**Programming in Python with the Initio Simulator: Part 4 Answer Booklet**

**Chapter 1**

**Question 1:** It prints an empty list.

**Question 2:** It created an empy list/array called readings.

**Question 3:** It prints an empty list (same as when data1.print\_readings() was called).

**Question 4:** take\_readings()

**Question 5:** print\_readings()

**Question 6:** The answer will depend upon where boxes have been placed and how many readings have been taken, but it will be a list of floats.

**Question 7:** data2 = DataLogger()

**Question 8:** Each data logger only stores a reading when its own take\_readings() method is called. Therefore, the two data loggers have different lists of readings because the objects were moved between taking readings with different loggers.

**Exercise 1:**

import simclient.simrobot as initio

class DataLogger:

def \_\_init\_\_(self):

self.right\_readings = []

self.left\_readings = []

def take\_irRight\_reading(self):

self.right\_readings.append(initio.irRight())

def take\_irLeft\_reading(self):

self.left\_readings.append(initio.irLeft())

def take\_reading(self):

self.right\_readings.append(initio.irRight())

self.left\_readings.append(initio.irLeft())

def print\_irRight\_log(self):

print(self.right\_readings)

def print\_irLeft\_log(self):

print(self.left\_readings)

def print\_readings(self):

print("Right Sensor:" + str(self.right\_readings))

print("Left Sensor:" + str(self.left\_readings))

**Exercise 2:**

import simclient.simrobot as initio

class DataLogger:

def \_\_init\_\_(self, name):

self.readings = []

self.logger\_name = name

def take\_reading(self):

self.readings.append(initio.getDistance())

def print\_readings(self):

print(self.readings)

def get\_name(self):

return self.logger\_name

def set\_name(self, name):

self.logger\_name = name

**Question 9:** The first logger prints its name as logger1 and the second as logger2. This is because they were given different names when initialised.

**Chapter 2**

**NOTE:** Exercises are reaching sufficient complexity that it is unreasonable to expect answers to be largely similar – particularly in the way functions, methods, fields and classes may be constructed. Sample answers have to be regarded as examples only, not definitive answers.

**Exercise 1:**

import simclient.simrobot as initio

import time

class WallFollower:

def \_\_init\_\_(self):

print("Initialising")

def drive\_to\_wall(self):

while (not initio.getDistance() < 20 and not initio.irLeft() and not initio.irRight()):

initio.forward(10)

initio.stop()

def spin(self, direction):

if (direction == 'right'):

initio.spinRight(10)

else:

initio.spinLeft(10)

def obstacle\_to(self, direction):

if (direction == 'right'):

return initio.irRight()

else:

return initio.irLeft()

def opposite\_direction(self, direction):

if (direction == 'right'):

return 'left'

else:

return 'right'

def follow\_wall(self, direction):

while (not initio.getSwitch() == True):

if (initio.getDistance() < 20):

while(initio.getDistance() < 20 or self.obstacle\_to(direction)):

self.spin(self.opposite\_direction(direction))

elif (self.obstacle\_to(direction)):

initio.forward(10)

time.sleep(2)

elif (not self.obstacle\_to(direction)):

self.spin(direction)

time.sleep(1)

initio.stop()

**Exercise 2:**

import simclient.simrobot as initio

import time

class WallFollower:

def \_\_init\_\_(self, side):

self.direction = side

def drive\_to\_wall(self):

while (not initio.getDistance() < 20 and not initio.irLeft() and not initio.irRight()):

initio.forward(10)

initio.stop()

def spin(self, direction):

if (direction == 'right'):

initio.spinRight(10)

else:

initio.spinLeft(10)

def obstacle\_to(self):

if (self.direction == 'right'):

return initio.irRight()

else:

return initio.irLeft()

def opposite\_direction(self):

if (self.direction == 'right'):

return 'left'

else:

return 'right'

def follow\_wall(self):

while (not initio.irLeftLine()):

if (initio.getDistance() < 20)

while(initio.getDistance() < 20 or self.obstacle\_to()):

self.spin(self.opposite\_direction())

elif (self.obstacle\_to()):

initio.forward(10)

time.sleep(2)

elif (not self.obstacle\_to()):

self.spin(self.direction)

time.sleep(1)

initio.stop()

initio.init()

right\_wall = WallFollower('right')

left\_wall = WallFollower('left')

right\_wall.follow\_wall()

initio.reverse(10)

time.sleep(5)

initio.spinRight(10)

time.sleep(5)

left\_wall.follow\_wall()

**Exercise 3:** Note we have a call to the ultrasonic sensor in follow\_policy to allow a user to stop the robot by placing something in front of it.

import simclient.simrobot as initio

import random, time

class MachineLearner():

def \_\_init\_\_(self, learning\_rate):

self.learning\_rate = learning\_rate

self.epsilon = 1

self.actions = ['forward', 'backward', 'left', 'right']

self.dictionary = self.action\_reward(self.actions, 1)

self.rewards = {(1, 1):1, (1, 0):2, (0, 1):1, (0, 0):0}

def action\_reward(self, action\_list, default):

action\_rewards = {}

for i in range(0, 2):

for j in range(0, 2):

for k in (action\_list):

action\_rewards[((i, j), k)] = default

return action\_rewards

def get\_max(self, state):

max\_reward = 0

action = 'forward'

for act in self.actions:

if (self.dictionary[(state, act)] > max\_reward):

action = act

max\_reward = self.dictionary[(state, act)]

return action

def action\_execute(self, action):

if (action == "forward"):

initio.forward(10)

elif (action == "backward"):

initio.reverse(10)

elif (action == "left"):

initio.spinLeft(10)

elif (action == "right"):

initio.spinRight(10)

else:

initio.stop()

def learn(self):

while (self.epsilon > 0):

explore = random.random();

state = (initio.irLeftLine(), initio.irRightLine())

if (explore < self.epsilon):

print("Exploring")

action = random.choice(self.actions)

else:

action = self.get\_max(state)

self.action\_execute(action)

time.sleep(3)

reward = self.rewards[(initio.irLeftLine(), initio.irRightLine())]

if (reward == 2):

self.epsilon = self.epsilon - 0.05

print("updating epsilon: " + str(self.epsilon))

self.dictionary[(state, action)] = self.dictionary[(state, action)] + ( reward - self.dictionary[(state, action)]) \* self.learning\_rate

def follow\_policy(self):

while (not initio.getDistance() < 20):

state = (initio.irLeftLine(), initio.irRightLine())

action = self.get\_max(state)

self.action\_execute(action)

time.sleep(3)

initio.stop();

initio.init()

learner = MachineLearner(0.5)

learner.learn()

learner.follow\_policy()

**Chapter 4**

**General Troubleshooting:** Watch out for complaints about apostrophes if students are cutting and pasting between the worksheet and IDLE.

**Question 1:** A dictionary is printed which will look something like:

{'distance': 630.0, 'obstacle\_right': 0, 'obstacle\_left': 0, 'line\_left': 0, 'line\_right': 0}

Though values may vary.

**Question 2:** This will depend upon the set up. In the beliefbase above the value is 630.

**Question 3:** The dictionary now contains 'test': 1

**Question 4:** 'name': 'initio'has been added.

**Question 5:** There is no ‘name’ key in the dictionary.

**Question 6:** It returns True because distance is a key in the dictionary.

**Question 7:** It prints out the value of the distance sensor stored in the belief base.

**Question 8:** It prints a list containing the string ‘a\_goal’

**Question 9:** It prints an empty list. check\_goals removed ‘a\_goal’ from the goalbase because ‘a\_goal’ was now an agent belief.

**Question 10:** The simulated Initio moves forward and then stops – just like typing initio.forward(10) etc. in earlier exercises. This is because the robot is a field in the agent object.

**Chapter 4**

**Exercise 1:**

import bdi.initioagent as cognitive

agent = cognitive.InitioAgent()

agent.init()

agent.getPercepts()

while not (agent.beliefbase['line\_left']):

if (agent.beliefbase['obstacle\_left']):

agent.robot.spinLeft(10)

elif (agent.beliefbase['obstacle\_right']):

agent.robot.spinLeft(10)

else:

agent.robot.forward(10)

agent.getPercepts()

agent.robot.stop()

**Exercise 2:**

import bdi.initioagent as cognitive

agent = cognitive.InitioAgent()

agent.init()

while True:

agent.getPercepts()

distance = agent.beliefbase['distance'];

if (distance < 50):

agent.add\_belief('obstacle')

elif ('obstacle' in agent.beliefbase):

agent.remove\_belief('obstacle')

if ('obstacle' in agent.beliefbase):

agent.robot.spinLeft(10)

else:

agent.robot.forward(10)

**Exercise 3:**

import bdi.initioagent as cognitive

agent = cognitive.InitioAgent()

agent.init()

agent.getPercepts()

agent.add\_goal('obstacle\_left')

while ('obstacle\_left' in agent.goalbase):

agent.robot.forward(10)

agent.getPercepts()

agent.check\_goals()

agent.robot.stop()

**Exercise 4:**

import bdi.initioagent as cognitive

agent = cognitive.InitioAgent()

agent.init()

agent.getPercepts()

while not (agent.beliefbase['obstacle\_left']):

if (agent.beliefbase['line\_left'] and not (agent.beliefbase['line\_right'])):

agent.robot.forward(10)

elif (agent.beliefbase['line\_right']):

agent.robot.spinRight(10)

else:

agent.robot.spinLeft(10)

agent.getPercepts()

agent.robot.stop()

**Exercise 5:**

import bdi.initioagent as cognitive

agent = cognitive.InitioAgent()

agent.init()

agent.getPercepts()

def edge\_following():

while ('edge\_square' in agent.goalbase):

if (agent.beliefbase['line\_left'] and not (agent.beliefbase['line\_right'])):

agent.robot.forward(10)

elif (agent.beliefbase['line\_right']):

agent.robot.spinRight(10)

else:

agent.robot.spinLeft(10)

agent.getPercepts()

if (agent.beliefbase['obstacle\_left']):

agent.drop\_goal('edge\_square')

def find\_edge():

agent.add\_goal('line\_left')

while ('line\_left' in agent.goalbase):

agent.robot.forward(10)

agent.getPercepts()

agent.check\_goals()

agent.add\_goal('edge\_square')

find\_edge()

edge\_following()

agent.robot.stop()

**Chapter 5**

**Question 1:** The agent continuously prints out its beliefs. These are the current beliefs so if, for instance, a block is moved in front of the agent then the distance belief will change.

**Exercise 1:**

import bdi.initioagent as cognitive

agent = cognitive.InitioAgent()

def print\_beliefs():

print(agent.beliefbase['distance'])

return

agent.add\_rule(print\_beliefs)

agent.run\_agent()

**Question 2:** The agent mostly apparently does nothing, but when something is closer than 50 to the distance sensor, then it prints out the belief base.

**Exercise 2:**

import bdi.initioagent as cognitive

agent = cognitive.InitioAgent()

def reverse\_rule():

agent.robot.reverse(10)

return

agent.add\_condition\_rule(agent.B('obstacle\_left'), reverse\_rule)

agent.run\_agent()

**Exercise 3:**

import bdi.initioagent as cognitive

import time

agent = cognitive.InitioAgent()

def reverse\_rule():

agent.robot.reverse(10)

time.sleep(5)

agent.stop()

agent.done()

return

agent.add\_condition\_rule(agent.B('obstacle\_left'), reverse\_rule)

agent.run\_agent()

**Chapter 6**

**Question 1:** When there is an obstacle on the left and on the right.

**Question 2:** When the there is an obstacle on the left and the agent does not believe ‘started’ because this has not been added to the belief base.

**Question 3:** When the switch is pressed and when the agent believes ‘started’ because this has been add to the belief base.

**Question 4:** When ‘started’ has been added to the belief base.

**Question 5:** When there is a line on the left ‘started’ is added to the belief base (and the agent sleeps for 5 seconds – to allow the line to “vanish”). Once it believes it has started the robot starts to move forward. When a line appears on the left again again (or if it is not removed within 5 seconds) then the started belief is dropped and a stopping belief is added. Once it believes it is stopped the agent stops the robot, the reasoning cycle and drops the stopping belief.

**Exercise 1:**

import bdi.initioagent as cognitive

import time

agent = cognitive.InitioAgent()

def start\_agent():

agent.add\_belief('started')

time.sleep(5)

return

def stop\_agent():

agent.drop\_belief('started')

agent.add\_belief('stopping')

time.sleep(5)

return

def forward():

agent.robot.forward(10)

return

def turn():

agent.robot.spinLeft(10)

return

def stop\_rule():

agent.robot.stop()

agent.done()

agent.drop\_belief('stopping')

return

start = agent.AND(agent.B('line\_left'), agent.NOT(agent.B('started')))

stop = agent.AND(agent.B('line\_left'), agent.B('started'))

no\_obstacle = agent.AND(agent.B('started'), agent.AND(agent.NOT(agent.B('obstacle\_right')), agent.NOT(agent.B('obstalce\_left'))))

obstacle = agent.AND(agent.B('started'), agent.OR(agent.B('obstacle\_left'), agent.B('obstacle\_right')))

agent.add\_condition\_rule(start, start\_agent)

agent.add\_condition\_rule(stop, stop\_agent)

agent.add\_condition\_rule(no\_obstacle, forward)

agent.add\_condition\_rule(obstacle, turn)

agent.add\_condition\_rule(agent.B('stopping'), stop\_rule)

agent.run\_agent()

**Chapter 7**

**Exercise 1:** Note that because this is a line following agent, I’m using obstacle\_left to start and stop it.

import bdi.initioagent as cognitive

import time

agent = cognitive.InitioAgent()

def start\_agent():

agent.add\_belief('started')

time.sleep(5)

return

def stop\_agent():

agent.drop\_belief('started')

agent.add\_belief('stopping')

time.sleep(5)

return

def stop\_rule():

agent.robot.stop()

agent.done()

agent.drop\_belief('stopping')

return

def forward():

agent.robot.forward(10)

return

def left():

agent.robot.spinLeft(10)

return

def right():

agent.robot.spinRight(10)

start = agent.AND(agent.B('obstacle\_left'), agent.NOT(agent.B('started')))

stop = agent.AND(agent.B('obstacle\_left'), agent.B('started'))

on\_line = agent.AND(agent.B('started'), agent.NOT(agent.OR(agent.B('line\_left'), agent.B('line\_right'))))

line\_on\_left = agent.AND(agent.B('started'), agent.B('line\_left'))

line\_on\_right = agent.AND(agent.B('started'), agent.B('line\_right'))

agent.add\_condition\_rule(start, start\_agent)

agent.add\_condition\_rule(stop, stop\_agent)

agent.add\_condition\_rule(on\_line, forward)

agent.add\_condition\_rule(line\_on\_left, left)

agent.add\_condition\_rule(line\_on\_right, right)

agent.add\_condition\_rule(agent.B('stopping'), stop\_rule)

agent.run\_agent()

**Exercise 2:**

import bdi.initioagent as cognitive

import time

agent = cognitive.InitioAgent()

def start\_agent():

agent.add\_belief('started')

time.sleep(5)

return

def stop\_agent():

agent.drop\_belief('started')

agent.add\_belief('stopping')

time.sleep(5)

return

def stop\_rule():

agent.robot.stop()

agent.done()

agent.drop\_belief('stopping')

return

def forward():

agent.robot.forward(10)

return

def left():

agent.robot.spinLeft(10)

return

def right():

agent.robot.forward(10)

time.sleep(1)

agent.robot.spinRight(10)

time.sleep(2)

return

def b\_obstacle\_centre():

if (agent.beliefbase['distance'] < 30):

return True

return False

start = agent.AND(agent.B('obstacle\_left'), agent.NOT(agent.B('started')))

stop = agent.AND(agent.B('obstacle\_left'), agent.B('started'))

wall\_on\_right = agent.AND(agent.B('started'), agent.AND(agent.B('obstacle\_right'), agent.NOT(b\_obstacle\_centre)))

wall\_in\_front = agent.AND(agent.B('started'), b\_obstacle\_centre)

lost\_wall = agent.AND(agent.B('started'), agent.NOT(agent.OR(agent.B('obstacle\_right'), agent.B('obstacle\_right'))))

floor = agent.AND(agent.B('started'), agent.B('line\_left'))

agent.add\_condition\_rule(start, start\_agent)

agent.add\_condition\_rule(stop, stop\_agent)

agent.add\_condition\_rule(floor, stop\_rule)

agent.add\_condition\_rule(wall\_on\_right, forward)

agent.add\_condition\_rule(wall\_in\_front, left)

agent.add\_condition\_rule(lost\_wall, right)

agent.add\_condition\_rule(agent.B('stopping'), stop\_rule)

agent.run\_agent()

**Chapter 8**

**Question 1:** `alice` is printed out.

**Question 2:** When the object is created the name field is set to “alice” and this is returned when getName() is called.

**Question 3:** `bob` is printed out.

**Question 4:** changeName(‘bob’) has changed the value of the name field to “bob”. This is returned when getName() is called.

**Question 5:** The virtual Initio moves forward and then stops.

**Question 6:** Because NameAgent sub-classes InitioAgent it can still use the methods and fields in InitioAgent.

**Question 7:** It creates a cognitive agent that reverses when an object is placed to the left of it, provided the reasoning cycle is running (i.e., the run\_agent() method has been called).

**Question 8:**

>>> bob = ReverseAgent()

>>> bob.run\_agent()

**Exercise 1:**

import bdi.initioagent as cognitive

import time

class ReverseAgent(cognitive.InitioAgent):

def \_\_init\_\_(self):

cognitive.InitioAgent.\_\_init\_\_()

self.add\_condition\_rule(self.B('obstacle\_left'), self.reverse\_rule)

self.add\_condition\_rule(self.B('obstacle\_righ'), self.done)

def reverse\_rule(self):

self.robot.reverse(10)

time.sleep(5)

self.robot.stop()

**Exercise 2:**

import bdi.initioagent as cognitive

import time

class ForwardAgent(cognitive.InitioAgent):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.add\_condition\_rule(self.AND(self.NOT(self.B('obstacle\_left')), self.NOT(self.B(‘obstacle\_right’))), self.forward\_rule)

self.add\_condition\_rule(self.B(‘line\_left’), self.done)

def forward\_rule(self):

self.robot.forward(10)

time.sleep(5)

self.robot.stop()

class ShortForwardAgent(ForwardAgent):

def forward\_rule(self):

self.robot.forward(10)

time.sleep(1)

self.robot.stop()

**Question 9:** The done() method and \_\_init()\_\_ method from the InitioAgent class.

**Question 10:** It creates a cognitive agent that, when running, will reverse if it encounters an obstacle on the left. If it detects an obstacle on the right it will stop running and print out “Exited the Reasoning Cycle”.

**Chapter 9**

**Exercise 1:** Note that this answer already includes proximity activation (expanded upon in next exercise) largely for convenience if porting the program to a physical Initio.

import bdi.initioagent as cognitive

import time

class LineFollower(cognitive.InitioAgent):

def \_\_init\_\_(self):

super().\_\_init\_\_()

start = self.AND(self.B('obstacle\_left'), self.NOT(self.B('started')))

stop = self.AND(self.B('obstacle\_left'), self.B('started'))

on\_line = self.AND(self.B('started'), self.NOT(self.OR(self.B('line\_left'), self.B('line\_right'))))

line\_on\_left = self.AND(self.B('started'), self.B('line\_left'))

line\_on\_right = self.AND(self.B('started'), self.B('line\_right'))

self.add\_condition\_rule(start, self.start\_self)

self.add\_condition\_rule(stop, self.stop\_self)

self.add\_condition\_rule(on\_line, self.forward)

self.add\_condition\_rule(line\_on\_left, self.left)

self.add\_condition\_rule(line\_on\_right, self.right)

self.add\_condition\_rule(self.B('stopping'), self.stop\_rule)

def start\_self(self):

self.add\_belief('started')

time.sleep(5)

return

def stop\_self(self):

self.drop\_belief('started')

self.add\_belief('stopping')

time.sleep(5)

return

def stop\_rule(self):

self.robot.stop()

self.done()

self.drop\_belief('stopping')

return

def forward(self):

self.robot.forward(10)

return

def left(self):

self.robot.spinLeft(10)

return

def right(self):

self.robot.forward(10)

time.sleep(1)

self.robot.spinRight(10)

time.sleep(2)

return

**Exercise 2:**

import bdi.initioagent as cognitive

import time

class ProximityActivatedAgent(cognitive.InitioAgent):

def \_\_init\_\_(self):

super().\_\_init\_\_()

start = self.AND(self.B('obstacle\_left'), self.NOT(self.B('started')))

stop = self.AND(self.B('obstacle\_left'), self.B('started'))

self.add\_condition\_rule(start, self.start\_self)

self.add\_condition\_rule(stop, self.stop\_self)

self.add\_condition\_rule(self.B('stopping'), self.stop\_rule)

def start\_self(self):

self.add\_belief('started')

time.sleep(5)

return

def stop\_self(self):

self.drop\_belief('started')

self.add\_belief('stopping')

time.sleep(5)

return

def stop\_rule(self):

self.robot.stop()

self.done()

self.drop\_belief('stopping')

return

**Exercise 3:**

import bdi.initioagent as cognitive

import time

class ProximityActivatedAgent(cognitive.InitioAgent):

def \_\_init\_\_(self):

super().\_\_init\_\_()

start = self.AND(self.B('obstacle\_left'), self.NOT(self.B('started')))

stop = self.AND(self.B('obstacle\_left'), self.B('started'))

self.add\_condition\_rule(start, self.start\_self)

self.add\_condition\_rule(stop, self.stop\_self)

self.add\_condition\_rule(self.B('stopping'), self.stop\_rule)

def start\_self(self):

self.add\_belief('started')

time.sleep(5)

return

def stop\_self(self):

self.drop\_belief('started')

self.add\_belief('stopping')

time.sleep(5)

return

def stop\_rule(self):

self.robot.stop()

self.done()

self.drop\_belief('stopping')

return

import bdi.initioagent as cognitive

import time

class LineFollower(ProximityActivatedAgent):

def \_\_init\_\_(self):

super().\_\_init\_\_()

on\_line = self.AND(self.B('started'), self.NOT(self.OR(self.B('line\_left'), self.B('line\_right'))))

line\_on\_left = self.AND(self.B('started'), self.B('line\_left'))

line\_on\_right = self.AND(self.B('started'), self.B('line\_right'))

self.add\_condition\_rule(on\_line, self.forward)

self.add\_condition\_rule(line\_on\_left, self.left)

self.add\_condition\_rule(line\_on\_right, self.right)

def forward(self):

self.robot.forward(10)

return

def left(self):

self.robot.spinLeft(10)

return

def right(self):

self.robot.forward(10)

time.sleep(1)

self.robot.spinRight(10)

time.sleep(2)

return

**Exercise 4:**

import bdi.initioagent as cognitive

import time

class SwitchActivatedAgent(cognitive.Pi2GoAgent):

def \_\_init\_\_(self):

super().\_\_init\_\_()

start = self.AND(self.B('switch\_pressed'), self.NOT(self.B('started')))

stop = self.AND(self.B('switch\_pressed'), self.B('started'))

self.add\_condition\_rule(start, self.start\_self)

self.add\_condition\_rule(stop, self.stop\_self)

self.add\_condition\_rule(self.B('stopping'), self.stop\_rule)

def start\_self(self):

self.add\_belief('started')

time.sleep(5)

return

def stop\_self(self):

self.drop\_belief('started')

self.add\_belief('stopping')

time.sleep(5)

return

def stop\_rule(self):

self.robot.stop()

self.done()

self.drop\_belief('stopping')

return

import bdi.pi2goagent as cognitive

import time

class WallFollower(SwitchActivatedAgent):

def \_\_init\_\_(self):

super().\_\_init\_\_()

wall\_on\_right = self.AND(self.B('started'), self.AND(self.B('obstacle\_right'), self.NOT(self.b\_obstacle\_centre)))

wall\_in\_front = self.AND(self.B('started'), self.b\_obstacle\_centre)

lost\_wall = self.AND(self.B('started'), self.NOT(self.OR(self.b\_obstacle\_centre, self.B('obstacle\_right'))))

floor = self.AND(self.B('started'), self.B('line\_left'))

self.add\_condition\_rule(floor, self.stop\_rule)

self.add\_condition\_rule(wall\_on\_right, self.forward)

self.add\_condition\_rule(wall\_in\_front, self.left)

self.add\_condition\_rule(lost\_wall, self.right)

def forward(self):

self.robot.forward(10)

return

def left(self):

self.robot.spinLeft(10)

return

def right(self):

self.robot.forward(10)

time.sleep(1)

self.robot.spinRight(10)

time.sleep(2)

return

def b\_obstacle\_centre():

if (agent.beliefbase['distance'] < 30):

return True

return False

**Chapter 10**

**Question 1:** It moves forward until it detects a line with its left line sensor.

**Question 2:** It moves forward until it reaches the square then it prints line\_left Goal Achieved!

**Exercise:**

import bdi.initioagent as cognitive

import time

class EnterAgent(cognitive.InitioAgent):

def \_\_init\_\_(self):

super().\_\_init\_\_()

only\_line\_goal = self.AND(self.G('line\_left'), self.NOT(self.G('obstacle\_right')))

goal\_and\_obstacle = self.AND(self.G('line\_left'), self.b\_obstacle\_centre)

goal\_and\_no\_wall = self.AND(self.G('line\_left'), self.NOT(self.B('obstacle\_right')))

goal\_and\_wall = self.AND(self.G('line\_left'), self.AND(self.B('obstacle\_right'), self.NOT(self.b\_obstacle\_centre)))

self.add\_condition\_rule(goal\_and\_obstacle, self.spin\_left\_rule)

self.add\_condition\_rule(self.G('obstacle\_right'), self.forward\_rule)

self.add\_condition\_rule(goal\_and\_no\_wall, self.spin\_right\_rule)

self.add\_condition\_rule(goal\_and\_wall, self.forward\_rule)

self.add\_goal('line\_left')

self.add\_goal('obstacle\_right')

def b\_obstacle\_centre(self):

if (self.beliefbase['distance'] < 30):

return True

return False

def forward\_rule(self):

self.robot.forward(10)

time.sleep(5)

self.robot.stop()

def spin\_left\_rule(self):

self.robot.spinLeft(10)

time.sleep(1)

self.robot.stop()

def spin\_right\_rule(self):

self.robot.spinRight(10)

time.sleep(1)

**Chapter 11**

**Exercise 1:**

import bdi.initioagent as cognitive

import random, time

class MachineLearner(cognitive.InitioAgent):

def \_\_init\_\_(self, learning\_rate):

super().\_\_init\_\_()

self.learning\_rate = learning\_rate

self.epsilon = 1

self.actions = ['forward', 'backward', 'left', 'right']

self.dictionary = self.action\_reward(self.actions, 1)

self.rewards = {(1, 1):1, (1, 0):2, (0, 1):0, (0, 0):1}

self.add\_goal('learn\_oval')

self.add\_condition\_rule(self.G('learn\_oval'), self.learn)

self.add\_condition\_rule(self.B('learn\_oval'), self.end\_learning)

def action\_reward(self, action\_list, default):

action\_rewards = {}

for i in range(0, 2):

for j in range(0, 2):

for k in (action\_list):

action\_rewards[((i, j), k)] = default

return action\_rewards

def end\_learning(self):

self.robot.stop()

print(self.dictionary)

self.done()

def get\_max(self, state):

max\_reward = 0

action = 'forward'

for act in self.actions:

if (self.dictionary[(state, act)] > max\_reward):

action = act

max\_reward = self.dictionary[(state, act)]

return action

def action\_execute(self, action):

if (action == "forward"):

self.robot.forward(10)

elif (action == "backward"):

self.robot.reverse(10)

elif (action == "left"):

self.robot.spinLeft(10)

elif (action == "right"):

self.robot.spinRight(10)

else:

self.robot.stop()

def learn(self):

explore = random.random();

state = (self.robot.irLeftLine(), self.robot.irRightLine())

if (explore < self.epsilon):

print("Exploring")

action = random.choice(self.actions)

else:

action = self.get\_max(state)

self.action\_execute(action)

time.sleep(3)

reward = self.rewards[(self.robot.irLeftLine(), self.robot.irRightLine())]

if (reward == 2):

self.epsilon = self.epsilon - 0.05

print("updating epsilon: " + str(self.epsilon))

if (self.epsilon < 0):

self.add\_belief('learn\_oval')

self.dictionary[(state, action)] = self.dictionary[(state, action)] + ( reward - self.dictionary[(state, action)]) \* self.learning\_rate

**Exercise 2:**

import bdi.initioagent as cognitive

import random, time

class MachineLearner(cognitive.InitioAgent):

def \_\_init\_\_(self, learning\_rate):

super().\_\_init\_\_()

self.learning\_rate = learning\_rate

self.epsilon = 1

self.actions = ['forward', 'backward', 'left', 'right']

self.dictionary = self.action\_reward(self.actions, 1)

self.rewards = {(1, 1):1, (1, 0):2, (0, 1):0, (0, 0):1}

self.add\_goal('follow\_oval')

self.add\_condition\_rule(self.G('learn\_oval'), self.learn)

self.add\_condition\_rule(self.AND(self.G('follow\_oval'), self.NOT(self.B('learn\_oval'))), self.first\_learn)

self.add\_condition\_rule(self.AND(self.G('follow\_oval'), self.B('learn\_oval')), self.execute\_best)

def action\_reward(self, action\_list, default):

action\_rewards = {}

for i in range(0, 2):

for j in range(0, 2):

for k in (action\_list):

action\_rewards[((i, j), k)] = default

return action\_rewards

def end\_learning(self):

self.robot.stop()

print(self.dictionary)

self.done()

def get\_max(self, state):

max\_reward = 0

action = 'forward'

for act in self.actions:

if (self.dictionary[(state, act)] > max\_reward):

action = act

max\_reward = self.dictionary[(state, act)]

return action

def action\_execute(self, action):

if (action == "forward"):

self.robot.forward(10)

elif (action == "backward"):

self.robot.reverse(10)

elif (action == "left"):

self.robot.spinLeft(10)

elif (action == "right"):

self.robot.spinRight(10)

else:

self.robot.stop()

def first\_learn(self):

self.add\_goal('learn\_oval')

def execute\_best(self):

state = (self.robot.irLeftLine(), self.robot.irRightLine())

best = self.get\_max(state)

self.action\_execute(best)

def learn(self):

explore = random.random();

state = (self.robot.irLeftLine(), self.robot.irRightLine())

if (explore < self.epsilon):

print("Exploring")

action = random.choice(self.actions)

else:

action = self.get\_max(state)

self.action\_execute(action)

time.sleep(3)

reward = self.rewards[(self.robot.irLeftLine(), self.robot.irRightLine())]

if (reward == 2):

self.epsilon = self.epsilon - 0.05

print("updating epsilon: " + str(self.epsilon))

if (self.epsilon < 0):

self.add\_belief('learn\_oval')

self.dictionary[(state, action)] = self.dictionary[(state, action)] + ( reward - self.dictionary[(state, action)]) \* self.learning\_rate

**Exercise 3:**

import bdi.initioagent as cognitive

import random, time

class ZigZag(cognitive.InitioAgent):

def \_\_init\_\_(self):

super().\_\_init\_\_()

only\_line\_goal = self.AND(self.OR(self.G('line\_left'), self.G('clear')), self.NOT(self.G('obstacle\_right')))

lgoal\_and\_obstacle = self.AND(self.G('line\_left'), self.b\_obstacle\_centre)

lgoal\_and\_no\_wall = self.AND(self.G('line\_left'), self.NOT(self.B('obstacle\_right')))

cgoal\_and\_obstacle = self.AND(self.G('clear'), self.b\_obstacle\_centre)

cgoal\_and\_no\_wall = self.AND(self.G('clear'), self.NOT(self.B('obstacle\_left')))

lgoal\_and\_wall = self.AND(self.G('line\_left'), self.AND(self.B('obstacle\_right'), self.NOT(self.b\_obstacle\_centre)))

cgoal\_and\_wall = self.AND(self.G('clear'), self.AND(self.B('obstacle\_left'), self.NOT(self.b\_obstacle\_centre)))

self.add\_condition\_rule(self.G('end'), self.end)

self.add\_condition\_rule(self.AND(self.B('line\_left'), self.B('clear')), self.remove\_clear)

self.add\_condition\_rule(self.AND(self.NOT(self.G('end')), self.AND(self.NOT(self.G('line\_left')), self.NOT(self.G('clear')))), self.ask)

self.add\_condition\_rule(self.NOT(self.OR(self.B('line\_left'), self.B('clear'))), self.add\_clear)

self.add\_condition\_rule(self.AND(self.G('obstacle\_right'), self.b\_obstacle\_centre), self.spin\_left\_rule)

self.add\_condition\_rule(self.G('obstacle\_right'), self.forward\_rule)

self.add\_condition\_rule(lgoal\_and\_obstacle, self.spin\_left\_rule)

self.add\_condition\_rule(lgoal\_and\_no\_wall, self.forward\_then\_spin\_right\_rule)

self.add\_condition\_rule(cgoal\_and\_obstacle, self.spin\_right\_rule)

self.add\_condition\_rule(cgoal\_and\_no\_wall, self.forward\_then\_spin\_left\_rule)

self.add\_condition\_rule(lgoal\_and\_wall, self.forward\_rule)

self.add\_condition\_rule(cgoal\_and\_wall, self.forward\_rule)

def end(self):

self.add\_belief('end')

self.done()

def ask(self):

self.drop\_belief('end')

goal = input('What would you like me to do? [B]find black,[C]find clear,[E]end')

if (goal == "B"):

self.add\_goal('obstacle\_right')

self.add\_goal('line\_left')

elif (goal == "C"):

self.add\_goal('obstacle\_right')

self.add\_goal('clear')

else:

self.add\_goal('end')

def remove\_clear(self):

print("dropping clear")

self.drop\_belief('clear')

def add\_clear(self):

self.add\_belief('clear')

def forward\_rule(self):

self.robot.forward(10)

time.sleep(5)

self.robot.stop()

def spin\_left\_rule(self):

self.robot.spinLeft(10)

time.sleep(1)

self.robot.stop()

def spin\_right\_rule(self):

self.robot.spinRight(10)

time.sleep(1)

self.robot.stop()

def forward\_then\_spin\_left\_rule(self):

self.robot.forward(10)

time.sleep(1)

self.robot.spinLeft(10)

time.sleep(2)

self.robot.stop()

def forward\_then\_spin\_right\_rule(self):

self.robot.forward(10)

time.sleep(1)

self.robot.spinRight(10)

time.sleep(2)

self.robot.stop()

def b\_obstacle\_centre(self):

if (self.beliefbase['distance'] < 30):

return True

return False

**Chapter 12**

**Note:** This chapter is supposed to give some idea how complex programs/classes can be built up from simpler ones by creating a class that draws upon multiple objects. While it serves as an example, the worksheet doesn’t really seek to *teach* how this may be done and students may benefit from discussion and encouragement to think about what is going on here.

**Question 1:** The students should get varying results depending upon whether they supply the values for left, right and centre or the objects gets the values from the virtual Initio’s sensors. Students may need some help importing the initio module, initialising the initio and passing it as an argument to the class.

**Exercise 1:**

import simclient.simrobot as initio

import time

class LineState:

def \_\_init\_\_(self, robot, left=None, right=None):

if (left is None):

self.left = robot.irLeftLine()

else:

self.left = left

if (right is None):

self.right = robot.irRightLine()

else:

self.right = right

def print\_state(self):

string = "(" + str(self.left) + " ," + str(self.right) + ")"

print(string)

**Question 2:** It prints False.

**Exercise 2:**

import simclient.simrobot as initio

import time

class ObstacleState:

def \_\_init\_\_(self, robot, left=None, right=None, centre=None):

if (left is None):

self.left = robot.irLeft()

else:

self.left = left

if (right is None):

self.right = robot.irRight()

else:

self.right = right

if (centre is None):

self.centre = robot.irCentre()

else:

self.centre = centre

def \_\_eq\_\_(self, other):

if self.left == other.left and self.right == other.right and self.centre == other.centre:

return True

else:

return False

def \_\_ne\_\_(self, other):

if self.left != other.left or self.right != other.right or self.centre != other.centre:

return True

else:

return False

def print\_state(self):

string = "(" + str(self.centre) + " ," + str(self.left) + " ," + str(self.right) + ")"

print(string)

**Exercise 3 & 4:** This is the answer to exercise 3 including the hash function from exercise 4. Going into depth about hash functions is outside the scope of these worksheets and has not been explained in detail. The Wikipedia page on hash functions is respectable, if quite technical, if students want to learn more.

class State:

def calculate\_reward(self):

return 1

def print\_state(self):

print("This is the base class")

class ObstacleState(State):

def \_\_init\_\_(self, robot, left=None, right=None, centre=None):

if (left is None):

self.left = robot.irLeft()

else:

self.left = left

if (right is None):

self.right = robot.irRight()

else:

self.right = right

if (centre is None):

self.centre = robot.irCentre()

else:

self.centre = centre

def calculate\_reward(self):

if self.right and not self.centre:

return 4

elif not self.left:

return 3

elif self.centre:

return 2

else:

return 0

def \_\_eq\_\_(self, other):

if self.left == other.left and self.right == other.right and self.centre == other.centre:

return True

else:

return False

def \_\_ne\_\_(self, other):

if self.left != other.left or self.right != other.right or self.centre != other.centre:

return True

else:

return False

def \_\_hash\_\_(self):

return self.left+3\*self.right+7\*self.centre

def print\_state(self):

string = "(" + str(self.centre) + " ," + str(self.left) + " ," + str(self.right) + ")"

print(string)

class LineState(State):

def \_\_init\_\_(self, robot, left=None, right=None):

if (left is None):

self.left = robot.irLeftLine()

else:

self.left = left

if (right is None):

self.right = robot.irRightLine()

else:

self.right = right

def calculate\_reward(self):

if self.right and not self.left:

return 2

elif self.right == self.left:

return 1

else:

return 0

def \_\_eq\_\_(self, other):

if self.left == other.left and self.right == other.right:

return True

else:

return False

def \_\_ne\_\_(self, other):

if self.left != other.left or self.right != other.right:

return True

else:

return False

def \_\_hash\_\_(self):

return self.left+3\*self.right

def print\_state(self):

string = "(" + str(self.left) + " ," + str(self.right) + ")"

print(string)

**Exercise 5:** Showing just the RewardDictionary classes – but the State classes will also be needed.

class RewardDictionary():

def \_\_init\_\_(self, robot):

self.reward\_dictionary = {}

self.learning\_rate = 0.5

self.robot = robot

def newState(self, robot):

print("Base Class: Not Implemented")

def max\_reward(self):

print("Base Class: Not Implemented")

def best\_action(self, state):

max\_reward = 0

for act in self.actions:

if (self.reward\_dictionary[(state, act)] > max\_reward):

action = act

max\_reward = self.reward\_dictionary[(state, act)]

return action

def update\_reward(self, state, action, reward):

self.reward\_dictionary([(state, action)]) = self.reward\_dictionary([state, action]) + (reward - self.reward\_dictionary([state, action]))\*self.learning\_rate

class ObstacleRewardDictionary():

def \_\_init\_\_(self, robot, actions):

super().\_\_init\_\_()

for i in range(0, 2):

for j in range(0, 2):

for k in range(0, 2):

for a in (actions):

self.reward\_dictionary[(ObstacleState(robot, i, j, k), a)] = 2

def newState(self):

return ObstacleState(self.robot)

def max\_reward(self):

return 4

class LineRewardDictionary():

def \_\_init\_\_(self, robot, actions):

super().\_\_init\_\_(robot)

for i in range(0, 2):

for j in range(0, 2):

for a in (actions):

self.reward\_dictionary[(LineState(robot, i, j), a)] = 2

def newState(self):

return LineState(self.robot)

def max\_reward(self):

return 2

**Exercise 6:**

import simclient.simrobot as initio

import time

import random

class State:

def calculate\_reward(self):

return 1

def print\_state(self):

print("This is the base class")

class ObstacleState(State):

def \_\_init\_\_(self, robot, left=None, right=None, centre=None):

if (left is None):

self.left = robot.irLeft()

else:

self.left = left

if (right is None):

self.right = robot.irRight()

else:

self.right = right

if (centre is None):

self.centre = robot.irCentre()

else:

self.centre = centre

def calculate\_reward(self):

if self.right and not self.centre:

return 4

elif not self.left:

return 3

elif self.centre:

return 2

else:

return 0

def \_\_eq\_\_(self, other):

if self.left == other.left and self.right == other.right and self.centre == other.centre:

return True

else:

return False

def \_\_ne\_\_(self, other):

if self.left != other.left or self.right != other.right or self.centre != other.centre:

return True

else:

return False

def \_\_hash\_\_(self):

return self.left+3\*self.right+7\*self.centre

def print\_state(self):

string = "(" + str(self.centre) + " ," + str(self.left) + " ," + str(self.right) + ")"

print(string)

class LineState(State):

def \_\_init\_\_(self, robot, left=None, right=None):

if (left is None):

self.left = robot.irLeftLine()

else:

self.left = left

if (right is None):

self.right = robot.irRightLine()

else:

self.right = right

def calculate\_reward(self):

if self.right and not self.left:

return 2

elif self.right == self.left:

return 1

else:

return 0

def \_\_eq\_\_(self, other):

if self.left == other.left and self.right == other.right:

return True

else:

return False

def \_\_ne\_\_(self, other):

if self.left != other.left or self.right != other.right:

return True

else:

return False

def \_\_hash\_\_(self):

return self.left+3\*self.right

def print\_state(self):

string = "(" + str(self.left) + " ," + str(self.right) + ")"

print(string)

class RewardDictionary():

def \_\_init\_\_(self, robot, actions):

self.reward\_dictionary = {}

self.learning\_rate = 0.5

self.actions = actions

self.robot = robot

def newState(self):

print("Base Class: Not Implemented")

def max\_reward(self):

print("Base Class: Not Implemented")

def best\_action(self, state):

max\_reward = 0

for act in self.actions:

if (self.reward\_dictionary[(state, act)] > max\_reward):

action = act

max\_reward = self.reward\_dictionary[(state, act)]

return action

def update\_reward(self, state, action, reward):

self.reward\_dictionary[(state, action)] = self.reward\_dictionary[state, action] + (reward - self.reward\_dictionary[state, action])\*self.learning\_rate

class ObstacleRewardDictionary(RewardDictionary):

def \_\_init\_\_(self, robot, actions):

super().\_\_init\_\_(robot, actions)

for i in range(0, 2):

for j in range(0, 2):

for k in range(0, 2):

for a in (actions):

self.reward\_dictionary[(ObstacleState(robot, i, j, k), a)] = 2

def newState(self):

return ObstacleState(self.robot)

def max\_reward(self):

return 4

class LineRewardDictionary(RewardDictionary):

def \_\_init\_\_(self, robot, actions):

super().\_\_init\_\_(robot, actions)

for i in range(0, 2):

for j in range(0, 2):

for a in (actions):

self.reward\_dictionary[(LineState(robot, i, j), a)] = 1

def newState(self):

return LineState(self.robot)

def max\_reward(self):

return 2

class MachineLearner():

def \_\_init\_\_(self, reward\_dictionary):

self.reward\_dictionary = reward\_dictionary

self.epsilon = 1

self.epsilon\_reduce = 0.05

def execute\_action(self, action):

if (action == "forward"):

initio.forward(10)

elif (action == "left"):

initio.spinLeft(10)

elif (action == "right"):

initio.spinRight(10)

else:

initio.stop()

time.sleep(3)

def learn(self):

while (self.epsilon > 0):

explore = random.random()

state = self.reward\_dictionary.newState()

if (explore < self.epsilon):

action = random.choice(self.reward\_dictionary.actions)

print("Random Action: " + action)

else:

action = self.reward\_dictionary.best\_action(state)

print("Best Action: " + action)

self.execute\_action(action)

new\_state = self.reward\_dictionary.newState()

reward = new\_state.calculate\_reward()

self.reward\_dictionary.update\_reward(state, action, reward)

if (reward == self.reward\_dictionary.max\_reward()):

self.epsilon = self.epsilon - self.epsilon\_reduce

print("New epsilon: " + str(self.epsilon))

initio.stop()

**Exericse 7:** Students should be encouraged to take their results from exercise 6 and adapt for this exercise by adding in cognitive agent aspects. They will need to take care to add self. in all the right places. They may also want to look at some of the exercise answers for chapters 10 & 11 to get ideas for how to do things like controlling getting input from the user.

import bdi.initioagent as cognitive

import time

import random

class State:

def calculate\_reward(self):

return 1

def print\_state(self):

print("This is the base class")

class ObstacleState(State):

def \_\_init\_\_(self, initio, left=None, right=None, centre=None):

if (left is None):

self.left = initio.irLeft()

else:

self.left = left

if (right is None):

self.right = initio.irRight()

else:

self.right = right

if (centre is None):

self.centre = initio.getDistance() < 30

else:

self.centre = centre

def calculate\_reward(self):

if self.right and not self.centre:

return 4

elif not self.left:

return 3

elif self.centre:

return 2

else:

return 0

def \_\_eq\_\_(self, other):

if self.left == other.left and self.right == other.right and self.centre == other.centre:

return True

else:

return False

def \_\_ne\_\_(self, other):

if self.left != other.left or self.right != other.right or self.centre != other.centre:

return True

else:

return False

def \_\_hash\_\_(self):

return self.left+3\*self.right

def print\_state(self):

string = "(" + str(self.centre) + " ," + str(self.left) + " ," + str(self.right) + ")"

print(string)

class LineState(State):

def \_\_init\_\_(self, initio, left=None, right=None):

if (left is None):

self.left = initio.irLeftLine()

else:

self.left = left

if (right is None):

self.right = initio.irRightLine()

else:

self.right = right

def calculate\_reward(self):

if self.right and not self.left:

return 2

elif self.right == self.left:

return 1

else:

return 0

def \_\_eq\_\_(self, other):

if self.left == other.left and self.right == other.right:

return True

else:

return False

def \_\_ne\_\_(self, other):

if self.left != other.left or self.right != other.right:

return True

else:

return False

def \_\_hash\_\_(self):

return self.left+3\*self.right

def print\_state(self):

string = "(" + str(self.left) + " ," + str(self.right) + ")"

print(string)

class RewardDictionary():

def \_\_init\_\_(self, actions, robot):

self.reward\_dictionary = {}

self.learning\_rate = 0.5

self.actions = actions

self.robot = robot

def newState(self):

print("Base Class: Not Implemented")

def max\_reward(self):

print("Base Class: Not Implemented")

def best\_action(self, state):

max\_reward = 0

for act in self.actions:

if (self.reward\_dictionary[(state, act)] > max\_reward):

action = act

max\_reward = self.reward\_dictionary[(state, act)]

return action

def update\_reward(self, state, action, reward):

self.reward\_dictionary[(state, action)] = self.reward\_dictionary[state, action] + (reward - self.reward\_dictionary[state, action])\*self.learning\_rate

class ObstacleRewardDictionary(RewardDictionary):

def \_\_init\_\_(self, actions, robot):

super().\_\_init\_\_(actions, robot)

for i in range(0, 2):

for j in range(0, 2):

for k in range(0, 2):

for a in (actions):

self.reward\_dictionary[(ObstacleState(robot, i, j, k), a)] = 2

def newState(self):

return ObstacleState(self.robot)

def max\_reward(self):

return 4

class LineRewardDictionary(RewardDictionary):

def \_\_init\_\_(self, actions, robot):

super().\_\_init\_\_(actions, robot)

for i in range(0, 2):

for j in range(0, 2):

for a in (actions):

self.reward\_dictionary[(LineState(robot, i, j), a)] = 1

def newState(self):

return LineState(self.robot)

def max\_reward(self):

return 2

class MachineLearner(cognitive.InitioAgent):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.epsilon = 1

self.epsilon\_reduce = 0.05

self.actions = ['forward', 'left', 'right']

self.oval\_reward\_dictionary = LineRewardDictionary(self.actions, self.robot)

self.wall\_reward\_dictionary = ObstacleRewardDictionary(self.actions, self.robot)

self.add\_condition\_rule(self.G('end'), self.end)

self.add\_condition\_rule(self.AND(self.NOT(self.G('end')), self.AND(self.NOT(self.G('follow\_oval')), self.NOT(self.G('follow\_wall')))), self.ask)

self.add\_condition\_rule(self.B('obstacle\_left'), self.drop\_goals)

self.add\_condition\_rule(self.G('learn\_oval'), self.learn\_oval)

self.add\_condition\_rule(self.G('learn\_wall'), self.learn\_wall)

self.add\_condition\_rule(self.AND(self.G('follow\_oval'), self.NOT(self.B('learn\_oval'))), self.first\_learn\_oval)

self.add\_condition\_rule(self.AND(self.G('follow\_wall'), self.NOT(self.B('learn\_wall'))), self.first\_learn\_wall)

self.add\_condition\_rule(self.AND(self.G('follow\_oval'), self.B('learn\_oval')), self.execute\_best\_oval)

self.add\_condition\_rule(self.AND(self.G('follow\_wall'), self.B('learn\_wall')), self.execute\_best\_wall)

def drop\_goals(self):

self.drop\_goal('follow\_oval')

self.drop\_goal('follow\_wall')

self.robot.stop()

def execute\_action(self, action):

if (action == "forward"):

self.robot.forward(10)

elif (action == "left"):

self.robot.spinLeft(10)

elif (action == "right"):

self.robot.spinRight(10)

else:

self.robot.stop()

time.sleep(3)

def ask(self):

self.drop\_belief('end')

goal = input('What would you like me to do? [O]follow\_oval,[W]follow\_wall,[E]end')

if (goal == "O"):

self.add\_goal('follow\_oval')

elif (goal == "W"):

self.add\_goal('follow\_wall')

else:

self.add\_goal('end')

def end(self):

self.add\_belief('end')

self.done()

def first\_learn\_oval(self):

self.add\_goal('learn\_oval')

def first\_learn\_wall(self):

self.add\_goal('learn\_wall')

def learn\_oval(self):

self.epsilon = 1

self.learn(self.oval\_reward\_dictionary)

self.add\_belief('learn\_oval')

def learn\_wall(self):

self.epsilon = 1

print("setting epsilon to 1")

self.learn(self.wall\_reward\_dictionary)

self.add\_belief('learn\_wall')

def execute\_best(self, reward\_dictionary):

state = reward\_dictionary.newState()

action = reward\_dictionary.best\_action(state)

self.execute\_action(action)

def execute\_best\_oval(self):

self.execute\_best(self.oval\_reward\_dictionary)

def execute\_best\_wall(self):

self.execute\_best(self.wall\_reward\_dictionary)

def learn(self, reward\_dictionary):

while (self.epsilon > 0):

explore = random.random()

state = reward\_dictionary.newState()

if (explore < self.epsilon):

action = random.choice(reward\_dictionary.actions)

print("Random Action: " + action)

else:

action = reward\_dictionary.best\_action(state)

print("Best Action: " + action)

self.execute\_action(action)

new\_state = reward\_dictionary.newState()

reward = new\_state.calculate\_reward()

reward\_dictionary.update\_reward(state, action, reward)

if (reward == reward\_dictionary.max\_reward()):

self.epsilon = self.epsilon - self.epsilon\_reduce

print("New epsilon: " + str(self.epsilon))

self.robot.stop()



University of Liverpool, 2020

This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0/).