1 The Programming Environment
-Becoming Familiar with programming

In this tutorial you will familiarize yourself with the Aseba Studio programming environment and build some small programs without using the Thymio II robot. These programs are designed to facilitate experimentation with the robot in the course.

1.1 Exploring the Aseba Studio Programming Environment

1. Log in into the computer.

2. Run the "`Aseba Studio for Thymio II`" from "`Applications`" folder (Mac OS X) or the "`Start`" menu (Windows). You may see the dialog such as the one in Figure 1. In this case click OK. You will then see the connection dialog box such as the one in Figure 2.
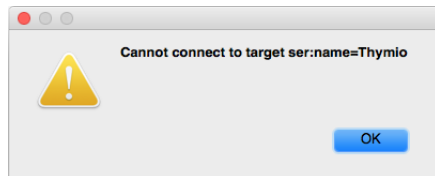

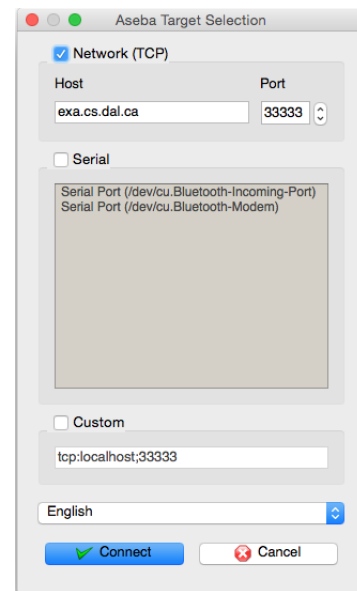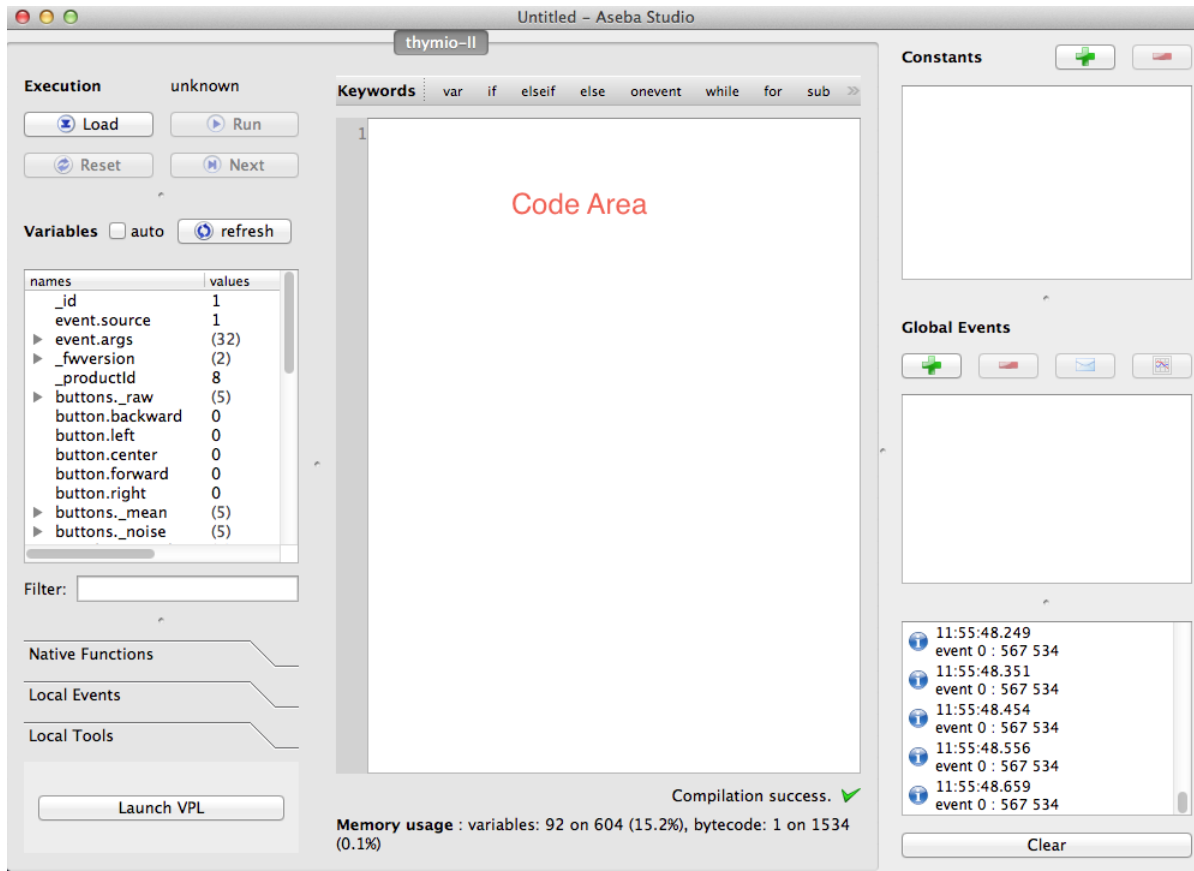Figure 1: The Thymio-II is not connected.

Figure 2: Select **Network (TCP)**.

3. Check the **Network (TCP)** option in the connection dialog box (Figure 2).

4. Enter the host as `exa.cs.dal.ca` and be sure the port is set to *33333*.

5. Click on **Connect**. The software will take a few seconds to load; you will then see the application window.

6. You should become familiar with the layout of this development environment. Click on the link: `https://www.thymio.org/en:asebastudio` to find documentation for Aseba text programming environment and read it carefully.

7. After getting familiar with the development environment, find the links for **Description of the Aseba language** and **Description of the native functions standard library** and read them to get familliar with the the Aseba language syntax and functions.

Untitled – Aseba Studio

thymio–II

Execution   unknown

Load   Run

Reset   Next

Variables □ auto ⟳ refresh

names	values
_id	1
event.source	1
▸ event.args	(32)
▸ _fwversion	(2)
_productId	8
▸ buttons._raw	(5)
button.backward	0
button.left	0
button.center	0
button.forward	0
button.right	0
▸ buttons._mean	(5)
▸ buttons._noise	(5)

Filter:

Native Functions

Local Events

Local Tools

Launch VPL

Keywords var if elseif else onevent while for sub »

1

Code Area

Compilation success. ✔

Memory usage : variables: 92 on 604 (15.2%), bytecode: 1 on 1534 (0.1%)

Constants + −

Global Events + − ✉ ✂

11:55:48.249
event 0 : 567 534
11:55:48.351
event 0 : 567 534
11:55:48.454
event 0 : 567 534
11:55:48.556
event 0 : 567 534
11:55:48.659
event 0 : 567 534

Clear

8. **Questions for Section 1.1:**

   (a) Identify the purpose and location of the following items in the Aseba text programming environment: (i) Code area, (ii) Debugger area, (iii) Data inspection area, (iv) Local events and native functions, (v) math native functions, (vi) leds native functions, (vii) sound native functions, (viii) Constants definition, and (ix) "`auto`" check box.

   (b) What happens when you click on the "`Load`" button? What about the "`Run`" button?

   (c) Why do we need to add comments in a program? How will you add a comment in the Aseba text programming environment?

   (d) Describe the difference between "`Variables`" and "`Constants`" and explain how each of them is declared in the Aseba environment?

   (e) Why do we need to use "`Arrays`" in programming? How do you define an array with name "`a1`" with 7 elements in the Aseba text programming environment?

   (f) What are "`Expressions`" and "`assignments`"? Name the operators which are only available from within "`if`", "`when`", and "`while`" structures and explain why do we need them?

   (g) Why do we need to use "`conditional statements`" in programming? What is the difference between the conditional statements in the Aseba text programming and how do you use them?

(h) Why do we need to use "`loops`" in programming? What is the difference between the the "`while`" and "`for`" loops in the Aseba text programming and how do you use them?

1.2 Simple Programs

We are now ready to write our first program. In this case, we will do something simple without using the robot.

1. Start a new program.

2. Create two constants: M with a value of 40, and N with a value of 10.

3. Define two variables a and b.

4. Build a simple program to multiply M by N and save the result into variable a and also divide M by N and save the result into variable b.

5. Click on the "`Load`" button in the top left corner and then run the program by clicking on the "`Run`" button.

6. Check the "`auto`" box above the Variable List, which causes an automatic update of the variables.

7. Scroll down the Variable List to find the results of the program (a and b).

8. **Questions for Section 1.2:**

   (a) Change the M and N constants into 500 and 90 and check the results. Then use a calculator and check the results again. Why there are differences?

   (b) What is the range of the values that you can set as a constant or as a variable? Briefly justify your answer.

1.3 Fahrenheit to Celsius Convertor Program

Your next task is to learn more about the expressions and assignments. You will build a program to convert Fahrenheit to Celsius using the Aseba programming language.

How to convert Fahrenheit to Celsius: The temperature in degrees Celsius (C) is equal to the temperature in degrees Fahrenheit (F) minus 32, times 5/9: $C = (F - 32) \times 5/9$

1. Start a new program.

2. Create a constants F for Fahrenheit input that you want to convert it into Celsius. Set to a value like 350.

3. Define a variables c to save the converted value in Celsius.

4. Build a simple program to convert Fahrenheit to Celsius using the formula.

5. Save your program under the name "`F2C_Convertor.aesl`" by clicking on the "`Save`" option of the "`File`" menu.

6. Click on the "Load" button and then run the program by clicking on the "Run" button.

7. Scroll down the Variable List to find the result of the program (c).

8. Try different numbers for the constant F and check the results.

9. **Questions for Section 1.3:**

   (a) Why the conversion is not accurate ($100\ °F$ is equal to $37.78\ °C$)?

1.4   Inches to Centimeters Convertor Program

Your last task is to learn about the loops, arrays, and native functions. You should build a program to convert Inches to centimeters using the Aseba programming language ($1\ inch = 2.54\ cm$).

1. Start a new program.

2. Define an array named "inches" with values of 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000.

3. Build a simple program using the "while" or "for" loops to multiply the "inches" array by 2.54 and store it into the "centimeters" array. Hint: As Aseba does not support decimal numbers (see the first question of the secound section), you should first perform a multiplication by 254 and then a division by 100 (254/100=2.54).

4. Save your program under the name "UnitConvertor.aesl".

5. Click on the "Load" button and then run the program by clicking on the "Run" button.

6. Scroll down the Variable List and expand the "centimeters". A 10-element array, *centimeters*, is displayed. The elements store the values calculated by your program.

7. **Questions for Section 1.4:**

   (a) Why there is the problem when you use simple multiplication and division?

   (b) Why do you need to use a native function "math.muldiv" to compute multiplication-division? Hint: Rebuild the program using the "math.muldiv" function and check the results again.

   (c) What is the maximum value for "inches" array that your program can convert into centimeter? Briefly justify your answer.

1.5   Answer these questions individually.

1. Write a program to convert Celsius to Fahrenheit.

2. Write a program to convert Centimeters to Inches.

Ask your TA to check if your programs work.

2   The Programming Environment and the Thymio II Robot
### -Becoming Familiar with the System

In this tutorial you will familiarize yourself with the Aseba Studio programming environment and build some small programs for the Thymio II robot. These programs are designed to facilitate experimentation with and calibration of the sensors and actuators used by the robot. 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.

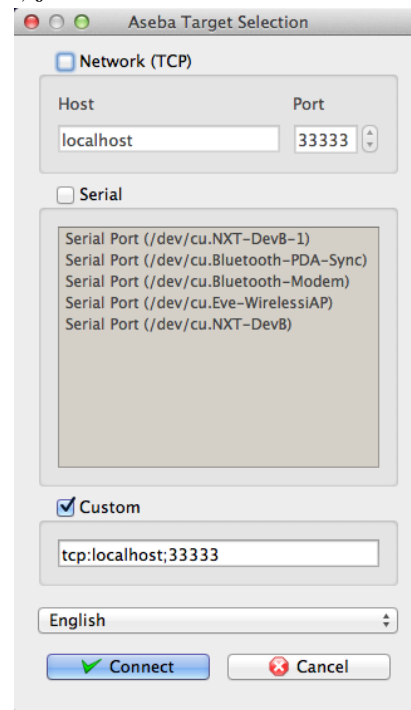### 2.1   Aseba Studio Programming Environment

1. Log in into the computer.

2. Plug in the Thymio II robot and turn it on by holding down the center button until the robot turns on. If a dialog, such as the one below, appears, just click "Cancel".
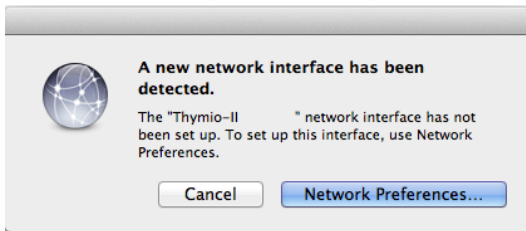
(a) Just click "Cancel".
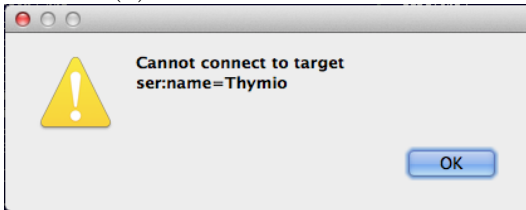
(b) Just click "OK".

(c) Just click "Cancel".

3. Run the "Aseba Studio for Thymio II" from "Applications" folder (Mac OS X) or the "Start" menu (Windows). If the robot was not plugged in, you will get warning about not being able to connect to the robot. In this case, click on "OK". A second dialog will appear. Click on "Cancel" to close the application, plug in the robot and reload the application. The software will take a few seconds to load; you will then see the application window.
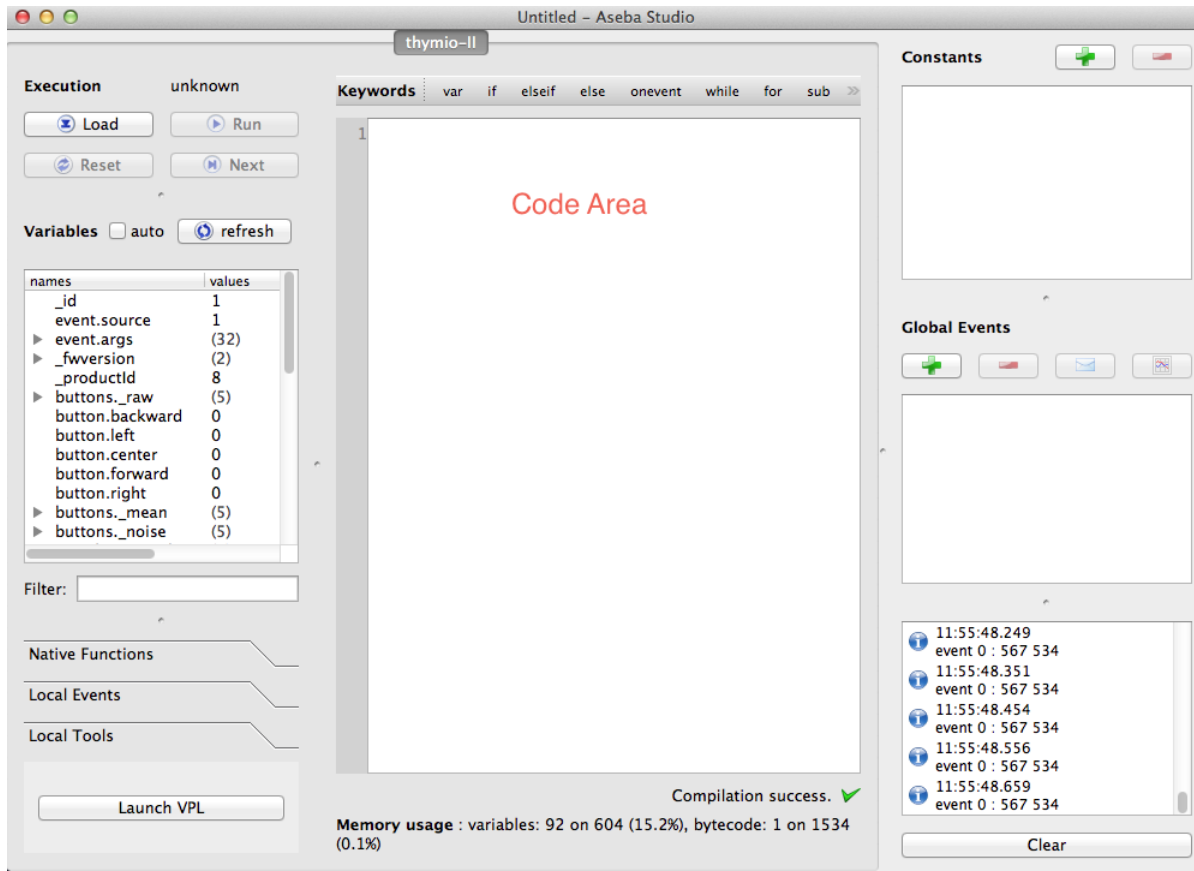
4. You should become familiar with the layout of this development environment. The center area contains the Code Area, where you will add code to control your robot. The left list-box lists all the variables: both system and programmer-defined—variable names and variable values are displayed in this box. By default the values are not automatically updated. However, in most instances automatic update is preferred.

5. Check the "auto" box above the Variable List, which causes automatic update of the variables.

6. Scroll down the Variable List and expand the "prox.horizontal". A 7-element array, *prox.horizontal*, is displayed. The elements store the values measured by the proximity sensors on the front (5) and back (2) of the robot. Place your hand in front of each of the proximity (**prox**) sensors and observe how the values change.

7. **Questions for Section 2.1:**

   (a) By placing your finger in front of each of the seven proximity sensors, identify the corresponding number of the sensor. I.e., the sensor numbers should range between 0 and 6. Draw a small diagram indicating the sensor numbers.

   (b) What value indicates that something is in close proximity to the sensor? What value indicates that there is nothing in the proximity of the sensor?

   (c) Are there other proximity sensors on the robot? If so, where are they?

Additional documentation can be found at: https://www.thymio.org/en:asebastudio

2.2 Moving in a Square

We are now ready to write our first program. In this case, we will do something simple like moving the robot in a square (approximately). A program comprises three parts:

Variable Declarations comprising a list of programmer defined variables, one per line. Each declaration begins with a *var* keyword followed by the variable name and an optional initializer, e.g., `var answer = 42`. There are no programmer defined variables in this example.

Initialization Code comprises the part of a program that is executed when the program starts running. This code initializes variables to initial values and prepares the program for execution. In the example below, the first three lines represent initialization code.

Event Handlers and Subroutines comprise code for all the event handlers and programmer-defined subroutines. Each event handler begins with the keyword *onevent* and each subroutine begins with the keyword *sub*. In this example, there are three event handlers.

As a general rule, variable declarations must come first, followed by initialization code, and lastly, event handlers and subroutines.

1. Start a new program and enter the program in Figure 3 into the Code Area.

```
motor.left.target = 0                    # reset motors amd timers
motor.right.target = 0
timer.period[0] = 0

onevent button.forward                   # on forward button start
  motor.left.target = SPEED              # motor and timer
  motor.right.target = SPEED
  timer.period[0] = FWD_PERIOD

onevent button.backward                  # on reverse button turn
  motor.left.target = 0                  # off motors and timer
  motor.right.target = 0
  timer.period[0] = 0

onevent timer0                           # on timer event switch
  motor.left.target = -motor.left.target # between turn and fwd
  if motor.left.target < 0 then          # if we are turning
    timer.period[0] = TURN_PERIOD        # set turn time to 1sec
  else                                   # else
    timer.period[0] = FWD_PERIOD         # set turn time to 2 secs
  end
```

Figure 3: The Square Program.

The **#** marks identify *comments*. A comment is ignored by the system, but is used by the programmer to document what her program is doing in a high-level easy to understand way. For example, the first statements of the program reset the motors and timer of the robot, the next piece of code specifies what should happen when the **Forward Button** is pressed (the motors are started as well as the timer), the third piece of code specifies what happens when the Back button is pressed, and the last part specifies what happens when the timer goes off.

This program works in the following manner:

(a) To move in a square, we need to make the robot move forward for a short period of time, say two seconds, and then make a 90 degree turn, then move forward another 2 seconds, make another 90 degree turn, and so on.

(b) The first three lines are executed when the program starts running. These lines reset the motors and the timer to 0. The **motor** devices are controlled by setting the *motor.left.target* and *motor.right.target* variables. The maximum value (speed) for the motors is 500 (forward) and −500 (reverse). To turn motors off, the variables are set to 0. To go straight, run both the left and right motors at the same speed. To turn the robot, run one of the motors forward and the other in reverse.

The robot has two timers, **timer0** and **timer1**. To use the timers, set the variable *timer.period[0]* and *timer.period[1]* to a period (in milliseconds) between 0 and 32767. If the period is nonzero, the timers will generate events `timer0` and `timer1` every so many milliseconds, as specified by the period. for example, if *timer.period[0]* is set to 1000., then **timer0** will generate a `timer0` event every second (1000 milliseconds).

By setting the motor and timer variables to 0, we ensure that the robot starts operation in a known state. The remainder of the program comprises three event handlers, which run when a corresponding event is generated.

(c) The `button.forward` event handler is executed whenever the **Forward Button** is pressed. This event handler starts the robot's motion. It runs both **motors** forward at the same speed by setting *motor.left.target* and *motor.right.target* to the constant *SPEED*. Lastly, the handler sets the period of **timer0** to 2 seconds, by setting the variable *timer.period[0]* to the constant *FWD_PERIOD*.

(d) The `button.reverse` event handler is executed whenever the **Reverse Button** is pressed. This event handler stops the robot's motion. It turns off both **motors** and **timer0** by setting *motor.left.target*, *motor.right.target*, and *timer.period[0]* to 0.

(e) The `timer0` event handler is executed whenever the period of the timer expires. The handler toggles the motion of the from forward to turn and vice versa by reversing the direction of one of the **motors**, and then toggling the period of **timer0** between *FWD_PERIOD* and *TURN_PERIOD*.

The program determines which period to use by comparing the speed of the afore mentioned **motor**. If the **motor** is rotating forward, then the robot is moving forward and the corresponding period is used. If the **motor** is rotating in reverse, then the robot is turning and the corresponding period is used.

2. Observe the message immediately below the Code Area, depicted in Figure 4. This indicates
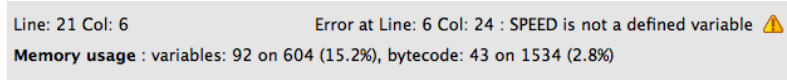
Figure 4: There is an error in your program.

that there is a problem with your program. Whenever you add code, the system will automatically check whether your program is correct and will notify you if it finds any errors by displaying the first error it discovers. In this case, the error message is:

```
Error at Line: 6  Col: 24 : SPEED is not a defined variable
```

Take a look at that line. Notice that we try to assign *SPEED* to *motor.left.target*, but we have never defined *SPEED*. In this case, we want to define a *constant* called *SPEED*. A *constant* is a named value that does not change, i.e., it's a value that remains constant.

3. Click on the $\boxed{+}$ button in top right corner. A dialog will appear to enter the name and value of the constant. Enter *SPEED* as the name and 200 as the value, as shown below. Then,
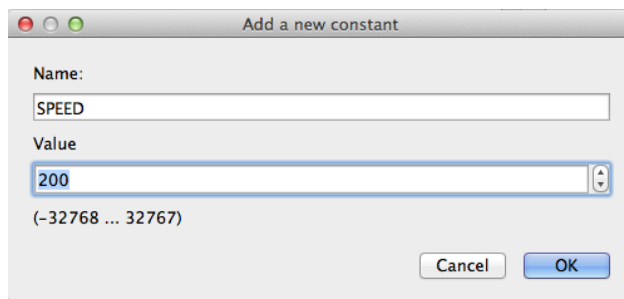

Figure 5: Declare a constant be entering the name and value.

click "OK". Notice that the error message changes, because the first error has been fixed.

4. Fix the remaining two errors by adding the *FWD_PERIOD* and *TURN_PERIOD* constants. The *FWD_PERIOD* constant should be 2000, i.e., we want the robot to move forward for 2 seconds (2000 milliseconds). The *TURN_PERIOD* constant should be set to 1000, i.e., the right angle turn takes approximately 1 second (1000 milliseconds) to perform.

5. Notice that your program should now compile correctly, i.e., it has no obvious errors. At the bottom, you should see the message like in Figure 6. We now load and run the program!
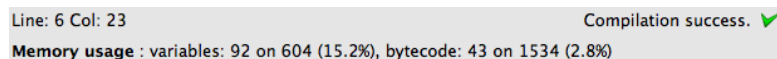

Figure 6: Your program is syntactically correct.

6. Save your program, by clicking on the "Save" option of the "File" menu. As in the previous module, it is important to save your work on a regular basis. Call your program "Square"

and save it to your shared drive. The resulting file will be called "`Square.aesl`". Be sure each group member has a copy of it as well by saving it on their USB key. This will ensure that everyone has copy of their work.

7. Load the program on to the robot by clicking on the "`Load`" button in the top left corner.

8. Run the program by clicking on the "`Run`" button. Note: the robot will not start moving because it is waiting for the **Forward Button** to be pressed.

9. Unplug the USB cable from the robot. Even though the USB cable is long and light, it will still affect the robot's motion. Hence, in most cases, it is recommended that you unplug the robot before hitting "Go". A dialog box will appear stating that the connection has been lost and that it is attempting to require the connection (Figure 7). **Do NOT press "Cancel".** If you do, you will need to exit the application and restart it in order to reconnect the robot.

Figure 7: **Do NOT press "`Cancel`".**

Once the robot is reconnected, everything will return to as it was prior to the disconnect. However, you will need to enable auto-update of the variables by clicking on the "`auto`" check-box. This is an annoying feature of the program.

10. You can now run the robot in a square. Find an empty surface, place the robot on it and press the **Forward Button**. The robot should start moving in a square. Note, that its turns may be slightly off, i.e., as time progresses, the square starts shifting.

11. Press the **Back button** to stop the robot. You can make it move again by pressing the **Forward Button** once more.

12. Stop the robot and reconnect it to the USB cable.

13. **Ask** your TA to check if your program work.

14. **Questions for Section 2.2:**

   (a) Suppose we changed the *SPEED* constant to a different value, say 400. How would this affect the behaviour of the robot?

   (b) Given the change above, would we need to change any other constants so that the robot continues to move in a square.

   (c) As you may notice, a 1 second turn, at a speed of 200, does not create a perfect 90 degree turn. Try adjusting the *TURN_PERIOD* constant to get as close as possible to the optimum value for a 90 degree turn. What is this value?

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

Congratulations! You completed your first robot programming task.

2.3 The Ground Proximity Sensors: Staying within the Lines

Your next task is to learn about the ground proximity sensors that the robot has and how to use them to build a program that stays within an area demarcated by a black line.

1. Start a new program.

2. Create two constants: *SPEED* with a value of 200, and *EDGE* with a value of 400.

3. Enter the program in Figure 8 in to the Code Area.

```
motor.left.target = 0                        # reset motors (turn them off)
motor.right.target = 0

onevent button.forward                       # when forward button is pressed
  motor.left.target = SPEED                  # start motors forward
  motor.right.target = SPEED

onevent button.backward                      # when backward button is pressed
  motor.left.target = 0                      # stop the motors.
  motor.right.target = 0

onevent prox
  if motor.left.target == 0 then             # if not moving no need to proceed
    return
  end

  # if we are moving forward
  if motor.right.target > 0 and motor.left.target > 0 then
    if prox.ground.delta[0] < EDGE then         #   if line is on the left
      motor.right.target = -SPEED               #      turn right
    elseif prox.ground.delta[1] < EDGE then     #   else if line on right
      motor.left.target = -SPEED                #      turn left
    end
  # else we are not moving forward, if no line
  elseif prox.ground.delta[0] > EDGE and prox.ground.delta[1] > EDGE then
    motor.left.target = SPEED                     #   move forward
    motor.right.target = SPEED
  end
```

Figure 8: Boundary Avoidance Program.

This program works in the following manner:

(a) To avoid the edges, we need to turn away from an edge whenever the robot detects that it is near one. The robot uses its two proximity ground sensors to detect the edge, which is a thick black line. The sensors work by detecting a change in brightness of the light from the surface at the front of the robot.

(b) The first three parts of the program are nearly identical to those of the preceding program. The first two lines reset the motors to 0 (no motion). The `button.forward` and `button.backward` event handlers start and stop the robot's motors. The only difference is that the timer is not used in this program and hence its period does not need to be initialized.

(c) The program has a `prox` event handler that is executed every tenth of a second (10 times per second). The event handler first checks whether the robot is moving; otherwise there is no need to do anything. The handler then checks whether the robot is moving forward or in reverse.

(d) If the robot is moving forward, the handler checks if either of the two proximity ground sensors sense a dark line. That is, if $prox.ground.delta[0]$ or $prox.ground.delta[1]$ are below $EDGE$ it means that sufficiently little brightness is being sensed, which indicates that one of the sensors is over a black line.

When one of the proximity ground sensors encounters a dark line, the program determines which way to turn. If only the right sensor encounters the line, the sensor robot turns left. Otherwise, the robot turns right.

(e) Otherwise, if the robot was not moving forward, the handler checks whether neither of the two sensors sense a black line. If this is the case, both motors are set for positive speed to move the robot in the forward direction.

4. Save the program under the name "`BoundaryAvoider`".

5. Load the program on your robot.

6. Create a small arena using black electrical tape provided by the lab facilitator.

7. Run the program and unplug the robot from the USB cable.

8. Place the robot into the arena and press the **Forward Button**.

9. Observe the robot's behaviour. The robot should remain within the boundary of the arena,

10. Stop the robot and reattach the USB cable.

11. **Ask** your TA to check if your program work.

12. **Questions for Section 2.3:**

(a) Does the speed at which the robot travels matter? Try increasing the speed to maximum (500) and see if it makes a difference.

(b) Suppose instead of the line being black, it was a lighter colour. What would you need to adjust? Why?

2.4 The Horizontal Proximity Sensors: Staying within the Walls

Your last task is to learn about the horizontal proximity sensors that the robot has and how to use them to build a program that avoids objects (walls) in front of the robot. You will also use programmer-defined variables and a math function provided by the system.

1. Start a new program.

2. Create two constants: *SPEED* with a value of 200, and *THRESHOLD* with a value of 0.

3. Enter the program in Figure 9 in to the Code Area.

   This program works in the following manner:

   (a) To avoid objects, we need to turn away from an object whenever the robot detects that it is near one. To detect nearby objects the robot can use its **horizontal proximity sensors**. The robot has seven (7) horizontal proximity sensors: five in the front and two in the rear. You can observe them in action by placing your hand or an object in front of a sensor—the closer the object is the brighter the associated LED lights up.

   The values measured by the sensors are stored in an array *prox.horizontal[0...6]*. The first five elements $[0 \ldots 4]$ store values from the front sensors (left to right) and the last two elements store the values from the rear sensors (left and right). The greater the value, the closer an object is to the sensor.

   (b) The first part of the program declares three programmer-defined variables. These are used to store statistics for the values measured by the horizontal proximity sensors.

   (c) The next three parts of the program are identical to those of the preceding program. The first two lines reset the motors to 0 (no motion). The `button.forward` and `button.backward` event handlers start and stop the robot's motors.

   (d) The program has a `prox` event handler that is executed every tenth of a second (10 times per second). The event handler first checks whether the robot is moving; otherwise there is no need to do anything.

   (e) If the robot is moving, the `prox` event handler computes the maximum value reported by the five horizontal proximity sensors (*prox.horizontal[0 : 4]*). This is accomplished by calling the built in `math.stat()` function. The result is stored in variable *max*.

   (f) The first *when ... do* statement checks whether there is an object in front of the robot by comparing the value in *max* to the *THRESHOLD*. If the value is greater than *THRESHOLD*, then there is an object in front of the robot.

   (g) When an object is encountered, the program determines if it is on the left side or right side of the robot, by comparing the values of the leftmost and rightmost sensors (the larger value) indicates a closer object). If the object is on the left side, the robot starts turning right, otherwise it starts turning left.

   (h) The second *when ... do* statement returns the robot to its forward motion. When the value in *max* is at or below *THRESHOLD*, this indicates that none of the horizontal proximity sensors are reporting an object in front of it. In this case, both motors should rotate forward.

4. Save the program under the name "`ObjectAvoider`".

```
var min                                # min over all sensor readings
var max                                # max over all sensor readings
var mean                               # average over all sensor readings

motor.left.target = 0                  # reset motors
motor.right.target = 0

onevent button.forward                 # when forward button is pressed
  motor.left.target = SPEED            # start moving forward
  motor.right.target = SPEED

onevent button.backward                # when backward button is pressed
  motor.left.target = 0                # stop motors
  motor.right.target = 0

onevent prox                           # check proximity sensors
  if motor.right.target == 0 then      # only if we are moving
    return
  end

  # compute min, max, and mean over the current sensor readings
  call math.stat( prox.horizontal[0:4], min, max, mean )

  if max > THRESHOLD then              # if a sensor is above threshold
    # if we are moving forward
    if motor.right.target > 0 and motor.left.target > 0 then
      # if object is closer on the left
      if prox.horizontal[0] > prox.horizontal[4] then
        motor.right.target = -SPEED       #   turn right
      else                                # else
        motor.left.target = -SPEED        #   turn left
      end
    end
  # else all sensors are below threshold, if we are also turning
  elseif motor.right.target < 0 or motor.left.target < 0 then
    motor.left.target = SPEED          # move forward
    motor.right.target = SPEED
  end
```

Figure 9: Object Avoidance Program.

5. Load the program on your robot.

6. Create a small enclosure using Duplo based walls provided by the lab facilitator (3×2).

7. Run the program and unplug the robot from the USB cable.

8. Place the robot into the enclosure and press the **Forward Button**.

9. Observe the robot's behaviour. The robot should avoid bumping into any of the walls.

10. Stop the robot and reattach the USB cable.

11. **Ask** your TA to check if your program work.

12. **Questions for Section 2.4:**

    (a) What would happen if we had used the *when ... do* statement instead of the *if ... then* statement? Would it make a difference? Why?

    (b) Does the speed at which the robot travels matter? Try increasing the speed to maximum (500) and see if it makes a difference.

    (c) Suppose you wanted to make the robot move closer to an object before turning away. What would you need to adjust? Why?

    (d) Suppose you were to use this program to run your Roomba, a robot that sweeps your floor while you are out. Would this program work well? Why or why not?

3 Robotics Simulation
-Using a Robot Simulator

In this tutorial you will learn how to use a robotics simulator. Simulation is an important aspect of robotics because the robots themselves tend to be expensive, bulky, and not always available when you are programming them. In fact, Aseba Studio requires that you have a Thymio-II robot plugged in in order to work.

Thymio on Remote (`Thor`) is a *simulator* that we will use to in lieu of a physical Thymio-II robot. `Thor` simulates a physical robot, but does not provide a way to manipulate it. You can pick up, move, and press buttons on the physical robot, but not on a simulated robot, which exists only as software. A program called a visualizer is used to manipulate the simulated robot.

A *visualizer* called `Hammer` allows the user to manipulate the simulated Thymio-II robot. `Hammer` is a Java GUI application, which allows you to interact with the simulated robot, manipulate it, press its buttons, tap it, and create an environment within which it operates. Without `Hammer` the simulated robot behaves as if it was in a sensory deprivation tank.

To use the simulator and visualizer you need to perform several steps:

1. Decide if you want to run the `Thor` simulator locally or remotely. If running `Thor` locally:

   (a) Download and install VirtualBox. (You only need to do this once.)

   (b) Download and install the `Thor` guest image. (You only need to do this once.)

   (c) Run the `Thor` guest image.

2. Download the `Hammer` visualizer. (You only need to do this once.)

3. Run the `Hammer` visualizer.

4. Connect Aseba Studio to `Thor`.

5. Connect the `Hammer` visualizer to `Thor`.

Running `Thor` locally is more efficient and avoids slowdown when the server is heavily loaded. However, you do have to spend a few minutes installing the `Thor` server locally. Using the remote server avoids the need to install it locally, but imay result in your simulation slowing down when the server is heavily loaded. If you wish to install `Thor` locally, on your own computer, please see the Appendix A to this tutorial.

Note: `Thor` should already be installed locally on the computers in the lab.

3.1 Getting Started

To use the `Hammer` visualizer:

1. Download the `Hammer` visualizer from the course website.

2. Drag the `Hammer.jar` file to your desktop.

3. Run `Hammer` by double clicking on the `Hammer.jar` file. If you are on a Mac and are not permitted to do this, then:

   (a) Run the `Terminal` program in the `Utilities` folder in the `Applications` folder.

(b) Enter the command:

```
java -jar Desktop/Hammer.jar
```

This should run the `Hammer`. You will see the user interface as depicted in Figure 10.
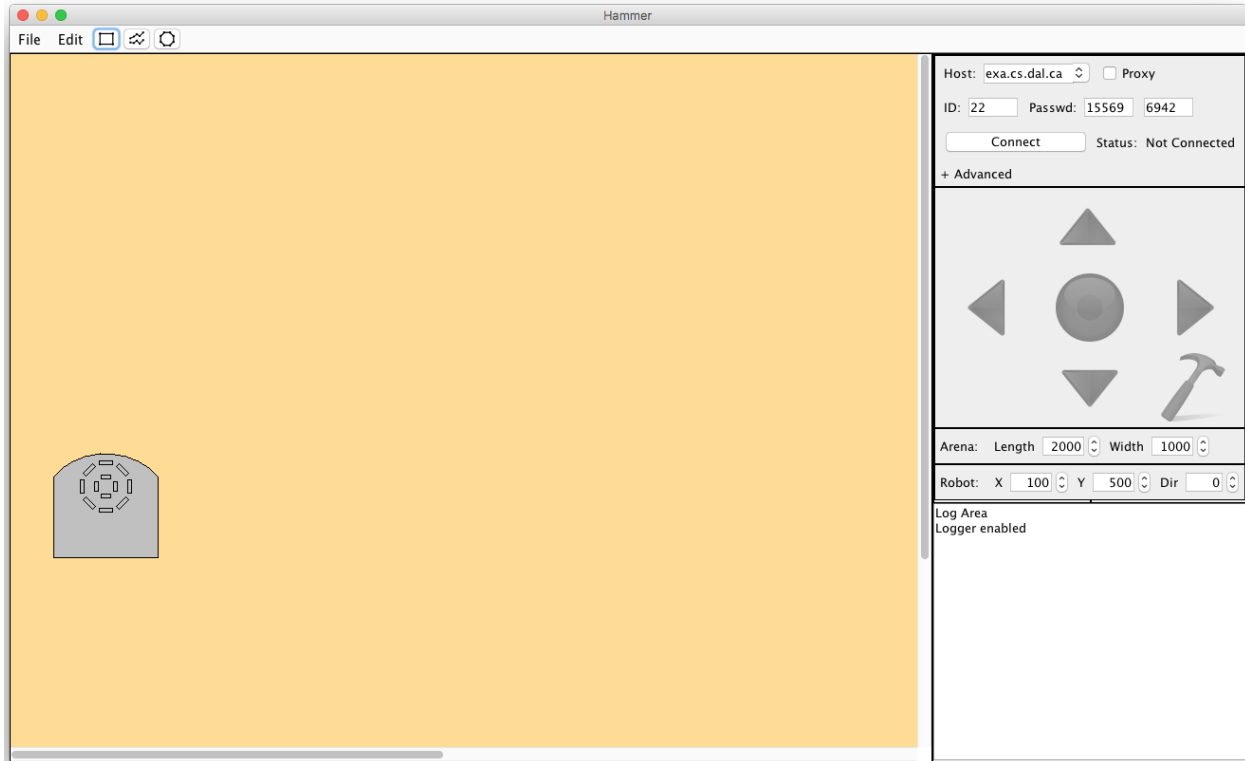

Figure 10: The visualizer's user interface.

3.1.1 Select the Appropriate Connection

The Connection Pane is located in the top right corner of the UI. The first field is the location of the Thor server: The default is set to `exa.cs.dal.ca`.

1. Check the *Proxy* check box if you are planning to use this (remote) server. There should be no need to change the *Host* field.

2. Otherwise, if you planning to use a local server, set the *Host* field to `localhost` by using the drop-down list-box.

Note: Using the proxy mode for remote servers, obviates the need for you set set the ID and password for the session, because `Hammer` will do this for you. In local mode, the default ID, which is 1, typically suffices, and the password is not used. (See Appendix C if you wish to set the ID and/or password manually.)

3.1.2 Run Thor and Aseba Studio

1. If you decide to use a local `Thor` Server

   (a) Run Virtual Box. You should see the main panel with *ThorV?* in the left panel. If not, see the Appendix A of this tutorial.

   (b) Double click on *ThorV?* in the left panel, or select it, and then click on *Start* in the top menu. You will see a window come up and text scroll by (see Figure 12). Once you see the message "`Starting Thor server`" the server is running.

2. Run Aseba Studio or Studio for Thymio II. You may see the dialog such as the one in Figure 11. In this case click OK. You will then see the connection dialog box such as the one in Figure 13

Figure 11: The Thymio-II is not connected.

Figure 12: The *Thor* server is ready to go.

Figure 13: Select *Custom*.

3. Check the **Custom** option of the connection dialog box (Figure 13). Specify the connection:

   `tcp:localhost;33333`

   which is typically the default.

4. Click on **Connect**.

The Aseba Studio IDE should appear and you can load, code, save, and run programs as if connected to a physical Thymio-II. To visulaize the robot, `Hammer` must connect to the simulation.

3.1.3   Connect to the Simulation

1. Switch back to `Hammer`. You can now set up and modify the arena. (See Appendix B for a description of how to edit the Arena.)

2. Click on the *Connect* button in the Connection Panel to connect to the simulation.

The success or failure of the connection will be displayed in the Log Pane in the bottom right corner of the interface, and in the connection status field to the right of the *Connect/Disconnect* button. Once you are connected, all visual Thymio actions can be viewed in the Arena (the yellow panel). A description of the user interface can be found in the Appendix B of this tutorial.

3.2   Moving in a Square, Again

We are now ready to write our first program for the simulator. In this case, we will do something simple like moving the robot in a square (approximately).

`Hammer`'s user interface is divided into three parts: the arena, the main menu, and the control panels. The arena, containing the large yellow area, is where the majority of the activity takes place. The arena contains the Thymio and the simulated environment, which is initially a flat surface $1000mm \times 2000mm$ in size (approximately the size of one of the tables in the lab). The size of the arena can be changed by using the *Arena Pane* on the right. See Appendix B for a fuller discussion.

1. Start a new program and enter the program in Figure 14 into the Code Area.

2. Add the three constants: *SPEED* should be set to 200; *FWD_PERIOD* should be set to 2000, and *TURN_PERIOD* should be set to 1000 for now.

3. Review how this program works by looking at Section 2.2 in the previous tutorial.

4. Save your program as "`SquareSim`". Note, if you saved your "`Square`" program from the previous tutorial, you can use that.

5. Load and run the program. Note: the robot will not start moving because it is waiting for the **Forward Button** to be pressed.

6. You can now run the robot in a the simulator. Switch focus to `Hammer` visualizer. Ensure that `Hammer` is connect to your simulation. The status line in the *Connection Pane* in the top right corner of the UI should indicate "Connected". If not, go back to the previous section and follow directions to connect.

7. Click on the **up** green arrow in the *Interaction Pane* on the the right. This corresponds to the **Forward** button on the Thymio.

   The robot should start moving forward, and after two seconds turn left. It will then move forward again and stop when it hits the edge of the table. Notice that when ever the robot hits the edge of the table it displays a sad face, indicating that it would have fallen off the table had this been the real world.

```
motor.left.target = 0                    # reset motors amd timers
motor.right.target = 0
timer.period[0] = 0

onevent button.forward                   # on forward button start
  motor.left.target = SPEED              # motor and timer
  motor.right.target = SPEED
  timer.period[0] = FWD_PERIOD

onevent button.backward                  # on reverse button turn
  motor.left.target = 0                  # off motors and timer
  motor.right.target = 0
  timer.period[0] = 0

onevent timer0                                 # on timer event switch
  motor.left.target = -motor.left.target       # between turn and fwd
  if motor.left.target < 0 then                # if we are turning
    timer.period[0] = TURN_PERIOD              # set turn time to 1sec
  else                                         # else
    timer.period[0] = FWD_PERIOD               # set turn time to 2 secs
  end
```

Figure 14: The Square Program .

8. Click on the **down** green arrow in the *Interaction Pane*. This corresponds to the **Back** button on the Thymio. The robot should stop moving.

9. Reposition the Thymio in the center of the arena by clicking on and dragging the robot to the center of the arena. Note: the robot's coordinates are shown in the *Robot Pane* on the right, as well as it's current direction.

10. Change the direction of the Thymio by dragging the red handle attached to the robot. Make it point (approximately) upwards so that the direction is near 0. As you drag the red handle you should see the *Dir* field in the *Robot Robot Pane* change.

11. Use the small up-down buttons on the *Dir* field to set the direction to 0. At thus point, the Thymio should be at the center of the screen and pointing straight up.

12. Again, click on the **Forward Button** in the *Interaction Pane* on the the right. The Thymio should start moving in a square without "falling off the table." Notice that it's turns are a little greater than 90 degrees. I.e. it is turning for too long.

   We can adjust the turn by adjust the time spent turning.

13. Stop the robot, using the **Back** button, and reset it's direction to be up.

14. Switch to Aseba Studio and change *TURN_PERIOD* to 900.

15. Load and Run the program.

16. Switch back to `Hammer` and click on the **Forward** button to start the Thymio moving again.

17. You can keep track how many degree the Thymio is turning by looking at the *Dir* field in the *Robot Pane*. Notice that the robot is now turning too little.

18. Find the optimal value for *TURN_PERIOD* so that the robot turns as close to 90 degrees as possible. Note: you will need to go back and forth between Aseba Studio and `Hammer` as you try different values. This is known as finding values experimentally.

19. **Ask** your TA to check if your program work.

20. **Questions for Section 3.2:**

    (a) How would you change the size of the arena?

    (b) What is the optimal setting for *TURN_PERIOD* so that the robot turns as close to 90 degrees as possible?

    (c) What happens as the robot continues to run for longer? I.e., what happens after the robot performs 10 squares?

    (d) Suppose we changed the *SPEED* constant to a different value, say 400. How would this affect the behaviour of the robot? Try it!

3.3 The Ground Proximity Sensors, Once More

Your next task is to investigate the ground proximity sensors in the simulator. You will use a program that stays within an area demarcated by a black line. One of the things you will need to do is add a black line to the simulator's arena (the yellow area). In order to do this, the visualizer must be in *Edit* mode.

`Hammer` has two modes: *Edit* (Not Connected) mode and *Simulation* (Connected) mode. In *Edit* mode the user can manipulate the Thymio's environment, but cannot see the Thymio move or interact with the Thymio, because the visualizer is not connected to `Thor`. In *Simulation* mode, the user can watch to Thymio move and interact with the Thymio, but not manipulate the Thymio's environment. A description of both modes can be found in the Appendix B of this tutorial.

1. Click on the *Disconnect* button in the *Connection Pane*. `Hammer` should now be in *Edit* mode.

2. Create a large black tape ring in the arena, similar to the one in Figure 15. To place some black tape on the arena, click on the button beside the *File* menu that looks like two wavy lines. You can then draw a path on the arena by placing vertices on the arena surface, which will be connected by black line segments. Double-click to place the last vertex. The width of these black lines is comparable to the width of electrical tape used in the last lab.

    Try drawing some different paths. You can delete a path by selecting it and then hitting the *delete* key, or the *Delete* action in the *Edit* menu.

    Notice that when you are creating the path or have the path selected, the *Object Pane* on the right shows the coordinates orientation, and colour of the object. You can rotate and manipulate the path in the same way you manipulated the Thymio. You drag the path, rotate it, and drag the vertices around. Feel free to experiment and play around with the paths.
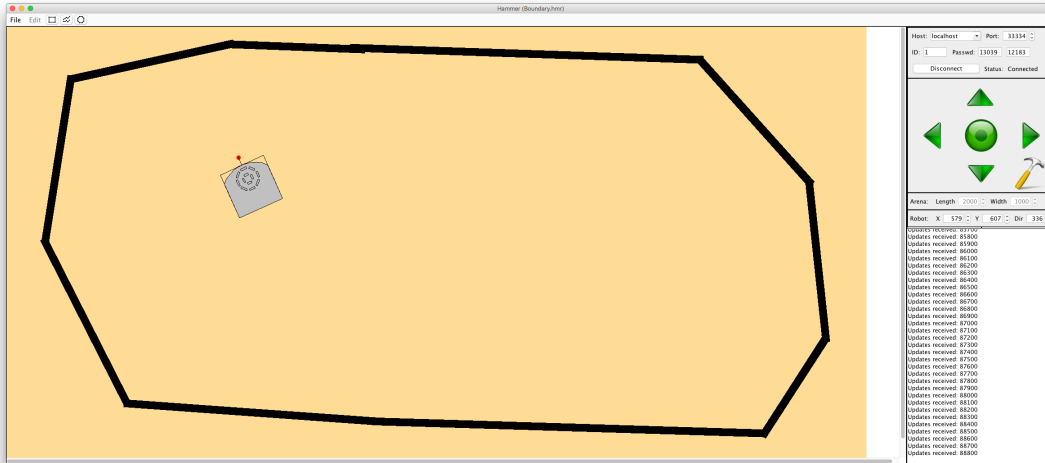
Figure 15: A ring of black tape with Thymio in the middle.

3. Drag the Thymio robot into the center of the ring that you constructed.

4. Save the environment by selecting the *Save* action from the *File* menu. The recommended file extension is `.hmr`. You can load the file into `Hammer` using the *Open* action from the *File* menu.

5. Click on *Connect* once you are done creating the circular path. Once `Hammer` connects to `Thor`, it enters *Simulation* and does not allow changes to the objects in the arena.

6. Switch to Aseba Studio and start a new program.

7. If you saved the program "`BoundaryAvoider`" from last lab, you can load it instead of reentering the code.

8. Create two constants: *SPEED* with a value of 200, and *EDGE* with a value of 400.

9. Enter the program in Figure 16 in to the Code Area.

10. Review how this program works by looking at Section 2.3 in the previous tutorial.

11. Save the program under the name "`BoundaryAvoider`".

12. Load and Run the program.

13. Start the robot by pressing the **Forward** button.

14. Observe the robot's behaviour. The robot should remain within the boundary of the arena.

15. Stop the robot by pressing the **Back** button.

16. **Ask** your TA to check if your program work.

17. **Questions for Section 3.3:**

```
motor.left.target = 0                       # reset motors (turn them off)
motor.right.target = 0

onevent button.forward                      # when forward button is pressed
  motor.left.target = SPEED                 # start motors forward
  motor.right.target = SPEED

onevent button.backward                     # when backward button is pressed
  motor.left.target = 0                     # stop the motors.
  motor.right.target = 0

onevent prox
  if motor.left.target == 0 then            # if not moving no need to proceed
    return
  end

  # if we are moving forward
  if motor.right.target > 0 and motor.left.target > 0 then
    if prox.ground.delta[0] < EDGE then         #   if line is on the left
      motor.right.target = -SPEED             #       turn right
    elseif prox.ground.delta[1] < EDGE then   #   else if line on right
      motor.left.target = -SPEED              #       turn left
    end
  # else we are not moving forward, if no line
  elseif prox.ground.delta[0] > EDGE and prox.ground.delta[1] > EDGE then
    motor.left.target = SPEED                       #   move forward
    motor.right.target = SPEED
  end
```

Figure 16: Boundary Avoidance Program.

(a) Does the speed at which the robot travels matter? Try increasing the speed to maximum (500) and see if it makes a difference.

(b) Suppose instead of the line being black, it was a lighter colour. What would you need to adjust? Try adjusting the colour of the path using the colour slider in the *Object Pane*. Again, you will first need to disconnect Hammer from Thor, perform the colour change, and then reconnect.

3.4   The Horizontal Proximity Sensors, Once More, with Feeling

Your last task is to investigate the the horizontal proximity sensors in the simulated robot and build environments with blocks that the robot has to avoid.

The blocks and black lines in the environment are based on real artifacts used in the lab. For example, the default block size is that of 14×2 Duplo block. And, the default line width and colour

is that of standard electrical tape. For most purposes, the defaults should suffice. See Appendix B for a fuller discussion.
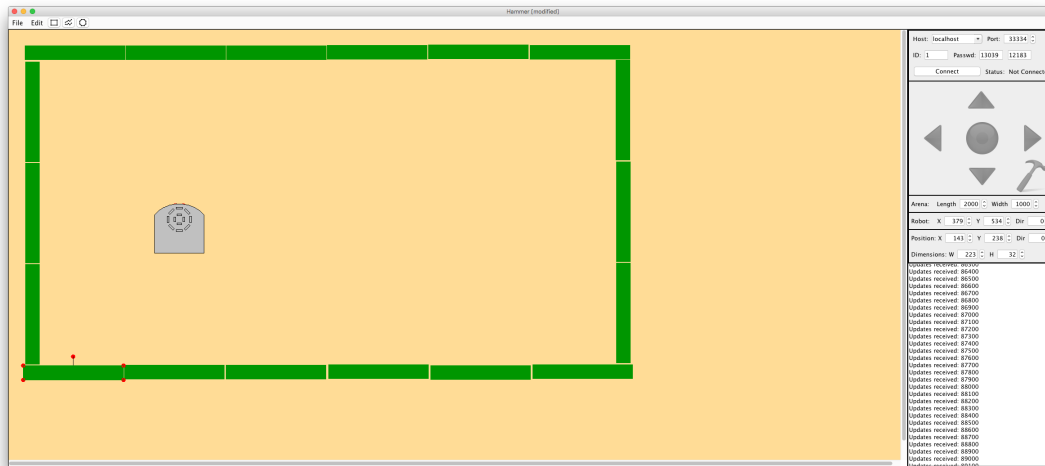

Figure 17: An enclosure of blocks with Thymio in the middle.

1. Click on the *Disconnect* button in the *Connection Pane*. `Hammer` should now be in *Edit* mode.

2. Create an enclosure of blocks in the arena, similar to the one depicted in Figure 17. To place a block in the arena, click on the button beside the *File* menu that looks like a block. You can then drag it anywhere in the arena and drop it by clicking. When selected, you can drag the block around and rotate it or resize it by using the handles. The dimension and orientation of the block are displayed in the *Object Pane* on the right.

   Try placing some blocks. You can copy a block by selecting it and using the standard hot-keys for copy and paste, or using the *Copy* and *Paste* actions in the *Edit* menu.

   Notice that if you drag a block over the Thymio, it is outlined in red and cannot be placed. This prevents you from sticking the Thymio inside a physical object.

3. Drag the Thymio robot into the center of the enclosure that you constructed.

4. Save the environment in the file "`BlockRing.hmr`".

5. Click on *Connect* to reconnect to `Thor`.

6. Switch to Aseba Studio, start a new program and enter the program in Figure 18 in to the Code Area.

7. Load and run the program.

8. Switch to `Hammer` and click on the **Forward** button to start the robot.

   Observe what happens when the robot touches a block. It stops and a sad face appears, indicating the robot cannot proceed. In `Hammer` blocks have infinite mass and cannot be

```
motor.left.target = 0                          # reset motors
motor.right.target = 0


onevent button.forward                         # when forward button is pressed
  motor.left.target = 200                      # start moving forward
  motor.right.target = 200


onevent button.backward                        # when backward button is pressed
  motor.left.target = 0                        # stop motors
  motor.right.target = 0
```
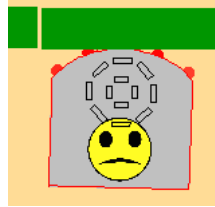
Figure 18: A simple go forward program.

Figure 19: Thymio stops when it hits a block.

moved by the robot. This is very different form the real world, where a robot can easily move a block. There are two reasons for this. First, simulating a robot pushing a block on the able requires more complex calculations than are implemented in the simulator. It requires knowing the friction coefficients of the table, the blocks, the robot's tires, as well as the weight of the blocks and the robot. While this can all be determined experimentally, it is not useful, because our robot generally wants to avoid touching the blocks in most cases.

9. Click on the **Back** button to stop the robot.

10. **Ask** your TA to check if your program work.

11. Switch to Aseba Studio and start a new program.

12. If you saved the program "ObjectAvoider" from last lab, you can load it instead of reentering the code.

13. Create two constants: *SPEED* with a value of 200, and *THRESHOLD* with a value of 0.

14. Enter the program in Figure 20 in to the Code Area.

15. Review how this program works by looking at Section 2.4 in the previous tutorial.

16. Save the program under the name "ObjectAvoider".

17. Load and run the program.

```
var min                               # min over all sensor readings
var max                               # max over all sensor readings
var mean                              # average over all sensor readings

motor.left.target = 0                 # reset motors
motor.right.target = 0

onevent button.forward                # when forward button is pressed
  motor.left.target = SPEED           # start moving forward
  motor.right.target = SPEED

onevent button.backward               # when backward button is pressed
  motor.left.target = 0               # stop motors
  motor.right.target = 0

onevent prox                          # check proximity sensors
  if motor.right.target == 0 then     # only if we are moving
    return
  end

  # compute min, max, and mean over the current sensor readings
  call math.stat( prox.horizontal[0:4], min, max, mean )

  if max > THRESHOLD then             # if a sensor is above threshold
    # if we are moving forward
    if motor.right.target > 0 and motor.left.target > 0 then
      # if object is closer on the left
      if prox.horizontal[0] > prox.horizontal[4] then
        motor.right.target = -SPEED       #   turn right
      else                            # else
        motor.left.target = -SPEED        #   turn left
      end
    end
  # else all sensors are below threshold, if we are also turning
  elseif motor.right.target < 0 or motor.left.target < 0 then
    motor.left.target = SPEED         # move forward
    motor.right.target = SPEED
  end
```

Figure 20: Object Avoidance Program.

18. Switch to `Hammer` and press the **Forward Button**.

19. Observe the robot's behaviour. The robot should avoid bumping into any of the walls.

20. **Ask** your TA to check if your program work.

21. **Questions for Section 3.4:**

    (a) Does the speed at which the robot travels matter? Try increasing the speed to maximum (500) and see if it makes a difference.

    (b) Suppose you wanted to make the robot move closer to an object before turning away. What would you need to adjust? Try it!

Appendix A: Installing `Thor` Locally

In an ideal world, you should be able to run the `Thor` server locally, on your own machine. However, in this world, there are a plethora of different operating systems and configurations. Providing an executable for each of the systems is not sustainable. Instead we have created a single guest Virtual Box image that will run the server on any platform that supports Virtual Box. While the install process is slightly more involved, it is an efficient way for us to make it possible for you to run your own copy of the `Thor` server.

1. Download and install Virtual Box from `https://www.virtualbox.org`.

2. Download the `Thor` guest image (`ThorV?.zip`) from the course website. Note: the `?` in `ThorV?.zip` is the version number.

3. Unzip the archive and place the folder somewhere safe on your machine. E.g., your Documents folder.

4. Run Virtual Box.

5. Select *Add Machine* from the *Machine* submenu of the main menu.

6. Navigate to the `Thor` folder that you created and select the `ThorV?.vbox` file and *Open* it. Once the process completes, *ThorV?* will appear in the left panel of the Virtual Box window.

When you are finished using the `Thor` server, you can shut it down by quitting Virtual Box.

1. Select *Quit* from the Virtual Box main menu, or use the *Cmd-Q* to quit.

2. When a dialog box comes up asking what to do, select *"Power off machine"*. This will shut down the server.

3. Select *Quit* from the Virtual Box main menu, or use the *Cmd-Q* to quit Virtual Box itself.

Note: `Thor` is still in the development stage, so it may crash. Be sure to save your work often. Send bug reports to `abrodsky@cs.dal.ca`.

Appendix B: The `Hammer` Visualizer

The User Interface

The visualizer's user interface is divided into three parts: the arena, the main menu, and the control panels. The arena, containing the large yellow area, is where the majority of the activity takes place. The arena depicts the Thymio and the simulated environment, which is initially a flat surface $1000mm \times 2000mm$ in size (approximately the size of one of the tables in the lab). The user can create an environment for the Thymio to negotiate, and then watch as the Thymio moves through it. The main menu, located at the top of the interface, provides functions for loading, storing, and editing the Thymio's environment. The three buttons allow the user to add blocks to the environment (square button), place marks on the surface of the arena (lines button), or place arbitrary polygons that behave either as blocks or as surface markings. All manipulations are performed via the control panels.

The control panels are located to the right of the arena. This area comprises several panels for manipulating various aspects of the visualizer. There are five fixed panels and one dynamic panel. The five fixed panels (top to bottom) are the Connection Pane, the Interaction Pane, the Arena Pane, the Robot Pane, and the Log Pane. The Object Pane, only appears when an object in the arena has been selected.

Connection Pane is used to start and stop connections to the `Thor` server. The panel is described in the *Getting Started* Section.

Interaction Pane has five buttons corresponding to the buttons on the Thymio. Pressing them causes the corresponding button event to occur in the simulated Thymio. The hammer button in the bottom right of the panel simulates a tap on the Thymio, causing the *tap* event to fire on the simulated Thymio. The panel is only enabled in *Simulation* mode, when the visualizer is connected to `Thor`.

Arena Pane is used to change the Arena dimensions. The pane is only enabled in *Edit* mode.

Robot Pane is used to change the position and the direction of the Thymio. This can also be done by dragging and dropping the Thymio.

Object Pane only appears when an object is selected. The panel allows the user to change the size, position, orientation and colour of the selected object. This panel will only appear in *Edit* mode.

Log Pane displays status updates and any information generated by the application as it is running. It is predominantly used for debugging purpose.

A Tale of Two Modes: Edit and Simulation

The visualizer has two modes: *Edit* (Not Connected) mode and *Simulation* (Connected) mode. In *Edit* mode the user can manipulate the Thymio's environment, but cannot see the Thymio move or interact with the Thymio, because the visualizer is not connected to `Thor`. In *Simulation* mode, the user can watch to Thymio move and interact with the Thymio, but not manipulate the Thymio's environment. The reason for this is that the current version of the protocol between the visualizer and `Thor` does not allow for changes to the environment once the simulation has begun.

Edit Mode

When started, the visualizer is in *Edit* mode. This mode allows the user to create or load the environment in which the Thymio is to be simulated. Artifacts, such as blocks and marks on the surface, can be added to the arena by clicking on one of the three buttons (Block, Line Mark, or Polygon) and placing the artifact on the arena. The shape, size, orientation, position, and colour of the artifacts can then be changed by selecting the desired artifact and manipulating it via its handles or the Object Pane located right above the Log Pane.

The artifacts in the environment are based on real artifacts used in the lab. For example, the default block size is that of 14×2 Duplo block. And, the default mark width and colour is that of standard electrical tape. For most purposes, the defaults should suffice. However, they can be changed by clicking on the artifact to select it, and them either manipulating the artifact's handles, or editing it in the Object Pane. Both the Block and Line Mark artifacts can be converted to Polygon artifacts by clicking on the action in the *Edit* menu.

The polygon is a general purpose artifact that can either be a block or a mark. It can be used to create objects in the environment that do not correspond to Duplo blocks or electrical tape marks. Once the environment is created, it can be saved and reloaded in a later session.

Environments can be saved and loaded via the File menu. Environment files have the extension `.hmr` and are simple text files that can be viewed with any text editor. The environment includes: the arena dimensions, the position and orientation of the Thymio, and the position, orientation, size, and shape of all the artifacts in the environment. This environment is sent to the `Thor` server when the visualizer enters *Simulation* mode.

Simulation Mode

The visualizer enters *Simulation* mode when it connects to the `Thor` server. In this mode the environment is immutable, the Interaction Pane is enabled, and the Arena and Object Panes are disabled. The only artifact that can be modified is the Thymio, which can be moved by either dragging it, or by using the Robot Pane. The Thymio's direction can also be changed. It is important to note that the actual simulation is done by the `Thor` server. The visualizer simply renders the simulation for the user. The visualizer will display the full arena during the simulation, which may appear smaller than in *Edit* mode.

Once in *Simulation Mode*, you can switch to Aseba Studio, load, and run your program on the simulated Thymio. You will be able to observe the Thymio moving in the Arena and interact with the Thymio via the Interaction Pane. You can also move the Thymio by dragging it or change its orientation. The movement of the Thymio is governed strictly by the speed of the wheel motors, i.e., basic physics.

The `Thor` server simulates only basic physics. It does not simulate drag, or friction, and it assumes that all artifacts have infinite weight and are immobile. Thus, if a Thymio bumps into a block, it stops, and must back up. If the Thymio makes contact with a block or moves beyond the boundary of the arena, it stops and displays a sad face. Moving the Thymio back into the arena and where it is not in contact with a block will remove the sad face and allow the Thymio to proceed. There is also no noise in the simulation, meaning that the Thymio may behave a little differently in the real world. E.g., if you tell the Thymio to move in a straight line, it will.

The visualizer also displays the same LED lights that an actual Thymio has. This makes debugging of your programs easier to do. You can tap the Thymio, using the hammer button, and

interact with it through the five buttons (up, right, down, left, and center).

The simulation is updated approximately at 20 to 60 frames per second. Thus, the visualizer may create a noticeable load on your system.

To switch back to *Edit* mode you need to disconnect from the `Thor` server. The Arena Pane will then be resized back to full size.

Note: Even though you have disconnected the `Thor` server keeps running the simulation. So, you may also want to stop your program in the Aseba Studio, at the same time.

Caveats

Note: Hammer is still in the development stage, so it may crash. Be sure to save your work often. Send bug reports to `abrodsky@cs.dal.ca`.

3.5   Appendix C: Setting the ID and Password Manually in `Hammer`

Since the Thor server runs multiple simulations concurrently, the fields: *ID* and *Password*, link your the visualizer to your simulation. To set these fields you need look at a couple of special variables in the Aseba Studio:

1. Switch to Aseba Studio (or Studio for Thymio II).

2. Click on the *View* sub-menu of the main menu. Ensure that "`Show hidden variables and functions`" is checked.

3. In the *Variables* panel on the left hand side, locate the `_id` variable (top of the list). Use the value of this variable to set the *ID* field in the Connection panel of the visualizer.

4. If you are using the remote server, you also need to set the password. In the *Variables* panel on the left hand side, locate the `_passwd` array variable (bottom of the list). Expand the array and use the two values in the array to set the *Password* fields in the Connection panel of the visualizer.

4   Modeling Sensors
-Creating Sensor Models

In this tutorial you will empirically evaluate your robot's sensors and create simple models of how the sensors behave. A *model* is a simplified description of a system (sensor) that we are trying to understand and use. We use the model to predict how the sensor behaves and interpret its response.

You will be provided with programs 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.

Once you have gathered the data, you will then construct a simple model of the sensor, which you will then be able to use when programming your robot in the future.

4.1   Modeling the Ground Proximity Sensors

In this section, we will investigate the response of the ground proximity sensors in typical lighting conditions. We will then construct a simple model of sensors' response.

1. Start a new program.

2. Add three constants: *SAMPLES* with a value of 5, *SAMPLE_PERIOD* with a value of 1000 and *SENSOR* with a value of 0. The first constant represents the number of times that we want the measurement performed, i.e., 5 measurements. The second constant represents the delay between measurements (one second). The third constant represents the sensor number; in this case, ground proximity sensor 0.

3. Copy and paste the code in Figure 21 into the Code Area.

```
var v[SAMPLES]                          # array of samples
var min                                 # stats variables
var max
var mean
var i = 0                               # counter

call math.fill( v, 0 )                  # initialize arrray
timer.period[0] = SAMPLE_PERIOD         # turn on timer

onevent timer0                          # when time goes off
  v[i] = prox.ground.delta[SENSOR]      # record sensor reading
  i++                                   # increment counter
  if i >= SAMPLES then                  # if we are done
    timer.period[0] = 0                 #  disable timer and
    call math.stat( v, min, max, mean ) #  compute stats
  end
  call sound.freq( 440, 10)             # beep to indicate progress
```

Figure 21: Sensor Sampling Program can be downloaded from the course website.

This program works in the following manner:

(a) The program takes a measurement of the ground proximity sensor, every second for 5 seconds. It stores the measurements in an array, and after completing the measurements computes the minimum, maximum, and mean of the values in the array. These values can be viewed from the Aseba Studio, by examining the variables in question in the Variable Area on the left.

(b) Each of the ground proximity sensors measures three values:

ambient light is the light intensity at ground level (the surrounding light level)
reflected light is the mount of light received while the sensor emits an infrared light
delta light is the difference between the reflected and the ambient light

To use an analogy, if you are out in the forest at night, there is certain amount of light around, even when your flashlight is off. This is the *ambient* light. If you turn your flashlight on, you see all the light emanating from the flashlight that *reflects* from the ground. The difference between your flashlight being on and off is the *delta* light. We are typically interested in the *delta* or *reflected* light because the ambient light has little effect on the sensors.

(c) The program first declares and initializes the variables it uses to store the sensor measurements and compute the various statistics.

(d) The program then initializes the period of **timer0** to *SAMPLE_PERIOD*, which is 1000ms (1 second). This causes the **timer0** event to occur every second.

(e) The **timer0** event handler reads the current value from one of the ground proximity sensors *prox.ground.delta[SENSOR]* and stores it in the ith element of the array, where i is then incremented by the event handler.

(f) If i is greater or equal to the number of *SAMPLES*, then the sampling is completed. The handler disables the timer by setting its period to 0 and computes the statistics.

(g) Lastly, the handler emits a beep, indicating that progress is being made.

(h) Once the beeping stops, the values in the array and the other variables can be examined in the Variable Area, located on the left side of the main pane of Aseba Studio.

4. Save the program under the name "`GroundSensorSampler`".

5. Load the program on the robot.

6. Locate Table 1 (located at the end of this tutorial). **Include with lab report.**

7. For each band in the gray-scale in Figure 22 (located at the end of this tutorial):

(a) Place the robot so that proximity ground sensor 0 is directly over the band.

(b) Run the program.

(c) Once the program finishes (no more beeps) examine the array of recorded values and statistics and record these in the corresponding column of Table 1.

Note, to speed things up, you can view the array and variables as the program runs and transcribe the values on the fly.

8. Create a line graph of "Grayscale Value" on the x-axis and "Ground Sensor Readings" on y-axis. Plot the line using the computed mean for each of the grey bands. **This is to be included with your lab report as well.**

9. Let the other groups know of your results by publishing a table of the computed means on the whiteboard at the front of the lab. Note: All groups will need to put their tables up, so don't take too much space. Your table should look like this:

Group #	Group Members: Alice, Bob, Carol					
Real Values	0%	20%	40%	60%	80%	100%
Mean Values (Sensor 0)						

10. We can construct a linear model for the ground proximity sensor.

   (a) Using a ruler, draw a line that best approximates the plotted values. Mark the start and end points of the line on the graph, and determine their coordinates. E.g., The start point s may be at $(0, 95)$ and the end point e at $(100, 700)$. This line represents the *linear* model of the sensor. From this we can derive an equation that models the sensor.

   (b) Compute the slope m of the line by dividing the rise by the run. I.e.,

   $$m = \frac{e_y - s_y}{e_x - s_x}$$

   In the example above, the slope would be:

   $$m = \frac{700 - 95}{100 - 0} = 6.05$$

   (c) Compute the y-intercept of the line, denoted by b, by noting where it crosses the y axis. In this example, the initial point $(0, 95)$ is on the y axis, so the y-intercept is $b = 95$.

   (d) We now insert the slope m and y-intercept b into the equation of a line

   $$y = m \times x + b$$

   to yield the equation for our model, where x is the actual value and y is the measured value. That is, if we measure a grey colour at brightness x, then y is what the sensor in question should return. We can use this equation to model (predict) how our sensor behaves. Typically, it is more useful to determine what the brightness is given the value from the sensor. Thus, we need to re-arrange the equation:

   (e) Re-arrange this equation by solving for x:

   $$x = \frac{y - b}{m} = \frac{y}{m} - \frac{b}{m}$$

   Thus, if we measure y, plug it into the equation above, the result (approximately) is the shade of grey that the sensor is sensing.

Congratulations! You have created a linear model of your robot's ground proximity sensors.

11. **Ask** your TA to check the results and if your program work.

12. Stick a piece of electrical tape onto your wooden table top.

13. Repeat the above procedure to model the response of the ground sensors for the following three conditions (instead of the grey scale values):

    (a) the robot is sensing the desk

    (b) the robot is sensing the edge of tape/desk, and

    (c) the robot is sensing the tape

14. Let the other groups know of your results, by publishing a summary table. The summary table on the whiteboard should look like this:

Group #	Alice, Bob, Carol		
Surface	Desk	Edge	Tape
Mean Values (Sensor 0)			

15. **Ask** your TA to check the results and if your program work.

16. **Questions for Section 4.1:**

    (a) What variables were fixed and what variables varied in the characterization that you performed?

    (b) Would the lighting conditions affect the readings? For example, would shadows or additional overhead lights change your measurements?

    (c) Suppose that you implemented your program on one robot but then had to run your program on a different robot (of the same type)? What problems would you encounter? How could these problems be addressed?

    (d) Would it be possible to create a single model (a single equation) for both sensors? Why or why not?

    (e) Given your measurements with the tape and table top. What threshold(s) would you use to distinguish between tape versus table top? Can the same threshold be used for both sensors? Why or why not?

    (f) How do your values (in both tables) compare to that of the other groups? Note: you may need to answer this question near the end of the lab.

4.2   Modeling the Horizontal Proximity Sensors

In this section you will modify the program that you used in the previous section to model the horizontal proximity sensors.

1. Load the program "`GroundSensorSampler`", which you used in Section 4.1.

2. Save the program under the name "`HorizontalSensorSampler`".

3. Modify the program in the following ways:

   (a) Change the *SENSOR* constant to 2.

   (b) On line 11, replace the variable access *prox.ground.delta*[*SENSOR*] with the variable access *prox.horizontal*[*SENSOR*].

4. Save the program.

5. Load the program on the robot.

6. Locate Figure 23, the 15cm ruler, which you will use for this portion of the lab.

7. Borrow a Duplo wall from the lab facilitator. The wall should be three bricks high.

8. Locate Table 3, entitled "Response of Horizontal Sensor 2", which you will fill in and include in your lab report.

9. For each of the distances in Table 3:

   (a) Place the robot on the ruler paper so that the horizontal sensor is located at centimeter 0 and the ruled line emanates directly from the center of the sensor and is perpendicular to the surface of the robot.

   (b) Place the Duplo wall at the corresponding distance such that it is directly parallel to the sensor (perpendicular to the ruler line).

   (c) Run the program.

   (d) Once the program finishes (no more beeps) examine the array of recorded values and statistics and record these in the corresponding column of the table. Note:, as before you can record the values as the program is running.

10. Plot a line graph of "Distance" on the x-axis and "Horizontal Sensor Readings" on y-axis. For each of the two horizontal sensors plot the computed mean for each of the distances. **This is to be included with your lab report as well.**

11. Using the same procedure as in Section 4.1 derive a linear model horizontal sensor 2 and derive the corresponding equation. That is:

    (a) Draw line that best matches the plotted averages.

    (b) Compute the slope (m) and y-intercept (b) of the line.

    (c) Plug the values into the standard equation of a line.

12. Let the other groups know of your results by publishing a table of the computed means on the whiteboard at the front of the lab. Note: All groups will need to put their tables up, so don't take too much space. Your table should look like this:

Group #	Group Members: Alice, Bob, Carol							
Real Values	0cm	2cm	4cm	6cm	8cm	10cm	12cm	14cm
Mean Values (Sensor 2)								

13. **Ask** your TA to check the results and if your program work.

14. **Questions for Section 4.2:**

    (a) What variables were fixed and what variables varied in the characterization that you performed?

    (b) Would the lighting conditions affect the readings? For example, would shadows or additional overhead lights change your measurements?

    (c) Compare your results to other groups. Do all the horizontal proximity sensors have the same response (behaviour)? If not, what are the differences? Note: you may need to answer this question near the end of the lab.

    (d) Would it be possible to create a single model (a single equation) for all sensors? Why or why not?
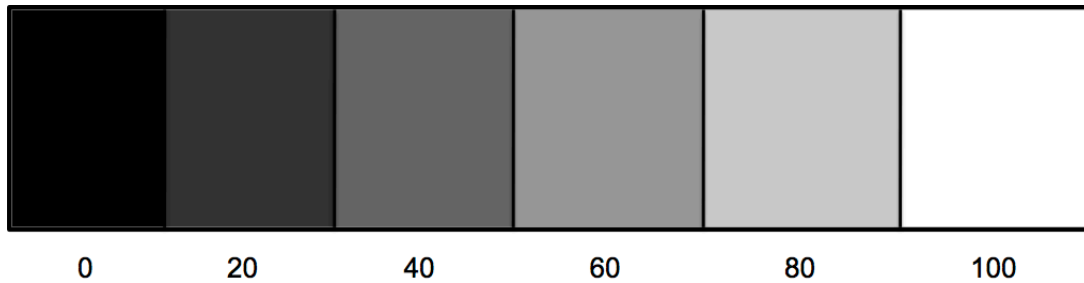
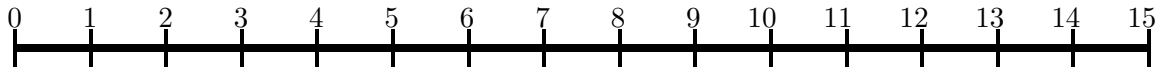Figure 22: Gray-scale for characterizing the ground proximity sensors.

Figure 23: Ruler for characterizing the horizontal proximity sensors.

	0%	20%	40%	60%	80%	100%
1						
2						
3						
4						
5						
Min						
Max						
Ave						

Table 1: Response of Proximity Ground Sensor 0. Include with lab report.

	Desk	Edge	Tape
1			
2			
3			
4			
5			
Min			
Max			
Ave			

Table 2: Edge Response of Proximity Ground Sensor 0. Include with lab report.

	0cm	2cm	4cm	6cm	8cm	10cm	12cm	14cm
1								
2								
3								
4								
5								
Min								
Max								
Ave								

Table 3: Response of Proximity Horizontal Sensor 2. Include with lab report.

5 Modeling Drive Actuators
-Speed and Divergence

In this tutorial you will empirically evaluate the robot's actuators. For example, suppose that you wanted your robot to make a 90 degree right turn. In this case, the left wheel has to turn forward, and the right wheel in reverse. Furthermore, for each degree that the robot turns, the motors need to run at the given speed for a specific duration. Without knowing the duration needed to turn the robot some fixed number of degrees, it is not possible to program the robot to make specific maneuvers, In fact, as we shall see, the duration depends not only on the amount of turning required, but also on the speed and possibly other factors as well.

5.1 Relating Target Setting to Speed

In this experiment you will create a program that runs the robot at various motor settings for a fixed period of time and measure the distance that the robot travels. You will then use this data to construct a linear model of how fast the robot moves for a given motor setting.

1. Start a new program.

2. Build a simple program that drives the robot forward 5 seconds when the **Forward Button** is pressed. You will not need any variables, but you should create a constant *TARGET*, initially set to 100, which you can use to set the speed of the robot. You will need two event handlers: One for the `button.forward` event, to start the robot, and the other for the `timer0` event to stop the robot after 5 seconds (5000 milliseconds). The entire program should between 8 to 10 lines of code! Hint: Take a look at the square program you wrote in the first tutorial, if you don't know where to start.

3. Save the new program as "`DistanceTest`".

4. The edge of the table will be the start line. Be sure that the robot has at least a meter of travel distance.

5. Locate Table 4, at the end of this tutorial, where you will record your measurements. Its five columns are labeled with a motor target setting: 100, 200, 300, 400, and 500 and each column contains 6 entries: 3 measurements, the minimum, maximum, and the mean. **This is to be included with your lab report.**

6. For each of the five motor target settings (100, 200, 300, 400, 500):

   (a) Set the *TARGET* constant to the target setting.

   (b) Save the program and load it on the robot.

   (c) Run the program and unplug the USB cable.

   (d) Perform the following measurement 3 times:

        i. Place the robot so that its rear edge is aligned with the start line.

        ii. Press the **Forward Button**. The robot will move forward for five seconds.

        iii. Measure this distance and record it in the table under the corresponding column.

   (e) Plug the USB cable back into the robot

(f) Compute the minimum, maximum, and mean of the column of measurements and record these at the bottom of the column. To compute the mean (average), sum the 3 measurements and divide the total by 3.

7. Plot a line graph of "Target Setting" on the x-axis and "Distance Traveled in Centimeters" on y-axis. Use the mean values computed above. **This is to be included with your lab report as well.**

8. Let the other groups know of your results by publishing a table of the computed means on the whiteboard at the front of the lab. Note: All groups will need to put their tables up, so don't take too much space. Your table should look like this:

Group #	Group: Alice, Bob, Carol				
Target Setting	100	200	300	400	500
Distance Traveled					

9. Create a linear model relating target setting to distance traveled. Recall from the previous tutorial, that we construct a model by:

   (a) Using a ruler, draw a straight line that best approximates the plotted points.
   (b) Compute the slope (m) of the line.
   (c) Compute the y-intercept.
   (d) Plug these into the equation of the line:

   $$y = m \times x + b$$

   (e) The resulting equation relates the target setting (x) to the distance (y) traveled in 5 seconds.

10. We can now create a linear model relating speed to target setting. Recall that speed is distance traveled divided by the time it took to travel the distance.

    $$v = \frac{d}{t}$$

    Hence, since our robot ran for 5 seconds, its speed is $\frac{y}{5}$, where y is the distance traveled. Thus, our model for speed of robot given target setting is:

    $$v = \frac{y}{5} = \frac{m \times x + b}{5}$$

    Thus, given a target setting (x), you can now predict at what speed (v) the robot will move. And if you wish to make the robot move at a specific speed (v), you can rearrange the equation above, solving for x, to figure out what the target setting should be.

11. **Ask** your TA to check the results and if your program work.

12. **Questions for Section 5.1:**

(a) What variables were fixed and what variables varied in the characterization that you performed?

(b) Would the surface on which the robot is moving affect your measurements?

(c) How do your measurements compare to that of the other groups? Note: you may need to answer this question near the end of the lab.

(d) Does the robot alway travel straight? If not, under what conditions does it veer from the straight path?

5.2 Measuring Divergence

In this experiment you will measure the amount of divergence from the straight path that robot undergoes as it moves in a straight line. Since the motors are imperfect, they may run at slightly different speeds, resulting in the robot veering to one side or another. This effect tends to be correlated with the speed of the robot. You will measure the divergence that occurs over a fixed distance that a robot travels at various speeds. You will then use this data to construct a linear model that predicts the amount of divergence, for a given speed per unit distance.

1. Start a new program.

2. Build a simple program that drives the robot forward when the **Forward Button** is pressed. The robot should stop *when* either of its ground proximity sensors encounters a black line (electrical tape). You can use the Boundary Avoidance Program from Tutorial 1 as an example, if you are not sure how to proceed.

   You will not need any variables, but you should create two constants: *TARGET*, initially set to 100, which you can use to set the speed of the robot, and *THRESHOLD*, set to 300, which is to be used to identify when the robot has encountered the black line. You will need two event handlers: One for the `button.forward` event, to start the robot, and the other for the `prox` event, which should use the *when ... do* statement to check whether either of the two ground sensors is registering the black line, and if so, stops both motors. As we learned in the last tutorial, a ground proximity sensor registers a dark patch on the ground if it returns low values that are typically less then 300, which is what our *THRESHOLD* is set to.

3. Save the new program as "`DivergenceTest`".

4. Using black tape, demarcate a start and finish line on the table, such that the lines are parallel, and exactly 111cm apart. The robot itself is 11cm deep, so when its rear is placed on the start line, it will need to travel exactly 1m (100cm) before stopping. Note: The finish line should be about 30cm to 50cm long. Using a straight edge and two small pieces of tape mark two points, representing a straight line that is perpendicular to the start and finish lines. (See Figure 24.) The first point should be on the finish line. The second, should be 11cm from the start line, when the nose of the robot will be when the rear of the robot is aligned with the start line. Note: the tape marks should be as small as possible.

5. Locate Table 5 where you will record in your measurements. **This is to be included with your lab report.**

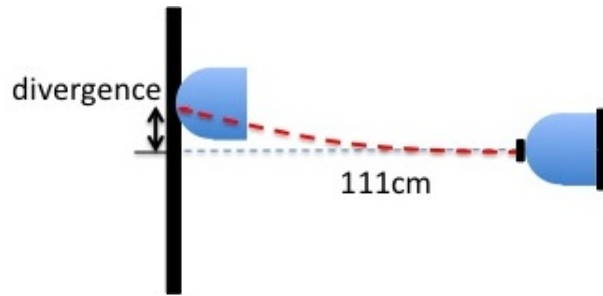6. For each of the five motor target settings (100, 200, 300, 400, and 500):

Figure 24: Setup for measuring divergence per unit distance.

(a) Set the *TARGET* constant to the target setting.

(b) Save the program and load it on the robot.

(c) Run the program and unplug the USB cable.

(d) Perform the following measurement 3 times:

    i. Place the robot so that its rear edge is aligned with the start line and its front is aligned with the tape mark.

    ii. Press the **Forward Button**. The robot will move forward until it encounters the finish line.

    iii. Measure the distance along the finish line between the nose of the robot and the mark on the finish line, denoting where the robot should end up if there is no divergence. Record the distance in the table under the corresponding column.

(e) Plug the USB cable back into the robot

(f) Compute the minimum, maximum and mean for the column of measurements and record them at the bottom of the column.

7. Plot a line graph of "Target Setting" on the x-axis and "Divergence Distance in Centimeters" on y-axis. Use the mean values computed above. **This is to be included with your lab report as well.**

8. Let the other groups know of your results by publishing a table of the computed means on the whiteboard at the front of the lab. Note: All ten groups will need to put their tables up, so don't take too much space. Your table should look like this:

Group #	Group: Alice, Bob, Carol				
Target Setting	100	200	300	400	500
Divergence Distance					

9. Create a linear model and the corresponding equation relating target setting to divergence distance.

10. **Ask** your TA to check the results and if your program work.

11. **Questions for Section 5.2:**

   (a) What variables were fixed and what variables varied in the characterization that you performed?

   (b) To minimize divergence, what target settings (speed) would you recommend? Why?

   (c) How do your measurements compare to that of the other groups? Note: you may need to answer this question near the end of the lab.

   (d) Describe a way that your robot could correct for divergence, assuming you had the model you developed in this section. If you have time, try out your solution.

	100	200	300	400	500
1					
2					
3					
Min					
Max					
Ave					

Table 4: Distance traveled in 5 seconds. Include with lab report.

	100	200	300	400	500
1					
2					
3					
Min					
Max					
Ave					

Table 5: Divergence after one meter of travel. Include with lab report.

6 Modeling Drive Actuators
-Turning and Divergence

In this tutorial you will empirically evaluate the robot's actuators. 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 robot's expected position in a given task.

6.1 How to Turn?

In this experiment you will measure the amount of time it takes to perform a 360-degree turn and relate that to the target setting. This is necessary information because if we want to make our robot turn. We need to know how long and how fast to run the motors to complete the turn. Once we know this, we can predict how long it would take to perform d degree turn, by multiplying the time it takes to perform a 360-degree turn by $\frac{d}{360}$. You will measure the time it takes to perform a 360-degree turn at a given target setting. You will then use this data to construct a linear model that predicts the time required to perform a turn of d degrees at a given target setting.

1. Start a new program.

2. Copy and paste the code listed in Figure 25.

3. You will need to add three constants:

   TARGET which is initially 100 and denotes the target setting of the motors.

   THRESHOLD which is initially 300 and denotes the threshold between dark (tape) and light (table) sensed by the ground proximity sensor

   SAMPLES which is 3 and denotes the number of samples that this program will perform.

4. Save the new program as "`TurnTimer`".

5. Place two 10cm (approximately) pieces of tape side by side on the table to create a dark $10cm \times 3cm$ patch. The pieces of tape should overlap slightly. There should be plenty of table space for the robot to do a complete turn, assuming it starts beside the black tape.

6. Locate Table 6 at the end of this tutorial, entitled "360-Degree Left Turn Time for a given Target Setting". You will record the measurements in this table. **This is to be included with your lab report.**

7. For each of the five motor target settings (100, 200, 300, 400, 500):

   (a) Set the *TARGET* constant to the target setting.
   (b) Save the program and load it on the robot.
   (c) Run the program and unplug the USB cable.
   (d) Place the robot so that the black tape patch you created is adjacent to the left side of the robot. The robot should be about 0.5cm from the patch.

```
var data[SAMPLES]                          # variable declarations
var turn                                   # id of turn being timed
var min
var max
var mean


motor.left.target = 0                      # reset motors
motor.right.target = 0


sub stop                                   # subroutine to stop robot
  motor.left.target = 0                    # reset motors and timer
  motor.right.target = 0
  timer.period[0] = 0
  call math.stat( data, min, max, mean )   # compute stats on data


onevent button.forward                     # ON forward button press
  call math.fill( data, 0 )                # Init data array & turn id
  turn = -1
  motor.left.target = -TARGET              # Activate motors and timer
  motor.right.target = TARGET
  timer.period[0] = 100                    # time turns in .1 secs


onevent button.backward                    # On back button press
  callsub stop                             # invoke subroutine to stop


onevent prox                               # when prox event occurs
  # when robot is running and it has encounterd black tape
  when motor.left.target != 0 and prox.ground.delta[0] < THRESHOLD do
    turn++                                 # next 360 turn
    if turn >= SAMPLES then                # if no more turns, stop
      callsub stop
    end
    call sound.freq( 440, 10)              # sound progress beep
  end


onevent timer0                             # on timer0 event
  if  turn >= 0 then                       # if performing turn
    data[turn]++                           #   inc time of turn
  end
```

Figure 25: Turn Timing Program

(e) Press the **Forward Button**. The robot will start turning. Every time its left ground proximity sensor encounters the black patch it will make a beep, indicating the start of

the next turn to be timed. The robot will perform three (3) turns as specified by the *SAMPLES* constant.

   (f) Plug the USB cable back into the robot and click on the "`auto`" checkbox to refresh the variable values from the robot.

   (g) Read variable values in the *data* array and the *min*, *max*, and *mean* variables and record them in the corresponding column of the table.

8. Plot a line graph of "Target Setting" on the x-axis and "Turn Time in Seconds" on y-axis. Note: your data is in tenths (0.1) of a second, so a value of 78 is interpreted as 7.8 seconds. Use the mean values recorded above. Plot left turn data. **This is to be included with your lab report as well.**

9. Let the other groups know of your results by publishing a table of the computed means on the whiteboard at the front of the lab. Note: All groups will need to put their tables up, so don't take too much space. Your table should look like this:

Group #	Group: Alice, Bob, Carol				
Target Setting	100	200	300	400	500
Time of Left Turn					

10. Create a linear model and the corresponding equation relating target setting to time it takes to make a 360-degree turn.

11. We can now use the model you created in the previous step to create a second model that relates the number of degrees that the robot needs to turn and the speed at which it is turning. Recall from our preliminary discussion that if we know the time it takes to turn 360 degrees (y), then we can multiply this by $\frac{d}{360}$ to yield the time necessary to turn d degrees. The linear model from the preceding step is of the form

$$y = m \times x + b$$

where y is the predicted time to turn 360 degrees at target setting x, and where m and b are the constants that you derived from the data. Hence, the time needed by your robot to turn d degrees at target setting x is

$$y \times \frac{d}{360} = (m \times x + b) \times \frac{d}{360} = \frac{(m \times x + b) \times d}{360}$$

For example, if the robot is to turn 90 degrees at a target setting of 200, then we plug in 90 for d, and 200 for x, to get the amount of time needed to make the turn. Then we can engage the the robot's motors with the target setting, and set the timer period to the computed time. When the timer goes off, the turn has completed.

12. **Ask** your TA to check the results and if your program work.

13. **Questions for Section 6.1:**

(a) Describe the structure and function of the program in Figure 25 in the same manner as the descriptions of programs in the previous tutorial.

(b) Modify the program to perform a right 360-degree turn instead of a left one.

(c) What variables were fixed and what variables varied in the characterization that you performed?

(d) Were the measurements for the left turns significantly different? I.e., can one model be used for both the left and right turns?

(e) How do your measurements compare to that of the other groups? Note: you may need to answer this question near the end of the lab.

(f) Suggest some general heuristics (rules of thumb) to make the robot's motion as accurate as possible.

6.2 Another Square

Now, we can use our knowledge to update our program that makes the robot run in a square.

1. Start a new program.

2. Build a program that drives the robot in a square whose sides are 30cm long. You will need the models from the previous experiments to determine the best settings. The goal is to create a program that drives the robot as accurately as possible. You can use the Square program from Tutorial 1 as a starting point if you are not sure where to begin.

   The robot should make one complete square and return to its original position when the **Forward Button** is pressed.

3. Save the new program as "`Square2`".

4. Load the program on the robot.

5. Run the program and unplug the USB cable.

6. Place a small piece of tape on the table indicating where the robot is to start.

7. Locate Table 7, entitled "Distance from Origin after n Rounds", where you will record in your measurements. **This is to be included with your lab report.**

8. Repeat the following ten (10) times:

   (a) Place the robot so that it is adjacent to the piece of tape.

   (b) Press the **Forward Button**. The robot will make a square and stop.

   (c) Record in the first column the distance from the robot to the piece of tape. (Ideally, it should be 0.). **Note: Do not move the robot.**

   (d) Press the **Forward Button** again and record the distance again (in the second column) after the robot comes to a halt.

   (e) Repeat the above step two (2) more times, completing a row of the table.

9. Compute the minimum, maximum, and mean distances for each of the columns.

10. **Ask** your TA to check the results and if your program work.

11. **Questions for Section 6.2:**

    (a) Does the average distance increase with the number of squares performed? If so, why?

    (b) Consider the maximums and minimums. How far away are they from the mean? To compute the relative distances use the formulae

    $$d_{min} = \frac{mean - min}{mean}$$
    $$d_{max} = \frac{max - mean}{mean}$$

    Does the relative distance change as the number of squares increases? If so, why?

    (c) Consider the maximums and minimums from the other previous experiments you performed today. Do the relative distances between the mean and the minimums and maximums increase as the target setting (speed) increases? If so, why?

    (d) Hence, what conclusions can you draw about the reliability of robot's drive actuators?

	100	200	300	400	500
1					
2					
3					
Min					
Max					
Ave					

Table 6: 360-Degree Left Turn Time for a given Target Setting. Include with lab report.

	1 turn	2 turns	3 turns	4 turns	5 turns
1					
2					
3					
Min					
Max					
Ave					

Table 7: Distance from Origin after n Rounds.

7 Modeling the Real World
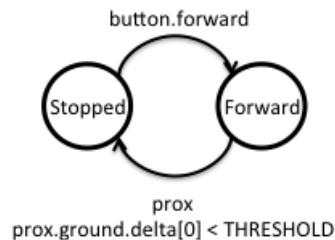## -States and Transitions

In this tutorial you will investigate methods for modeling the robot's behaviour and environment. You will begin by writing a small program that pauses the robot when it encounters an object ahead of it and another program that makes the robot 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 "Square" program from the previous tutorials. Initially the robot is in a Stopped state where it does not do anything. Once the `button.forward` event occurs, it starts moving forward and transitions to the Forward state. Once the `timer0` event occurs, the robot starts turning and transitions to the Turn state. Once the `timer0` event occurs again, the robot starts moving straight once more and transitions back to the Forward state. If the `button.backward` event occurs, the robot stops and transitions to the Stopped state.
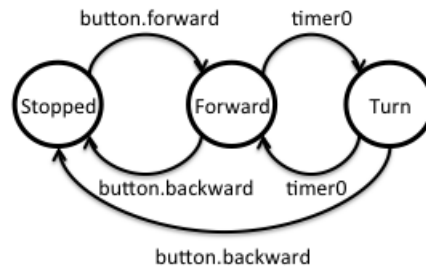
In another example, consider the "DivergenceTest" program from the previous tutorial. In this program the robot is initially in a Stopped state. Once the `button.forward` event occurs, the robot starts moving forward and enters the Forward state. Once the robot detects a black line during the `prox` event it stops, transitioning to the Stopped state.

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 that involve counters. In this tutorial you will investigate the state transition approach.

7.1 A Simple Stop and Go Program

1. Start a new program.

2. Download from the website (or copy and paste) the code listed in Figure 26.

```
var min                              # min over all sensor readings
var max                              # max over all sensor readings
var mean                             # average over all sensor readings
var state = STOPPED                  # state of the robot

motor.left.target = 0                # reset motors
motor.right.target = 0

onevent button.forward               # if forward button is pressed
  state = FORWARD                    # transition to FORWARD state
  motor.left.target = TARGET         # start moving forward
  motor.right.target = TARGET

onevent button.backward              # if backward button is pressed
  state = STOPPED                    # transition to STOPPED state
  motor.left.target = 0              # stop motors
  motor.right.target = 0

onevent prox                         # check proximity sensors
  if state != STOPPED then           # only if we are not STOPPED
    # compute min, max, and mean over the current sensor readings
    call math.stat( prox.horizontal[0:4], min, max, mean )

    # if we are in state FORWARD and sense a blockage
    if state == FORWARD and max > THRESHOLD then
      state = BLOCKED                # transition to BLOCKED state
      motor.left.target = 0          # stop motors
      motor.right.target = 0
    end

    # if we are in state BLOCKED and blockage is removed
    if state == BLOCKED and max <= THRESHOLD then
      state = FORWARD                # transition to FORWARD state
      motor.left.target = TARGET     # start moving forward
      motor.right.target = TARGET
    end
  end
```

Figure 26: A Simple Stop and Go Program

3. Add the following constants:

TARGET the target setting for the motors when the robot is moving, set to a value like 200.

THRESHOLD the threshold used to determine whether the horizontal sensors are sensing an object ahead, set to 0.

STOPPED the Stopped state identifier, set to 0.

FORWARD the Forward state identifier, set to 1.

BLOCKED the Blocked state identifier, set to 2.

Note that the identifiers of the states need to be unique. The easiest way to ensure this is to number the states consecutively, starting from 0. The state transition diagram for the program is depicted in Figure 27.
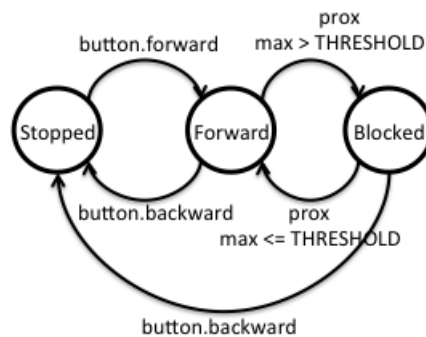
Figure 27: State transition diagram for the Stop and Go program.

4. Save the program as "`StopNGo`".

5. Load the program on your robot and give it a try. You should not need to unplug it.

6. **Ask** your TA to check if your program work.

7. **Questions for Section 7.1:**

   (a) Briefly explain how this program works.

   (b) How does the program encode its current state?

   (c) How did the programs that we developed previously encode their state? (Hint: Compare the `prox` event handler in this program to the `prox` event handler in the previous programs.)

   (d) Identify where the transitions depicted by the state-transition diagram occur in the program. Briefly describe the cause of each of the transitions.

7.2   A Simple Line Following Program

We now implement a line following program that will allow the robot to follow a black line (electrical tape). Ideally we would first derive a state-transition diagram and then implement it. However, it is instructive to first do the reverse: derive a state-transition diagram from the program.

1. Start a new program.

2. Copy and paste the code listed in Figure 28.

```
var state = STOPPED                          # variable declarations

motor.left.target = 0                        # reset motors
motor.right.target = 0

onevent button.forward                       # on forward button press
  state = LEFT                               # transition to LEFT state

onevent button.backward                      # on backward button press
  state = STOPPED                            # transition to STOPPED state
  motor.left.target = 0                      # stop motors
  motor.right.target = 0

onevent prox                                 # on prox event
  if state != STOPPED then                   # if robot is moving
    # if moving left and not sensing black line
    if state == LEFT and  prox.ground.delta[0] >= THRESHOLD then
      state = RIGHT                          #  transition to RIGHT state
      motor.left.target = TARGET             #  move right
      motor.right.target = 0
    end

    # if moving right and sensing black line
    if state == RIGHT and prox.ground.delta[0] < THRESHOLD then
      state = LEFT                           #  transition to LEFT state
      motor.left.target = 0                  #  move left
      motor.right.target = TARGET
    end
  end
```

Figure 28: Line Following Program

3. Add the following constants:

   TARGET the target setting for the motors when the robot is moving, set to a value like 300.

   THRESHOLD the threshold used to determine whether the horizontal sensors are sensing an object ahead, set to 300, for now.

   STOPPED the Stopped state identifier, set to 0.

   LEFT the Left state identifier, set to 1.

   RIGHT the Right state identifier, set to 2.

4. Save the program as "`LineFollow`".

5. Load the program on your robot.

6. Using black tape, create a closed loop on the table to be used as a track by the robot. It should have a couple curves in it and one or two straightaways. (Don't make the corners too sharp.

7. Run the robot on the track to see how it performs.

8. Try adjusting the *TARGET* constant (speed of the motors) and the *THRESHOLD* used to distinguish tape from table.

9. Find the optimal settings for your line-follower so that it moves as quickly as possible without losing track of the line.

10. **Ask** your TA to check if your program work.

11. **Questions for Section 7.2:**

    (a) Briefly explain how this program works.

    (b) Draw the state-transition diagram for the program. Be sure to label all the states and transitions.

    (c) Compare this state-transition diagram to the one from the previous question. How similar are they? Why?

    (d) Identify where the transitions depicted by the state-transition diagram occur in the program. Briefly describe the cause of each of the transitions.

    (e) What is the sharpest corner that this program can negotiate in both directions? I.e., the robot should be able negotiate the corner regardless of which direct (along the line) it is traveling.

    (f) Under what conditions will the robot go off-track? Why?

7.3 A Stop and Go Line Follower

One problem with our current line follower is that the robot will bump into things as it is following the line. Ideally, we would like to make use of the horizontal proximity sensors to detect objects in the robot's path and pause the robot until the object is moved. Our next goal is to extend the line-following program to achieve this requirement.

1. Using the preceding state-transition diagrams as possible starting points, derive a new state-transition diagram for a program that follows the line but pauses the robot when an object appears in its path. Hint: The state-transition diagram requires at most four or five states. **This should be submitted with your lab report.**

2. Load the "`LineFollow`" program.

3. Save it as a new program called "`StopNFollow`".

4. Using the state-transition diagram you developed as a guideline, and the code from both the "StopNGo" program and the "LineFollow" program as guidelines, extend this copy of the "LineFollow" program to stop the robot whenever it is blocked. That is:

   - The robot should start following the line when the **Forward Button** is pressed.
   - The robot should stop when the **Backward Button** is pressed.
   - If the robot is moving and it encounters an object, The robot should pause while its path is blocked.
   - If the robot is paused because it is blocked and the path becomes unblocked, the robot should resume following the line.

   Note: Your program will have two different thresholds, one for the ground proximity sensors and the other for the horizontal proximity sensors. Hence you will need distinct names for the threshold constants. Be sure that all the states have distinct values.

5. Save your program and load it on the robot.

6. Try running your program. Fix any problems you encounter.

7. **Ask** your TA to check if your program work.

8. **Questions for Section 7.3:**

   (a) Briefly explain how this program works.
   (b) Identify where the transitions depicted by the state-transition diagram occur in the program. Briefly describe the cause of each of the transitions.
   (c) When the robot becomes unblocked, how does your program determine whether to transition to the Left or Right state?

7.4 Building a Better Line Follower

Interestingly, the line follower we have developed only uses one of its ground proximity sensors (the left one). The robot also wastes a lot of time and energy swinging from side to side when it could be going straight. The question you should is: Can we do better?

1. Load the "LineFollow" program.

2. Save it as a new program called "BetterLineFollow"

3. Think about how you can improve on the design of this program. As mentioned already, two possible things to consider is using the other sensor and allowing the robot to travel in a straight line, when it can.

4. Plan your program extension by designing the state transition diagram for the extension. The state transition diagram for the "LineFollow" program may be a good starting point.

5. Implement your extensions.

6. Test your extensions. Are your extensions an improvement?

7. Iterate over your design and improve it as much as possible.

8. Compare your design to the original by timing how long each of the programs takes to race around your track. Repeat the races several times. Note: If your design loses the line much more often than the original design, this may be a drawback.

9. Record the outcomes of your comparison in a table and determine how much faster your design is compared to the original.

10. **Ask** your TA to check if your program work.

11. **Questions for Section 7.4:**

    (a) Describe the improvements you made to the original program and justify your decisions.

    (b) Include the state-transition diagram of your final design.

    (c) What is the performance improvement of your design? Include the timings you gathered when comparing your design to the original.

    (d) Are there any drawback or flaws in your design that you have noticed? If yes, describe them. If no, justify why.

8 Dealing with an Imperfect World
-Fault Identification and Recovery

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 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 common failures of sensors and actuators.

8.1 Determining Failure Modes

1. Load the "LineFollow" program that was implemented in the last tutorial.

2. Load the program on your robot.

3. Create a black tape track, like in the previous tutorial and test how well the robot follows the track. If the track is too simple, feel free to make it more complex, with sharper corners. Note, there is asymmetry in the program in the sense that the robot only uses its left ground proximity sensor and uses it to detect the left edge of the line. Consequently, the robot may have more difficulty traversing the track in one direction than another. Specifically, the robot may be more likely to lose track of the line during a sharp left turn than a sharp right turn.

4. Try increasing the likelihood of the robot losing the line by increasing its speed.

5. Place a piece of paper on the track and observe what happens when the robot runs over it.

6. **Ask** your TA to check if your program work.

7. **Questions for Section 8.1:**

    (a) At what point do you know (from watching the robot) that it has lost the line?

    (b) In what direction (left or right) is the robot turning when it loses the line?

(c) Consequently, what state is the robot in when it loses the line?

(d) Can the robot (running this specific program) lose the line while turning in the other direction? Why or why not?

(e) What would cause the robot to lose the line? Justify your answer.

(f) Suppose we were sitting inside the robot. How could we identify that this failure has occurred?

8.2 Failure Identification

We will now extend the "LineFollow" program by adding in a simple mechanism for detecting that the line has been lost. The failure that we need to identify occurs when the robot's ground proximity sensor either swings over the line too quick to detect it, or is confused by other debris covering the line. In both cases the sensor does not detect the line and the robot continues to turn in the same direction, assuming incorrectly, that its sensor has not yet reached the line. A close analogy is a driver missing the turn-off either because she was driving to fast or because the road is very foggy. The driver would identify such a failure mode by noting that she has been driving too long without seeing the turn-off and concludes that she has missed the turn-off.

1. Save a copy of the "LineFollow" program as "FailFollow".

2. Update the state-transition diagram for the program by adding an additional state (Lost). As you have already observed, the robot can only lose the line when turning right. I.e., it is in the Right state. Thus, there must be a transition from the Right state to the Lost state. The question is, when does this transition occur?

   Just like the driver, the robot can assume that it has lost the line, if it has been turning right (in the Right state) for too long. That is, a timer event has occurred. Thus, the transition from Right to Lost will happen as a result of a timer event. Once the robot enters the Lost state, it should, at least for now, beep to alert the user.

   We are now ready to make the modifications to our program based on our new state-transition diagram. **Be sure to include the diagram with your lab report.**

3. Make the following modifications to the program:

   (a) In the initialization code, after the variable declarations, initialize the period of **timer0** to 0. (For examples, see programs from previous tutorials.)

   (b) Add a new constant $LOST$ with a value of 3. I.e., it should be different from all other state identifiers.

   (c) At the end of the program add a **timer0** event handler that
       i. Sets $state$ to $LOST$,
       ii. Resets the period for **timer0** to 0, and
       iii. Generates a beep.

   (d) Immediately after the line that sets $state$ to $LEFT$, add a line that sets the period of **timer0** to 0.
       This line turns the identification mechanism off, because it is not needed when the robot is turning left, and in fact could adversely affect the program if triggered, i.e., like a false fire alarm.

(e) Similarly, in the `button.backward`, set the period of **timer0** to 0 as well.

(f) Immediately after the line that sets *state* to *RIGHT*, add a line that sets the period of **timer0** to 2 second (2000ms).

This line is essential because it turns the timer on and begins the countdown. Once two seconds has elapsed, the `timer0` event is triggered, and its handler transitions the system to the **Lost** state,

4. Save the program and load it onto the robot.

5. Try out this program on the track and see if it beeps when the line is lost. Note, the program makes no attempt, yet, to recover from the failure, all it does is identify that a failure has occurred.

6. **Ask** your TA to check if your program work.

7. **Questions for Section 8.2:**

(a) According to the state transition diagram, once the robot enters the **Lost** state, it can never leave.[1] Is this actually the case? There are two missing transitions in the state-transition, both emanating from the **Lost** state.

 i. Identify the transitions and add them to the state-transition diagram.

 ii. Justify these transitions by referring to the program to explain how they could occur.

(b) The length of time to wait before setting off the timer is really a function of speed. That is, we typically, want to wait at least until the robot turns almost 180 degrees before declaring the line lost. The reason for this is that some sharp right turns may cause the robot not to see the line for a long time. Since the time it takes for a robot to turn is a function of its speed, this is another place where the models we derived in the previous tutorials are useful. Using the model we derived for turning time as a function of degrees and speed, give formula for determining the time period the robot should wait before transitioning into the **Lost** state.

(c) Once we identify that the line is lost, how could the robot recover from this failure?

8.3 Failure Recovery

We will now complete the "`FailFollow`" program by adding in a simple recovery mechanism for returning to line that was lost. Recall, that our failure occurs when the robot's sensor misses the line because it was moving too quickly, or there was some debris on the line. We can deal with this in the same manner that a driver would deal with the failure of missing her turn-off: Simply backtrack and try again. This approach works remarkably well for many simple failure modes.

1. Until the robot recovers, it remains in the **Lost** state. Recovery ensues when the robot detects the line once again. Once it does, it can transition into the **Left** state, and continue as before. Consequently, our state-transition diagram should already be up to date. All we need to do is to implement the recovery behaviour associated with the **Lost** state, which in this case is simply reversing direction and backtracking to to the line.

---

[1]Like Hotel California.

2. Add code to the end of the `timer0` event handler to move in reverse of the present trajectory. That is, both the left and right motors should be assigned target settings that are negative of their current ones. That's all! Once the robot finds the line and transitions to the Left, the motors' target settings will again be set to forward motion during the transition.

3. Save the program and load it onto the robot.

4. Try out this program on the track and see if it recovers from its blunders.

5. **Ask** your TA to check if your program work.

6. **Questions for Section 8.3:**

   (a) Suppose the robot reaches the end of the line (not a loop). What happens?

   (b) Is it possible for the robot to get stuck in the same place and not make any progress? Why or why not?

   (c) Under what conditions could this recovery mechanism fail?

   (d) Could these identification and recovery mechanisms be used with your "`StopNFollow`" program? Why or why not? If yes, are there modifications that you would need to make to your program in addition to the ones described in your tutorial? If so, what are they? If not, why not?

8.4   Acknowledging Defeat

Suppose your robot comes to the end of the line. The robot will assume it lost the line and will attempt to recover. However, if the robot has arrived at the end of the line, then it will continue to search for the lost line that is no longer there, forever. This is not desirable. Ideally, the robot should attempt to locate the line, but if it does not succeed after a couple attempts, it should let the user know, stop, and save its batteries.

There are two ways to accomplish this. The first is to use a timer. For example, if the robot remains in the Lost state for more than ten seconds, the search is hopeless. The other approach is to use a counter. For example, if the robot enters the Lost state too many times in a given period of time, then there is clearly something wrong and the robot should respond to this. Typically, both approaches are used because they are complementary.

You will extend your "`FailFollow`" program by implementing one or both of these methods.

1. Save a copy of the "`FailFollow`" program as "`TryFollow`".

2. Update the state-transition diagram for the program by adding another transition from the Lost state to the Stopped state. The transition will be triggered by a timer expiring. That is, when the Lost state is entered, a timer will be set. If the timer expires while the system is still in the Lost state, then the robot can transition into the Stopped state.

3. Implement the additions to the state-transition diagram by modifying the `timer0` event handler. Instead of changing the state of the system to the Lost state, the handler should check, using an *if ... then ... else* statement if the system is already in the Lost state. If not, then the handler should transition to the Lost state, emit a beep, and set the motor speeds as

before, but instead disabling **timer0** by setting its period to 0, the period should be set to ten seconds, the amount of time after which the robot is expected to give up.

If the `timer0` event handler executes and the robot is in the Lost state, then the handler, should disable the timer and the motors, and transition to the Stopped state.

4. Save your program, load it on the robot, and give it a try. Try running your robot on a blank table top, with no tape. See if the robot stops after ten seconds.

5. We will now add a mechanism to stop the robot if it loses the line too often.

   Update the state-transition diagram for the program by adding another transition from the Lost state to the Stopped state. The transition will be triggered by a timer expiring in conjunction with a counter variable being too high.

6. Implement the additions to the state-transition diagram by adding a variable, adding a second timer and modifying the `timer0` event handler. First, add a variable called *LostCount*. In the `button.forward` handler initialize it to 0 and set the period of **timer1** to ten seconds. This will cause the second timer to go off every ten seconds.

   Second, whenever the robot transitions into the Lost state, increment the *LostCount* variable. Hence, we are counting the number of times that the robot loses the line.

   Third, add a `timer1` event handler and a constant called *MAX_LOST*, which should be set to 3. Using an *if ... then ... else* statement, the handler should compare *LostCount* to *MAX_LOST*. If *LostCount* is less than *MAX_LOST*, the handler should simply reset the variable to 0. Otherwise, the handler should transition the robot to the Stopped state and disable all motors and timers.

   Lastly, you may wish to disable **timer1** whenever the robot transitions to the Stopped state by setting its period to 0.

7. Save your program, load it on the robot, and give it a try. Try running your robot on a track and use a blank piece of paper to make the robot lose the line several times in quick succession. See if the robot stops.

8. **Ask** your TA to check if your program work.

9. **Questions for Section 8.4:**

   (a) Suppose the robot only had one timer instead of two. Is it possible to implement the same behaviour as above? If yes, how? If no, why not?

   (b) Currently, the timeout periods are set to 10 seconds.
      i. Why is this not necessarily a good idea?
      ii. Could you suggest a better way of setting the timeout period? Hint: What is the timeout period a function of?

9 Programming Techniques
-Subroutines and Development Cycle

In this tutorial you will learn about subroutines, a useful programming feature, and go through the process of solving a problem by analysis, design, implementation and testing. Subroutines provide a simple mechanism for making your programs simpler and avoiding replication of code. Solving a problem in robotics, as in any other area can be challenging. However, having a process to follow does make the task easier, but still requires significant work on your behalf.

9.1 Subroutines

A subroutine is a piece of code that can be called by other parts of the program. When the subroutine is called, it executes its code and then returns to the caller, from where it was called. As an analogy, think about how you check your email. That is, every time you check your email you do the same thing: stop what you are doing, get your phone, enter the access code, run the mail application, check the messages, lock the phone, put the phone back, go back to whatever you were doing. Because you check your email or text messages quite often, you most likely have a routine (subroutine) that you perform each time you do it. When a program needs to perform a specific task, it calls (executes) the subroutine, and when the subroutine completes, it returns and the program continues running from where it left off. Subroutines can be called from multiple locations in a program, which means that they can be used to organize your program into modules or parts, making it easier to both understand and to modify.

1. Load the "`TryFollow`" program that was implemented in the last tutorial.

2. Save a copy of the program as "`TryFollowSub`".

3. Observe that whenever the program transitions into the **Stopped** state, the program performs the same operations: perform transition, disable motors, and disable timers (see Figure 29).

```
state = STOPPED
motor.left.target = 0
motor.right.target = 0
timer.period[0] = 0
timer.period[1] = 0
```

Figure 29: Operations performed on transition to **Stopped** state.

The transition to the **Stopped** state occurs at least three times in the program, and whenever it occurs, the same code is repeated. This is a problem!

There are actually a couple problems. First, your program is longer than it needs to be, taking up more memory and seeming more complex than it needs to be. Second, adding new transitions to the **Stopped** state will require you to duplicate the code for each new transition. Third, any changes to the code associated with transitions into the **Stopped** state will need to be duplicated in multiple locations in the code. For example, recall from the last tutorial, that you had to update the code for all transitions into the **Stopped** state when you started

using the second timer. This adds further complexity to the program and potential for bugs. Ideally, there should be a single instance of the code to transition into the Stopped state. We can use a subroutine to solve these problems.

4. Immediately prior to the first event handler in your program define a subroutine called "to_stop" that performs the statements in Figure 29. A subroutine comprises three parts:

```
sub to_stop
  state = STOPPED
  motor.left.target = 0
  motor.right.target = 0
  timer.period[0] = 0
  timer.period[1] = 0
```

Figure 30: Operations performed on transition to Stopped state.

(i) the sub keyword, (ii) the subroutine name (a single alphanumeric string), and (iii) the subroutine body, comprising the statements to be executed when the subroutine is called.

It is strongly recommended that all subroutines be defined *before* any of the event handlers because subroutines must be defined before they can be called. However, subroutines must be defined *after* the variable declarations and initialization code. Consequently, the initialization code cannot call any subroutines.

5. In the event handlers, replace the code for transitioning into the Stopped state (see Figure 29) with a call to the subroutine. A call consists of the keyword callsub followed by the name of the subroutine, e.g., callsub to_stop

Observe that your code has become shorter, because the event handlers are shorter and easier to read, and if you do need to make changes to the transition into the Stopped state, you only need to make them in one place.

6. Save, load, and test the program. It should behave exactly the same way as the original program did.

7. **Ask** your TA to check if your program work.

8. **Questions for Section 9.1:**

   (a) What other programs that you have written over the past tutorials could be improved through the use of subroutines. Briefly justify your answer.

   (b) One use for subroutines is to make a subroutine for each state transition in your program. Is this a good idea? Why or why not?

9.2   A Spin through the Development Cycle

Developing a program for the robot to solve a given task can be a challenging experience. As we discussed previously, the world is uncertain and many things can go wrong, Furthermore, the robot

has to make decisions and perform actions in a given order which can be complicated in its own right. Fortunately, we can follow a process to make the task more manageable.

The process involves several steps: analyzing the problem, developing a strategy, designing a solution, implementing the solution, and testing the solution. For the remainder of this tutorial, we will run through this process.

First the problem. Suppose we wanted our robot to negotiate a maze composed of walls of Duplo blocks. Furthermore, we do not know the layout of the maze. We may be told that all passageways will be 20cm wide and that there are no cycles in the maze, but we do not have a map of the maze. How do we solve this problem?

1. We begin with an analysis of the problem. First, since the walls of the maze will be composed of Duplo blocks, we can use the robot's horizontal proximity sensors to sense the walls. Second, since we do not know the shape or layout of the maze, the robot will need to search the maze for the exit. Third, since all passage ways will be at least 20cm wide, the robot should have plenty of space to maneuver. Thus, we need a strategy for finding the maze exit.

2. Next, we develop a strategy. Our analysis reveals that our robot will need to search the maze for an exit. We now consider several strategies: the first one would be the random strategy: drive forward until encountering a wall, turn a random number of degrees, and drive forward again. Although this strategy may take a very long time to succeed, it will, with high probability, eventually succeed. Furthermore, the strategy is easy to implement—we did this in the first tutorial. Can we do better?

   Another strategy that would yield a more systematic (and faster search) is to use the right-hand rule. The idea is that the robot follows the wall on its right-hand size. For example, suppose we were in an unlit building. We walk until we walk into a wall; place our right hand on the wall; and start walking forward, keeping our hand on the wall. Eventually, assuming there are no cycles in the building, we will find the exit. Our robot can use the same strategy. This strategy is guaranteed to work in a bounded amount of time, because our robot will traverse each passageway in the maze at most twice (once in each direction). So, we will use this strategy.

3. We now need to design a solution based on this strategy. How do we implement a right-hand rule? What tactics should the robot use? Well, the robot must track the wall on its right.

   - If the robot senses no wall in front of it or on its right, the robot should move rightward, until it senses a wall.

   - If the robot senses no wall in front of it but a wall on its right, it should move along the wall, and perhaps slightly leftward to avoid crashing into the wall—once it stops sensing the wall on its right, it can start moving in a rightward direction again.

   - If the robot senses a wall in front of it, but no wall on its right, the robot should make a sharp right turn until it no longer senses the wall in front—the right-hand rule dictates that the robot must always turn right if it can.

   - If the robot senses a wall in front and on its right, then its only option is to make a sharp left turn.

Note how these four rules essentially define the right-hand rule strategy. We are getting closer to our solution, because if you observe carefully, the above four rules define the robot's states that it will use to implement the strategy.

We should also suggest that robot begin in a nonmoving state, and start moving rightward when the **Forward Button** is pressed. It should stop then the **Backward Button** is pressed.

4. We now derive the corresponding state-transition diagram. There will be five (5) states:

   Stopped Robot is motionless.

   Rightward Robot is moving in a rightward direction.

   Leftward Robot is moving in a leftward direction.

   RightTurn Robot is making a sharp right turn.

   LeftTurn Robot is making a sharp left turn.

   The rules in the previous step define the transitions. Note, that the robot can transition from any of the latter four states to any of the states, and it transitions from the Stopped state to the Rightward state.

   Submit this State transition diagram as part of your lab report.

5. Start a new program.

6. We now implement the solution, represented by the state-transition diagram as a program. We will only need to use two horizontal proximity sensors: $prox.horizontal[2]$ (front) and $prox.horizontal[4]$ (right). Based on the description of the transitions, the program will need to have three event handlers: `prox`, `button.forward`, and `button.backward`. As with our previous programs, your program will need to have a *state* variable, and several constants representing the states: $STOPPED$, $RIGHTWARD$ $LEFTWARD$, $RIGHTTURN$, and $LEFTTURN$, all with unique values. You will also need a $TARGET$ constant and a $THRESHOLD$ constant to define the speed of the robot and to compare the values returned by the horizontal proximity sensors. Use programs that you implemented previously as templates. Hint: To implement the transitions into the latter four states, your `prox` event handler will have four *when … do* statements, one for each of states.

   For the left turn and right turn settings, use the sharpest turn possible, i.e., one motor should be set to $TARGET$ and the other to $-TARGET$. For the Leftward and *Rightward* states the turns must be wider. In fact, the rightward turn should be wider than the leftward turn. You will need to try out a variety of values for the target settings. Hint: To make things consistent, the target setting of one motor should be a fraction of the other. For example, for the rightward turn, if the target setting for the left motor is $TARGET$, then the target setting for the right motor should be $TARGET \times \alpha$ where α is a fraction between 0 and 1, e.g., 0, $\frac{1}{2}$, 1, etc.

   Save your program as "`MazeRunner`".

7. Lastly, test and refine your program. Using the Duplo blocks provided by the lab facilitator, create a maze. Note that one wall (2 by 20 pips) is approximately 20cm wide. Run your robot through the maze. You may need to adjust the settings for your leftward and rightward

fractions of *TARGET*, as well as possibly the *TARGET* and *THRESHOLD* as well. This process may take some time.

8. Congratulations! You have completed a program that solves a specific task.

9. **Ask** your TA to check if your program work.

10. **Questions for Section 9.2:**

    (a) Give the state-transition diagram for the "Random" strategy described in Step 2.

    (b) For each of the *when ... do* statements in your program give the conditions under which the statement executes its body and identify the transitions associated with it.

    (c) What target settings did you use for the leftward and rightward motion?

    (d) Suppose the robot is placed in the middle of a large area in the maze such that there are no walls near by. What would happen once the **Forward Button** was pressed?

    (e) How long (how many trials) did it take you to determine the optimal settings?

    (f) Suppose we wanted the robot to stop at the exit to the maze, which is demarcated by black tape. How would this change the state-transition diagram and your program?

10 Advanced Concepts in Robotics
-PID Controller

In this tutorial, you will investigate a **PID Controller** as a common technique used to control a wide variety of machinery including vehicles, robots and even rockets. You will learn how to create a PID-controller to do line following. Once created, the same PID can be used, with only minor modifications, with any other Thymio applications.

In the example discussed here, we will limit ourselves to a single ground sensor. Even with a single sensor we should be able to build a robot that can track the line very precisely even if the line has curves in it. Similar to previous tutorials, we will try to follow the edge of the line and we will apply a **left hand line follower** method.

You will be provided with programs that make the robot follow a black line (electrical tape) using a *P controller*, *PI controller*, and *PID controller*. You will then experiment with the programs, change the constants parameters **Kp**,**Ki** and **Kd**, and derive a general idea for controlling a robot using PID controller.

Our goal is to find good parameters to get a good line follower robot which moves as fast as possible with the ability to handle lines with sharp corners. There are different parameters in PID controller as follows:

- **SETPOINT:** The $SETPOINT$ value (the average of white and black light sensor readings) is the constant value. You will need to measure the light levels (a *"black"* and a *"white"* value) with your robot, calculate the average, and put it into the $SETPOINT$ constant.

- **SPEED:** This parameter controls how fast the robot is moving along the line. The target power $SPEED$ value is a tunable parameter which has to be determined by trial and error. You will need to find the maximum $SPEED$ value that your robot can track the line with sharp curves. You will put the maximum speed value into the $SPEED$ constant.

- **Kp,Ki and Kd parameters**: These tunable parameters control how fast the robot is moving along the line edge when it has moved away from it.

Reminder: Doing math on a computer that doesn't know what a decimal point is: In the process of tuning Kp, Ki, and Kd, you will be changing their values up and down, and it might even be necessary to have fractional values. As Aseba does not support decimal numbers, it causes some challenge. Suppose you want to multiply variable "`error`" by the decimal number $Kp = 1.5$ for your P controller. You can not enter 1.5 so you can not try $Kp = 1.5$. It is pretty easy to fix the problem. The solution is to first perform a multiplication and then a division, so, for example, $(error \times 15)/10$ or $(error \times 2)/3$. In the program, we can define a constant value $Kp = KP/KPd$ and set the values $KP = 15$ and $KPd = 10$ for the value $Kp = 1.5$.

Integer values are used in Aseba to represent numbers, they are comprised between -32768 and 32767, which is the range for signed integer numbers that can be represented by a 16 bit binary word. You might need to use a native function "`math.muldiv`" to compute multiplication-division using internal 32-bit precision.

10.1 The "P" in "PID": Proportion(al)

1. Start a new program

```
var state = 0
var lightValue
var error
var p
var turn

motor.left.target  = 0 # reset motors
motor.right.target = 0

onevent button.forward
  state = 1

onevent button.backward      # on reverse button turn off motors
  state = 0
  motor.left.target  = 0
  motor.right.target = 0

onevent prox
  if state == 1 then
     lightValue = prox.ground.delta[0]
     error = lightValue - SETPOINT
     call math.muldiv(p, error,KP,KPd)

     turn = p
     motor.left.target  =  SPEED + turn
     motor.right.target =  SPEED - turn
  end
```

Figure 31: The P controller Program

2. Copy and paste the code listed in Figure 31.

3. Add the following constants"

   SPEED the target setting (tunable parameter) for the motors when the robot is moving in a straight line. Set to a value like 250.

   SETPOINT the average of white and black light sensor readings. You need to measure the light levels (a *"black"* and a *"white"* value) with your robot, calculate the average, and put it into the *SETPOINT* constant. You might find a value between 450 and 550.

   KP and *KPd* represent the Kp parameter in the P controller. We have guestimated a starting value for KP of maximum error (a value between 350 and 500) and a starting value for KPD of maximum target speed (500).

4. Save the program as "`LineFollow-P`".

5. Load the program on your robot.

6. Using a black tape, create a close loop on the table to be used as a track by the robot. It should have a couple of sharp curves in it and one or two straightaways.

7. Run the robot on the track to see how it performs.

8. Try adjusting the KP and KPd constants (Kp term in P controller). (Hint: Just tweak the KP value.)

9. Find the optimal Kp term for the P controller so that your line-follower robot moves as quickly as possible (less oscillation on straight lines) without losing track of the line.

10. **Ask** your TA to check if your program work.

11. **Questions for Section 10.1:**

    (a) Breifly explain how this program works.

    (b) Compare this program (code and performance) with the "`LineFollow`" program from the tutorial 5. (You may need to load that program on your robot and see how it performs.)

    (c) What is the behaviour of your robot in straight lines and curves when you increase or decrease the Kp term?

    (d) If $SPEED$ is bigger than that maximum, the robot will lose the line when it encounters a sharp curve. Can you solve the problem by changing the Kp value? Justify your answer.

10.2 Adding "I" To The Controller: The PI Controller

We want to improve our p controller. We will add the *integral* term, the **I** in PID, as the running sum of the error. Each time we calculate an error, we will add it to a variable. We call this variable as **integral**. The integral can be a large factor in the error correction and also helps to remove small errors.

1. Start a new program

2. Copy and paste the code listed in Figure 32.

3. Add the following constants"

    $SPEED$ the target setting (tunable parameter) for the motors when the robot is moving in a straight line. Set to a value like 250.

    $SETPOINT$ the average of white and black light sensor readings. You need to measure the light levels (a "black" and a "white" value) with your robot, calculate the average, and put it into the $SETPOINT$ constant. Set the value you have found for the P-controller in the previous experiment.

    KP and KPd represent the Kp parameter in the PI controller. We have guestimated a starting value for Kp is half of the Kp value you found for the P controller. (Hint: Set the value of KPd into 100 and change the KP value.)

```
var state = 0
var lightValue
var error
var p
var turn
var integral = 0
var i
var sign
var lastsign

motor.left.target  = 0
motor.right.target = 0

onevent button.forward
  state = 1
  integral = 0

onevent button.backward
  state = 0
  motor.left.target  = 0
  motor.right.target = 0

onevent prox
  if state == 1 then
     lightValue = prox.ground.delta[0]
     error = lightValue - SETPOINT
     call math.muldiv(p, error,KP,KPd)
     integral = integral + error
     call math.muldiv(i, integral,KI,KId)
     if error>=0 then sign = 0 else sign = 1 end
     if error == 0 or sign!=lastsign then integral = 0 end
     lastsign = sign

     turn = p + i
     motor.left.target  =  SPEED + turn
     motor.right.target =  SPEED - turn
  end
```

Figure 32: The PI controller Program

KI and KId represent the Ki parameter in the PI controller. We have guestimated a starting value for Ki is one-tenth of Kp value. (Hint: Set the value of KId into 1000 and change the KI value.)

4. Save the program as "`LineFollow-PI`".

5. Load the program on your robot.

6. Run the robot on the track to see how it performs.

7. Try adjusting the KP and KI constants (Kp and Ki terms in PI controller).

8. Find the optimal Kp and Ki terms for the PI controller so that your line-follower robot moves as quickly as possible (less oscillation on straight lines) without loosing track of the line .

9. **Ask** your TA to check the results and if your program work.

10. **Questions for Section 10.2:**

    (a) Breifly explain how this program works.

    (b) Compare this program with the "`LineFollow-P`" program.

    (c) Why we need to zero the **integral** variable?

    (d) Without changing the optimal Kp value, what is the behaviour of your robot in straight lines and curves when you increase or decrease the Ki term?

    (e) Without changing the optimal Ki value, what is the behaviour of your robot in straight lines and curves when you increase or decrease the Kp term?

10.3   Adding "D" To The Controller: The PID Controller

We already have a controller contains a *proportional (Kp)* term which tries to correct the current error and an *integral (Ki)* term that tries to correct past errors. The *derivative (Kd)* term tries to look ahead in time and to correct the error that hasn't occurred yet.

1. Start a new program

2. Copy and paste the code listed in Figure 33.

3. Add the following constants"

    SPEED the target setting (tunable parameter) for the motors when the robot is moving in a straight line. Set to a value like 250.

    SETPOINT the average of white and black light sensor readings. You need to measure the light levels (a *"black"* and a *"white"* value) with your robot, calculate the average, and put it into the *SETPOINT* constant. Set the value you have found for the P-controller/PI controller in the previous experiment.

    KP and *KPd* represent the *Kp* parameter in the PID controller. We have guestimated a starting value for *Kp* is less than the *Kp* value you found for the PI controller. (Hint: Set the value of *KPd* into 100 and change the *KP* value.)

```
var state = 0
var lightValue
var error
var p
var turn
var integral = 0
var i
var sign
var lastsign
var derivative
var d
var lasterror

motor.left.target  = 0
motor.right.target = 0

onevent button.forward
  state = 1
  integral = 0

onevent button.backward
  state = 0
  motor.left.target  = 0
  motor.right.target = 0

onevent prox
  if state == 1 then
     lightValue = prox.ground.delta[0]
     error = lightValue - SETPOINT
     call math.muldiv(p, error,KP,KPd)
     integral = integral + error
     call math.muldiv(i, integral,KI,KId)
     if error>=0 then sign = 0 else sign = 1 end
     if error == 0 or sign!=lastsign then integral = 0 end
     lastsign = sign
     derivative = error - lasterror
     call math.muldiv(d, derivative,KD,KDd)
     lasterror = error

     turn = p + i + d
     motor.left.target  =  SPEED + turn
     motor.right.target =  SPEED - turn
  end
```

Figure 33: The PID controller Program

KI and KId represent the Ki parameter in the PID controller. We have guestimated a starting value for Ki is greater than the Ki value you found for the PI controller. (Hint: Set the value of KId into 1000 and change the KI value.)

KD and KDd represent the Kd parameter in the PID controller. We have guestimated a starting value for Kd is around the Ki value you set for Ki for PID this PID controller. (Hint: Set the value of KDd into 1000 and change the KD value.)

4. Save the program as "`LineFollow-PID`".

5. Load the program on your robot.

6. Run the robot on the track to see how it performs.

7. Try adjusting the KP, KI and KD constants (Kp, Ki and Kd terms in PID controller).

8. Find the optimal Kp, Ki and Kd terms for the PID controller so that your line-follower robot moves as quickly as possible (less oscillation on straight lines) without losing track of the line.

9. **Ask** your TA to check the results and if your program work.

10. **Questions for Section 10.3:**

    (a) Breifly explain how this program works.

    (b) Compare this program with the "`LineFollow-PI`" program.

    (c) Why we need to save the **error** variable into the **lasterror** variable?

    (d) Without changing the optimal Kp and Ki values, what is the behaviour of your robot in straight lines and curves when you increase or decrease the Kd term?

11 Advanced Concepts in Robotics
-Localization

In this tutorial, you will implement a simple version of **Markov Localization** as a technique in robotics used to integrate sensor and internal model information to estimate the localization of a robot within a map. You will learn how the robot can know where it is. This tutorial is taken from programming examples and projects at `http://www.weizmann.ac.il/sci-tea/benari/`.

Your Thymio robot navigates within a known environment for which it has a *map*. The task of the Thymio is to find its location. So how can the robot know where it is? The purpose of a *localization* algorithm is to determine the position within the known environment from sensor readings and internal knowledge of mast moves.

You will be provided with programs that make the robot to determine its position in a circle with five dark gray circular sectors (doors on the wall) and three light gray circular sectors (no door). Figure 34 shows a map with five doors (dark gray sectors) and three sectors where there is no door (light gray sectors). Note that the environment is periodic so that there is no start or end of the sequence of doors and walls.
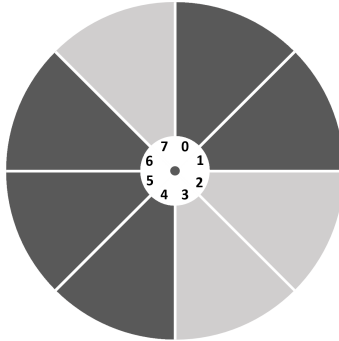

Figure 34: A map with five doors (dark gray sectors) and three sectors where there is no door (light gray sectors).

Your task is to understand and experiment with the programs, and derive a general idea for uncertainty in sensing and motion. The task for our Thymio is to find a specific door (a dark gray circular sector) within a known environment without knowing the starting position.

The *map* in this tutorial contains eight positions (five dark grays for positions that we label 0,1,4,5,and 6, and three light grays for positions 2,3 and 7) and Thymio must assign a probability to each position where it might be located. Initially, our robot has no idea where it is (uncertainty), so it assigns each position the probability $1/8 = 0.125$, which corresponds to a uniform distribution:
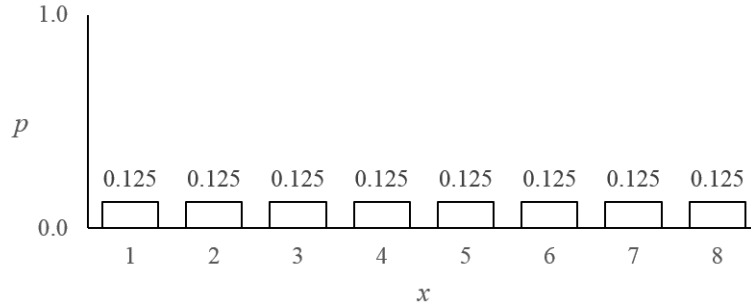
Now suppose that the Thymio's sensors detect a dark gray area. As the robot knows that it is in front of one of the five doors, its uncertainty is reduced. The probability distribution shows $1/5 = 0.2$ for each of the doors (dark gray areas) and 0 for each of the walls (light gray areas):
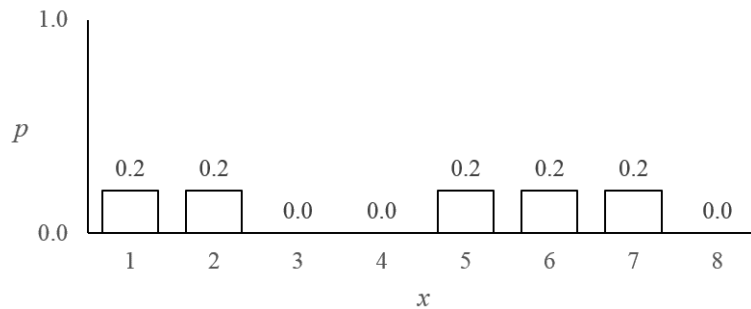
Next our Thymio moves and again senses a dark gray area. There are now only three possibilities: it was at position 0 and moved to 1, it was at 4 and moved to 5, or it was at 5 and moved to 6. If the robot's initial position was 6, after moving right it would no longer detect a dark gray area, so it could not have been there. The probability is now $1/3 = 0.33$ for each of these three positions:
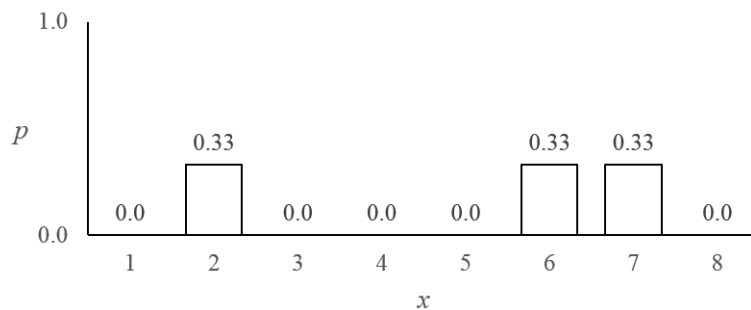
After the next step of the robot, if it again detects a door (dark gray area), it is without a doubt at position 6, while if it does not detect a door, it is either at position 2 or position 7.

Doing math on a computer that doesn't know what a decimal point is: As Aseba does not support decimal numbers, we need some workaround to represent the fractional probabilities. Suppose you want to assign a probability $1/8 = 0.125$. You can not enter 0.125 so you can not assign it to a variable. However, it is pretty easy to fix the problem. The solution is to first perform a multiplication and then a division, so, for example, $(1 \times 1000)/8 = 125$. As a result, the probabilities for the positions are scaled to the range 0 through 1000. In this way, the probabilities are measured in units per million. You might also be familiar with the representation of probabilities in % (percent or per hundred). By increasing the number we divide by we are increasing the precision with which we can represent the fractional values with integer numbers.

BTW, integer values are used in Aseba to represent numbers, they are comprised between -32768 and 32767, which is the range of 16 bit signed integers. You might need to use a native function "`math.muldiv`" to compute multiplication-division using internal 32 bit precision.

11.1 Simple localization program

1. Start a new program.

2. Copy and paste the code listed in Figure 35.

3. Add the following constants:

   LED scale factor for displaying beliefs in leds. Set to a value 32.

   THRESHOLD value for detecting dark gray.

   HIT value for detecting dark gray when on dark gray or light gray when on light gray. Set to a value 1.

   MISS value for detecting dark gray when on light gray or light gray when on dark gray. Set to a value 0.

4. Save the program as "`Localization-simple`".

5. Load the program on your robot.

6. Press the **center button**, look at LEDs and read the values in the *beliefs* array. **Explain your experience as part of your lab report.**

7. Place the robot on one of the circular sectors in the map shown in Figure 36 (located at the end of this tutorial) so that proximity ground sensor 1 is directly over the circular sector. Then press the **forward button** and check LEDs as well as the values in the *beliefs* array.

8. Repeat (16 times) the previous step (step 7) by placing the robot over the next right circular sector and **explain the results in your lab report as well**.

9. **Ask** your TA to check the results and if your program work.

10. **Questions for Section 11.1:**

    (a) Breifly explain how this program works.

    (b) Why we need *rotate* subroutine? Justify your answer.

```
var world[8]= [1,1,0,0,1,1,1,0]
var beliefs[8]
var i
var hit
var temp
var sum

onevent button.center
    when button.center == 0 do
          call math.fill(beliefs, 1000/8)
          callsub display_beliefs
    end
onevent button.forward
    when button.forward == 0 do
          callsub rotate
          callsub sense
          callsub display_beliefs
    end

sub display_beliefs
    call leds.circle( beliefs[0]/LED, beliefs[1]/LED, beliefs[2]/LED,
                      beliefs[3]/LED, beliefs[4]/LED, beliefs[5]/LED,
                      beliefs[6]/LED, beliefs[7]/LED)
sub rotate
    temp = beliefs[7]
    for i in 7:1 step -1 do beliefs[i] = beliefs[i-1] end
    beliefs[0] = temp
sub sense
    if  prox.ground.delta[1] < THRESHOLD  then hit = 1 else hit = 0 end
    for i in 0:7 do
          if  (hit==1 and world[i]==1) or (hit==0 and world[i]==0) then
               call math.mul(beliefs[i], beliefs[i], HIT)
          else
               call math.mul(beliefs[i], beliefs[i], MISS)
          end
    end
    callsub normalize
sub normalize
    sum = 0
    for i in 0:7 do sum += beliefs[i] end
    for i in 0:7 do call math.muldiv(beliefs[i], beliefs[i],1000,sum) end
```

Figure 35: The simple localization program

(c) What is the role of *normalize* subroutine? Why we **must** use it?

(d) Press the **center button** again and repeat the experiment without moving the robot and see if the algorithm works. Explain your observations.

11.2 Uncertainty in sensing

As you have experienced, the robot cannot detect with complete certainty whether it senses a dark gray area or a light gray sector. Such uncertainties happen for many reasons such as fading colors for walls and doors, different lighting conditions, or noise in the sensors of the robot so that it could not distinguish between light and dark gray in question *(d)*. We can model such uncertainties by assigning probabilities to the sensor detection. If the robot senses dark gray sector, we specify that the probability is 0.9 that it has (correctly) detected a door and 0.1 that it has (mistakenly) detected a wall. Conversely, if it senses light gray, the probabilities are 0.1 for a door and 0.9 for a wall.

1. Save a copy of the "`Localization-simple`" program as "`Localization-uncertain`".

2. Make modifications to the program that can accept $HIT = 0.9$ and $MISS = 0.1$. Hint: You can use "`math.muldiv`" function.

3. Save your program and load it on your robot.

4. Press the **center button**, look at LEDs and read the values in the *beliefs* array. **Explain your experience as part of your lab report.**

5. Place the robot on one of the circular sectors in the map shown in Figure 34 (printable version located at the end of this tutorial) so that proximity ground sensor 1 is directly over the circular sector. Then press the **forward button** and check LEDs as well as the values in the *beliefs* array.

6. Repeat (16 times) the previous step (step 5) by placing the robot over the next right circular sector and **explain the results in your lab report as well**.

7. **Ask** your TA to check the results and if your program work.

8. **Questions for Section 11.2:**

   (a) What is the difference between the results of this program and previous one?

   (b) Press the **center button** again and repeat the experiment without moving the robot and see if the algorithm works. Explain your observations..

   (c) Experiment with different values of HIT and MISS (HIT+MISS=1). How low can you specify HIT and have the algorithm still work?

11.3 Localization of a moving robot

Let us now test the localization algorithm on the moving robot. We want our robot to find a specific location on the map (i.e. position 4) and generates a beep.

1. Save a copy of the "`Localization-uncertain`" program as "`Localization-uncertain-motion`".

2. Add a constant variable LOC for the specific location on the map.

3. Make modifications to the program that can **rotate right** on the map (the right wheel should be on the black dot at the center of the map). The robot should stop on each circular sector and show the probability of its location on circle LEDs. After finding the location LOC, the robot must generate a beep.

4. Save your program and load it on your robot.

5. **Ask** your TA to check if your program work. Your TA places the robot on a random circular sector and your robot must find the specific location LOC.
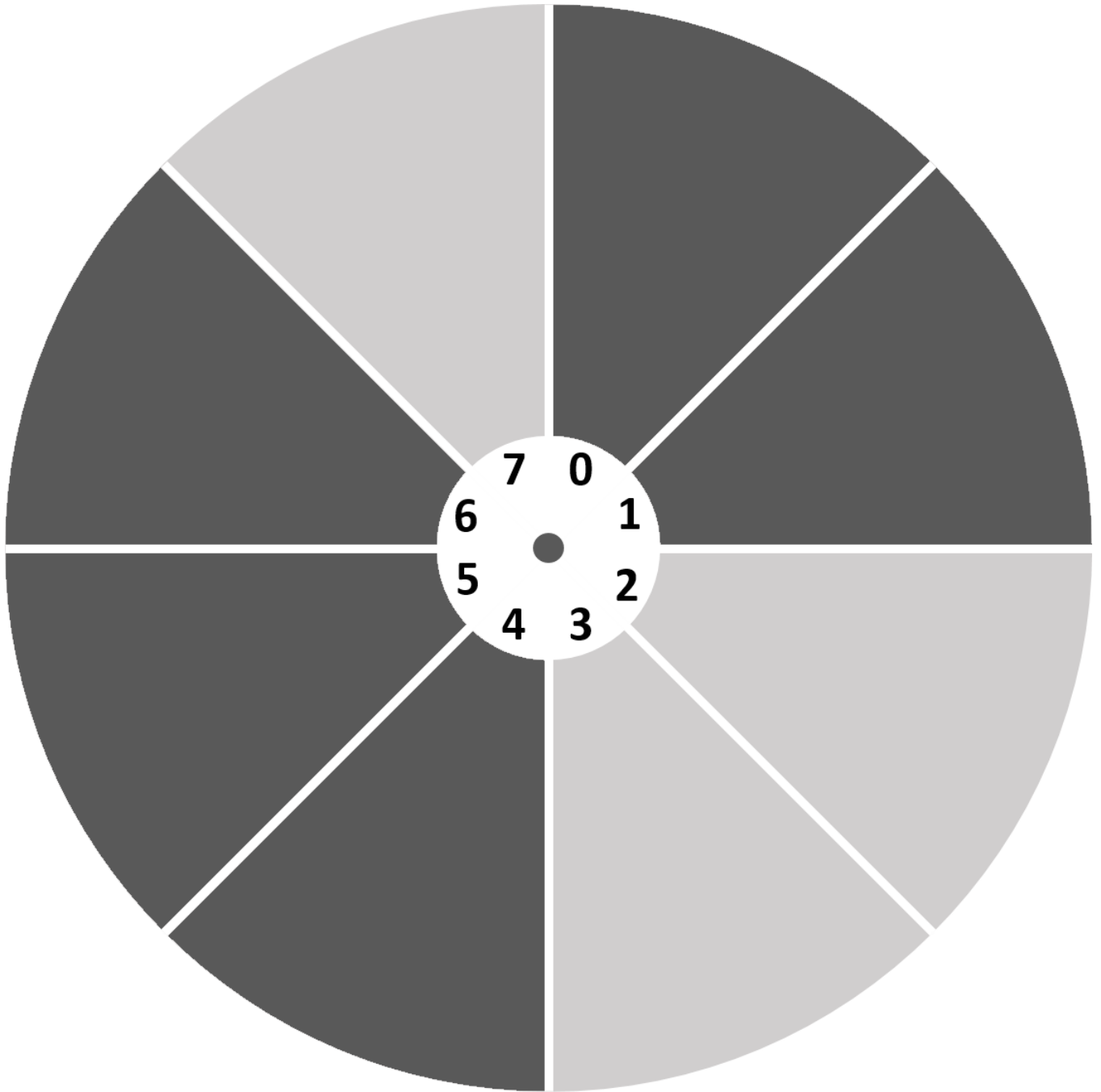
Figure 36: A printable map with five doors (dark gray sectors) and three sectors where there is no door (light gray sectors).

12 Advanced Concepts in Robotics
-Object Recognition

Robots must interact with the environment and usually must be able to detect specific objects. Most robots are therefore equipped with some form of a camera and some form of program to recognize objects in images. Indeed, our proximity sensors can be seen as a simple form of a camera. In this tutorial, you will investigate methods for object recognition using the proximity sensors in different ways to detect specific signatures of objects.

As usual, we will provide you with some example *programs* that make use of the proximity sensors, and we will be using LEDs on the circle to communicate the robots' observation. You will then experiment with the programs and derive a general idea for object recognition.

12.1 Recognizing objects with different colors and shape

In this experiment, you will measure the width of the bands (black and gray bands in Figure 39 located at the end of this tutorial) using ground sensors. Moreover, you will identify the color of each band.

The Thymio travels across the band. When it detects a band (Gray or Black) during a *prox* event, a counter is set to zero. The counter is then incremented regularly during each subsequent move until the band is no longer detected. The robot shows the width of the band as measured by the value of the counter on the circle LEDs. Moreover, the robot will also report the detected color on the top RGB LED. You will have to experiment with the parameters of this experiment to get it to work properly.

1. Start a new program.

2. Copy and paste the code listed in Figure 37.

3. Add the following constants"

    $STOPPED$, $SEARCH$, $BLACK$, $GRAY$ the Stopped, Search, Black, and Gray state identifiers.

    $TARGET$ the target setting for the motors when the robot is moving.

    $BLACK\_TH$ **and** $GRAY\_TH$ the thresholds use to determine whether the ground sensors are sensing Black or Gray.

4. Save the program as "`color-code`" and load the program on your robot.

5. Place the robot on one of the white areas toward a band so that ground sensors are directly over the white areas in Figure 39 (located at the end of this tutorial).Then press the **forward button** and check top RGB LED and circle LEDs. You **must** adjust the *TARGET*, *BLACK_TH* and *GRAY_TH* constants in a way that your robot shows the width of bands on circle LEDs as well as the color on top RGB LED (blue for black bands and green for gray bands).

6. **Ask** your TA to check if your robot can distinguish between different colors as well as shows the width of different bands correctly.

```
var state = STOPPED
var width = 0
var reading

onevent button.forward
  if button.forward == 0 then
    state = SEARCH
    motor.left.target = TARGET
    motor.right.target = TARGET
    callsub set_circle_leds
  end

onevent button.backward
  callsub stop
  call leds.top(0, 0, 0)

onevent prox
  if  state != STOPPED then
    reading = (prox.ground.delta[0] + prox.ground.delta[1]) / 2

    when state == SEARCH and reading <= BLACK_TH do
      state = BLACK
    end

    when state==SEARCH and reading<=GRAY_TH and reading>BLACK_TH do
      state = GRAY
    end

    if state==BLACK then
      if reading <= BLACK_TH then
        width++
        call leds.top(0,0,32)
      else
        callsub stop
      end
    end

    if state==GRAY then
      if reading <= GRAY_TH then
        width++
        call leds.top(0,32,0)
      else
        callsub stop
      end
    end
  end
```

```
sub stop
  state = STOPPED
  motor.left.target = 0
  motor.right.target = 0
  callsub set_circle_leds
  width = 0

sub set_circle_leds
  if width==0 then call leds.circle(0,0,0,0,0,0,0,0) end
  if width==1 then call leds.circle(32,0,0,0,0,0,0,0) end
  if width==2 then call leds.circle(32,32,0,0,0,0,0,0) end
  if width==3 then call leds.circle(32,32,32,0,0,0,0,0) end
  if width==4 then call leds.circle(32,32,32,32,0,0,0,0) end
  if width==5 then call leds.circle(32,32,32,32,32,0,0,0) end
  if width==6 then call leds.circle(32,32,32,32,32,32,0,0) end
  if width==7 then call leds.circle(32,32,32,32,32,32,32,0) end
  if width==8 then call leds.circle(32,32,32,32,32,32,32,32) end
```

Figure 37: The simple color code program

7. **Questions for Section 12.1:**

   (a) Briefly explain how this program works.

   (b) What happens if your robot calculates the width greater than eight? **Justify your answer.**

   (c) Suppose that you want to show the width greater than eight. What strategy will you take? **Justify your answer.**

12.2   Simple barcode reading program

In this experiment, you will investigate how to read a barcode consisting of different strips of black and white with different width using just one proximity horizontal sensor. The robot scans the barcode using just one sensor (horizontal sensor 2) and shows the barcode on circle LEDs.

1. Start a new program.

2. Copy and paste the code listed in Figure 38.

3. Add the following constants"

   $STOPPED, SEARCH, LEFT, RIGHT$ the Stopped, Search, Left, and Right state identifiers.

   $TARGET$ the target setting for the motors when the robot is scanning.

   $THRESHOLD$ uses to determine whether the horizontal sensors are sensing the barcode.

   $LENGTH$ the length of array for storing samples. Set the value to 100.

LEN the maximum width of tiny white strips. Set the value to 7 (you might need to change the value based on the *TARGTE* value).

4. Save the program as "`barcode-reader`".

5. Load the program on your robot.

6. Stack together four 8×2 Duplo blocks and cover the Duplo wall with one of the barcodes in Figure 40 (located at the end of this tutorial).

7. Place the robot in front of the barcode (directly parallel to the wall) so that the horizontal sensor 2 is located around one-centimeter distance from the middle of the wall. Then press the **forward button**.

8. You **must** adjust the *TARGET* and *THRESHOLD* constants in a way that your robot shows the barcodes on circle LEDs. Hint: Your robot should turn on LEDs for white areas and turn off LEDs for black areas).

9. **Ask** your TA to check if your robot can show **all** different barcodes on circle LEDs correctly.

10. **Questions for Section 12.2:**

    (a) Briefly explain how this program works.

    (b) Try to change the distance between the horizontal sensor 2 and the wall and find the minimum distance that your robot can read the barcode correctly or it can distinguish different barcodes.

    (c) Try to change the initial position of the robot (the wall is not parallel to the wall or the sensor is not located in front of the center of the wall). Can your robot read all barcodes correctly or distinguish different barcodes? **Justify your answer.**

```
var min
var max
var mean
var i=0
var j=0
var len=0
var samples[LENGTH]
var pattern[8]
var state= STOPPED

onevent button.backward
    callsub stop

onevent button.forward
    state = LEFT
    motor.left.target = -TARGET
    motor.right.target = TARGET
    i = 0
    call math.fill(samples, 0)
    call math.fill(pattern, 0)
    call leds.top(0,0,0)
    call leds.circle(pattern[0],pattern[1],pattern[2],pattern[3],
                     pattern[4],pattern[5],pattern[6],pattern[7])

onevent prox
    if state != STOPPED then
        call math.stat( prox.horizontal[0:4], min, max, mean )
        when state == LEFT and  mean == 0 do
            state = RIGHT
            motor.left.target = TARGET
            motor.right.target = -TARGET
        end
        when state == RIGHT and mean > 0 do
            call leds.top(0,32,0)
            state = SEARCH
        end
        if state == SEARCH then
            samples[i] = prox.horizontal[2]
            i++
        end
        when state == SEARCH and mean == 0 do
            callsub stop
            callsub compute
        end
    end
```

```
sub stop
    call leds.top(0,0,0)
    motor.left.target = 0
    motor.right.target = 0
    state = STOPPED

sub compute
    i=0
    for j in 7:0 step -1 do
        if samples[i] < THRESHOLD then
            while samples[i] < THRESHOLD and i<LENGTH-1 do
                i++
                pattern[j]=0
            end
        else
            len=1
            while samples[i] >= THRESHOLD and i<LENGTH-1 and len <LEN do
                i++
                len++
                pattern[j]=32
            end
        end
    end
    call leds.circle(pattern[0],pattern[1],pattern[2],pattern[3],
                     pattern[4],pattern[5],pattern[6],pattern[7])
```

Figure 38: The simple barcode reader program

12.3   Image recognition

In this experiment, you will investigate how a robot can use its proximity sensors as a camera for image recognition. You should develop a simple program that reads a pattern shown in Figure 41 (located at the end of this tutorial).

Same as the previous experiment, stack together four 8×2 Duplo blocks and cover the Duplo wall with one of the patterns in Figure 41. Then place the robot in front of the wall and directly parallel to the wall so that the horizontal sensor 2 is located around one-centimeter distance from the middle of the wall. Then read the value of all front horizontal sensors (**at least horizontal sensors 1,2 and 3**) and display their values on circle LEDs 2 to 6 (or circle LEDs 3 to 5 if you use sensors 1,2 and 3). For each pattern, you should put a different color on top RGB LED.

1. Start a new program.

2. Build a program that your robot reads a pattern shown in Figure 41 using at least three front horizontal sensors. We named these pattern based on different color codes (RED, GREEN,

BLUE, and WHITE). Your robot should display the observed pattern on circle LEDs number 2 to 6 when the **forward button** is pressed. Moreover, the robot should identify different patterns by displaying the color code on top RGB LED. Hint: your robot does not move in this experiment. Moreover, your robot must observe patterns using **at least three horizontal sensors (sensors 1,2 and 3)**.

3. Save the program as "`image-recognition`".

4. Load the program on your robot.

5. Stack together four 8×2 Duplo blocks and cover the Duplo wall with one of the patterns in Figure 41 (located at the end of this tutorial).

6. Place the robot in front of the wall (directly parallel to the wall) so that the horizontal sensor 2 is located around one-centimeter distance from the middle of the wall. Then press the **forward button**.

7. **Ask** your TA to check if your robot can show **all** different patterns on circle LEDs correctly as well as on top RGB LED.

8. **Questions for Section 12.3:**

   (a) Try to change the distance between the horizontal sensor 2 and the wall and find the minimum/maximum distance that your robot can read the pattern correctly or it can distinguish different patterns.

   (b) Try to change the initial position of the robot (the wall is not parallel to the wall or the sensor is not located in front of the center of the wall). Can your robot read all patterns correctly or distinguish different patterns? **Justify your answer.**

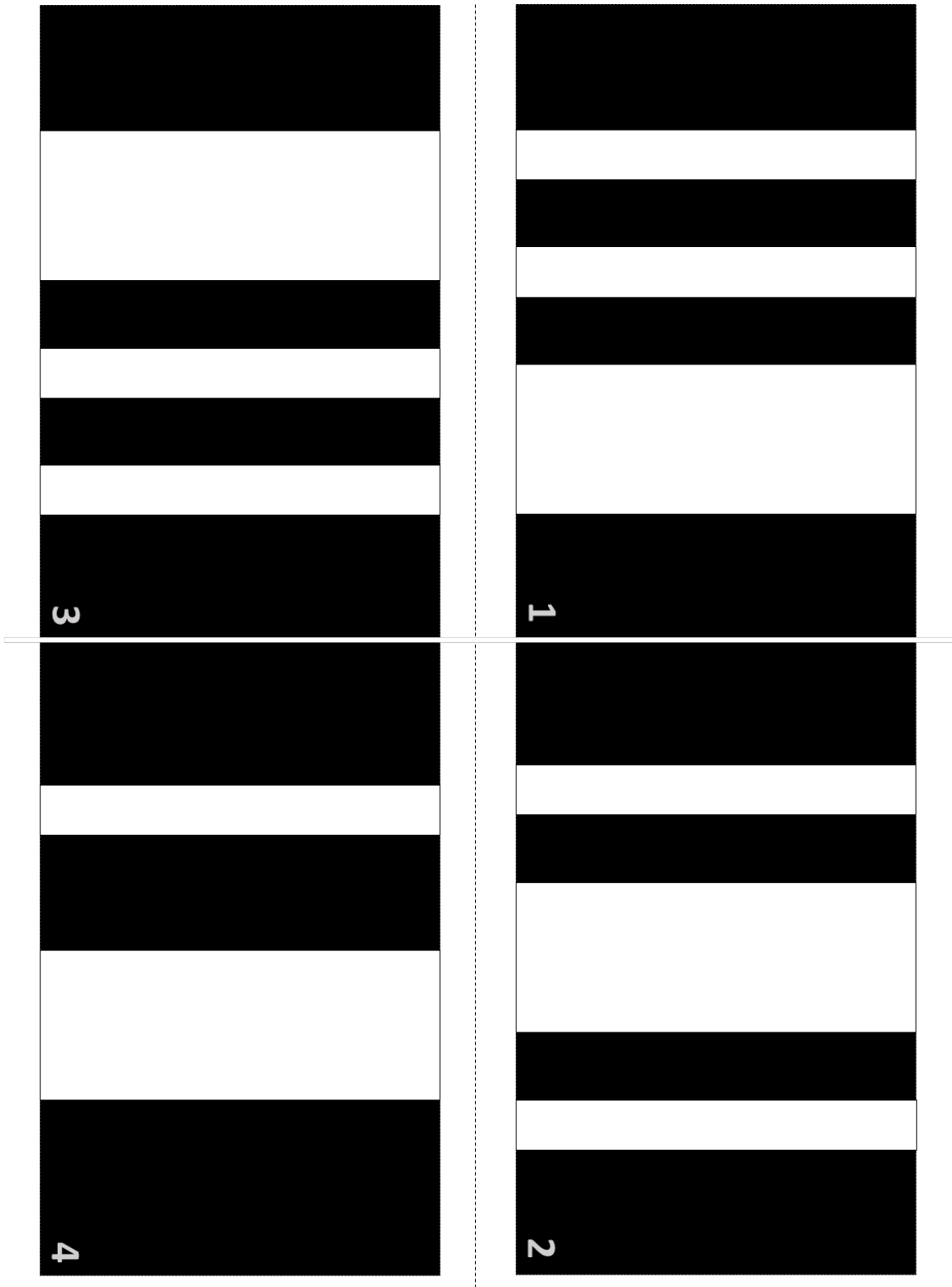Figure 39: A printable color codes for the simple color code program.

Figure 40: A printable barcodes for the barcode reader program.
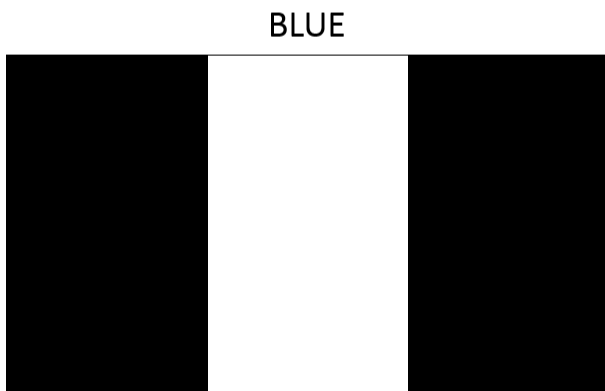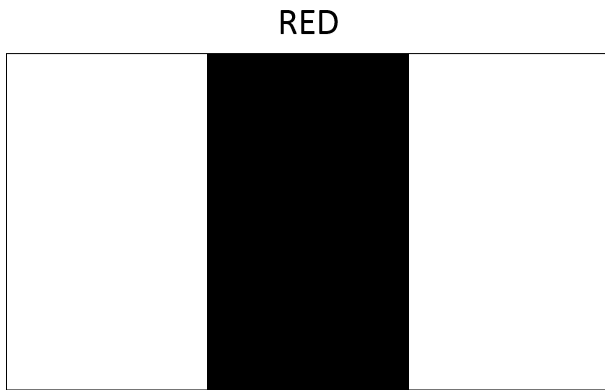
RED

GREEN

BLUE

WHITE

Figure 41: A printable patterns for the image recognition program.