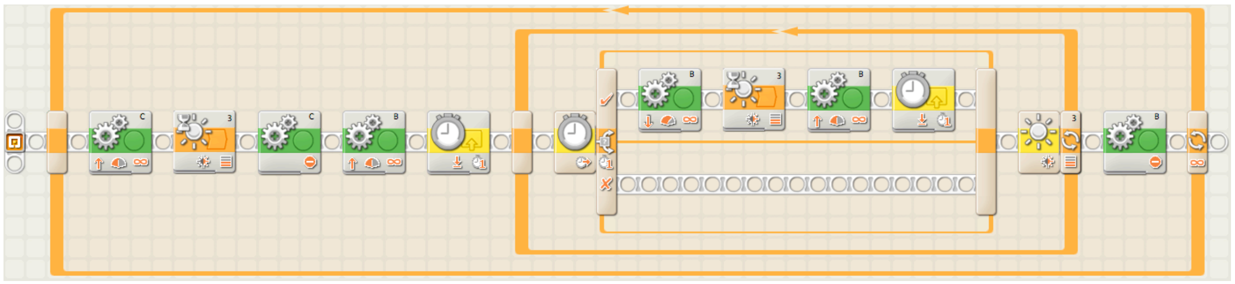
[1] Click on the first Move block, set the Duration to *unlimited*, and set the Port to *(c)*.
[2] Click on the first Wait block (in the loop) and change the sensor type from *Touch Sensor* to *Light Sensor*. Adjust the sensitivity of the sensor to detect white (no line) using the observations you made in the previous tutorial.
[3] Click on the second Move block, set the Port to *(c)*, and set the Direction to *Stop* ⊖.
[4] Click on the third Move block, set the Duration to *unlimited*, and set the Port to *(b)*.
[5] Click on the second Wait block (in the loop) and change the sensor type from *Touch Sensor* to *Light Sensor*. Change the Until to trigger on dark instead of light and adjust the sensitivity of the sensor to detect black (line) using the observations you made in the previous tutorial.
[6] Click on the fourth Move block, set the Port to *(b)*, and set the Direction to *Stop* ⊖.

3. Observe that this program is nearly identical to the previous one, except that the driving is done a little differently.
4. Using the ring you created in the previous tutorial, test how well the Tribot follows the line.
5. **Questions for Section 1:**
   (a) What would cause the Tribot to lose the line?
   (b) Take a small blank piece of paper and place it over a portion of the line. What happens when the Tribot passes over this portion?
6. **Things to think about:**
   (a) How do we know (how do we identify) when the Tribot has lost the line?
   (b) How would a Tribot know (identify) that it has lost the line?
   (c) Once we (or the Tribot) has identified that the line has been lost, how do we recover?

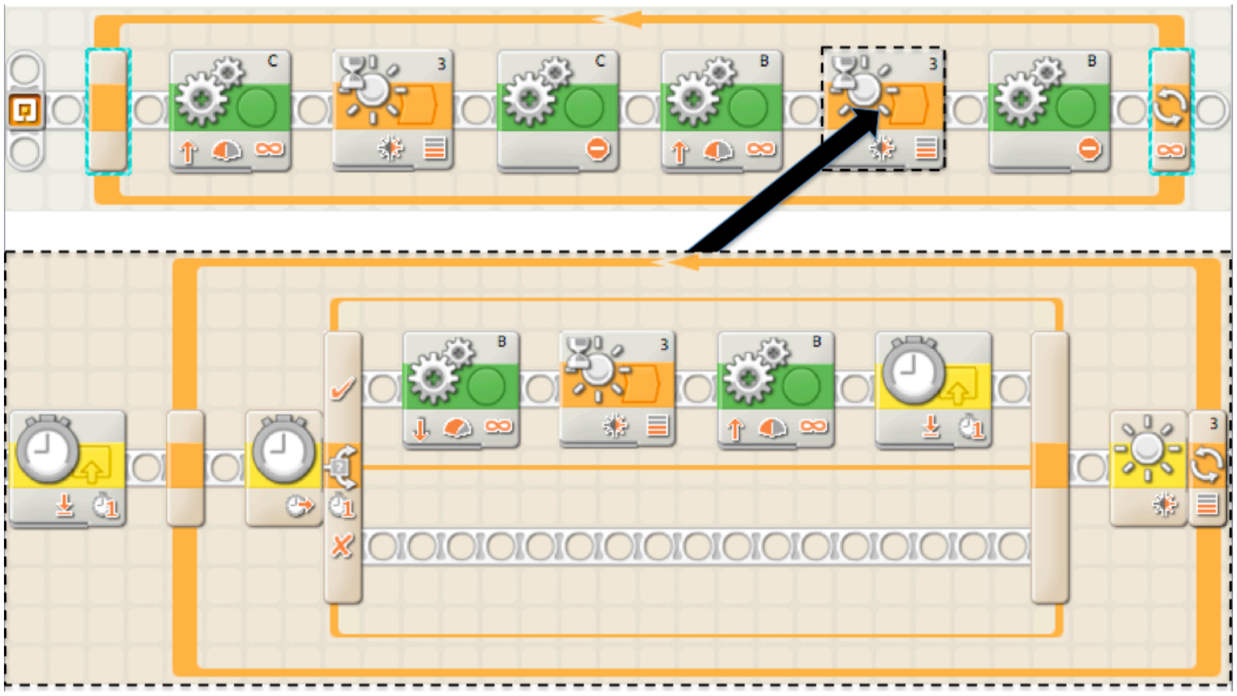
2 Failure Mode Identification and Recovery

We will now extend the "FollowPreset" program by adding in a simple mechanism for detecting the lost line failure mode and another simple mechanism for recovering the lost line.

1. Save a copy of the "FollowPreset" program as "FollowReset" and create the following program:

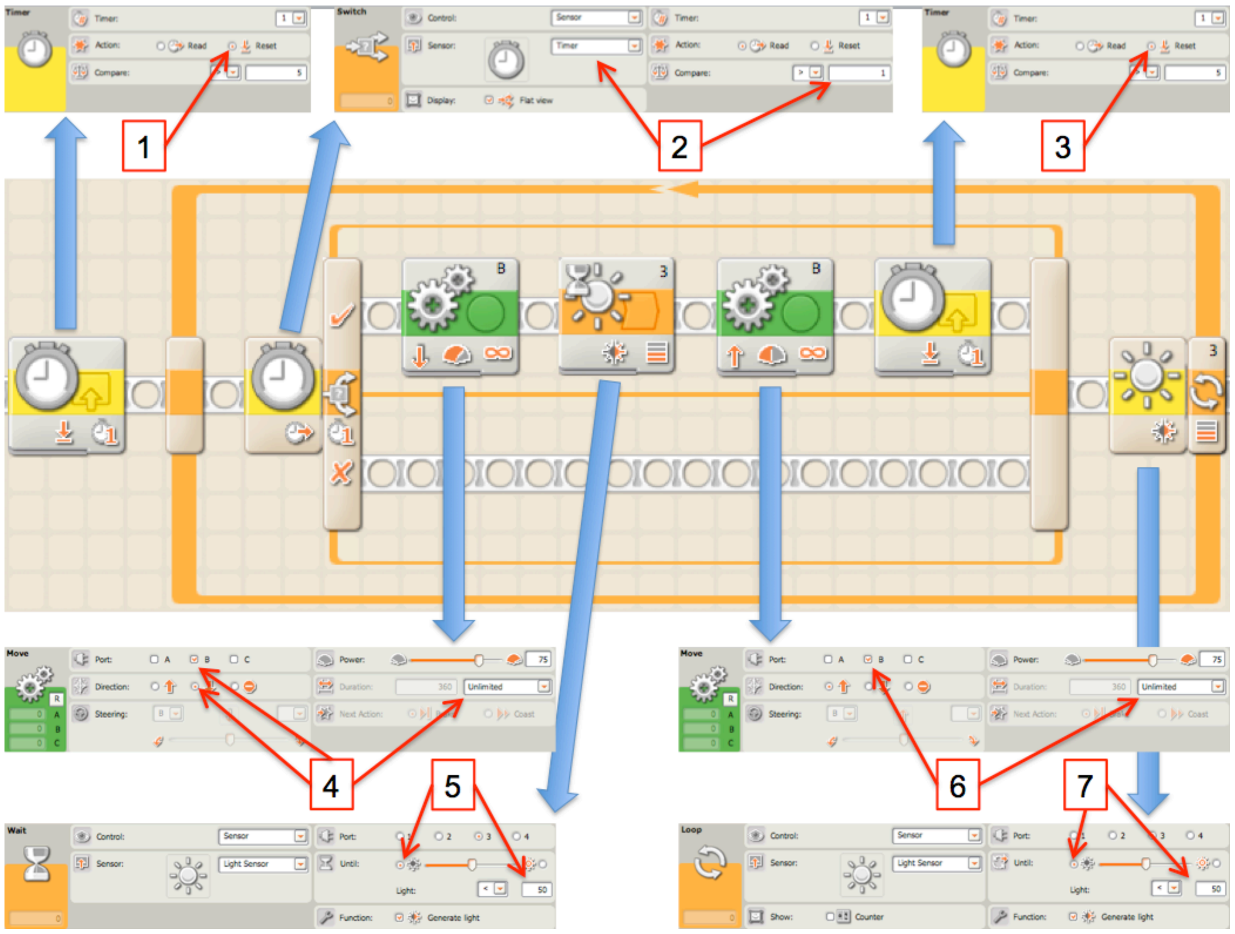


by replacing the second Wait block with a sequence of blocks as illustrated below:



the new block that you may not be familiar with is the Timer block, which is found in the Sensor Palette. You will also need a Loop block and a Switch (Conditional) block.

2. Configure each of the five blocks in the loop in the following manner.



$\boxed{1}$ Click on the first Timer block, set the Action to *Reset*.

$\boxed{2}$ Click on Switch block, set the Sensor to *Timer* and set the Compare to > 1.

$\boxed{3}$ Click on the second Timer block, set the Action to *Reset*.

$\boxed{4}$ Click on the first Move block, set the Direction to *Reverse* ⇩, the Duration to *unlimited*, and set the Port to *(b)*.

$\boxed{5}$ Click on the Wait block and change the sensor type from *Touch Sensor* to *Light Sensor*. Adjust the sensitivity of the sensor to detect dark (line) using the observations you made in the previous tutorial.

$\boxed{6}$ Click on the second Move block, set the Duration to *unlimited*, and set the Port to *(b)*.

$\boxed{7}$ Click on the Loop block, set the Control to *Sensor* and set the Sensor to *Light Sensor*. Adjust the sensitivity of the sensor to detect dark (line) using the observations you made in the previous tutorial.

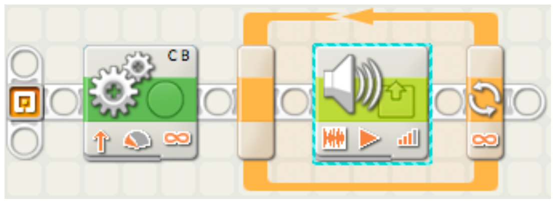
3. Try out this program on the track and see how well it recovers the line when it does lose it.

4. **Questions for Section 2:**
    (a) What mechanism is used to identify when the Tribot has lost the line?
    (b) Observe that the lost line failure mode is assumed to only occur when the Tribot is in the second state. Why will this failure mode not occur while the Tribot is in the first state? (Think about what assumption is being made here.)
    (c) Once the failure mode is identified, how does the Tribot attempt to recover the line?
    (d) Under what conditions could this identification mechanism fail?
    (e) Under what conditions could this recovery mechanism fail?
5. **Things to think about:**
    (a) Under what conditions could the Tribot lose the line while being in the first state? (The Armdale Round-about is a good example of where this could happen.)
    (b) Can you make the identification or recovery mechanism fail? If so, how?
    (c) Can you think of other mechanisms to identify that a line has been lost?
    (d) Can you think of other mechanisms for recovering the line?
    (e) Is it possible to develop a perfect identification and/or recovery mechanism? Why or why not?

3 Getting out of a Jam

Suppose we wanted to write a Roomba-like program that bounces off walls. We could use the touch or the ultrasonic sensor for this, but the Touch sensor can only be mounted on one side of the Tribot, and the ultrasonic sensor does not work well at oblique angles to a wall. What else can we do? We know the Tribot is stuck when it cannot move. Can we detect this behaviour? To a degree, yes: The actuators have built in rotation sensors that work in a similar manner as Timers, except that instead of measuring time (in seconds) they measure the rotation of the chosen motor (in degrees or rotations). Your task will be to create a simple Roomba-like program that propels the Tribot in a single direction. When the Tribot bumps into a wall, it should detect that it is not moving, back-off a short distance, turn right, and then resume its forward motion.

1. Start a new program called "Hoomba" and construct the following program:



   The Move block should be configured to move the Tribot forward at a Power of *25* (not to fast) and the Sound block should be configured to create a sound of your choice.
2. Now extend this program in a similar manner to the way we extended the "FollowPreset" program. You will need to use Rotation Sensor block at the start of the loop to reset the rotation sensor on one of the two drive motors, say (b). Then after the Sound block you will need to use a Switch (Conditional) block, configured to use the rotation sensor

of motor (b) to check whether motor (b) has rotated more than say 10 degrees. If yes, then no action needs to be taken, otherwise you will need to use three Move blocks to back-off the Tribot, turn to the right, and resume the forward progress (at Power 25).

3. Try your new Hoomba out. Does it work? No? Hmmm... How much time passes between when your rotation sensor is reset and when it is checked? Is it enough time to move 10 degrees? How could a wait block be helpful?
4. **Questions for Section 3:**
   (a) What mechanisms are used to identify and recover when the Tribot bumps into a wall?
   (b) Under what conditions may the identification mechanism fail? Hint: try increasing the power to the motors, then place the Tribot against a wall (facing the wall), and run it.
   (c) Under what conditions may the recovery mechanism fail?
5. **Things to think about:**
   (a) How could you make the recovery mechanism more robust so that the chances of the Tribot getting stuck are reduced?
   (b) Is it possible to guarantee that the Tribot will not get stuck?
6. Extend the recovery mechanism of your Tribot (as you described) to reduce its chances of getting stuck.
7. Using a few of the chairs create an obstacle course for your Hoomba. Try it out and see how well the Hoomba negotiates the course.

4 If You have Time

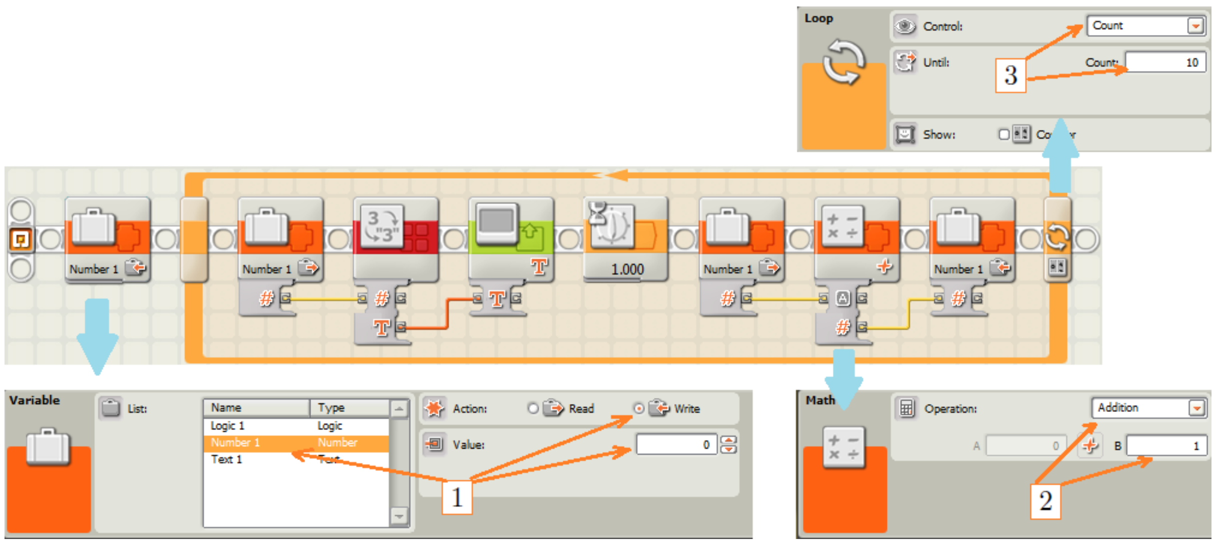
If you have any additional time extend your Hoomba further to use the ultrasonic sensor to detect upcoming objects as well (reduces the number of collisions) and also try using the touch sensor to avoid collisions when the Tribot is backing up. To make your code more manageable use the Robot Educator to learn about how to create your own blocks, called “My Blocks”.

Variables and Threads
Tutorial 6

In this tutorial you will learn about two useful programming features: variables and threads. Variables, pictured as briefcases in the NXT programming language, are used to store pieces of information that are useful to track. When you need to make a decision about what to do next, a variable can help. Threads are used to run multiple paths of instructions at the same time. So, while your robot is being asked to do one thing, say, moving forward and spinning around, a second path of instruction could be monitoring the light sensor, say, to prevent the robot from crossing a boundary.

1 Variables for Tracking Values

1. Start a new program called "IncrementCounter" and construct the following program:



1 Click on the first Variable block, set the Variable Name to *Number 1*, the Action to *Write* and the Value to 0. This initializes the variable to 0. Otherwise, the variable could have any random initial integer value.

2 Click on the Math block, set the operation to *Addition* and type 1 into the 'B' field.

3 Click on the loop and set the Control to *Count* and the Until field to 10,

4 All variable blocks should similarly be set to have the Variable name *Number 1*. The second and third variable blocks will Read the variables (to see what's inside the briefcase), while the first and fourth will write to the variable (to put something in the briefcase).

5 Connect all of the wires as shown. A wire can be connected by placing your mouse over one end of the connection, clicking, moving to the other end of the connection, and clicking again. A solid line (usually coloured) means that the connection has been made, while a dashed line says that the connection was not made. When you move a block, all of its connections are lost.

2. **Questions for Section 1:**

   (a) Run this program. Look at the display. What does it do?

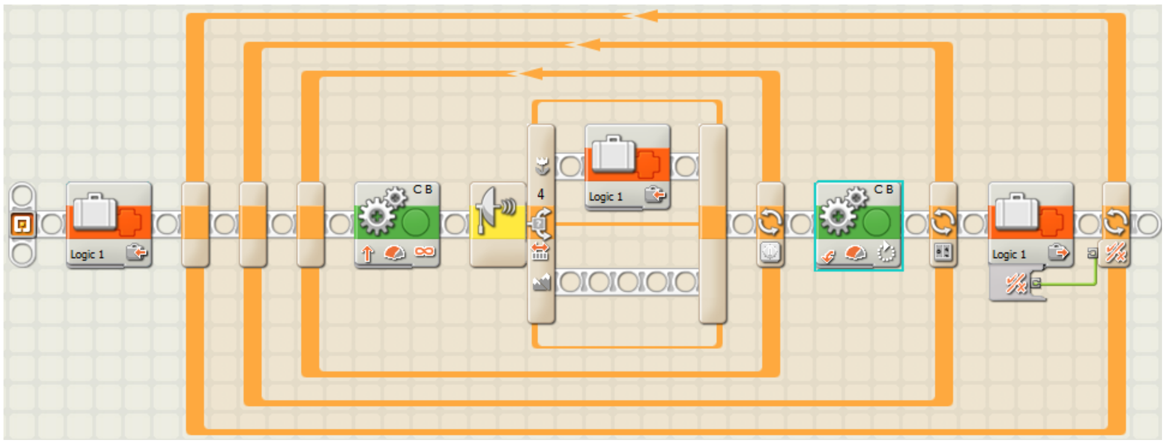
   (b) In two or three sentences, describe how the program accomplishes this task.

   (c) How would you make the output show odd numbers only?

   Incrementing counters like this can help keep track of things like how many turns you have made or how many objects you have seen. Such information can be very helpful when executing a difficult task.

2 Variables for Tracking State

1. Start a new program called "SpyRobot" and construct the following program:



1 The variable initialization block *Writes* False to *Logic 1*. The second variable block *Writes* True to *Logic 1*. The final variable block is a *Read* block.

2 The ultrasonic sensor switch block uses a small distance of 20cm.

3 The first move block moves forward for an *unlimited* period of time. The second move block makes a 90 degree turn to the left.

4 The innermost loop is *timed* for 3 seconds. Moving outward, the next loop repeats for a *Count* of 4. Finally the outermost loop continues until the logical input reads *True*.

2. **Questions for Section 2:**

   (a) Run this program. What does it do?

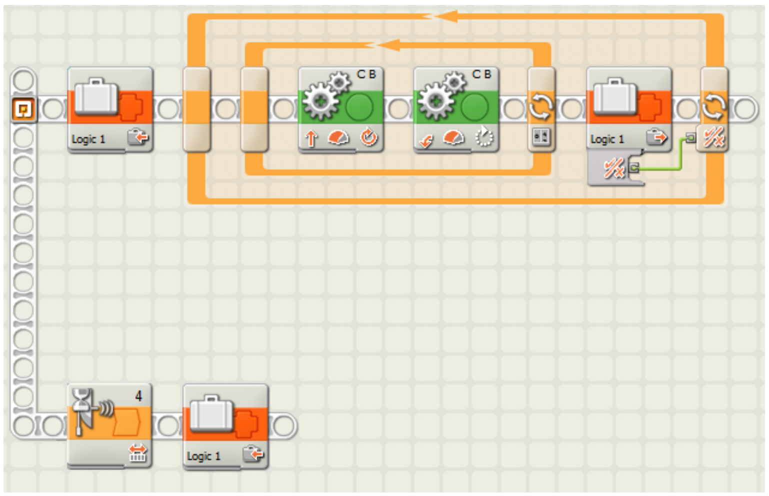
(b) Under what circumstances is the *Logic 1* (boolean) variable set? How and when is this used to stop the robot?

It is sometimes appropriate to monitor sensors, but delay the response. Boolean variables are an easy way to remember that an event occurred in the task.

3 A More Intuitive Spy Robot

Although the spy robot accomplishes its mission rather well, the 3-loop structure and sensor switch approach is not an obvious way to accomplish this. Instead, it would be nice if the robot could separately monitor the ultrasonic sensor for an object and move the robot along the perimeter. Threads do just that. Threads allow you to execute more than one program path at (about) the same time.

1. Start a new program called "SpyRobotThreads" and construct the following program, by first copying and pasting the "SpyRobot" program:



$\boxed{1}$ Notice that the main innermost loop from "SpyRobot" is eliminated. Change the forward move block to have a duration of 3 rotations.
$\boxed{2}$ Create a sensor wait block, setting it to be an ultrasonic sensor that waits until it sees something closer than 20cm. Place the block below the main loop, unattached to a girder.
$\boxed{3}$ Drag a girder down to meet this orphaned block.
$\boxed{4}$ Add a Variable block that *Writes* True to Logic 1, following the ultrasonic sensor wait block.

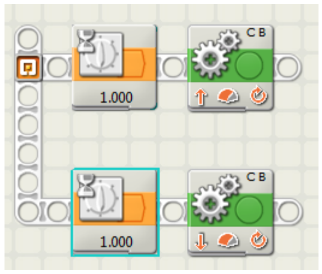
2. **Questions for Section 3:**

(a) Run this program. Does it differ from the previous program?

4 Racing Threads

Threads are useful. However, there is one little problem. What happens when two threads want to control the same motor?

1. Start a new program called "RacingThreads" and construct the following program:



$\boxed{1}$ Create two timer blocks, followed by move blocks, one going forward for one rotation and the other backward for one rotation.
$\boxed{2}$ Initially set the wait times equal (1.000).

2. **Questions for Section 4:**
   (a) Run this program. What happens?
   (b) Change the timer durations so that one timer completes by 0.1s before the other. What happens now?
   (c) Again change the timer durations, but now so that the other timer completes first. Is there a change in the behaviour?

   Using your new understanding of variables and hints from previous programs in this tutorial make the threads take turns. That is, the robot should move forward one rotation and then backward one rotation.
   Although threads will compete for resources there are effective ways of managing this. Threads can ultimately make some tasks possible and other tasks easier to solve.