Projects for the Thymio Robot in the Aseba Studio Environment

Version 1.9.0

Moti Ben-Ari

http://www.weizmann.ac.il/sci-tea/benari/

and other contributors (see `authors.txt`)

Contents

Chapter 1

Introduction

This document describes projects for the Thymio robot using *AESL*, the textual programming language of Aseba, and its *Studio* development environment. The document assumes that you are familiar with the Thymio robot, for example, by constructing projects using VPL, the visual programming language. You must understand the concept of an *event* which causes an *action*. Elementary versions of some of these projects are contained in the VPL tutorial; here we present better solutions that use the capabilities of AESL.

Project structure

Each chapter is structured as follows:

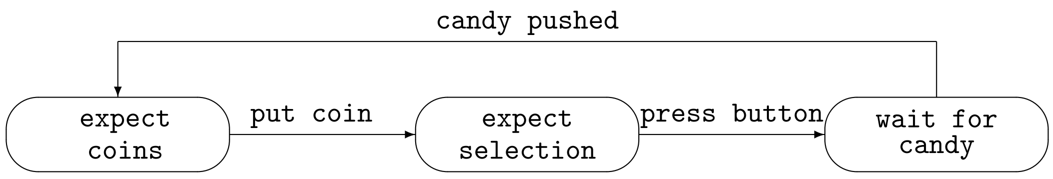
- Specification of the problem to be solved.
- Design of the program.
- Construction of a state machine for the robot.
- Declarations of constants and variables.
- Description of the event handlers and subroutines.
- Programming notes.
- Experiments for you to do.

I strongly suggest that you work on each project by yourself and only then compare your solution to mine.

State machines

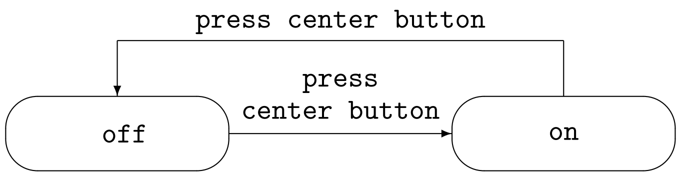
You stand in front of a vending machine and wait for a candy bar to be pushed out. Nothing happens. You push a button to select which candy you want. Nothing happens. Finally, you put some coins in the machine. Nothing happens. Again, you push a button to make a selection. Now, the machine starts working and the candy is pushed out.

A vending machine is really a robot; it's task is to give candy to people. The task of a vending machine is divided into smaller subtasks and only one subtask is active at any time. We say that the machine can be in several *states* and that there are conditions that determine when the machine makes a *transition* from one state to the next. The states and transitions can be described in the diagram of a *state machine*, where the states are the ovals and the transitions are the arrows:



The state machine makes it clear why you won't receive any candy just by standing in front of the vending machine. Candy is pushed out only in the rightmost state which can be reached only after putting coins and making a selection.

All our projects use state machines to describe the behavior of the robots. The state machine can be as simple as:



but usually it will be more complex because the robot—like a vending machine—must perform several subtasks.

Overview of the projects

The document is divided into two parts. The first part (Chapters 2–7) presents projects similar to those you constructed in VPL, extended to use the additional features of the Thymio robot that can be accessed from AESL. The second part (Chapters 8–11) introduces advanced concepts in robotics. Although the Thymio is too simple to fully explore these concepts, it is powerful enough so that you can get a taste of what advanced work in robotics is like.

Chapter 2: The cat catches the mouse The robot is a cat which searches for and catches a mouse. This type of requirement is common when a robot has to locate an object and move it someplace.

Chapter 3: How fast is the Thymio? The performance of the robot is measured. We compute the speed and acceleration of the robot, and the relationship between the amount of power applied to the motors and the acceleration of the robot.

Chapter 4: Line following An autonomous robot must be able to *navigate*, that is, to move from its current position to a new position, for example, by following a line displayed on the floor.

Chapter 5: Composing music on-the-fly The Thymio contains accelerometers which can be used to measure both the attitude of the robot (because gravity is an acceleration that always points downwards) and to sense if the robot is hit or shaken. This project uses the accelerations obtained when the robot is shaken to compose music.

Chapter 6: Odometry Odometry is the calculation of a robot's position and heading obtained by starting at an initial position and using the measured velocity and time.

Chapter 7: Image recognition A robot must sense its environment in order to carry out its tasks. This project demonstrates how a robot can search for a particular object by recognizing its barcode.

Chapter 8: Nonlinearity A sensor is linear if the values it returns are a linear function of what it measures. For example, a linear distance sensor might return the value $50 \cdot d$, where d is the distance to the object. It will return 50 if an object is 1 meter away and 100 if the object were 2 meters away. The nonlinearity of a sensor can be measured and used to calibrate the sensor.

Chapter 9: Odometry in Two Dimensions A wheeled robot that travels in a two-dimensional plane can turn. This makes odometry calculations more complex.

Chapter 10: Control An autonomous robot must make decisions, such as in what direction to turn and how fast to travel. The robot uses control algorithms such as proportional control to make these decisions.

Chapter 11: Fuzzy logic The algorithms in the previous chapter used precise mathematical computations to control the robot. Fuzzy logic uses imprecise language to specify control algorithms.

Chapter 12: Localization Consider a robot is navigating within a known environment such as a building for which it has a *map*. The robot uses *localization* algorithms in order to determine its position within the environment.

Chapter 13: Surveying This project uses the Thymio to determine the location of an object using two techniques: measuring the angle and distance to the object, and measuring the angle to the object from two different positions.

Chapter 14: Infrared communication The infrared horizontal proximity sensors of the Thymio robot are normally used for detecting objects, but they can also be used for communication with another Thymio. This chapter describes techniques for programming communications between two robots.

Chapter 15: Distributed algorithm Infrared communications is used to implement a simplified version of the Ricart-Agrawala algorithm for distributed mutual exclusion.

Increasing the detection range of the sensors

Ordinary objects need to be very close to the Thymio before they are detected by the horizontal proximity sensors. You can greatly increase the range by attaching *reflecting tape*, such as used on bicycles, to the objects.[a] The settings of the thresholds and motor powers in the programs in the archive will, of couse, have to be changed.

[a]My thanks to Francesco Mondada for showing this to me!

References

- Visual programming and VPL: https://www.thymio.org/en:visualprogramming.
- Text programming: the AESL language, the Studio environment and the Thymio interface: https://www.thymio.org/en:asebausermanual.

Part I

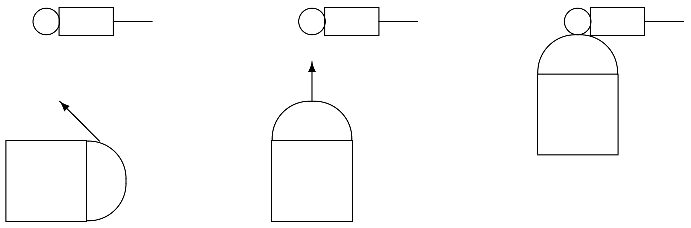
Projects

Chapter 2

The Cat Catches the Mouse

Specification

A cat turns left and right searching for a mouse. When it senses a mouse, it turns in the direction of the mouse and pounces on it quickly. When the cat collides with the mouse, it stops and announces its success.

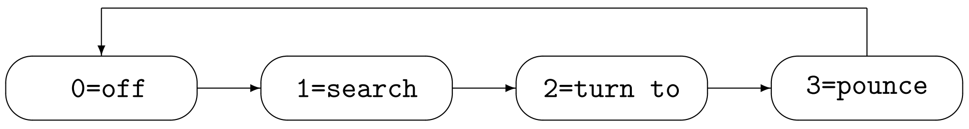


System design

The Thymio robot will be the cat which searches for a toy mouse. When the center button is touched, the robot turns left and then right repeatedly searching for the mouse. When the mouse is detected by one of the front horizontal proximity sensors, the robot turns until the mouse is detected by the *center* proximity sensor. Then it quickly moves forward in that direction and stops when the mouse is very close. The robot announces the cat's success by playing a victory tune: the first four notes of Beethoven's Fifth Symphony. The robot displays its current state in the top LEDs.

State machine

The program transitions sequentially through four states:



- The transition from `off` to `search` occurs when the center button is touched.

- The transition from `search` to `turn to` occurs when the mouse is detected.
- The transition from `turn to` to `pounce` occurs when the center sensor detects the mouse.
- The transition from `pounce` to `off` occurs when the center sensor detects that the mouse is very close.
- Additionally, in all the states except `off`, touching the center button causes a transition to state `off`. (These transitions are not shown in the above diagram.)

The state `search` has substates for turning left and right, as does the playing of the sound.

Constants

- `SCAN`: The power of the motors during the left and right search. One wheel receives positive (forward) power of this magnitude, while the other wheel receives negative (backwards) power of this magnitude.
- `TIMER`: The time between changes of direction during the search.
- `TURN`: The low power of the motors when turning to face the mouse.
- `POUNCE`: The high power of motors when pouncing on the mouse.
- `DETECTION`: The threshold at which a proximity sensor detects the mouse.
- `COLLISION`: The threshold at which a proximity sensor detects a collision.

Variables

- `state`: The current value of the state machine (0 .. 3).
- `motor_state`: The direction (+1 = right, -1 = left) in which the cat is turning.
- `sound_state`: The state of playing the victory tune (0 .. 4).
- `i`: An index variable for the loop that checks if the mouse is detected.

Event handlers and subroutines

- Event center button is touched: The event occurs when the button is *released*. The subroutine `on_off` is called.

- Subroutine `on_off`: The robot is turned on by setting the state to 1 (`search`) if it is 0 and turned off by setting the state to 0 (`off`) if it is not. The subroutine `initialize` is called.
- Subroutine `initialize` sets the initial values of the variables (`state`, `sound_state`, `motor_state`), the motors, the LEDs and the timer (`timer[0]`).
- Event `timer0` expires: If the robot is in state 1 (`search`), subroutine `search` is called.
- Subroutine `search` changes the direction of the motor from left to right or from right to left. The five front sensors are sampled (in a loop) to determine if the mouse is detected; if so, the state is changed to 2 (`turn to`).
- Event proximity sensor sampled (10 times per second): If the current state is 2 (`turn to`), the subroutine `turn_to` is called, while if the current state is 3 (`pounce`), the subroutine `pounce` is called.
- Subroutine `turn_to`: If the mouse is detected by the central sensor, the state is changed to 3 (`pounce`) and the robot is driven forward rapidly. Otherwise, if the mouse is detected by either of the left sensors, the robot is slowly turned to the left. Otherwise, the mouse is detected by either of the right sensors, the robot is slowly turned to the right.
- Subroutine `pounce`: If the central sensor detects that the mouse is very close, the motors are stopped and the state is set to 0 (`off`). The first note of the victory tune is played.
- Event sound finishes: This event occurs when the note that is currently being played has finished. The next note in the sequence is played and the variable `sound_state` is incremented, until all four notes have been played.
- LEDs: The LEDs on top of the robot are lit with distinct colors in the following subroutines: `initialize`, `search`, `turn_to`, `pounce`, and in the event handler `sound.finished`. The colors depend on the direction of the robot and on which sensors detect the mouse.

Programming notes

Program file `cat.aesl`

Start each program with a title, author and copyright notice:

```
# Thymio program: cat hunts mouse
# Copyright 2013 by Moti Ben-Ari
# CreativeCommons BY-SA 3.0
```

I suggest using the CreativeCommons BY-SA license, which allows anyone to copy and modify your program, provided that your name remains in the program and provided that they agree to let other people share their work.

Next, write comments explaining the constants and declarations of the variables with comments:

```
# Constants
# SCAN      - power of motors during search
# TURN      - power of motors during turn towards mouse
# POUNCE    - power of motors during pounce on mouse
# TIMER     - time between change of directions
# DETECTION - threshold for the detection of the mouse
# COLLISION - threshold for the detection of a collision with the mouse

# Variables
var state          # 0 = off, 1 = search, 2 = turn to, 3 = pounce
var motor_state    # +1 = right, -1 = left
var sound_state    # 0-4 for playing Beethoven's Fifth
var i              # loop variable
```

The values of the comments are set in the upper right corner of Aseba Studio, but comments cannot be added directly so you should document them within the program.

The initialization should be comprehensive since you don't know what the current state of the robot is when you run the program. For example, I turned off the temperature LED so that it won't confuse me when I observe the display on the robot:

```
call leds.temperature(0,0)
```

I use a lot of subroutines to keep the size of program components small and easy to read. In particular, I do very little in the event handlers themselves and put most of the processing in a subroutine:

```
onevent prox
  if state == 2 then
    callsub turn_to
  elseif  state == 3 then
    callsub pounce
  end
```

The order of alternatives in an if-statement is important and you may have to experiment to get the best results. The center sensor is checked before the side ones:

```
sub turn_to
  if     (prox.horizontal[2] > DETECTION) then
             # pounce
  elseif (prox.horizontal[0] > DETECTION) or
         (prox.horizontal[1] > DETECTION) then
             # turn right
  elseif (prox.horizontal[3] > DETECTION) or
         (prox.horizontal[4] > DETECTION) then
             # turn left
  end
```

This means that if the center sensor *and* another one detect the mouse in the same sample, the robot starts to pounce. Alternatively, if the center sensor were checked as the last alternative in the `if`-statement, the robot would pounce *only* when the center sensor alone detects the mouse. This is more precise but in the real world it might not occur.

Experiments

1. Experiment with “mice” of different sizes, colors and materials (metal, plastic).
2. Experiment with different combinations of sensors. In my program, both right and both left sensors are checked to see if they detect the mouse. What happens if only the rightmost and leftmost sensors are checked? What happens if the robot pounces on the mouse when it is detected by the center sensor *and* one of the side sensors?
3. You may want to define different thresholds for different sensors.
4. Experiment with the speeds `SCAN`, `TURN` and `POUNCE` to see how they affect the ability of the robot to detect and pounce on the mouse.

Chapter 3

How Fast is the Thymio?

There are three related projects in this chapter. The first project measures the speed of the robot by measuring how long it takes the robot to move a fixed length. The second project enhances the first one by enabling you to change the speed without making a change in the program. The third project measures the acceleration of the robot.

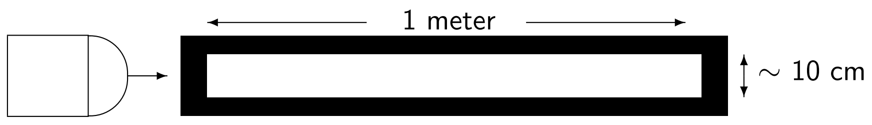
3.1 Measuring the speed of the Thymio

Specification

Measure the speed of the Thymio robot.

System design

Construct a long, narrow rectangle of black tape on the floor and place the robot near the left end of the rectangle facing right:[1]



The distance between the right edge of the left tape and the left edge of the right tape should be exactly one meter, while the distance between the top and bottom tapes should be sufficient so that both ground sensors can detect the floor, but a relatively small turn will cause the robot to detect the tapes.

When the forward button is touched, the robot moves slowly straight ahead until it detects the left tape; it continues to move over the tape until it detects the right edge of the left tape. At that point, it starts moving rapidly until it reaches the left edge of the right tape, at which point it stops. If the robot detects the top or bottom tapes, it turns back to a straight course.

[1] I am assuming that the floor has a light color. If the floor has a dark color, you should use white tape and change the program accordingly.

When the robot detects the right edge of the left tape, it sets a time counter to zero. The value of the time counter when the robot stops is used to compute the time that the robot takes to travel one meter and thus its velocity.

Touching the center button when the robot is moving causes it to stop.

State machine

The program transitions sequentially through four states:

$$\begin{tikzcd}
\texttt{0=off} \arrow[r] & \texttt{1=find} \arrow[r] & \texttt{2=skip} \arrow[r] & \texttt{3=straight} \arrow[lll, bend right]
\end{tikzcd}$$

- The transition from `off` to `find` occurs when the center button is touched.
- The transition from `find` to `skip` occurs when the left edge of the left tape is detected.
- The transition from `skip` to `straight` occurs when the right edge of the left tape is detected.
- The transition from `straight` to `off` occurs when the left edge of the right tape is detected.
- Additionally, in all the states except `off`, touching the center button causes a transition to state `off`. (These transitions are not shown in the diagram above.)

The value of the current state is displayed by turning on a LED associated with a button: 0=front, 1=right, 2=rear, 3=left.

Constants

- `THRESHOLD`: The threshold below which a sensor detects that it is on a tape.
- `MOTOR`: The power setting of the motors.
- `CHANGE`: The percentage change in the motor power when turning.

Variables

- `state`: The current value of the state machine (0 .. 3).
- `time`: A counter of tenths of seconds.

Event handlers and subroutines

- Initialization: Initialize `state` to 0, set the LEDs accordingly, initialize `timer0` to occur every 100 milliseconds = 0.1 seconds, and turn the motors off.
- When the center-button event occurs: if the button is released, the subroutine `stop` is called. It sets the state to 0 (`off`) and initializes the motors.
- When the `timer0` event occurs: if the current state is 3 (`straight`), the variable `time` is incremented.
- When a proximity event occurs, the subroutine called depends on the state:
  - In state 1 (`find`) subroutine `start_found` is called. If the start of the tape is found (both ground sensors sense black—a value less than the threshold), the motors are set to run at a reduced speed. The state is changed to 2 (`skip`).
  - In state 2 (`skip`) subroutine `end_of_start_found` is called. When both ground sensors no longer sense black, this is the end of the start tape. The state is changed to 3 (`straight`), the motors are set to run at the value `MOTOR` and the variable `time` is initialized to 0.
  - In state 3 (`straight`) subroutine `drive_straight` is called. If both ground sensors sense black, this is the finish line and `stop` is called. If only one sensor senses black, the robot is turned to get it going straight again by increasing the speed of one wheel and decreasing the other. If neither sensor senses black, the robot continues straight.

Programming notes

Program file `speed1.aesl`

The initialization statements and those in the subroutine `stop` are almost the same, but have to be repeated because the initialization cannot call a subroutine.

The constant `CHANGE` specifies the percentage by which the motor power setting `MOTOR` should be increased or reduced to turn the robot. The percentage 25% is expressed in mathematical calculation by the decimal number 0.25, but Aseba does not support decimal numbers. If the value of `CHANGE` is 25, we cannot write `CHANGE / 100`, because division in Aseba is *integer division*: 25/100 equals 0 with remainder 25.[2] The solution is to first perform a multiplication and then a division, so, for example, $(300 \times 25)/100 = 75$. The Aseba statements to change the power setting are:

```
motor.left.target  = MOTOR + (MOTOR * CHANGE)/100
motor.right.target = MOTOR - (MOTOR * CHANGE)/100
```

[2]You can obtain the remainder using the *modulo* operation in Aseba: `CHANGE % 100`.

The value of the variable time when the robot stops can be obtained by looking in the table labeled **Variables** at the left of the Aseba Studio window. When the robot stops, click refresh. Better, click the check box auto and the display will be dynamically refreshed.

Use the top circle LEDs to display the value of the variable time. Since there are only 8 LEDs the display can only updated at long intervals, for example, every 4 seconds to cover the range from 0 through 32 seconds. Remember that the variable time counts in tenths of a second, so the subroutine set_circle_leds is:[3]

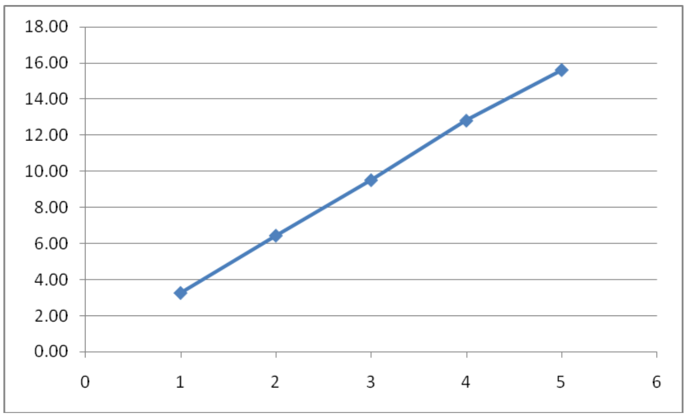
```
sub set_circle_leds
  call leds.circle(
    (time/ 40)*31, (time/ 80)*31, (time/120)*31, (time/160)*31,
    (time/200)*31, (time/240)*31, (time/280)*31, (time/310)*31)
```

Measuring the speed

The robot was run five times, one for each setting of the constant MOTOR: 100, 200, 300, 400, 500. The value of the variable time was recorded for each run and divided by ten since the variable records tenths of a second. Dividing 100 centimeters (= 1 meter) by the time gives the (average) speed of the robot:

Power	Time (sec)	Speed (cm/sec)
100	30.5	3.3
200	15.5	6.5
300	10.5	9.5
400	7.8	12.8
500	6.5	15.6

Here is a graph of power (x-axis) vs. speed (y-axis):



[3]31 is used instead of 32 to prevent the overflow of the intensity value.

The relation is linear! Linearity is a desirable property because you can easily change the power to obtain any desired speed. If the relation were not linear, you would have to adjust the power setting by including a table of power vs. speed in the program, as described in Chapter 8.

3.2 Setting the power from the Thymio

Specification

It is difficult to measure the speed of the robot for different power settings because for each one you have to change the program and reconnect the robot to the computer to load the new program. Modify the program so that the power can be changed by pressing buttons on the robot.

System design

The right and left buttons are used to increase and decrease the power setting in the range 0 .. 500. The setting is displayed by turning on 0 .. 5 circle LEDs, given by the power setting divided by 100.

Constants

- The constant `MOTOR` is no longer needed.
- `INCREMENT` is the amount by which the motor power is increased or decreased when the right and left buttons are pressed.

Variables

- `motor`: The motor power setting.
- `motor_change`: The change to the motor power setting.

Event handlers and subroutines

Two event handlers `button.left` and `button.right` are needed to detect when these buttons are released.

The subroutine `set_circle_leds` lights the correct number of circle LEDs.

Programming notes

Program file `speed2.aesl`

All occurrences of the constant `MOTOR` must be replaced by the variable `motor`.

In order to avoid recomputing the expression `(motor * CHANGE)/100` frequently, this value is assigned to the variable `motor_change` whenever the value of `motor` is changed.

```
motor_change = (motor * CHANGE)/100
```

The assignment of power setting to the power uses this value:

```
motor.right.target = motor - motor_change
motor.right.target = motor - motor_change
```

When increasing or decreasing the motor power, we have to be careful not to give values that are outside the range 0 .. 500 that the motor accepts. To avoid *saturation*, the range is checked when the left and right buttons are touched:

```
if  motor < 0 then motor = 0 end
if  motor > 500 then motor = 500 end
```

The subroutine `set_circle_leds` turns on 0 through 5 LEDS depending on the value of `motor`:

```
sub set_circle_leds
  call leds.circle(
    (motor/100)*32, (motor/200)*32, (motor/300)*32,
    (motor/400)*32,(motor/500)*32, 0,0,0)
```

for each value in the range 0 .. 5. Division in Aseba is integer division with truncation so if the value of `motor` is 328, the result of the division is 3.

3.3 Measuring acceleration of the Thymio

Acceleration is the rate of change of the speed. For example, if the speed of the robot starts at 0, becomes 5 cm/sec after 1 second, then 10 cm/sec after 2 seconds, and finally 15 cm/sec after 3 seconds, the acceleration is 5 cm/sec per second, written 5 cm/sec^2.

Specification

Measure the acceleration of the robot as the power setting is changed.

System design

The robot moves to the end of the start tape and stops.

The timer event `timer1` is set to occur every *t* seconds. When it occurs, the power of the robot is increased by 100. Therefore, the robot runs at each power setting 100, 200, 300, 400, 500 for *t* seconds. When it has run at power 500 for *t* seconds, the robot stops. The measurements are explained below.

Constants

- `MOTOR`: The initial power setting of the motors.
- `ACCEL`: The time between occurrences of the event `timer1`.

Event handlers and subroutines

- Subroutine `end_of_start_found` is modified to set `motor` to 0 and stop the robot.
- Subroutine `drive_straight` is modified to remove the condition of stopping when the end tape is reached.
- When the `timer1` event occurs and the state is 3 (`straight`):
  - If `motor` is less than 500, increase the value of `motor` by 100.
  - If `motor` is equal to 500, call subroutine `stop`.

Programming notes

Program file `speed3.aesl`

- The counter `time` should be initialized to 0 in the handler for the `timer1` event when `motor` is 0. This will be more accurate since the robot starts to move when this event handler is run and not when the robot reaches the end of the start tape.
- Initialization of `motor` to `MOTOR` should occur when the center button event occurs so that you can run the program multiple times without reloading.

Measuring the acceleration

Set the value of `timer1` successively to three values (500 ms = 0.5 second, 1000 ms = 1 second, 2000 ms = 2 seconds) and run the program. Use a tape measure to measure the distance that the robot moves and note the time as recorded in the counter `time`.

A body that accelerates from rest travels a distance $s = \frac{1}{2}at^2$, where a is the acceleration. Therefore, we can calculate the (average) acceleration as $a = 2s/t^2$.

For the values of the timer 0.5, 1, 2 seconds, I measured that the robot traveled 25, 50, 100 centimeters in 2.5, 5, 10 seconds, respectively. The accelerations are 8, 4, 2 cm/sec^2, showing that the faster you "press the gas pedal down to the floor," the faster the robot accelerates, and that the acceleration is inversely proportional to the time it takes to press the pedal to the floor.

Experiments

1. Experiment to see how consistent the measurements are. Perform the speed measurement at the five power settings several times.

2. How sensitive is the speed measurement to the experimental setup. What result do you get if you measure the speed over a 2-meter course. Try making the top and bottom tapes closer together or farther apart and see if that affects the results.

3. Experiment with the effect of friction on the speed of the robot. Measure the speed on a bare tile or concrete floor, on a wooden desk, on a sheet of plastic, and outside on the ground.

4. Making corrections to keep the robot within the rectangular frame can cause inaccuracy in the measurement of the speed. Instead, you can calibrate the motors so that the robot runs straight and then measure the speed from on small strip of tape to another place one meter away. Make small adjustments in the powers of the left and right motors until the robot runs straight. Alternatively, use the calibration procedure in the Thymio robot as explained here.

5. Modify the third program so that the acceleration is smooth.

6. Experiment with *deceleration* where you start the robot at a power setting of 500 and slowly reduce the power.

Chapter 4

Line Following

Specification

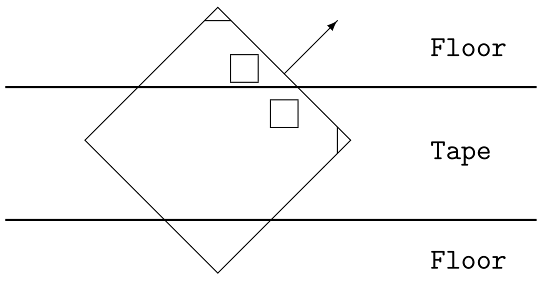
Write a program that causes the robot to follow a line on the floor.

System design

If the floor is light-colored, the ground proximity sensors will detect a lot of reflected light, so we need a dark line that will reflect very little light. This is easy to do by placing black electrician's tape on the floor:

The tape must be wide enough (at least 4 cm) so that both ground sensors will sense black when the robot is successfully following the tape.[1] When the robot strays off the line, the direction of the turn can be determined by which sensor measures a low level of light (over the tape) and which measures a high level of light (over the floor):

[1]You can download and print out two closed black-on-white lines from:
`https://aseba.wikidot.com/en:thymiobehaviourinvestigator`.
If your floor is dark-colored, use white tape and modify the program accordingly.



Once we know this direction we can turn the robot back onto the tape by increasing the power of one motor and decreasing the power of the other motor. In the drawing above, to cause the robot to turn right back onto the tape, we cause the left motor to run forward at high power and the right motor to run forward at low power, or even backwards to achieve a tighter turn.

The difficult part of a program for line following is handling the case when *both* sensors no longer detect the line. The solution is to remember the direction the robot was going when the first sensor detects the floor. However, the sampling rate of the proximity sensors (10 times a second) is not fast enough to do this if the robot is traveling at a high speed. Therefore, a timer event is added and set to occur more often, say, 50 times a second (every 20 milliseconds). When the event occurs, the computation of the direction is performed as explained below.

The direction of the turn is indicated by different colors of the top LEDs.

State machine

There are two states: 0 for `off` and 1 for `on`.

Constants

- `THRESHOLD`: The threshold below which a sensor decides that it is on a tape.
- `DIFF`: The difference in sensor values above which the direction is remembered.
- `CHANGE`: The percentage change in the motor power when turning.
- `TIMER`: The period of the timer that computes the last direction.
- `INCREMENT` is the amount by which the motor power is increased or decreased when the right and left buttons are pressed.

Variables

- `state`: The state: `off` = 0 or `on` = 1.
- `motor`: The motor power setting.
- `motor_change`: The change to the motor power setting.
- `diff`: The difference between the values of the left and right ground sensors.
- `dir`: The direction the robot was turning when it left the tape (1 = right, -1 = left).

Event handlers and subroutines

- Setting the motor power: the handlers for the events `button.left` and `button.right` and the subroutine `set_circle_leds` were described in Chapter 3.
- When the center-button event occurs: if the button is released, the state and the motors are changed from on to off and from off to on.
- When a proximity event occurs, the subroutine `drive` is called.
  - If both sensors detect the tape, the robot drives straight.
  - If only one of the sensors detects the tape, the robot turns in the correct direction to return to the tape.
  - If neither sensor detects the tape, the robot turns in the direction indicated by the variable `dir`.
- When the `timer0` event occurs: The range of values of the ground sensors is very large—between 0 and 1023—so the two sensors will almost certainly measure slightly different values even if both are above the tape or above the floor. Therefore, the direction is computed only if the absolute value of the difference between the measured values is significant—above a threshold determined by the constant `DIFF`. If so, the variable `dir` is set to 1 if the value of the left sensor is greater than that of the right sensor and, if not, it is set to -1.

Programming notes

The operator for absolute value is defined in AESL. The outline of the `timer0` event is:

```
onevent timer0
  diff = prox.ground.delta[0] - prox.ground.delta[1]
  if  abs diff >= DIFF then
    if  diff > 0 then dir = 1 else dir = -1 end
  end
```

Program file `line.aesl`

Experiments

1. What is the maximum speed at which the robot can reliably following the line? (Here, *reliably* means that the robot can successfully follow the line if you run the program several times.) The faster the robot, the more likely it is that the robot will lose the line and not be able to return to it.

2. Experiment with the power setting of the turns. My program sets one motor to move forward and one to move backwards. This turns the robot rapidly in the correct direction but slows it down. A gentle turn may or may not be better.

3. What is the largest angle that the robot can follow? Can it follow a tape that has a full (90°) left or right turn?

4. Modify the program so that instead of using a single value for the detection threshold, there are separate thresholds t_w and t_b such that the robot is considered to be on the tape if the sensor value is less than t_b and off the tape if the sensor value is greater than t_w.[2]

5. Modify the program so that the robot calibrates the sensors—determines the detection thresholds—by itself. Place the robot with both sensors over the tape and press a button; the program will calculate t_b. Now place the robot with both sensors over the floor and press another button; the program will calculate t_w.

[2]This experiment and the following one are based on the line-following program from: `https://aseba.wikidot.com/en:thymiobehaviourinvestigator`.

Chapter 5

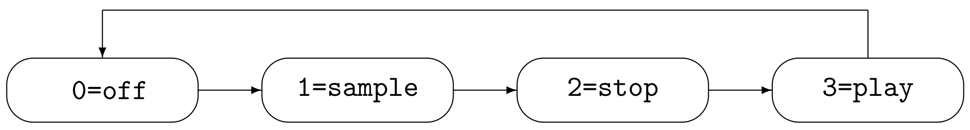
Composing Music On-the-Fly

The Thymio contains *accelerometers* that measure the acceleration of the robot on each of the three axes: left-right, front-back and top-down. When the robot is at rest, the acceleration of the left-right and front-back axes will be zero, but the accelerometer for the top-down axis will measure a positive acceleration corresponding to the earth's gravity. The accelerometers are used to detect shocks (like the *tap* event) and they can be used to detect the orientation of the robot relative to the ground. The project in this chapter uses movement of the robot to compose music.

Specification

Hold the Thymio in your hand and move it quickly left and right. The program will sample the left-right acceleration and store the samples. When enough samples have been taken, their values will be used to play music using the sound synthesizer.

State machine



The initial state is `off`. When the center button is touched, the transition to `sample` is taken and the robot begins to sample the accelerations. The samples are stored in a data structure; when it is full, the transition to `stop` is taken. Touching the center button again causes a transition to `play` and the sounds to be played. When all the sounds have been played (or the center button is touched again), the transition to state `off` is taken.

Constants

- `SIZE`: The number of samples of the acceleration.
- `PERIOD`: The number of milliseconds between samples.

- `BASE`: The base frequency for sound.
- `FACTOR`: The factor for changing the frequency of each sound.
- `DURATION`: The duration of each sound.

Variables

- `save`: An array of length `SIZE` for saving the samples.
- `index`: An array index variable.
- `state`: The state.

Event handlers and subroutines[1]

- Event `button.center`: When the center button is released, the state transitions described above are performed. During the transition from state `stop` to state `play` a short sound is played so that the event `sound.finished` will occur. During the transition from state `play` to state `off`, `sound.freq` is called with a duration of -1 to stop the sound. In all cases, the variable `index` is set to 0.
- Event `timer1`: When the timer expires in state `sample`, the value of `acc[0]` is stored in `save` and `index` is incremented. When the array is full, the state is set to `stop`.
- Event `sound.finished`: When a sound is finished, initiate the next sound by calling `sound.freq`. When all sounds have been played, go to state `off`.

Programming notes

Program file `music1.aesl`

- To initialize the array elements to zero, use the native function `math.fill`.
- Initialize the timer to `PERIOD`.
- The values of the acceleration are between -32 and 32, so multiply them by `FACTOR` and add to `BASE` to obtain an audible note (several hundred hertz):

  ```
  call sound.freq(BASE+save[index]*FACTOR, DURATION)
  ```

  The units of DURATION are $\frac{1}{60}$ths of a second.

[1]Before continuing, review sections *Accelerometer* and *Synthetic sound* of the *Programming Interface* at `https://aseba.wikidot.com/en:thymioapi`.

Experiments

- Experiment with the values of the constants. Make sure that the note played—the first parameter to `sound.freq`—doesn't go below or above a sound you can hear.

- Set the LEDs on the top of the robot to reflect its current state.

  Program file `music2.aesl`

- The sounds obtained are not "real" notes but just sounds of different frequencies. Modify the program so that only real notes are played. Define an array of frequencies: the frequencies from *middle C* to *high C*, rounded to integers, are:

  ```
  var notes[8] = [261, 294, 330, 349, 392, 440, 494, 523]
  ```

  Use the values in `save` to compute an index to `notes` to select a frequency for `call sound.freq(notes[...], DURATION)`. Use the modulo operator `% 8` to ensure that the index is in the range 0–7.

  Program file `music3.aesl`

- In addition to the left-right acceleration, save the forward-back acceleration in another array. Use the left-right values to compute the note and the forward-back values to compute the duration. Explain what happens if you use top-bottom acceleration instead.

  Program file `music4.aesl`

- Instead of calling `leds.top` whenever the state is changed, write an event handler that checks the current value of `state` and sets the LEDs accordingly. Since the `prox` event is not otherwise used in this project and occurs frequently (10 times a second), it is convenient to use it for this purpose.

  Program file `music5.aesl`

- By default, one circle LED is lit at the lowest point of the robot. Modify this so that two circle LEDs are lit: left or right and front or back to show the directions of the accelerations in these axes.

  Program file `music6.aesl`

Chapter 6

Odometry

Suppose that you are in a car and your GPS navigation system instructs you: "In 700 meters turn right." You no longer need the system to choose the correct turn. Simply check the car's *odometer* which measures how far you have traveled and when it approaches 700 meters beyond its initial reading, look for a road on the right. *Odometry*—the measurement of distance—is a fundamental method used by robots for navigation. However, even with a relatively accurate odometer, you will want to look carefully for an intersection, because you don't want to turn even two or three meters too early or too late.

Odometers measure time and speed, and then compute distance traveled by multiplying them. In this chapter we demonstrate odometry with the Thymio robot.

Feel free to make the programs more exciting by including lights and sounds.

Specification

The Thymio robot travels straight at a constant speed for a fixed period of time. Compute how far it has traveled and compare the computed distance with the actual distance.

Design

The motor power is set by assigning values to `motor.X.target` (where `X` is `left` or `right`). The internal computer of the robot tries to maintain this power level, but it is possible to measure the actual power level by reading the values of `motor.left.speed` and `motor.right.speed`. In theory, once we set the motor power, it will remain constant, but in practice it changes frequently, so we will sample the actual power level at frequent intervals. The distance traveled during each interval will be computed and these distances will be added to obtain the total distance traveled by the robot.

The values returned by `motor.X.speed` are in the range 0–500 and have to be converted into speeds measured in centimeters per second. Implement the project in Chapter 3 to compute the relationship between power levels and speed of your Thymio.

From the table on page 14, I determined that my Thymio travels over the floor at about $\frac{Power}{100} \times 3.2$ cm/sec. Multiply this by the duration of the sampling interval in seconds to obtain the distance that the robot has traveled during that interval. For example, if the value in `motor.X.speed` is 350 and the interval is 0.5 seconds, the distance traveled is $\frac{350}{100} \times 3.2 \times 0.5 = 5.6$ cm.

Constants

- `MOTOR`: The motor target speed
- `DELTA`: The time between samples
- `COUNT`: The number of samples
- `SPEED`: The speed of robot per 100 power setting

Programming

Program file `odo1.aesl`

- The actual motor speeds should be measured starting from the *second* occurrence of the timer event so that the robot has time to achieve its target speed.
- Sample the speed of either the left or the right motor on the assumption that they are roughly the same. Alternatively, sample both speeds and take their average.
- The distance traveled during an interval is $\Delta d = \frac{p}{100} \times s \times \Delta t$, where p is the sampled speed (power), s is the power-to-speed conversion factor and Δt is the time length of the interval.

  Since real numbers are not supported, $s = 3.2$ cm/sec and $t = 500$ msec $= 0.5$ sec are scaled by multiplying by ten: $s' = 32$ **mm/sec** and $t' = 5$ **sec**. The distance is then divided twice by ten. To preserve the precision of Δd, the division should be done on the total distance obtained by summing each Δd.
- Use the native function for 32-bit computation (see Section 10.2):

  ```
  call math.muldiv(deltaD, p, s'*t', 100)
  ```

Running the program

- Run the program several times, measure the distances traveled by the Thymio, and compare them with the computed distances. How accurate is the odometry computation? How consistent are the measured and computed distances?

- Does the robot travel straight? If it turns to one side, you might want to add or subtract a value from the constant `MOTOR` when assigning it as the target speed to one of the motors.
- Place an object at a known distance in front of the Thymio. Modify the program so that when the odometry computation determines that the robot is near the object, it detects the object using the horizontal sensors and then moves slowly until the robot touches the object.

Experiments

Program file `odo2.aesl`

The motors are set to run at a constant speed, so we would probably get reasonably accurate measurements of the distance by taking a single sample. Modify the program so that the robot *randomly* changes the target speeds of the motors in each interval after the speed is sampled.

The native function `math.rand` returns a random number in the range $-32768..32767$. Before using this value as a target speed you will have to perform a computation to ensure that the value is in the range of the motor power values 0..500.

Chapter 7

Image Recognition

A robot must be able to look at its environment in order to locate and identify the targets of its actions or objects that may be in its way. Image recognition normally requires a camera; this project will use the proximity sensors.

Consider a robot in a warehouse where all objects are identified by barcodes. Let us see how a robot could search for a specific object in an aisle with many objects.

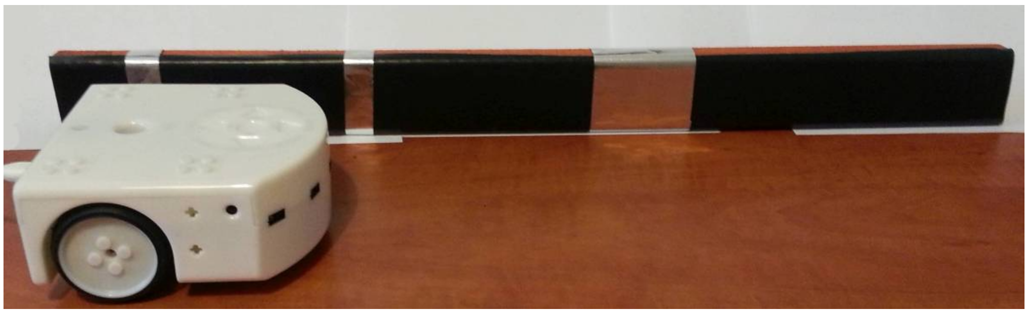
Specification

The Thymio moves straight and at frequent intervals turns left to identify to object on a shelf. After sampling the objects, it turns back right and then continues straight. When the robot locates the object, it emits a signal.

Design

Program file `barcode.aesl`

Take a piece of wood about the height of the robot and about 50 cm long. Cover it with black tape so the wooden background will not be detected by the sensors. The objects are represented by strips of aluminum foil fastened over the black tape. Narrow strips are objects we don't want and the wide strip is the object we want.



The robot computes the sum of the values returned by the five horizontal sensors. When this value is above a threshold, the robot detects the wide strip and the top LEDs are turned on. All the sums of the samples are stored in an array so that they can be examined after the robot stops.

State machine

The program transitions sequentially through four states:

$$\begin{tikzcd}
\boxed{\texttt{0=off}} \arrow[r] & \boxed{\texttt{1=straight}} \arrow[r] & \boxed{\texttt{2=left}} \arrow[r] & \boxed{\texttt{3=right}} \arrow[ll, bend right]
\end{tikzcd}$$

- The transition from `off` to `straight` occurs when the forward button is touched.
- The transition from `straight` to `left` occurs when `TIME_STRAIGHT` has passed.
- The transition from `left` to `right` occurs when `TIME_LEFT` has passed.
- The transition from `right` to `straight` occurs when `TIME_RIGHT` has passed.
- In addition to the transitions shown in the diagram, touching the center button causes a transition to state `off`, as does detecting the end of the table.

Constants

- `MOTOR`: The power of the motors.
- `THRESHOLD`: The threshold for detection of the wide aluminum strip.
- `TIME_STRAIGHT`, `TIME_LEFT`, `TIME_RIGHT`: Values for the timer to control the search and the turns. Turning left and right should take the same amount of time, but inconsistencies may require the use of slightly different times.

Variables

- `state`: The current state.
- `samples`: An array to store the sums of the samples.
- `i`: Index for sampling all the sensors.
- `j`: Index for the array `samples`.

Event handlers and subroutines

The events handlers for touching the center and forward buttons and for detecting the edge of the table are straightforward. Most of the computation is done in the handler for `timer0`. It contains an `if`-statement with four alternatives, one for each state. Each alternative changes the motor power as required and sets the timer for the next state.

Before turning right in state 2, the sensors are sampled. If the sum is greater than `THRESHOLD`, the top LEDs are turned on.

Running the program

It is quite difficult to get this program to run correctly, because turning left and then right may not return the robot to exactly the same heading. If the table surface if not perfectly smooth, these deviation can change as the robot moves along the piece of wood.

Experiments

1. Experiment with the speed of the robot. How fast can you get it to search for the object and still identify it.
2. See if the program can identify multiple objects (wide strips).
3. Measure how far the robot moves for each sample. Modify the program to count the number of samples until it finds the object and then to compute to distance from the starting point to the object.
4. Experiment with the time between samples (`TIME_STRAIGHT`). More frequent samples will cause the robot to take a longer time to locate the object; however, it will locate its position with greater accuracy.
5. The accuracy of the search can be improved by placing a tape on the surface and using a line-following program (Chapter 4). This will not only ensure that the robot travels straight, but it will make it easy to determine when the right turn has moved to robot back to its initial heading.

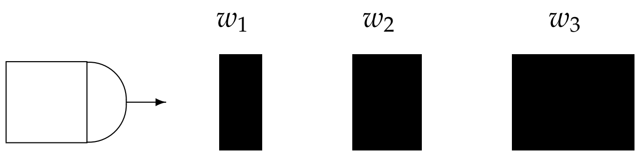
Project

Develop a program that reads a barcode consisting of three strips of tape placed on a table. The strips are detected by the ground sensors and interpreted as encodings of RGB colors. When the strips have been decoded, the top LEDS will display that color.

Program file `color-code.aesl`

Guidance:

Place three strips of tape of varying widths on a table.



Define two constants `MOTOR_SPEED` and `THRESHOLD`. The first is the power setting for both the left and right motors, and the second is the value of the ground sensors below which the black tape is detected. (Should you use the value of the left ground sensor, the right one or some combination of the two?)

The Thymio travels across the strips of tape. When it detects a black tape during a `prox` event, it zeros a counter. The counter is incremented during each subsequent `prox` event until the black tape is no longer detected. The counter is then stored in an array. When three strips have been detected, the three values stored in the array are used to compute the values of the red, green and blue components and `leds.top` is set accordingly. The higher the value of the counter (that is, the more events that occurred while moving over a strip), the wider the strip and the higher the value of the color component.

You will have to experiment with the parameters of this project to get it to work properly: the width of the strips of tape, the constants `MOTOR_SPEED` and `THRESHOLD`, and the mapping between counter values and the values of the color components.

Run the program with different arrangements of the strips of tape in order to display different colors.

Part II

Advanced Concepts in Robotics

Chapter 8

Nonlinearity

In Chapter 3 we measured the speed of the Thymio robot for various settings of the motor power and found that the speed v was a linear function $v \approx 0.03p$ of the power p. For example, a power setting of 300 gave a speed of about 9 cm/sec. In this chapter we will look at the values returned by the horizontal proximity sensors and ask: Are the values a linear function of the distance to an object?

8.1 Measuring the horizontal proximity sensors

Specification

Construct a graph showing the values returned by the center horizontal sensor for an object placed at 1 cm intervals in front of the robot.

System design

Tape a ruler on your table and carefully place the robot so that its front is at the 0 mark of the ruler. Place an object exactly at the 1 cm mark on the ruler and touch the center button. Store the value of the center proximity sensor. Repeat for 2 cm, 3 cm, . . . , until the value goes to zero.

The value returned by the sensor depends not only on the distance of the object but also on how precisely it is centered in front of the object. For each distance, move the object slightly left and right until the value is as large as possible.[1] You can read the value of the sensor in the `Variables` pane at the left of the Aseba Studio window; click the check box `auto` so that the display will be dynamically refreshed.

Variables

- `proximity`: An array for storing the values read from the sensor. Eight elements should be sufficient: by experiment, I found that the sensor is unlikely to detect an

[1]The values change all the time so you won't be able to get the true maximum.

object further than 8 cm away. Initialize each element of the array to zero so that you can determine which values have been entered from a new run of the program.

- `count`: An index for the array.

Event handlers and subroutines

When the center button is released, the current value of the center horizontal sensor is stored in the first unused element of the array `promixity`.

Programming notes

Program file `nonlinear1.aesl`

Aseba can determine the size of an array from the number of elements in an initial value.

```
var proximity[] = [0,0,0,0,0,0,0,0]
```

Running the program

Run the program and touch the center button for each position of the object. The values in the array `proximity` will be displayed in the `Variables` pane. Construct a graph with distance on the x-axis and the sensor values on the y-axis. Better, run the program several times and use the average values of the sensors for each distance.

Figure 8.1 shows the results of two runs and their average. The function is reasonably linear in the range 3–5 cm, but nonlinear outside that range.

Exporting the values of the variables

It is not necessary to copy the values by hand from the `Variables` pane. After running the program, select `Files/Export memory content` and enter a file name such as `data.txt`. If you open the file in an editor, you will see the values of each variable, for example:

```
thymio-II.proximity 1333 2278 2865 3338 3710 4092 4274 4349
```

You can use spreadsheet software to compute with these value or to draw a graph. Here are the instructions for Microsoft Excel:

- Select `File/Open` from the menu and write in the name of your file.[2] Click `Open`.

[2]The file will not be displayed unless you select `All Files` in the list next to the `File name` field.

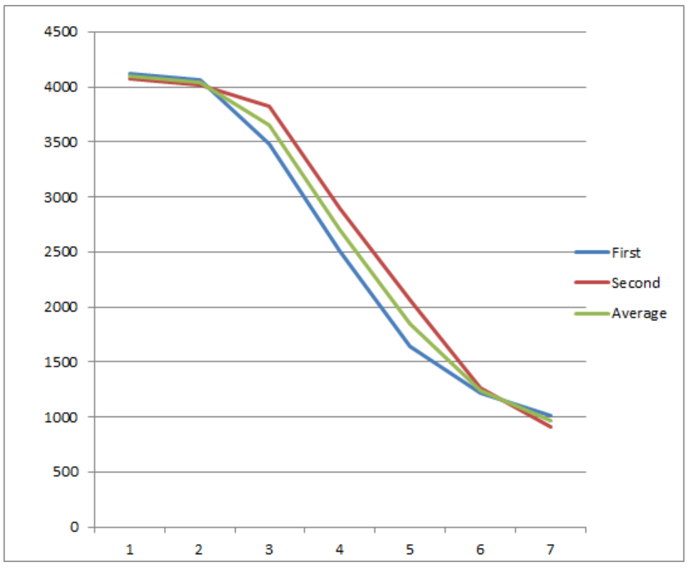


Figure 8.1: Sensor value for distances 1–7 cm

- Select `Delimited` and click `Next`.
- Select `Space` and click `Next`.
- Click `Finish`.

To draw a graph, select the values in the line labeled `thymio-II.proximity`, click `Insert/Line` and choose a style.

Experiments

1. Modify the program so that it stores the results of three sequences of measurements and computes the average.
2. Do the values measured by sensor depend on the object? Experiment with objects of different shapes and materials.
3. Modify the program so that it records the values of the other four horizontal sensors. Do the different sensors return similar values?

8.2 Calibrating the horizontal proximity sensors

Nonlinear sensors can be used if they are *calibrated*. The results of an experimental measurement (as described in the previous section) can be used to translate the raw sensor data into actual distances.

Specification

When an object is placed in front of the center horizontal sensor and the center button touched, the circle LEDs on the top of the robot display the distance from the robot.

System design

An array stores the values measured for each distance. When the button is released, the value of the sensor is compared with the elements of the array. The first index at which the element of the array is lower than the measured value is taken to be the distance, and that number of LEDs are turned on. For example, let the values in the array be:

```
var proximity[] = [4100, 4045, 3650, 2705, 1850, 1245, 965, -1]
```

If the sensor measures a value of 3000, then `proximity[3]` (2705) is less than 3000, so the loop stops with the value 3 in `distance`. If the sensor measures 2000, the distance is 4 cm.

Variables

- `proximity`: The array stores the values of the sensors previously measured.
- `distance`: The array index; its final value is the distance to the object.

Event handlers and subroutines

- When the center button is released, a loop searches the array `proximity` comparing the values with the value measured by the sensor.
- `set_circle_leds`: Light the number of LEDs given by the variable `distance`; it is similar to the subroutine in Chapter 3 that displayed the motor setting.

Programming notes

Program file `nonlinear2.aesl`

We used a well-known programming trick called searching an array with a *sentinel*. The last element of the array is given the value -1, which is less than 0, the *smallest* possible value of the sensor. This ensures that the loop will *always* terminate.

Running the program

When I ran the program, the results were accurate for distances in the range 1–5, but less accurate for larger distances.

Experiments

1. The transition between n cm and $n + 1$ cm is taken when the object is bit over n cm from the robot. Modify the program so that the transition is taken halfway between the two distances.

2. Modify the program so that it can measure distances of half a centimeter. One solution is to extend the calibration to sample values for every one-half centimeter. Alternatively, you could try *interpolation*, where the transition value for a half centimeter distance are taken as the average of the two neighboring values.

Chapter 9

Odometry in Two Dimensions

In the previous chapter, the distance traveled in one dimension was measured. For a robot like the Thymio that can turn, we want to track its position in the plane in two dimensions x and y, as well as its *heading*, the direction in which the robot is pointing.

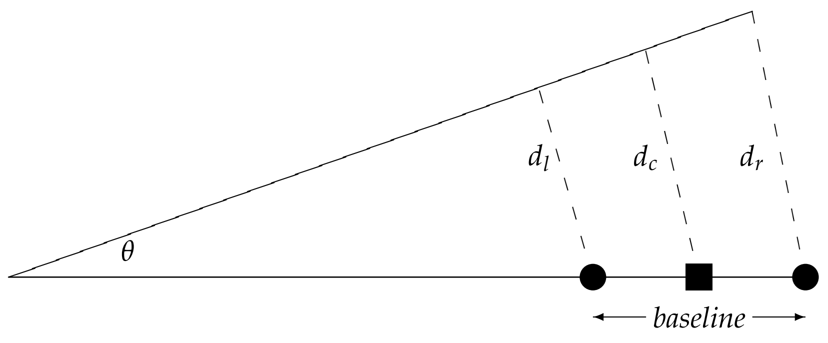
This chapter presents the basic concepts of odometry for a turning robot. Since Aseba only supports 16-bit integer computation, while the trigonometric computations that are needed are usually done with real numbers, scaling factors must be used and it will be rather difficult to keep track of them.

Specification

The Thymio robot moves a short distance with the right wheel moving faster than the left wheel. Compute the new position and heading of the robot.

Design

Consider a robot that moves a short distance at an angle:



The square indicates the center of the robot and the circles indicate the left and right wheels. The distance between them is *baseline* and d_l, d_r, d_c represent the distances moved by the two wheels and the center of the robot.

The circumference of a circle of radius r is given by $2\pi r$ and in general the length of an arc of angle θ (in radians)[1] is given by θr. Therefore, we have:

$$\theta = d_l/r_l = d_r/r_r = d_c/r_c,$$

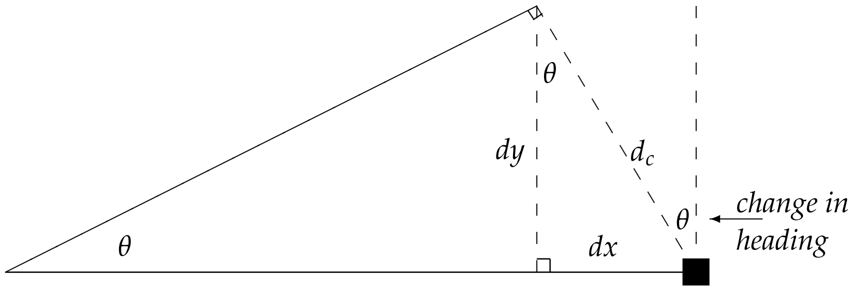
where r_l, r_r, r_c are the distances of the wheels and the center from the origin of the turn.

The angle θ can be computed from the distances traveled and the baseline:

$$\begin{aligned}
r_r \cdot \theta &= d_r, \\
r_l \cdot \theta &= d_l, \\
r_r \cdot \theta - r_l \cdot \theta &= d_r - d_l, \\
\theta &= (d_r - d_l)/(r_r - r_l), \\
\theta &= (d_r - d_l)/\mathit{baseline}.
\end{aligned}$$

We assume that the center of the robot is halfway between its wheels, so $r_c = (r_l + r_r)/2$, and therefore $r_c = d_c/\theta = (d_l/\theta + d_r/\theta)/2$; multiplying by θ gives $d_c = (d_l + d_r)/2$.

If the distance moved is small, the line labeled d_c is approximately perpendicular to the radius through the final position of the robot:



By similar triangles, we see that θ is the change in heading of the robot.

By trigonometry, we have that:

$$\begin{aligned}
dx &= -d_c \cdot \sin\theta, \\
dy &= d_c \cdot \cos\theta.
\end{aligned}$$

These formulas show how to compute the changes dx, dy and θ when the robot moves a short distance. To implement odometry, the changes are computed at frequent intervals and the actual position and heading are updated with these changes. For simplicity and given the limitations of the measurements of d_l and d_r, we will implement the computation of a single change over a relatively long period of time.

[1]A radian is a unit used to measure angles. A circle of 360° is defined to be $2\pi \approx 6.28$ radians. It follows that $180° = \pi$ radians, $90° = \pi/2$ radians and so on.

Constants

- `TIME`: Duration of the run (30 tenths of a second)
- `BASELINE`: The distance between the left and right wheels (95 mm)
- `LEFT`: Power setting of left wheel (200)
- `RIGHT`: Power setting of right wheel (300)
- `SPEED`: The speed of robot per 100 power setting (32 mm/sec)

Programming

Program file `odo3.aesl`

It would be easy if we could ask for a single timer event to occur after three seconds, but the Thymio timers cause an event to occur immediately when the program runs. Therefore, the timer is set to expire every tenth of a second and the odometry computation is done when 30 timer events have occurred.

The distances traveled by the two wheels are computed from the power settings:

```
call math.muldiv(dleft, motor.left.speed, SPEED*TIME, 100*10*10)
call math.muldiv(dright, motor.right.speed, SPEED*TIME, 100*10*10)
dcenter = (dleft+dright)/2
```

The result is divided by a factor of 100 for the motor powers, 10 for the power to speed conversion (3.2 cm/sec) and again by 10 because the time is tenths of a second.

The change of heading is computed from the distances traveled by the two wheels and the baseline:

```
theta = ((dright-dleft)*10*100)/BASELINE
```

The factor 10 cancels out the factor of 10 in the baseline that is given in millimeters. The factor 100 gives the result in hundreths of radians, transforming the real range 0–3.14 to the integer range 0–314. (For simplicity we consider only positive angles.)

The sine and cosine are computed:

```
call math.cos(tcos, theta*100)
call math.sin(tsin, theta*100)
```

In Aseba, the parameter to the trigonometric functions is within the entire 16-bit range -32768–32767, which represents the range from $-\pi$ to π radians. Multiplying the angle by 100 will cause the range of positive angles 0–314 to become 0–31400, approximately the range of the positive angles as 16-bit integers.

We can now compute the change in the x- and y-positions:

```
call math.muldiv(dx, dcenter, -tsin, 32767)
call math.muldiv(dy, dcenter, tcos, 32767)
```

The result is divided by 32767 because the result of the trigonometric functions in Aseba is within the entire 16-bit range -32768–32767, instead of the normal range of real numbers -1.0–1.0.

Running the program

Running the program gave the following results which are reasonable:

`dleft`	20
`dright`	29
`dcenter`	24
`theta`	.94
`dx`	-18
`dy`	15

The units are centimeters except for θ which is in radians (.94 radians $\approx 54^\circ$).

Running the program several times gives different results because the values returned by `motor.X.speed` are not consistent.

Experiments

- Run the program for different periods of time. How does this affect the accuracy and consistency of the odometry computation?
- The Thymio robot has a hole that can be used to hold a marker. Mark the actual path of the robot and measure d_c. Can you use this to compute d_l and d_r? If so, perform the odometry computation with these values. Is the computation more accurate and consistent than the computation using the measurements of `motor.X.speed`?

Chapter 10

Control

All our projects contain an algorithm that *decides* what to do. The proximity sensors measure a value and if it is above or below a threshold, the program decides (using an `if`-statement) whether to start or stop the robot, or to turn it in one direction or another. These algorithms are called *control algorithms* and there is a sophisticated mathematical theory of control that is fundamental in robotics. In this chapter, we develop several projects that demonstrate various control algorithms.

The projects concern searching for and approaching an object, as was done in the cat-catches-mouse project in Chapter 2. We simplify the projects in order to the focus on the control algorithms. The presentation of the projects will be more informal than in previous chapters on the assumption that you now have enough experience to design state machines, declare constants and variables, and write event handlers and subroutines.

I built an object from a lipstick container wrapped in aluminum foil. The object must be narrow so that an individual horizontal sensor will be able to locate it accurately.

Before you proceed with the projects, experiment to find out the range of values returned by the horizontal sensors as the object gets closer. Recall that you can examine the values in the `Variables` pane of Aseba after checking the box labeled `auto`. Define a constant `THRESHOLD` to be close to the highest value. I found that the values ranged between 0 and 4100 or a bit more, so I defined `THRESHOLD` as 4000.

The first part of the chapter described two projects for searching for the object, one using a simple on-off controller and the other using a proportional controller. The second part describes three projects for approaching the object, using an on-off and a proportional controller, as well as a more sophisticated *proportional-integral* controller.

Tip: You can follow the changes in the motor power by examining the variables table as described on page 14. A better way is to program the circle LEDs to display the motor power as described on page 16.

10.1 Searching

An on-off controller

Specification: When the leftmost horizontal sensor of the Thymio detects the object, it turns rapidly to the right until the rightmost sensor detects the object and then stops.[1]

Program file `scan-on-off.aesl`

The first control algorithm is very simple. When the value of the leftmost sensor is greater than the threshold turn the motors on to their highest positive and negative values (500 and -500). When the value of the rightmost sensor is greater than the threshold, turn the motors off:

```
onevent prox
  if  prox.horizontal[0] > THRESHOLD then
    motor.left.target  = -MOTOR
    motor.right.target = MOTOR
  elseif  prox.horizontal[4] > THRESHOLD then
    motor.left.target  = 0
    motor.right.target = 0
  end
```

This is called the *on-off* algorithm, because the motors are either on at full power or off.[2] Run the program. What happens? When I ran the program, the robot stopped with the object *past* the rightmost sensor. When the robot is moving at full speed, the rate at which the `prox` event occurs is simply not fast enough to stop the robot in time. As you know from riding in a car, starting and stopping as fast as possible is very uncomfortable. Fast acceleration and braking is also bad for the engine and brakes. For these reasons, the on-off algorithm is not used in practice.

[1] In all the projects, the touching the center button starts and stops the program and motors.
[2] Another, colorful, name is the *bang-bang* algorithm!

A proportional controller

To develop an algorithm that is better than the on-off algorithm, we take inspiration from riding a bicycle. Suppose that you are riding your bicycle and see that the traffic light ahead has turned red. You *don't* wait until the last moment and then squeeze hard on the brake lever; if you did so, you might be thrown from the bicycle! What you do is to slow your speed gradually: First, you stop pedaling; then, you squeeze the brake gently to slow down a bit more; finally, when you are at the light and going slowly, you squeeze harder to fully stop the bicycle.

The algorithm used by a bicycle rider can be expressed as: to stop gradually, reduce your speed more as you get closer to the object. The decrease in speed is *proportional* to how close you are to the light: the closer you are, the more you slow down.

Program file `scan-p.aesl`

To implement a proportional algorithm we have to *measure* how close the object is from rightmost sensor. The language of the following description might seem strange but the intent should be clear:

> When detected by the leftmost sensor, the object is 0 units close;
> When detected by the next sensor to the right, it is 1 unit close;
> When detected by the center sensor, it is 2 units close;
> When detected by the next sensor to the right, it is 3 units close;
> When detected by the rightmost sensor, the robot is at the object.

The algorithm reduces the motor power setting in proportion to the number of units that are measured. Define a constant `DELTA` for the amount by which the power is reduced for each unit. When the object is detected by the n'th sensor ($n = 0 .. 3$), the power is reduced by n times `DELTA`. When the 4th sensor detects the object, the robot stops. The event handler is shown in Figure 10.1.

Run the program, experimenting with the values for `DELTA`, so that the robot starts turning very fast but then gently slows down, just as if you were applying more and more pressure to the brake lever on the bicycle. The robot should stop with the rightmost sensor precisely opposite the object.

Programming note: Why are the sensors checked in the order 4, 3, 2, 1, 0 and not in the order 0, 1, 2, 3, 4? The answer is in the footnote, but think about it before you look.[3]

[3] If the object is detected by *two* sensors at the same time, the lower power setting of the sensor *closer to* the rightmost sensor should be used. This way the slowing down of the robot is smoother. Experiment with the other order and see what happens.

```
onevent prox
  if  prox.horizontal[4] > THRESHOLD then
    motor.left.target  = 0
    motor.right.target = 0
  elseif  prox.horizontal[3] > THRESHOLD then
    motor.left.target  = -MOTOR + DELTA * 3
    motor.right.target =  MOTOR - DELTA * 3
  elseif  prox.horizontal[2] > THRESHOLD then
    motor.left.target  = -MOTOR + DELTA * 2
    motor.right.target =  MOTOR - DELTA * 2
  elseif  prox.horizontal[1] > THRESHOLD then
    motor.left.target  = -MOTOR + DELTA * 1
    motor.right.target =  MOTOR - DELTA * 1
  elseif  prox.horizontal[0] > THRESHOLD then
    motor.left.target  = -MOTOR + DELTA * 0
    motor.right.target =  MOTOR - DELTA * 0
  end
```

Figure 10.1: Searching with a proportional algorithm

10.2 Approaching

In this section we develop algorithms for slowing down and stopping the robot. It is similar to the `pounce` behavior of the cat-catches-mouse project in Chapter 2. The proportional algorithm in the previous section used a rather arbitrary measure of closeness: the number of sensors between the current position of the object and the rightmost sensor. In this section, the power setting of the motors will be proportional to a measure of the distance between the sensor and the object.

Specification: The object is placed in front of the center horizontal sensor. The robot moves towards the object and stops just before hitting it.

Previously, you were asked to determine the upper limit of the value of the horizontal sensor when it detected the object. For the projects in this section, a value close to the upper limit is given by the constant `TARGET`. Instead of checking if the value of the sensor is above a threshold, we compute the *error* defined as:

$$error = target\ distance - measured\ distance$$

The robot stops when the error is below a value given by the constant `TARGET_ERROR`.

An on-off controller

Program file `approach-on-off.aesl`

Here is the event handler for an on-off controller. It turns the motors off when the error is small, but leaves them at full power when the error is large:

```
onevent prox
  error = abs (TARGET - prox.horizontal[2])
  if error < TARGET_ERROR then
    callsub stop
  else
    motor.left.target  =  MOTOR
    motor.right.target =  MOTOR
  end
```

Programming note: The absolute value is used because the value of the sensor can be slightly higher than `TARGET` so the error can be negative. What problem can this cause? The answer is in the footnote, but think about it before you look.[4]

When I ran this program the robot moved so fast that it collided with the object!

A proportional controller

Program file `approach-p.aesl`

In a proportional controller the motor power is proportional to the error: The larger the error—the farther the robot is from the object—the higher the power; as the robot approaches the object, the error becomes small and the power is reduced. The power is determined by multiplying the error by a number called the *gain* for which we define a constant `GAIN_P`. Be careful to avoid saturation: the power must never be higher than the highest allowed value, which is 500 for the Thymio robot:

```
  motor_power = error * GAIN_P
  if motor_power > 500 then motor_power = 500 end
```

Experiment to find a value for the gain that gives good results. I was able to get the robot to stop just before hitting the object.

Programming note: The variable `error` has values in the range from 0 to about 4000, while we want the motor power to have values in the range 0 to 500. Therefore, the gain

[4]If the value of the sensor is large, the absolute value of the difference could be larger than `TARGET_ERROR` and the robot will continue moving instead of stopping. Modify the program to solve this problem.

should be $1/8 = 0.125$ (or less). In Section 3.1, we solved this problem by scaling the integer division, but there is a better solution.

The Aseba native function `math.muldiv(A, B, C, D)` computes $A = (B \cdot C)/D$ in 32-bit arithmetic. Therefore, `GAIN_P` can be defined to be 8 and the value of `motor_power` computed without using a scaling factor:

```
error = abs (TARGET - prox.horizontal[2])
call math.muldiv(motor_power, error, 1, GAIN_P)
if motor_power > 500 then motor_power = 500 end
```

Programming note: In my program, the statements in the event handler for `prox` are only run if the state is `on` and not `off`. However, I placed the computations of `error` and `motor_power` before checking the state so that I can observe them in the `Variables` pane without actually running the algorithm.

A proportional-integral controller (advanced)

Program file `approach-pi.aesl`

While the proportional controller does not collide with the object, it may not get as close as we want it to. A *proportional-integral controller* can to a better job by using the accumulated error to control the power setting. In mathematics, an accumulated sum is called an integral.

To implement a proportional-integral controller, use a variable `error_sum` to accumulate the error and then add it to the motor power after scaling by the reciprocal of the integral gain `GAIN_I`. To prevent `error_sum` from growing too large, the `error` is divided by a scale factor `SUM_FACTOR` before it is added to `error_sum`:

```
error = abs (TARGET - prox.horizontal[2])
error_sum = error_sum + (error / SUM_FACTOR)
call math.muldiv(motor_power, error, 1, GAIN_P)
motor_power = motor_power + error_sum / GAIN_I
if motor_power > 500 then motor_power = 500 end
```

Experiment with the gains to see if you can detect a difference between the proportional controller and the proportional-integral controller.

Chapter 11

Fuzzy Logic Control

The control algorithms in the previous chapter used mathematical computations. An alternate approach is to use *Fuzzy logic*, which was originally proposed by Lotfi A. Zadeh in 1965. In fuzzy logic, the control algorithms use *rules*:

- If the *temperature is low* and the *humidity is high*, set the *heater to high*.
- If the *temperature is low* and the *humidity is low*, set the *heater to medium*.

The logic is "fuzzy" because the rules are expressed in terms of *linguistic variables* like *temperature* which do not have precise mathematical specifications, but only imprecise linguistic specifications like *high*.

In this chapter we implement a fuzzy logic controller for the task from the previous chapter of approaching an object.

> This project is complex. It assumes that you have studied an introduction to fuzzy logic, for example, from this online tutorial or from Section 2.2 of Kevin M. Passino and Stephen Yurkovich. *Fuzzy Control*, Addison-Wesley, 1998.

Overview

A fuzzy logic controller consists of three phases that are run sequentially:

Fuzzify The values of the sensors are converted into values of linguistic variables called *premises*. Each premise specifies the *certainty* with which we believe that the variable is true. There will be three variables: `far`, `closing`, `near`.

Apply rules A set of *rules* expresses the control algorithm. Given a set of premises, a *consequent* is inferred. There will be five variables for the consequents: `very fast`, `fast`, `cruise`, `slow`, `stop`.

Defuzzify The consequents are combined in order to produce an *crisp* output, which for our project is the power applied to the motors.

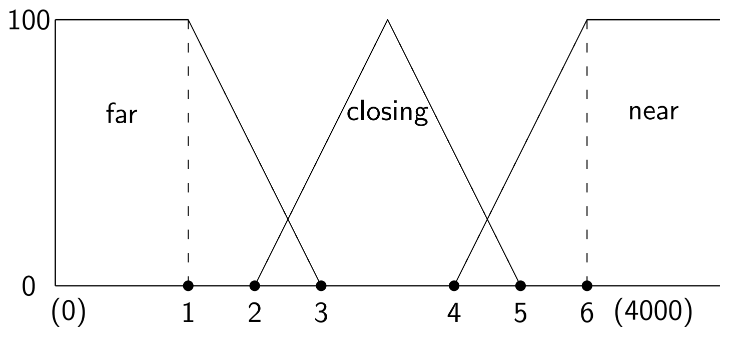


Figure 11.1: Fuzzify the center horizontal sensor

Certainties of the premises and consequents are given in the range 0.0 (not at all certain) through 1.0 (completely certain). Since Aseba does not support floating-point numbers, they will be scaled to the range 0 through 100.

Fuzzify

When approaching an object, the value read by the center horizontal sensor increases from 0 to a large number: over 4000, depending on your robot and the reflecting properties of the object.[1] The precise value returned by the sensor needs to be fuzzified—converted to the value of a linguistic variable. Figure 11.1 shows three graphs for converting the sensor values into certainties of the variables for `far`, `closing` and `near`. The x-axis is the value returned by the sensor and the y-axis is the certainty.

The labeled points on the x-axis refer to thresholds: `(1) FAR_LOW, (2) CLOSING_LOW, (3) FAR_HIGH, (4) NEAR_LOW, (5) CLOSING_HIGH, (6) NEAR_HIGH`. If the value of the sensor is below `(1) FAR_LOW`, then we are completely certain that the object is far away. If the value is between `(2) CLOSING_LOW` and `(3) FAR_HIGH` then we are somewhat certain that the object is far away but also somewhat certain that it is closing. The fuzziness results from the overlapping ranges: when the value is between point (2) and point(3), we can't say with complete certainty if the object is far away or closing.

[1]It is a good idea to use reflecting tape as noted on page 4.

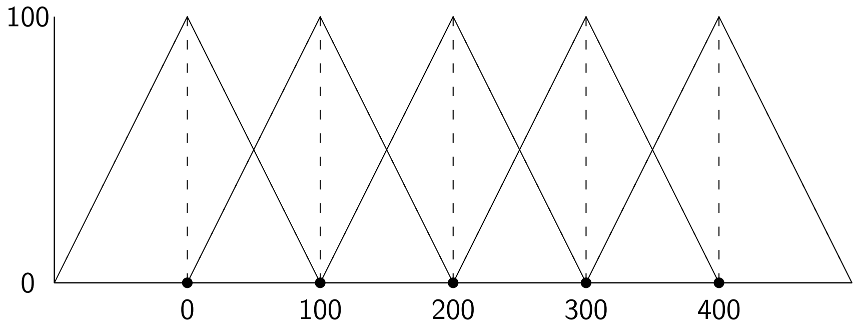


Figure 11.2: Defuzzify to obtain the crisp motor setting

Apply rules

The three premises—the certainties of `far`, `closing` and `near`—are used to compute five consequents using the following rules:

1. If `far` then `very fast`
2. If `far` and `closing` then `fast`
3. If `closing` then `cruise`
4. If `closing` and `near` then `slow`
5. If `near` then `stop`

The certainties of the consequents resulting from rules 1, 3, 5 are the same as the certainties of the corresponding premises. When there are two premises, as in rules 2 and 4, the certainties of the consequents are the minimum of the certainties of the premises. Since *both* of the premises must apply, we can't be more certain of the consequent than we are of the smaller of the premises.

Defuzzify

The next step is to combine the consequents, taking into account their certainties. Figure 11.2 shows the output motor powers for each of the five consequents. For example, if we were completely certain that the output is `cruise`, the center graph in the figure shows that the motor power would be set to 200, but if we were less certain, the motor power would be less or more.

For our rules more than one consequent can have positive values. These are combined using the areas of the trapezoids defined by the certainties. Figure 11.3 shows the consequent `cruise` with a certainty of 40 and the consequent `fast` with a certainty of 20.

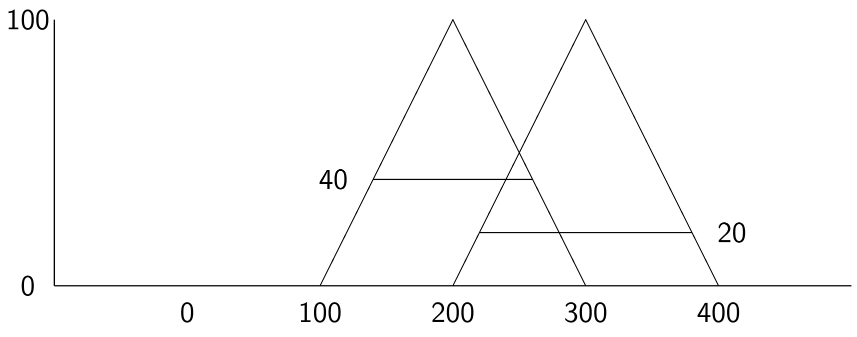


Figure 11.3: Areas defined by the certainties of the consequents

Let w and h be the width and height of the triangle. Then the area of the trapezoid bounded by the certainty h' is given by the formula:[2]

$$wh' - wh'(\frac{h'}{2h}).$$

For $w = h = 100$ and $h' = 20, 40$, the areas are:

$$\begin{array}{c} 100 \cdot 20 - 100 \cdot 20 \cdot \frac{20}{200} = 1800 \\ 100 \cdot 40 - 100 \cdot 40 \cdot \frac{40}{200} = 3200. \end{array}$$

To obtain a crisp value, the *center of gravity* is computed. This is the sum of the areas weighted by the center of each triangle divided by the sum of the areas. For the example—and removing the scaling factor of 100—this gives:

$$\frac{18 \cdot 3 + 32 \cdot 2}{18 + 32} = 39.$$

This is the crisp value can now be scaled to the range 0 through 500 of the motors.

State machine

There are two states: 0 for `off` and 1 for `on`.

Constants

Define the six thresholds as shown in Figure 11.1.

[2]The derivation of the formula is given in an appendix to this chapter.

Variables

The main variables are a three-element array `premises` and a five-element array `consequents`. Another five-element array contains the `centers` of the output membership functions. The output is stored in the variable `crisp`.

Event handlers and subroutines

- The state is turned on and off using the front and center button event handlers.
- When a proximity event occurs, three subroutines `fuzzify`, `apply_rules`, `defuzzify` are called in sequence, and then the value of `crisp` is assigned to the motor targets.
- Subroutine `fuzzify` checks the value of the center horizontal sensor against the threshold values. The certainties defined by the functions in Figure 11.1 are stored in the array `premises`.
- Subroutine `apply_rules` checks for non-zero values in `premises` and then assigns certainties to the array `consequents` as described above.
- Subroutine `defuzzify` computes the areas and center of gravity to obtain a crisp value.

Programming notes

- You can't really see how the algorithm works by observing the robot. Instead, observe the variables in the table at the left of the window. Even better, assign the values of the premises and consequents to the circle LEDs after calling `apply_rules`. Since the certainties are scaled to the range 0 through 100, by dividing by three, the certainties can be seen from the intensity of the LEDs:

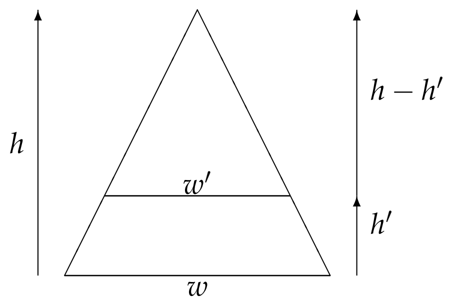
```
call leds.circle(premises[0]/3, premises[1]/3, premises[2]/3,
                 consequents[0]/3, consequents[1]/3, consequents[2]/3,
                 consequents[3]/3, consequents[4]/3)
```

- Scale the values appropriately since floating-point arithmetic is not supported.
- The native function `math.multdiv` is very useful for carrying out the computations.

Program file `fuzzy.aesl`

Appendix: Computing the area from the certainty

The following diagram shows a triangle of width w and height h cut off at height h':



The area of the trapezoid is the difference between the areas of the two triangles:

$$a = wh/2 - w'(h - h')/2.$$

By similar triangles:

$$\frac{h}{h - h'} = \frac{w}{w'},$$

so:

$$w' = \frac{w(h - h')}{h}$$

Substituting:

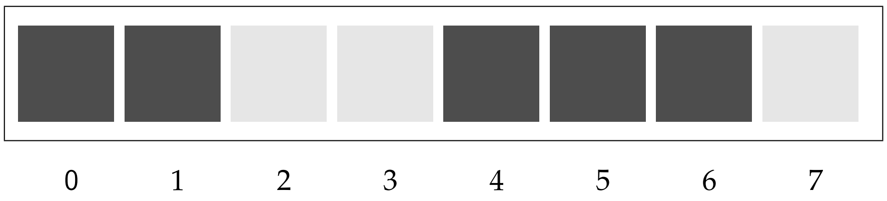
$$\begin{aligned} a &= \frac{wh}{2} - \frac{w(h - h')(h - h')}{2h} \\ &= \frac{w(h^2 - (h - h')^2)}{2h} \\ &= \frac{w(h^2 - h^2 + 2hh' - h'^2)}{2h} \\ &= \frac{w(2hh' - h'^2)}{2h} \\ &= wh' - wh'(\frac{h'}{2h}). \end{aligned}$$

Chapter 12

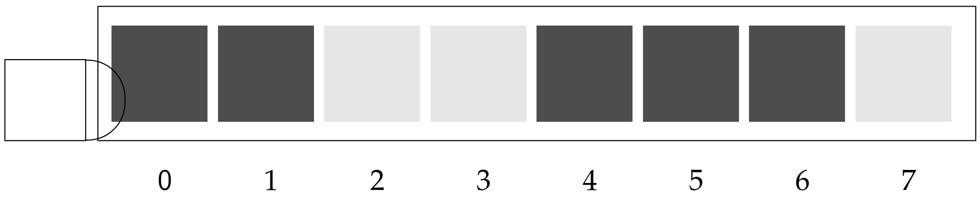
Localization

How can the robot know where it is?

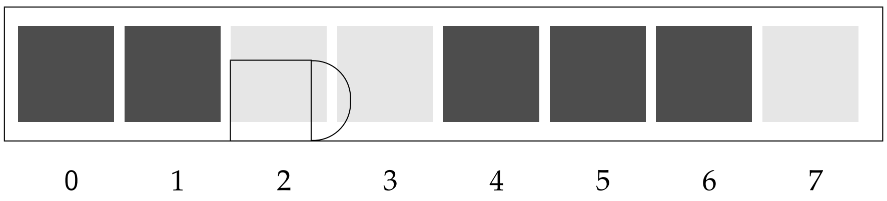
Consider a robot that is navigating within a known environment for which it has a *map*. The following map shows a wall with five doors (dark gray blocks) and three areas where there is no door (light gray blocks):



The task of the robot is to enter a specific door, say the one at position 4. But how can the robot know where it is? In Chapters 6, 9, you learned about odometry, which enables a robot to determine its current position given a know starting position. For example, if the robot is at the left end of the wall:



it knows that it has to travel four times the width of each door, while if the robot is at the following position:

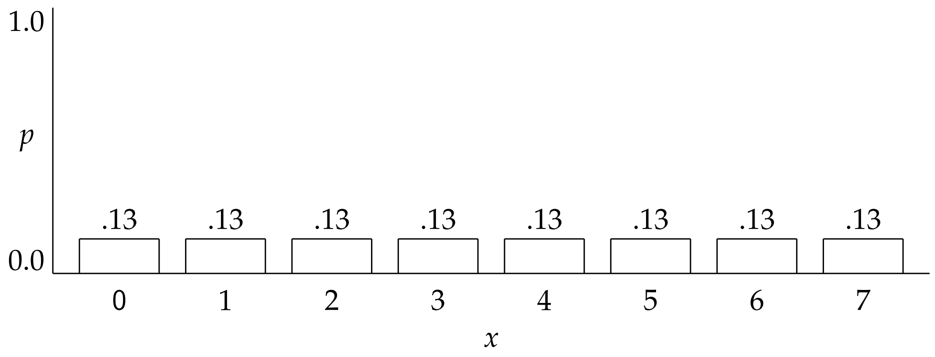


the required door is the next one to the right. Unfortunately, odometry is subject to error in the measurement of the robot's speed and direction, so it is quite likely that as time goes by the robot will get lost. The purpose of *localization* algorithms is to determine the robot's position within a known environment. In this chapter, we implement a simple one-dimensional version of the *Markov algorithm*.

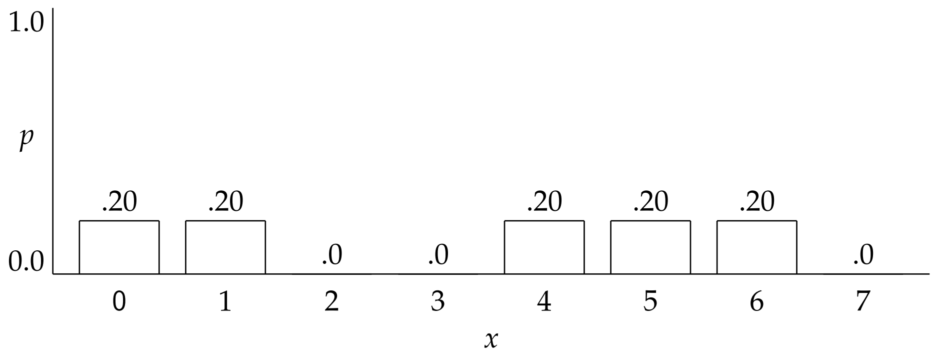
> This chapter follows the explanations of the algorithms given in the first week of the online course *Artificial Intelligence for Robotics* by Sebastian Thrun, which can be found at https://www.udacity.com/course/artificial-intelligence-for-robotics–cs373.

Sensing increases certainty

The robot assigns a probability to the eight positions where it might be located. Initially, it has no idea where it is, so each position will be assigned the probability 1.0/8 = .125:[1]

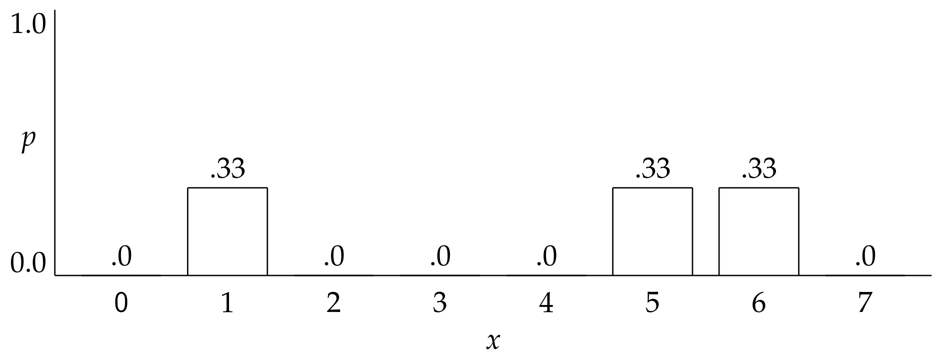


Suppose now that the robot uses its sensors and detects a dark gray area. Now its uncertainty is reduced, because it knows that it is in front of one of the five doors. The probability distribution shows .2 for each of the doors and .0 for each of the walls:



[1]The labels on the bars in the graphs in this chapter are rounded.

Next our mobile robot moves forwards (to the right) 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 .33 for each of these three positions:



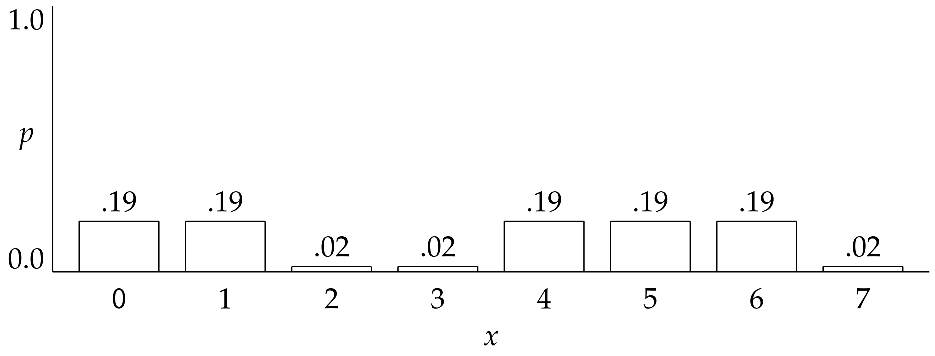
After the next step of the robot, if it again detects a door, it is without doubt at position 6, while if it does not detect a door, it is either at position 2 or position 7.

Uncertainty in sensing

The difference between a dark gray door and a light gray wall is not too great, and as the paint ages it might become more and more difficult to distinguish between them. Furthermore, as you have experienced, the robot's sensors do not give uniform responses and they can change over time. Therefore, the robot cannot detect with complete certainty whether is senses a door or a wall.

We will model this by assigning probabilities to the detection. If the robot senses dark gray, we specify that the probability is .9 that it has (correctly) detected a door and .1 that it has (mistakenly) detected a wall. Conversely, if it senses light gray, the probabilities are .1 for a door and .9 for a wall.

After sensing dark gray at a position where there is a door, we only know with probability $.125 \times .9 = .1125$ that a door has been correctly detected; however, there is still a $.125 \times .1 = .0125$ possibility that it has mistakenly detected a door. After normalizing (see sidebar), the probability distribution is:



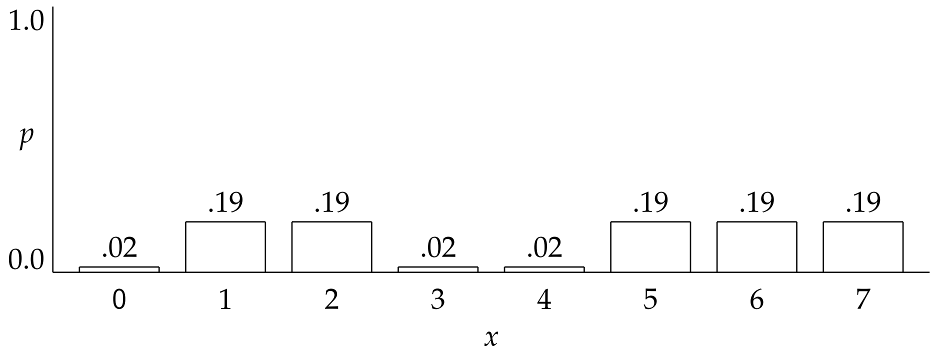
The set of probabilities of the possible outcomes of an event must add up to 1 since one of the outcomes must occur. The probabilities that we computed add up to much less than 1:

$$.1125 + .1125 + .0125 + .0125 + .1125 + .1125 + .1125 + .0125 = .575.$$

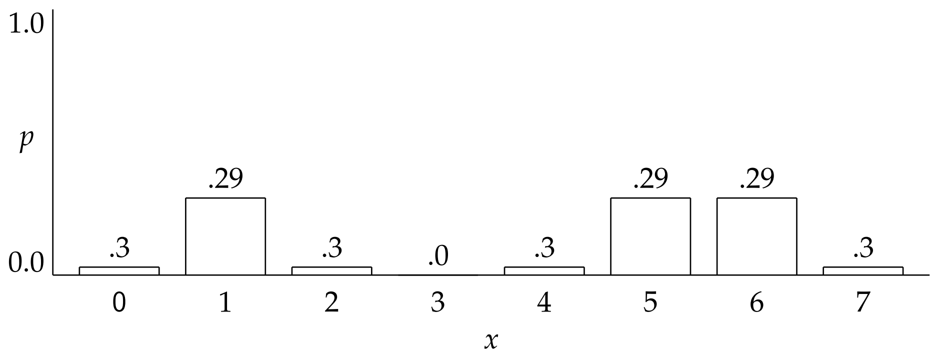
We must *normalize* the probabilities by dividing by the sum .575 so that the sum will be 1. We have $.1125/.575 \approx .19$ and $.0125/.575 \approx .02$, so:

$$.19 + .19 + .02 + .02 + .19 + .19 + .19 + .02 \approx 1.$$

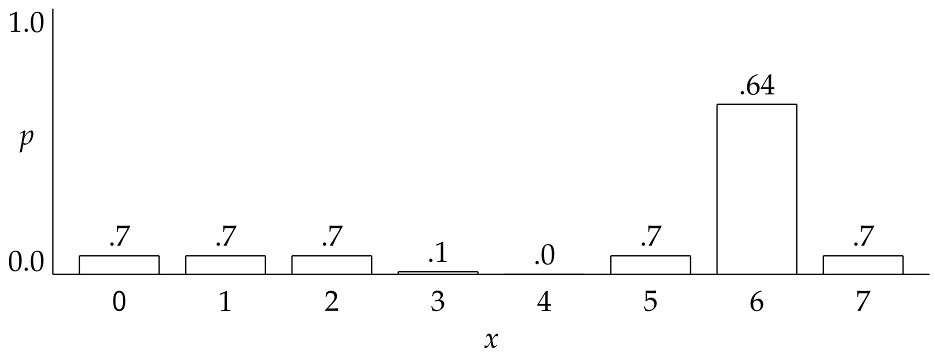
The robot now moves to the right and we will assume that the probability distribution is cyclic, that is, the value associated with position 7 becomes the value associated with position 0:



If the robot again senses dark gray, the probability of being at positions 1, 5 or 6 should increase. Computing the probabilities by multiplying and normalizing gives:



If the robot moves right again and senses a third dark gray area, the distribution becomes:



Not surprisingly, the robot is almost certainly at position 6.

Specification

In order the careful examine the computations, our initial implementation will not move the robot. Instead, we will place the robot over black and white areas on the table and touch a button: first, to cause the robot to read the ground sensors and to compute the new probability distribution, and then to move the distribution (cyclically) to the right. The values of the variables can be examined in the **Variables** table (see page 3.1).

The values of the probability distributions will also be displayed in the circle LEDs. The robot will be localized—will have computed its position with a high degree of certainty—when exactly one LED is bright and the others are dim.

System design

There are two subroutines: `sense` which computes the changes in the probabilities as a result of reading the sensors, and `move` which virtually moves the robot one position to the right.

State machine

There are two states that remember whether `sense` or `move` is to be performed next.

Constants

- LED: A scale factor for displaying probabilities in the LEDs.
- THRESHOLD: The value below which black is detected.
- P_HIT: The probability of detecting black when on a black area or detecting white when on a white area.
- P_MISS: The probability of detecting black when on a white area or detecting white when on a black area.

Variables

- `world`: An 8-element array with the *map* of the environment. It is initialized by `[1, 1, 0, 0, 1, 1, 1, 0]`, corresponding to the diagram at the beginning of the chapter, where 1 means that a door exists at that position and 0 means that a wall exists at that position.
- `beliefs`: An 8-element array with the current probability distribution of what the robot believes is its position. The array is initialized by the uniform distribution with 125 in each element.
- `state`: State variable 0 = sense, 1 = move.
- Auxiliary variables: `i`, `hit`, `temp`, `sum`.

Event handlers and subroutines

- Event `button.center`: Initialize `beliefs` to the uniform distribution and set `state` to 0.
- Event `button.forward`: Call subroutine `sense` or `move` depending on the state and update the value of `state`.
- Subroutine `display_beliefs`: Display the values of the elements of `beliefs` in the circle LEDs after scaling approximately to the range 0 through 32.
- Subroutine `sense`: Perform the following operations:

  - If both ground sensors are below the `THRESHOLD`, detect a door, otherwise detect a wall.
  - For each position in `world`, if the detection matches the element (door or wall), multiply the corresponding position in `beliefs` by `P_HIT`; otherwise, multiply by `P_MISS`.
  - Normalize the elements in `beliefs` by dividing each one by the sum of the elements.

- Subroutine `move`: Move elements of `beliefs` one position to the right cyclically.

Programming notes

- The probabilities in `beliefs` are scaled to the range 0 through 1000.
- The factors `P_HIT` and `P_MISS` are scaled to the range 0 through 10.
- Use the native function `math.multdiv` to carry out the computations.

Program file `local-simple.aesl`

Experiments

1. We showed how three measurements are sufficient to localize the robot if it starts at position 4. Try other positions and see if the algorithm works.
2. Experiment with the values of `P_HIT` and `P_MISS`. How low can you specify `P_HIT` and have the algorithm still work?

Uncertain motion

As well as uncertainty in the sensors, robots are subject to uncertainty in their motion. We can ask the robot to move one "position" to the right, but it might move two positions, or it might move very little and remain in its current position. Let us modify the program for localization to take this uncertainty into account.

Let b be the belief array. The subroutine `sense` computed b', the new values of the elements, using the formula:[2]

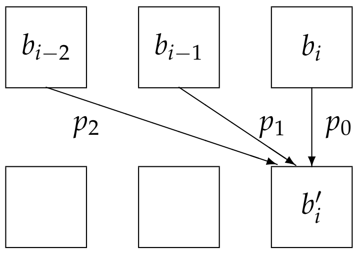
$$b'_i = b_i \times p,$$

[2]The computations must take account of the cyclic movement.

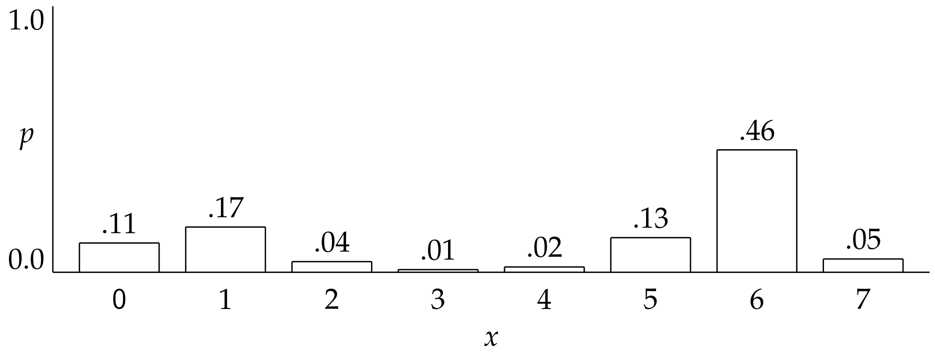
where p is `P_HIT` or `P_MISS`, depending on whether the sensor value matches the map or not. The subroutine `move` simply moved the values to the right $b'_i = b_{i-1}$. With uncertainty, `move` must perform the following computation:

$$b'_i = (b_{i-2} \times p_2) + (b_{i-1} \times p_1) + (b_i \times p_0),$$

where p_n is the probability that the robot has moved n positions:



It is highly likely that the robot will move correctly, so reasonable values are $p_1 = 0.8$ and $p_0 = p_2 = 0.1$. (Don't forget to normalize after performing the computation.) With these values for uncertainty, the beliefs after sensing three black areas are:



The robot is likely at position 6, but it is less certain than before (.46 instead of .64).

Programming notes

Declare an array `save` of the same length as `beliefs` and copy to `save` (using the native function `math.copy`) the values of `beliefs` before starting the computation. The original values are thus remain available.

Program file `local-uncertain-motion.aesl`

Localization of a moving robot

Let us now attempt localization of a moving robot. Place three strips of tape at regular intervals on a table to simulate the three doors at positions 4, 5 and 6. Set the motors to a constant speed and set a timer to the amount of time it takes the robot to travel from one strip to the next. When the timer event occurs, call the subroutines `sense` and `move`. Start the robot before the first strip and see if it locates itself at position 6 after sensing the third strip.

Program file `local-move.aesl`

Appendix: The mathematics of localization

Bayes rule

The problem of localization is to determine the probability $p(x_i)$ that the robot is at position x_i for all i. In particular, given the reading z of the sensor, what is the probability that we are at position x_i? This is expressed as a *conditional probability* $p(x_i \mid z)$. The data we have available are the *current* probability $p(x_i)$ that we are at x_i, and $p(z \mid x_i)$, the conditional probability that the sensor reads z if we are in fact at x_i. In the example, initially, $p(x_i) = .125$ for all positions x_i, and, if we are at a door, the probability that the sensor detects this correctly is .9. We can now *multiply* these two probabilities to obtain (the unnormalized) probability $\bar{p}(x_i \mid z)$, in the example, $.125 \times .9 = .1125$.

To normalize the probability, we have to divide it by the sum of the results obtained by multiplying at each position:

$$p(z) = \sum_i p(z \mid x_i) \cdot p(x_i).$$

In the example, the sum is .575 and $.1125/.575 = .19$.

The formula:

$$p(x \mid z) = \frac{p(z \mid x) \cdot p(x)}{p(z)}$$

is called *Bayes rule*.

Total probability

To compute the change in probability when the robot moves, we used the formula:

$$b'_i = (b_{i-2} \cdot p_2) + (b_{i-1} \cdot p_1) + (b_i \cdot p_0).$$

Expressed in terms of probability, this is:

$$p'(x_i) = p(x_{i-2}) \cdot p(x_i \mid x_{i-2}) \;+\; p(x_{i-1}) \cdot p(x_i \mid x_{i-1}) \;+\; p(x_i) \cdot p(x_i \mid x_i),$$

where $p(x_i \mid x_j)$ is the probability that the current position is x_i given that the previous position was x_j. The general case is:

$$p'(x_i) = \sum_j \, p(x_j) \cdot p(x_i \mid x_j),$$

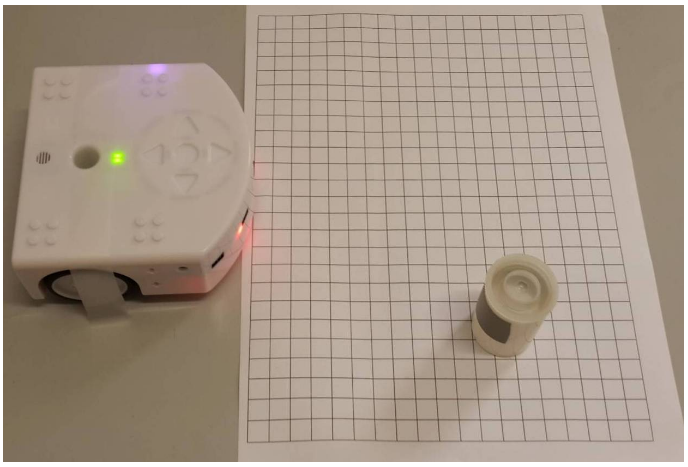
which is called the *total probability*.

Chapter 13

Thymio the Surveyor

Land surveying is a set of techniques for determining the precise position of points on the earth.[1] Since ancient times, surveying was used to mark out fields and to build large structures. This project uses the Thymio to determine the location of an object using two techniques: measuring the angle and distance to the object, and measuring the angle to the object from two different positions. The results will be very inaccurate, but the techniques and computations reflect actual practice.

Take a piece of paper marked with a grid of 1 cm squares. Place the Thymio so that its front panel is at the left edge of the grid and place an object somewhere on the grid:[2]



Measuring angles using the Thymio

The sensors are on the front panel of the robot, which forms an arc of a circle whose center is the hole used to place a pencil for drawing. Place a protractor with its center over the hole and use a thread to form a line from the center to a sensor (Figure 13.1). Read the angle from the protractor. Since the sensors are spaced uniformly, we define a constant `THETA` and the five angles will be:

[1]See the Wikipedia article on surveying for an overview.

[2]Print the file `grid.pdf` in the `images` directory. The file was generated from the LaTeX file `grid.tex`. We used reflecting tape on the object as described on page 4 to increase the detection range.

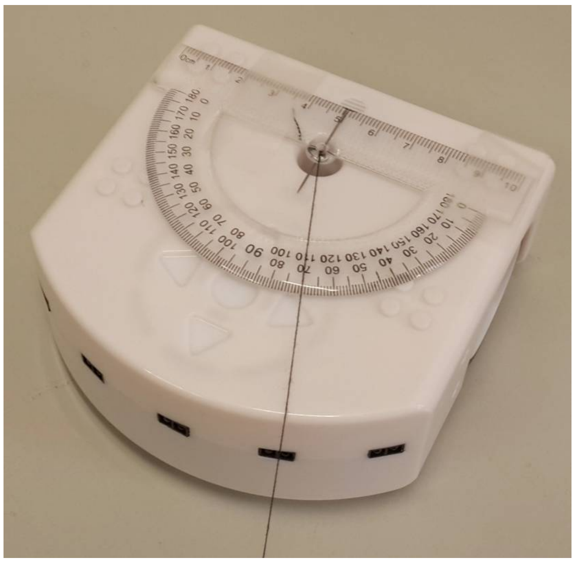


Figure 13.1: Measuring the angle between the sensors

```
-2*THETA, -THETA, 0, -THETA, 2*THETA.
```

13.1 Determining position from an angle and a distance

Chapter 8 describes how to determine the distance to an object by relating the values returned by the horizontal proximity sensors to distance in centimeters. The angle to the object is determined by the horizontal sensor that returns the highest value.

Computing the coordinates

Figure 13.2 shows the geometry of the surveyor. The object is indicated by the dark point. The Thymio is at the origin of a coordinate system with $(0, 0)$ defined as the hole. The x-axis goes through the axis of the wheels, while the y-axis points forwards. Given the angle θ to the object as defined by the horizontal proximity sensor with the highest value, and the distance d to the object defined by the value returned by that sensor, the coordinates can be computed using elementary trigonometry.

Variables and constants

- `THETA`: The angle between two sensors.

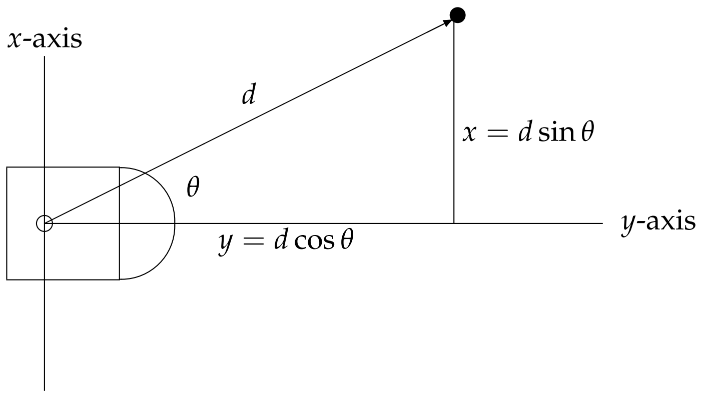


Figure 13.2: The geometry of the surveyor

- `OFFSET`: The distance between the hole in the Thymio and its front panel.
- `proximity`: An array for storing the values read from the sensor.
- `max_prox`: The proximity sensor with the maximum value.
- `max_value`: The value returned by sensor `max_prox`.
- `distance`: The distance from the Thymio hole to the object.
- `angle`: The angle from the y-axis to the object.
- `x, y`: The coordinates of the object.

You may need additional variable for indices and for storing temporary values.

Event handlers and subroutines

It is sufficient to use one event like touching a button to initiate the measurements and computations, but I used two events so that the distance and angle are displayed on the top, bottom and circle LEDs before the computation is done, and later the coordinates are displayed. The subroutines for displaying data on the LEDs will not be described here; you are invited to design your own method for doing this.

There are three subroutines for performing the computations:

- `get_angle`: Get the angle by checking which horizontal proximity sensor returns the maximum value.
- `get_distance`: Compute the distance corresponding to the maximum sensor value.

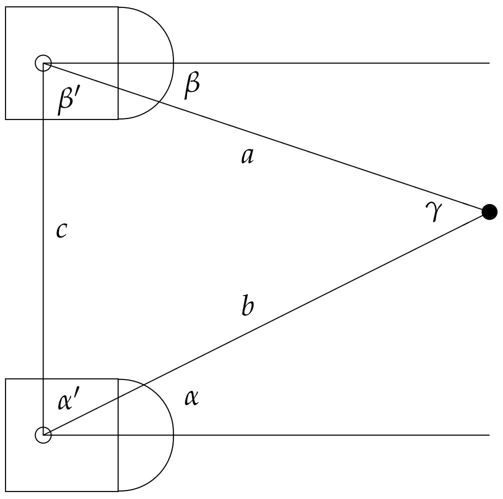


Figure 13.3: Triangulation

- `get_coordinates`: The computation of the coordinates from the distance and angle is conceptually very simple, but difficult to do on the Thymio because the trigonometric functions do not use standard values for parameters like degrees or radians. Study the computations in Chapter 9 to see how this is done.

Programming notes

Program file `survey1.aesl`

Experiments

1. Make a more accurate determination of the angle. If two horizontal sensors return values that are "more or less" the same, use the average of their two angles.

13.2 Determining position by triangulation

Surveyors use microwave and light waves to measure distance. Before these were available, it was extremely difficult to accurately measure distances, especially distances to very distant or inaccessible objects. Angles were always relatively easy to measure. Therefore, *triangulation* was used in surveying. If you know two angles of a triangle and the length of the included side, you can compute the lengths of the other sides. Once these are known, the position of a distant object can be computed as we did above.

Consider the setup shown in Figure 13.3. The Thymio first measures the angle α; then it is moved a distance of c and the angle β can be measured. The lengths a and b can be computed using the *law of sines* for a triangle:

$$\frac{a}{\sin\alpha'} = \frac{b}{\sin\beta'} = \frac{c}{\sin\gamma}.$$

We have used the interior angles of the triangle $\alpha' = 90^\circ - \alpha$, $\beta' = 90^\circ - \beta$.

Given the angle α, let us we compute the length b. We have:

$$\gamma = 180^\circ - \alpha' - \beta' = 180^\circ - (90^\circ - \alpha) - (90^\circ - \beta) = \alpha + \beta.$$

From the law of sines:

$$b = \frac{c\sin\beta'}{\sin\gamma} = \frac{c\sin(90^\circ - \beta)}{\sin(\alpha+\beta)} = \frac{c\cos\beta}{\sin(\alpha+\beta)}.$$

Programming notes

Two events were used: touch the forwards button to measure α, then move the Thymio to the new position and touch the backwards button to measure β and compute b.

The object was placed at coordinates $(8, 12)$ and the two measurements were done after moving the Thymio 24 cm. The results were $\alpha = 20^\circ$ and $\beta = 40^\circ$, giving $b = 22$. Measuring the distance from the center hole to the object showed that this is a reasonable value. Of course, the extremely coarse measurement of the angles makes an accurate determination of the position impossible.

Program file `survey2.aesl`

Experiments

1. Program the Thymio so that it moves to the second position by itself after making the first measurement.

Part III

Multiple Robots

Chapter 14

Infrared Communications

The infrared horizontal proximity sensors of the Thymio are normally used for detecting objects, but they can also be used for communication with another Thymio. This chapter describes techniques that enable two robots to communicate. A projects using communications will be presented in the next chapter.

Using Aseba Studio with two robots

Run two copies of Studio from the desktop. If you have two Thymio robots connected to two different USB ports, you will see that the two robots are listed in the initial target-selection window. Choose a different robot for each copy of Studio. When you load and run a program from one copy of Studio, it will run on the robot connected to that copy. In the following sections, each robot will have a separate program; to identify them, we append `tx` to one file name and `rx` to the other.

Establishing communications

To enable and disable infrared communications (IRC), use the native function:

```
  call prox.comm.enable(parameter)
```

A value of 1 for the parameter enables communication, while a value of 0 disables it.

For convenience, in each program in this chapter we associate a button with enabling and disabling IRC. Use the top LEDs to display the state:

```
onevent button.forward
  call leds.top(32,32,32)
  call prox.comm.enable(1)

onevent button.backward
  call leds.top(0,0,0)
  call prox.comm.enable(0)
```

Test IRC

One Thymio TX transmits a message to another Thymio RX. The content of the message changes every second. RX changes its top color depending on the received message.

When IRC is enabled, the transmitter transmits the value of the predefined variable `prox.comm.tx` every 100 ms. We use a timer to change its value every second:

```
timer.period[0]=1000

onevent timer0
  if  prox.comm.tx == 1 then
   prox.comm.tx = 2
  else
   prox.comm.tx = 1
  end
```

When a robot receives a message, the event `prox.comm` occurs and the value that was transmitted in the message is copied into the predefined variable `prox.comm.rx`. The value of this variable is used to decide on the top color:

```
onevent prox.comm
  if prox.comm.rx == 1 then
   call leds.top(0,32,0)
  else
   call leds.top(32,0,0)
  end
```

Program file `test-*.aesl`

The value transmitted and received can use 10 bits: the values 0–1023.

Come here

One Thymio TX will send a message to ask a second Thymio RX to come meet it. When RX receives a message it will move forward until it is near TX, at which point it will stop. This demonstrates that IRC does not interfere with the normal functioning of the sensors for detecting objects.

The content of the message is not important so TX sets `prox.comm.tx` to an arbitrary value. RX turns the motors on when it receives a message and turns the motors off when its center sensor detect TX:

```
onevent prox.comm
  motor.left.target = 100
  motor.right.target = 100
  call leds.top(0, 32,0)

onevent prox
  if prox.horizontal[2] > 3000 then
    motor.left.target = 0
    motor.right.target = 0
    call leds.top(32, 0,0)
    call prox.comm.enable(0)
  end
```

Since the message is transmitted repeatedly (ten times per second), the event `prox.comm` will occur repeatedly, but that is not a problem since the motor is always set to the same value. However, when RX detects TX, IRC must be disabled so that subsequent events do not turn on the motor again.

Program file `come-*.aesl`

The behavior depends on the message

Let us extend the previous project so that the movement of the receiving robot RX depends on the message received. TX will send a message whose value (1 or 2) depends on which button is touched. RX will go either forwards or backwards:

```
onevent prox.comm
  if prox.comm.rx == 1 then
    motor.left.target = 100
    motor.right.target = 100
    call leds.top(0, 32,0)
  else
    motor.left.target = -100
    motor.right.target = -100
    call leds.top(0, 0, 32)
  end
```

Program file `come-go-*.aesl`

Two-way communications

So far, one Thymio has always transmitted messages and the other received them. Let us extend the above program so that when the moving robot RX detects the robot TX, it stops and also transmits a message so that TX can turn its top LEDs red. RX adds an assignment of the value 2 to `prox.comm.tx` meaning that it has detected TX:

```
onevent prox
  if prox.horizontal[2] > 3000 then
    prox.comm.tx = 2
    motor.left.target = 0
    motor.right.target = 0
    call leds.top(32, 0,0)
  end
```

Add an event handler to TX so that it changes color when it receives this message:

```
onevent prox.comm
  if prox.comm.rx == 2 then
    call leds.top(32, 0, 0)
  end
```

Unfortunately, this program does not work as required. The reason is that the robot TX continues transmitting which causes the moving robot RX to start moving again. Since TX is near RX, the center sensor of RX continues to detect TX, changes its color to red, and sends the value 2 again. To prevent this behavior, we have to disable IRC once the value 2 is sent and received. We can either disable TX when it receives the value:

```
onevent prox.comm
  if prox.comm.rx == 2 then
    call leds.top(32, 0, 0)
    call prox.comm.enable(0)
  end
```

or we can disable IRC in RX, but we must do so only after the value 2 is sent. This is ensured by waiting until the *next* occurrence of the `prox` event:

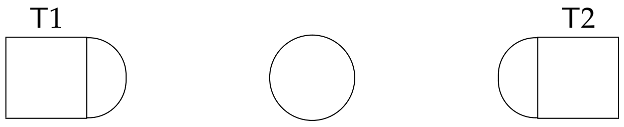
```
onevent prox
  if  prox.comm.tx == 2 then
    call prox.comm.enable(0)
  end
  if prox.horizontal[2] > 3000 then ...
```

Program file `come-arrived-*.aesl`

Chapter 15

Distributed Mutual Exclusion

Place two Thymios facing each other with a reflective object about halfway between them:



What will happen if we run the following program in both robots?

```
onevent button.forward
  motor.left.target = 100
  motor.right.target = 100

onevent prox
  if prox.horizontal[2] > 4000 then
    motor.left.target = 0
    motor.right.target = 0
  end
```

Program file `no-mutex.aesl`

If both forward buttons are touched at about the same time, the two robots will simultaneously move towards the object; each robot will stop when it is close to the object.

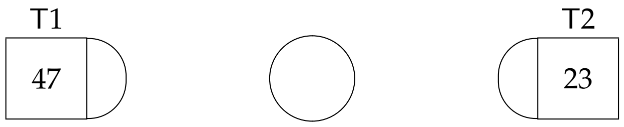
Distributed algorithms

Suppose that we want the robots to approach the object one-by-one. This is called the *critical section problem* or the *mutual exclusion problem*. It is a fundamental problem the area of computing called *concurrent and distributed programming*. Obviously, we can achieve this goal by touching the button of one robot, waiting until it has reached the object and then touching the button of the other robot. The difficulty with this solution is that it assumes a central computing device that makes decisions for both robots. This is unrealistic in practice: consider the Internet where millions of computing devices synchronize and communicate with no central control. The basic requirement for distributed algorithms is that the identical algorithm run on all participating computing devices and that the access to services is fair, that is, some devices will not receive preferential treatment.

The Ricart-Agrawala algorithm

We now describe a *distributed* algorithm for mutual exclusion.[1] How do people synchronize access to ticket counters, food service stations, and so on? They take slips of paper with numbers and are served in numerical order, thus implementing a queue. The queuing is facilitated by an electronic display, but a queue can be implemented simply by having people compare numbers with each other.

The two robots choose *random* numbers:



Each robots sends its number in a message to the other robot:

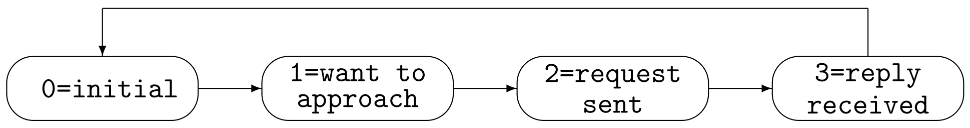
- T1 receives 23 from T2 and compares it with its own number 47. Since T2 has the lower number, it should be served before T1. Therefore, T1 sends a *reply* to T2 to indicate that it can approach the object.
- T2 receives the reply from T1 and approaches the object. *At the same time*, T2 has received the message from T1 with its number. Since 47 is greater than 23, T2 *does not* send a reply to T1; instead, it records that the sending of the reply is *deferred*. Since T1 does not receive a reply, it does not approach the object.
- When T2 is close to the object it stops. It also recalls that it has deferred the reply to T1 and sends it now.
- T1 receives the reply and approaches the object.

It is important to note that *both* robots run the same program and that the algorithm is fair, in the sense that if run repeatedly, the random numbers will give priority to each robot about half the time.

State machine

Touching the forward button indicates that the robot wants to approach the object. It sends a request and starts moving when the reply is received:

[1]The algorithm is based on the Ricart-Agrawala algorithm, which achieves mutual exclusion for any number of nodes repeatedly trying to enter a critical section. The algorithm it takes into account several difficulties that we ignore here for simplicity, for example, the possibility that two robots choose the same random number. A full explanation can be found in Chapter 10 of M. Ben-Ari. *Principles of Concurrent and Distributed Programming (Second Edition)*, Addison-Wesley, 2006.



Variables

- `number`: The number chosen by this robot (0–1022). The value 1023 is used for the reply message.
- `state`: The current state of the state machine.
- `deferred`: Set to 1 if the reply must be deferred.

Programming notes

The event handler for the center button will stop the motors, enable IRC, and set initial values of the variables, including `prox.comm.tx` and `prox.comm.rx`. The event handler for the forward button will set `state` to 1 to initiate the algorithm.

Most of the processing is done in the event handler for IRC communication. In state 1, a random number is chosen and sent to the other Thymio:

```
onevent prox.comm
  if state == 1 then
      call math.rand(number)
      number = abs(number) &amp; 1022
      prox.comm.tx = number
      state = 2
      return
  end
```

In state 2, the `prox.comm.rx` is checked to see if it is the reply message (1023); if so, the motors are started:

```
  if state == 2 then
    if prox.comm.rx == 1023 then
      motor.left.target = 100
      motor.right.target = 100
      state = 3
      return
    end
  end
```

If the robot is not competing to approach the object (it is in state 0) or if a lower number is received from the other robot, the reply is sent; otherwise it is deferred:

```
  if state == 0 or prox.comm.rx < number then
    prox.comm.tx = 1023
  else
    deferred = 1
  end
```

When the robot is close to the object the motors can be turned off. The variable `deferred` is checked to see if a reply must be sent to the other robot:

```
onevent prox
  if prox.horizontal[2] > 4000 then
    motor.left.target = 0
    motor.right.target = 0
    if deferred == 1 then
      prox.comm.tx = 1023
    else
      prox.comm.tx = 0
    end
  end
```

Program file `ra.aesl`

Testing the program

Place the robots and an object as shown in the above diagram.[2] Load and run the program into both robots. Touch the center buttons to initialize. Touch the forward button on one of the robots; only that robot should approach the object. Move the robot back to its starting position and touch the center button to reset. Now touch both forward buttons at the same time. One robot should approach and stop, and then the other robot should approach and stop.

[2]The robots have to be relatively close so that IRC works and slightly off-center so that the object doesn't obstruct IRC.