Artificial Intelligence Programming for Robots

October 5, 2019

This document forms a very basic tutorial for the BDIPython library. Please note that this library is still very preliminary and under intermittent development. There is no guarantee that this tutorial is up to date.

It is adapted from some Python programming exercises aimed at children in Year 9 in the UK so some of the instructions are quite basic - experienced Python programmers can probably skip some of the early sections on how to run IDLE and so on.

We are going to explore programming robots using something called *rational agent programming*. In rational agent programming an *agent* has *beliefs* about the world and *goals* it wants to achieve. It uses *rules* in order to decide what to do based on its beliefs and goals.

We will explore rational agent programming using a simulator of Pi2Go Robot.

1 Setting up the Simulator

Check clone pirover_simulator from `https://github.com/legorovers/pirover_simulator`

In the pirover_simulator folder you should find two files pysim.py and pysimosx.py. The first is for use on Windows and Linux machines and the second for use on Macs. You should run the simulator either from the command line or from a Python development environment such as IDLE.

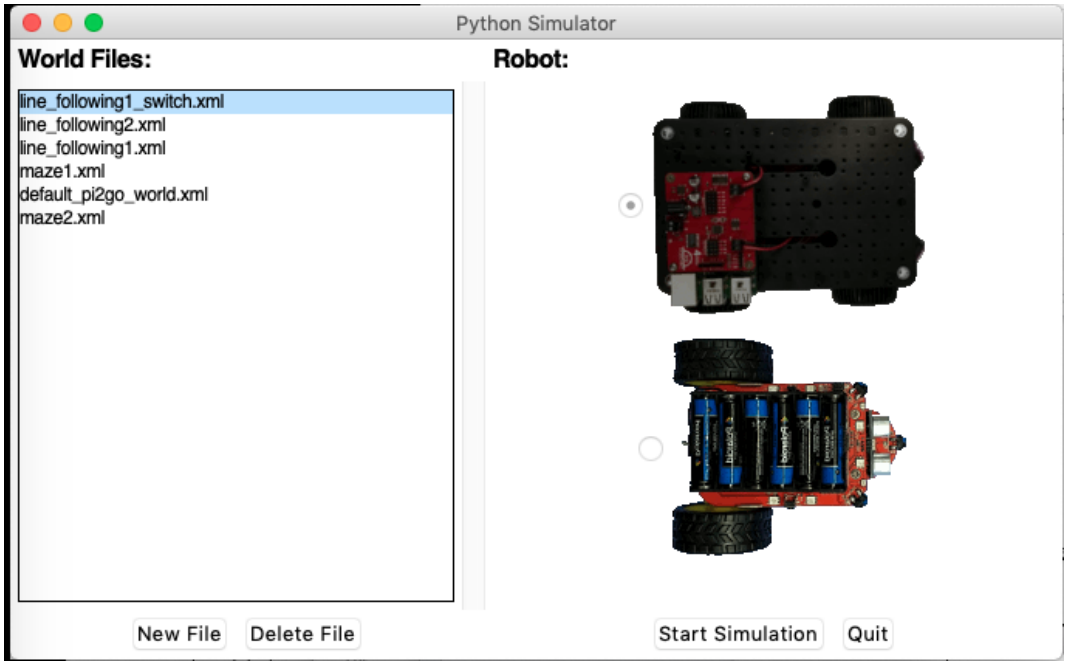
From the command line: Type python pysim.py (pysimosx.py on Macs) in the pirover_simulator folder then press return.

From IDLE: Start IDLE. A window should open and you should see something like:

```
Python 3.7.2 (v3.7.2:9a3ffc0492, Dec 24 2018, 02:44:43)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

Open pysim.py (pysimosx.py on Macs) from the File Menu. A file will open in Idle (ignore this) and a new menu item **Run** should appear. From **Run** select **Run Module**.

You should see:



From here you can start a simulation for either the Initio robot (the top image) or the Pi2Go robot (the bottom image) in a number of settings (the "World Files" on the left).

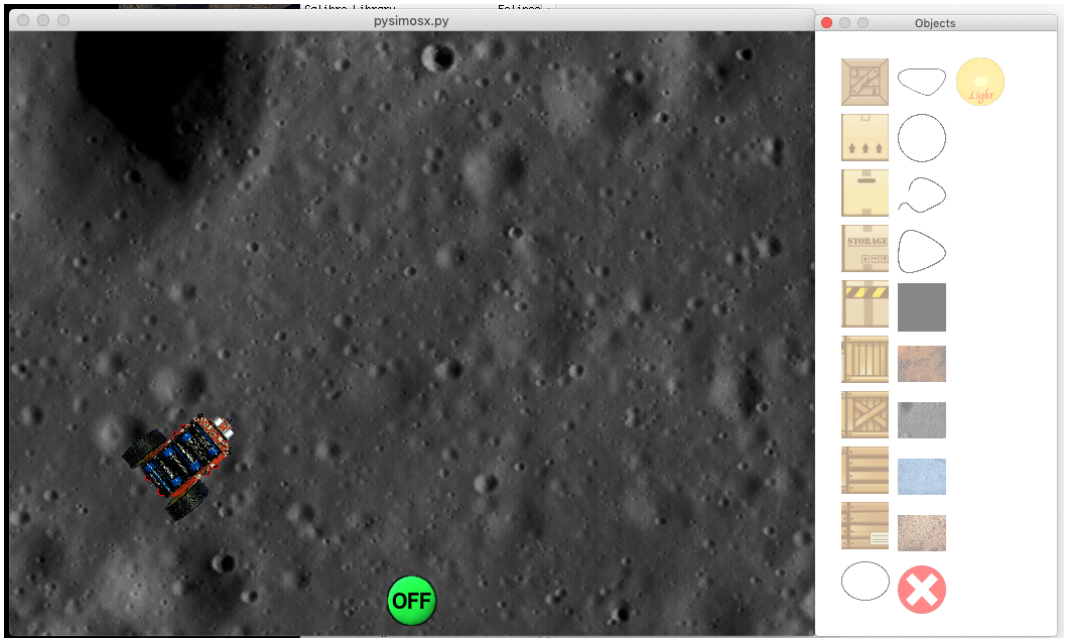
Select the Pi2Go robot and the default_pi2go_world .xml Then click on *Start Simulation*

Once the simulation has started, you can stop it by pressing Q.

1.1 Opening the Edit Window

You can change the environment your virtual Pi2Go encounters by using objects from the Edit window.

Press E in the PiRover Simulator. A second window, the Objects window, should open as shown in the screenshot below



1.2 Programming the Robot

Open a new IDLE window or a new command line. Try typing the following commands at the command line.

```
import simclient.simrobot as pi2go
pi2go.init()
pi2go.forward(10)
pi2go.stop()
```

2 BDIPython: Basics

In rational agent programming an *agent*, in this case your robot, acts according to various rules. The agent continually checks which of its rules are applicable, picks one of them and then executes that rule.

2.1 PYTHONPATH

Make sure pirover_simulator is on your PYTHONPATH. In BDIPython make sure BDIPython itself is on your PYTHONPATH and BDIPython/tests.

2.2 Adding a Rule

Task We will start programming with a very simple rule that is always applicable and which prints the input from a sensor to your screen. We will start with the input from

the small switch on the side of the real robot but displayed at the bottom of the simulator window. This switch is shown in figure 1.

Figure 1: The Switch

```
1  import pi2goagent
2
3  agent = pi2goagent.Pi2GoAgent()
4
5  def print_switch_rule():
6      switch_pressed = agent.sensor_value('switch_pressed')
7      print("Switch Pressed: ", switch_pressed)
8      return
9
10 agent.add_rule(print_switch_rule)
11 agent.run_agent()
```

Figure 2: A First Agent Program

Task Copy program shown in figure 2 into a file. Save the file and run it.

We will discuss each line of this program in turn. Take some time to make sure you understand what it is doing.

Line 1 First we import the Python code for the agent `import pi2goagent`.

Line 3 Then we create the *agent* using `agent = pi2goagent.Pi2GoAgent()`. This creates the agent as a Python *object*. It isn't necessary for this tutorial to know much about Python objects. They are much like other Python libraries and contain a number of functions which can be called using the `agent.function()` syntax which you can see in lines 6, 10 and 11. Unlike normal python libraries they can also contain data specific to the object, in this case a *belief base* and a *goal base* which we will discuss later.

Line 5 Here we declare a Python *function* called `print_switch_rule`. This represents our agent's rule.

A Python function is defined with the keyword `def` followed by the function name, brackets and then a colon. On the line below and indented (like when using `if`) are the Python commands to be executed by the function. These are mostly just normal Python commands and should look familiar to you.

In this case the rule contains three lines of Python commands - on lines 6, 7 and 8 - which will be executed when the function is executed (in this case when the agent executes the rule).

Line 6 Uses one of the agent's functions, `agent.sensor_value(sensor_key)`. This function returns the current value being sent by a particular sensor, in this case the switch. This value is accessed using a string called the *sensor key*. This string is passed to the function as an *argument*. In the case of this program the sensor key is `'switch_pressed'`. So on this line we find the value of the the switch and store it in the Python variable `switch_pressed`.

Line 7 prints out the value of the switch's sensor using Python's `print command` and taking the value stored in `switch_pressed`.

Line 8 The function ends by *returning*.

Line 10 Here we add our rule to the agent. `agent.add_rule(rule_name)` is a function that takes the name of a rule as an argument.

Line 11 Here we start the agent running. `agent.run_agent()` is a function that takes no arguments but starts the agent's *reasoning cycle* which is discussed next.

Task Run your program using the `python` command at the command line in your second tab.

What happens when the switch is pressed?

You will need to use Ctrl-C to stop the program.

All Pi2Go agent programs follow this basic format:

1. Create an agent object.
2. Define rules for the agent as python functions.
3. Add the rules to the agent.
4. Run the agent.

When the agent runs it operates a *reasoning cycle*. The Pi2Go agent reasoning cycle is shown in figure 3. The program in figure 2 has only one rule. So each time around the

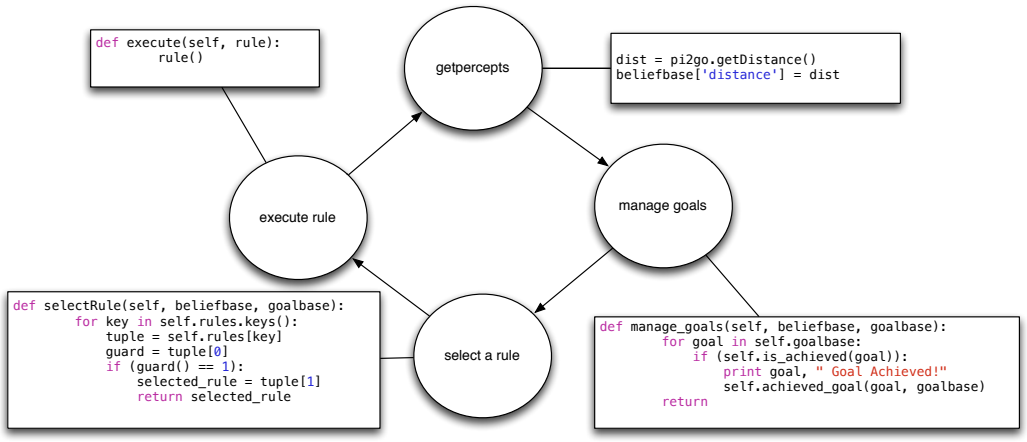


Figure 3: The Pi2Go Agent Reasoning Cycle. This starts by 1. checking all the sensors and updating the *belief base* with values from the sensor. 2. Checking all *goals* and removing any that have been achieved (described in section 6). 3. Checking all rules and picking the first that can be executed. 4. Executes that rule or does nothing if no rules can be executed.

reasoning cycle that rule is executed and it prints out the current value of the switch sensor.

There are several other sensors. Figure 4 shows the ones on the front of the robot. These sensors all operate in a similar way. They emit something, either a pulse of infra red light (the three small sensors) or a pulse of ultrasonic noise (the large sensor), and then measure the value of the reflection or echo.

You can access the values returned by these sensors with the strings ' obstacle_centre ', ' obstacle_left ', ' obstacle_right ' (the infra red sensors) and ' distance ' (the distance sensor).

Task Create a new python program by copying your first program to a new file and then modify this program so that the rule prints the value of a different sensor. Check both ' obstacle_centre ' and ' distance '.

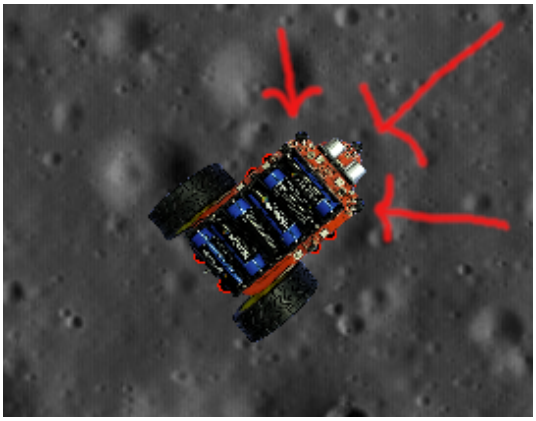


Figure 4: Sensors on the front of the robot. There are three small *infra red* sensors and a large *ultrasonic* sensor at the front

2.3 Stopping the Program

It is inelegant to use Ctrl-C to stop a program so instead we will use a sensor and a *conditional rule*.

Task Create a Python function called `stop_rule`, like `print_switch_rule` in section 2.2. Instead of calling `agent.sensor_value(sensor_key)` this program should simply print the message `"Stopping Agent"` and then (on the next line) call `agent.done()` (this will stop the agent's reasoning cycle and so finish execution of your whole program).

You want this rule to be executed *only if* the switch is pressed. This means we need to add it as a *conditional rule* to our agent. The function `agent.add_condition_rule(guard, rule_name)` adds a conditional rule to an agent. This takes two arguments. The first is a *guard* which is a function that returns 1 (for true) and 0 (for false) when it is called. The second argument is the name of a rule.

All the sensor values we looked at in section 2.2 are stored in the agent's *belief base*. If the sensor returns either true or false (like `'switch_pressed'` and `'obstacle_centre'`) then they be accessed by calling the function `agent.believe(sensor_key)` with the sensor's key string as an argument. `agent.believe(sensor_key)` returns the value 1 if the sensor is returning true and 0 if the sensor is returning false.

So, assuming your stop rule is called `stop_rule`, you add a conditional rule that the agent should stop if the switch is pressed by writing:

```
agent.add_condition_rule(agent.believe('switch_pressed'), stop_rule)
```

Important: The conditional rule needs to be added *before* the print sensor value rule. This is because the agent always executes picks the first rule that applies in the order that they were added.

Side Note for the Curious We can't use agent. sensor_value (sensor_key) as a guard because it will store the value of the function *at the time the rule is added to the robot* while agent. believe (sensor_key) waits until the rule is executed before checking the value. Delaying the execution of a python function is quite advanced programming and you do not need to understand how this works here. Normally it can be achieved just by passing the name of one function as an argument to another (this is why we use rule names as arguments in agent. add_rule (rule_name) and agent. add_condition_rule (guard, rule_name)) but it is complicated in agent. believe (sensor_key) by the fact the function takes the sensor key as an argument.

Task Create a program which prints out the readings from one sensor until the switch is pressed.

2.4 Moving the Robot

We will now get the robot to drive forwards until the switch is pressed. For this you will need to replace your print sensor rule with a go forward rule.

To make the robot go forwards you want to use a function contained in the robot's library rather than in the agent object. In order to to this you need to import simclient .simrobot as pi2go as well as pi2goagent at the start of your program. This is a normal library so you do not need to create any objects.

Task Use pi2go. forward(20) to create a program where the robot moves forwards at speed 20 until the switch is pressed. Use a rule that first prints the message Going Forward and then moves the robot. You may want to include the command pi2go. stop() in your stop rule. This stops the Pi2Go's motors.

2.5 Using Internal Beliefs to Control Rules

You will have noticed in section 2.4 that the robot keeps printing Going Forward. This is because the rule is selected and executed on each turn of the reasoning cycle. It would be better if the rule is executed just once, to start the robot moving and isn't executed again. Ideally we would like to use a conditional rule with the guard that the robot believes it isn't moving.

To do this we will need to add an *internal belief* to the belief base. The agent function for adding internal beliefs is shown in line 2 of figure 5.

We only want the agent to move if we *don't believe* we are moving. We can create our guard using the function agent. NOT(belief) – i.e.,

- agent. NOT(agent.believe('moving')) or

```
def forward_rule():                                    1
        agent.add_belief('moving')                     2
        pi2go.forward(20)                              3
        return                                         4
```

Figure 5: A Rule that adds an Internal Belief

- agent.NOT(agent.B('moving'))

NB: agent.B is a shortened form of agent.believe.

Task Write a program to make your start moving forward until the switch is pressed. This program should only start the robot moving once.

2.6 Using the Switch to Start and Stop Operation

You are probably becoming frustrated by the way the keyboard and monitor make it difficult to move the robot properly.

Instead of just using the switch to stop the robot program we could also use it to start the program by getting it to add an internal belief started and a rule for starting. This will let you delay the start of robot motion in order to give you time to unplug the keyboard, mouse and monitor and let the robot move about.

However, if you have unplugged the monitor you will no longer be able to use a print statement to tell if the agent has noticed your switch press (sometimes if you press and release too quickly it can miss the switch press). Fortunately the Pi2Go has a set of LEDs around it which can light up in various different colours.

```
def start_rule():                                      1
        print "Starting Agent"                         2
        agent.add_belief('started')                    3
        pi2go.setAllLEDs(4095, 4095, 4095)             4
        time.sleep(1)                                  5
        pi2go.setAllLEDs(0, 0, 0)                      6
        return                                         7
```

Figure 6: A Start Rule that flashes the LEDs white when a switch press is detected.

Figure 6 shows a start rule that adds the belief 'started' in line 3. In line 4 the rule calls the function setAllLEDs. This takes three arguments for the value of the red, green and blue light to be displayed. We are passing in the maximum for all three values which gives us a bright white light, but many other colours can be displayed by changing the numbers. We then get the rule to sleep for a millisecond using time.sleep(1). **Important:** for this you will need to import the time library. Then it sets the value of all LEDs to 0 (i.e., switches all the lights off). Figure 7 lists the functions for controlling the Pi2Go LEDs and these are also listed at the end of this booklet.

setLED(LED, Red, Green, Blue) Sets the LED specified to required RGB value. $1 \leq LED \leq 4$, $0 \leq Red, Green, Blue \leq 4095$

setAllLEDs(Red, Green, Blue) Sets all LEDs to required RGB. $0 \leq Red, Green, Blue \leq 4095$

Figure 7: Pi2Go LED Functions

The sleep function isn't included just to allow the flash of light to be seen. You also want time to release the switch before the agent checks again otherwise it will instantly execute the stop rule.

We want

- The agent to start if it believes the switch is pressed and doesn't believe it has started.
- We want the agent to move it if believes it has started and doesn't believe it is already moving.
- We want the agent to stop if it believes it has started and believes the switch is pressed.

We can use agent.AND(belief1, belief2) along with agent.NOT(belief) in order to create the guards we need.

If you feel your calls to agent. add_condition_rule (guard, rule_name) are getting very long (this is particularly irritating with a small monitor) then you can use python variables just as in a normal python program. You can see an example of this in figure 8

```
b_switch = agent.B('switch_pressed')
b_started = agent.B('started')
b_moving = agent.B('moving')

b_ready_to_start = agent.AND(b_switch, agent.NOT(b_started))
b_ready_to_move = agent.AND(b_started, agent.NOT(b_moving))
b_ready_to_stop = agent.AND(b_switch, b_started)

agent.add_condition_rule(b_ready_to_start, start_rule)
agent.add_condition_rule(b_ready_to_move, forward_rule)
agent.add_condition_rule(b_ready_to_stop, stop_rule)
agent.run_agent()
```

Figure 8: Using Python variables for complex conditions

Task Write a program for your Pi2Go so that when the switch is pressed it starts moving forwards and when the switch is pressed again it stops executing.

2.7 Summary

This section has used several very simple programs to illustrate the basics of rational agent programming using the Pi2Go. It has introduced the agent object and illustrated several useful functions. There are two further functions that you will need in the next sections.

agent.drop_belief(belief) This function allows you to drop an internal belief. This can be important if you have, for instance, several different types of moving behaviour you want to switch between and the agent needs to know which one it is using at the moment.

agent.OR(belief1, belief2) is used in guards. This returns true if the agent believes *either* belief1 or belief2 .

Figure 9 summarises all the agent functions. Figure 10 summarises the sensor keys. Note that some of these functions and sensor keys have not been introduced yet. They will be described when you need them.

3 Obstacle Avoidance Behaviour

You are now ready to start programming some more complex behaviours for the robot.

As well as forward the pi2go library has a number of other functions for moving the robot. These are summarised in figure 11.

Task Program the robot so that if it believes there is an obstacle in front of any of its infra-red sensors it turns until there is no longer an obstacle, at which point it carries on moving forwards. You may want to use agent.OR(belief1, belief2) to construct the guard for your rules and agent. drop_belief (belief) to help the robot know whether it is moving forwards or turning.

Option Customise this program so that the direction in which the robot turns depends upon which sensors are detecting the obstacle so that it turns in the most sensible direction to clear the obstacle depending upon whether it is on the right or on the left.

4 Abstractions: Light Following Behaviour

So far we have been able to directly use the beliefs that come in from sensors, but now we are going to look at the light sensors which do not return True or False but a number between 0 and 1023 depending upon how strong a light they are detecting.

The light sensors are contained in the two LEDs at the front and at the back of the robot. They can be accessed with the sensor keys 'lightFL' (front left), 'lightFR' (front right), 'lightBL' (back left), 'lightBR' (back right).

agent.sensor_value(sensor_key) Returns the value of the sensor associated with sensor_key.

agent.add_belief(belief) Adds an internal belief.

agent.drop_belief(belief) Removes an internal belief.

agent.believe(belief_key) Returns true if the agent believes belief.

agent.B(belief_key) Returns true if the agent believes belief.

agent.AND(belief1, belief2) This returns true if the agent believes *both* belief1 and belief2.

agent.OR(belief1, belief2) This returns true if the agent believes *either* belief1 or belief2.

agent.NOT(belief) This returns true if the agent doesn't belief belief

agent.add_goal(goal) Adds a goal to the agent's goal base.

agent.drop_goal(goal) Removes a goal from the agent's goal base.

agent.has_goal(goal) Returns true if the agent has goal goal.

agent.G(goal) Returns true if the agent has goal goal.

agent.goal_is_achieved_when(goal, belief) Tells the agent what must be believed for goal to be achieved.

agent.add_rule(rule_name) Adds an unconditional rule to the agent.

agent.add_condition_rule(guard, rule_name) Adds a conditional rule to the agent with guard, guard.

agent.run_agent() Starts the agent's reasoning cycle.

agent.done() Stops the agent's reasoning cycle.

Figure 9: Summary of all the functions of the agent object

switch_pressed True if the switch is pressed, false otherwise.

obstacle_centre True if something is close to the front infra red sensor.

obstacle_right True if something is close to the front right infra red sensor.

obstacle_left True if something is close to the front left infra red sensor.

distance Returns the distance something is in front of the ultrasonic sensor.

no_line_left True if there is a bright colour beneath the left downward pointing infra red sensor.

no_line_right True if there is a bright colour beneath the right downward pointing infra red sensor.

lightFL Returns the light intensity recorded by the front left light sensor.

lightFR Returns the light intensity recorded by the front right light sensor.

lightBL Returns the light intensity recorded by the back left light sensor.

lightBR Returns the light intensity recorded by the back right light sensor.

Figure 10: Summary of all the Sensor Keys

stop() Stops both motors

forward(speed) Sets both motors to move forward at speed. $0 \leq speed \leq 100$

reverse(speed) Sets both motors to reverse at speed. $0 \leq speed \leq 100$

spinLeft(speed) Sets motors to turn opposite directions at speed. $0 \leq speed \leq 100$

spinRight(speed) Sets motors to turn opposite directions at speed. $0 \leq speed \leq 100$

turnForward(leftSpeed, rightSpeed) Moves forwards in an arc by setting different speeds. $0 \leq leftSpeed, rightSpeed \leq 100$

turnreverse(leftSpeed, rightSpeed) Moves backwards in an arc by setting different speeds. $0 \leq leftSpeed, rightSpeed \leq 100$

go(leftSpeed, rightSpeed) controls motors in both directions independently using different positive/negative speeds. $-100 \leq leftSpeed, rightSpeed \leq 100$

go(speed) controls motors in both directions together with positive/negative speed parameter. $-100 \leq speed \leq 100$

Figure 11: Summary of the Pi2Go Movement Functions

To use these as beliefs we are going to have to define *abstractions* from these values to the "true" and "false" used by beliefs. To do this we will define them as Python functions which return either 1 (for true) or 0 (for false).

Figure 12 defines a function b_light_on_front_right which represents that the belief that there is a brighter light on the right at the front of the robot than on the left. It does this getting the sensor values from the front two light sensors and comparing them. If lightFR is greater than lightFL by 10 then it believes the light is brighter on the right.

```
def b_light_on_front_right():
        lightFL = agent.sensor_value('lightFL')
        lightFR = agent.sensor_value('lightFR')
        print "Left Light: ", lightFL, "Right Light: ", lightFR
        if (lightFL < lightFR and lightFR - lightFL > 10):
                return 1
```

Figure 12: A Python Function that abstracts values from the Light Sensors

To add a conditional rule to a program that will make the robot turn towards the right if the light is brighter on that side then you add the *name of* the function as the guard. For instance:

```
agent.add_condition_rule ( b_light_on_front_right , turn_right_rule )
```

The name of the function can also be used with the functions agent.AND(belief1, belief2), agent.OR(belief1, belief2) and agent.NOT(belief). For instance:

```
agent.AND( b_light_on_front_right , agent.B('started '))
```

represents the condition that the light is brighter at the front *and* the robot has started.

Task Program the robot to turn on the spot so that it faces the brightest light in the room. Create abstractions using all four light sensors to achieve this.

Option Modify your program so that the robot drives towards the brightest light in the room. If the light is moving it should follow the light. You may want to use turnForward instead of spinLeft and spinRight.

Option Go back to your obstacle avoidance program and adapt it to use the ultrasonic distance sensor, so that it detects obstacles sooner.

5 Line Following Behaviour

The Pi2Go has two additional infrared sensors underneath the robot. Since they are pointing downwards they measure how much light is reflected from the floor and this depends upon what colour the floor is. The sensors return True if the floor is a light colour (such as white) and False if it is a dark colour (such as black). These sensor values are returned with the keys ' no_line_left ' (to indicate a light colour on the left) and ' no_line_right ' (to indicate a light colour on the right).

This means if you place the robot so the two sensors are either side of a thick black line it should be able to travel along the line by moving forwards (when neither sensor detects the line) and turning left or right (as appropriate) when one of the sensors detects the line.

Task Program your robot to follow a line.

6 Goal Directed Behaviour

So far the agents have been working only with beliefs. To get more complex behaviour it is necessary to consider what the robot is trying to do. Rational agent programming uses the idea of a *goal* that the agent is trying to achieve to represent this.

The Pi2Go Agent has several functions that support programming with goals:

agent.add_goal(goal_key) This adds a goal to the agent's *goal base*

agent.has_goal(goal_key) Is used in rule guards to detect whether the agent has a goal.

agent.G(goal_key) is identical to agent.has_goal(goal_key)

A goal remains in the agent's goal base until it is believed to be true. So, in a simple case, if our goal key is the same as one of the sensor keys the then agent will continue to have that goal until that sensor returns True.

Task Write a program which adds the goal 'obstacle_centre' during the start rule (this is the *initial goal*). The program should have a rule that makes the robot drive forwards if it has started and has the goal 'obstacle_centre'.

6.1 Complex Goals

The function agent.goal_is_achieved_when(goal_key, belief) can link a goal key with a complex belief allowing you to create more complex goals.

Consider the python code

```
b_obstacle = agent.OR(b_obstacle_centre, agent.OR(b_obstacle_left, b_obstacle_right))

agent.goal_is_achieved_when('obstacle', b_obstacle)
```

What do you think this code does? _____

Task Adapt your previous program to use 'obstacle' as the goal rather than 'obstacle_centre'.

6.2 Abstract Goals

agent.goal_is_achieved_when(goal_key, belief) can also be used to define abstract goals using functions. Consider

```
def b_in_the_light():
        lightFL = agent.sensor_value('lightFL')
        lightFR = agent.sensor_value('lightFR')
        if (lightFL > 250 and lightFR > 250):
                return 1
        return 0

agent.goal_is_achieved_when('in_the_light', b_in_the_light)
```

What do you think this code does? _____

Task Write a program to make a robot drive to a very bright part of the room and then stop.

6.3 Subgoals

A robot can have multiple goals at once. This allows you to use some goals as *subgoals*. For instance a robot with a goal to find an obstacle might need to find a line first and then follow that line to the obstacle. So the first thing it does, if it has a goal to find an obstacle is set up a subgoal to fine a line (e.g., by driving forwards). Once that subgoal is achieved then it can start line following behaviour to find the obstacle.

Task Write a program to find a line and then follow it until it meets an obstacle.

7 More Complex Programs for you to try

Task Write a program to drive to a very bright part of the room, then follow a line until it meets an obstacle.

Task Write a program where an agent in a very bright part of the room gets a goal to find an obstacle, and an agent that has found an obstacle gets a goal to move to a very bright part of the room.

8 Command Summaries

8.1 LED Functions

setLED(LED, Red, Green, Blue) Sets the LED specified to required RGB value. $1 \leq LED \leq 4$, $0 \leq Red, Green, Blue \leq 4095$

setAllLEDs(Red, Green, Blue) Sets all LEDs to required RGB. $0 \leq Red, Green, Blue \leq 4095$

8.2 Agent Functions

agent.sensor_value(sensor_key) Returns the value of the sensor associated with sensor_key.

agent.add_belief(belief) Adds an internal belief.

agent.drop_belief(belief) Removes an internal belief.

agent.believe(belief_key) Returns true if the agent believes belief.

agent.B(belief_key) Returns true if the agent believes belief.

agent.AND(belief1, belief2) This returns true if the agent believes *both* belief1 and belief2 .

agent.OR(belief1, belief2) This returns true if the agent believes *either* belief1 or belief2 .

agent.NOT(belief) This returns true if the agent doesn't belief belief

agent.add_goal(goal) Adds a goal to the agent's goal base.

agent.drop_goal(goal) Removes a goal from the agent's goal base.

agent.has_goal(goal) Returns true if the agent has goal goal.

agent.G(goal) Returns true if the agent has goal goal.

agent.goal_is_achieved_when(goal, belief) Tells the agent what must be believed for goal to be achieved.

agent.add_rule(rule_name) Adds an unconditional rule to the agent.

agent.add_condition_rule(guard, rule_name) Adds a conditional rule to the agent with guard, guard.

agent.run_agent() Starts the agent's reasoning cycle.

agent.done() Stops the agent's reasoning cycle.

8.3 Sensor Keys

switch_pressed True if the switch is pressed, false otherwise.

obstacle_centre True if something is close to the front infra red sensor.

obstacle_right True if something is close to the front right infra red sensor.

obstacle_left True if something is close to the front left infra red sensor.

distance Returns the distance something is in front of the ultrasonic sensor.

no_line_left True if there is a bright colour beneath the left downward pointing infra red sensor.

no_line_right True if there is a bright colour beneath the right downward pointing infra red sensor.

lightFL Returns the light intensity recorded by the front left light sensor.

lightFR Returns the light intensity recorded by the front right light sensor.

lightBL Returns the light intensity recorded by the back left light sensor.

lightBR Returns the light intensity recorded by the back right light sensor.

8.4 Motor Functions

stop() Stops both motors

forward(speed) Sets both motors to move forward at speed. $0 \leq speed \leq 100$

reverse(speed) Sets both motors to reverse at speed. $0 \leq speed \leq 100$

spinLeft(speed) Sets motors to turn opposite directions at speed. $0 \leq speed \leq 100$

spinRight(speed) Sets motors to turn opposite directions at speed. $0 \leq speed \leq 100$

turnForward(leftSpeed, rightSpeed) Moves forwards in an arc by setting different speeds. $0 \leq leftSpeed, rightSpeed \leq 100$

turnReverse(leftSpeed, rightSpeed) Moves backwards in an arc by setting different speeds. $0 \leq leftSpeed, rightSpeed \leq 100$

go(leftSpeed, rightSpeed) controls motors in both directions independently using different positive/negative speeds. $-100 \leq leftSpeed, rightSpeed \leq 100$

go(speed) controls motors in both directions together with positive/negative speed parameter. $-100 \leq speed \leq 100$