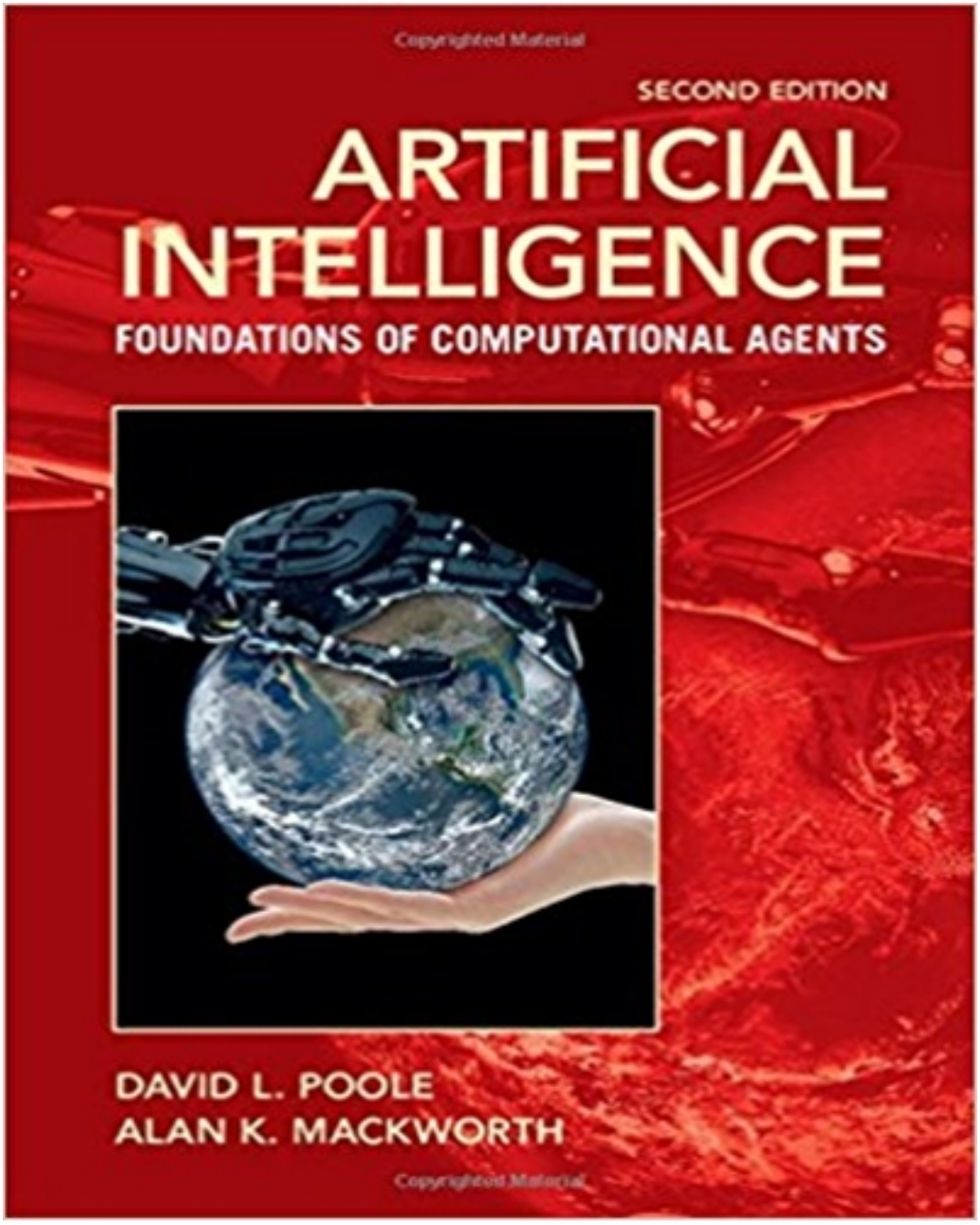
Solutions for Artificial Intelligence Foundations of Computational Agents 2nd Edition by Poole

CLICK HERE TO ACCESS COMPLETE Solutions



Solutions

Chapter 2

Agent Architectures and Hierarchical Control

Exercise 2.1 The start of Section 2.3 on page 56 argued that it was impossible to build a representation of a world independently of what the agent will do with it. This exercise lets you evaluate this argument.

Choose a particular world, for example, the things on top of your desk right now.

i) Get someone to list all of the individuals (things) that exist in this world (or try it yourself as a thought experiment).

ii) Try to think of twenty individuals that they missed. Make these as different from each other as possible. For example, the ball at the tip of the rightmost ball-point pen on the desk, the part of the stapler that makes the staples bend, or the third word on page 72 of a particular book on the desk.

iii) Try to find an individual that cannot be described using your natural language (such as a particular component of the texture of the desk).

iv) Choose a particular task, such as making the desk tidy, and try to write down all of the individuals in the world at a level of description relevant to this task.

Based on this exercise, discuss the following statements.

(a) What exists in a world is a property of the observer.

(b) We need ways to refer to individuals other than expecting each individual to have a separate name.

(c) Which individuals exist is a property of the task as well as of the world.

(d) To describe the individuals in a domain, you need what is essentially a dictionary of a huge number of words and ways to combine them, and this should be able to be done independently of any particular domain.

Exercise 2.2 Consider the top level controller of Example 2.6 on page 63

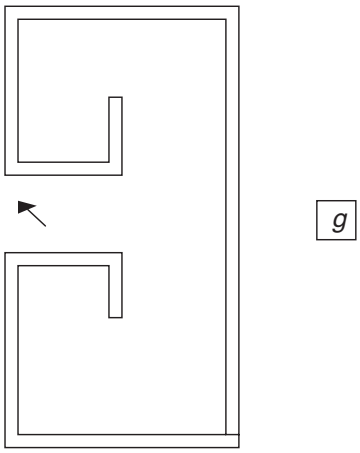


Figure 2.1: A robot trap

(a) If the lower level reach the timeout without getting to the target position, what does the agent do?

(b) The definition of the target position means that, when the plan ends, the top level stops. This is not reasonable for the robot that can only change directions and cannot stop. Change the definition so that the robot keeps going.

Exercise 2.3 The obstacle avoidance implemented in Example 2.5 on page 61 can easily get stuck.

(a) Show an obstacle and a target for which the robot using the controller of Example 2.5 on page 61 would not be able to get around (and it will crash or loop).

(b) Even without obstacles, the robot may never reach its destination. For example, if the robot is close to its target position, but not close enough to have arrived, it may keep circling forever without reaching its target. Design a controller that can detect this situation and find its way to the target.

Exercise 2.4 Consider the "robot trap" in Figure 2.11.

(a) This question is to explore why it is so tricky for a robot to get to location g. Explain what the current robot does. Suppose one was to implement a robot that follows the wall using the "right-hand rule": the robot turns left when it hits an obstacle and keeps following a wall, with the wall always on its right. Is there a simple characterization of the situations in which the robot should keep following this rule or head towards the target?

(b) An intuition of how to escape such a trap is that, when the robot hits a wall, it follows the wall until the number of right turns equals the number of left turns. Show how this can be implemented, explaining the belief state, and the functions of the layer.

Solution Replace the middle level with `http://cs.ubc.ca/~poole/aibook/figures/ch02/robot-middle-trap.txt`

Exercise 2.5 If the current target location were to be moved, the middle layer of Example 2.5 on page 61 travels to the original position of that target and does not try to go to the new position. Change the controller so that the robot can adapt to targets moving.

Solution Instead of remembering the coordinates, the robot remembers the location name, then look up its coordinates every time. Replace the first rule of the high layer with:

```
assign(target_pos,Loc,T) <-
     arrived(T) &
     was(to_do,[goto(Loc)|_],_,T).
```

Replace the first rule of the middle layer with:

```
target_direction(G,T) <-
     val(robot_pos,(X0,Y0),T) &
     val(target_pos,Loc,T) &
     at(Loc,(X1,Y1)) &
     direction((X0,Y0),(X1,Y1),G).
```

Replace the eighth rule of the middle layer with:

```
arrived(T) <-
     was(target_pos,Loc,_,T) &
     at(Loc,Target_Coords) &
     robot_pos(Robot_Coords,T) &
     close_enough(Target_Coords,Robot_Coords).
```

Exercise 2.6 The current controller visits the locations in the *to_do* list sequentially.

(a) Change the controller so that it is opportunistic; when it selects the next location to visit, it selects the location that is closest to its current position. It should still visit all the locations.

(b) Give one example of an environment in which the new controller visits all the locations in fewer time steps than the original controller.

(c) Give one example of an environment in which the original controller visits all the locations in fewer time steps than the modified controller.

(d) Change the controller so that, at every step, the agent heads toward whichever target location is closest to its current position.

(e) Can the controller from part (d) get stuck and never reach a target in an example where the original controller will work? Either give an example in which it gets stuck and explain why it cannot find a solution, or explain why it gets to a goal whenever the original can.

Solution See `http://www.ubc.ca/~poole/aibook/figures/ch02/opportunistic-top.txt` for the `AISpace.org` applet robot controller. Note that this applet is really difficult to use, as good debugging facilities have not been implemented.

Exercise 2.7 Change the controller so that the robot senses the environment to determine the coordinates of a location. Assume that the body can provide the coordinates of a named location.

Exercise 2.8 Suppose the robot has a battery that must be charged at a particular wall socket before it runs out. How should the robot controller be modified to allow for battery recharging?

Exercise 2.9 Suppose you have a new job and must build a controller for an intelligent robot. You tell your bosses that you just have to implement a command function and a state transition function. They are very skeptical. Why these functions? Why only these? Explain why a controller requires a command function and a state transition function, but not other functions. Use proper English. Be concise.

Solution A robot only has access to what it observes and what it remembers. At each time step it has to act (hence the command function) and it has to determine what to remember for the future (which leads to the state transition function). As the robot is completely determined by its internal state and its behaviour, these are the only two function necessary.

Sample Exam Questions

Exercise 2.10

(a) Explain why we use hierarchical (layered) controllers.

(b) What does it mean that the higher layers run at a different time scale than lower layers?

(c) Give the intuition behind the notion of a transduction? Why do we define causal transductions?

(d) Why does an agent have a state?

Chapter 2

Agents and Control

This implements the controllers described in Chapter 2.

In this version the higher-levels call the lower-levels. A more sophisticated version may have them run concurrently (either as coroutines or in parallel). The higher-levels calling the lower-level works in simulated environments when there is a single agent, and where the lower-level are written to make sure they return (and don't go on forever), and the higher level doesn't take too long (as the lower-levels will wait until called again).

2.1 Representing Agents and Environments

An agent observes the world, and carries out actions in the environment, it also has an internal state that it updates. The environment takes in actions of the agents, updates it internal state and returns the percepts.

In this implementation, the state of the agent and the state of the environment are represented using standard Python variables, which are updated as the state changes. The percepts and the actions are represented as variable-value dictionaries.

An agent implements the $go(n)$ method, where n is an integer. This means that the agent should run for n time steps.

In the following code `raise NotImplementedError()` is a way to specify an abstract method that needs to be overidden in any implemented agent or environment.

agents.py — Agent and Controllers

```
11 import random
12
13 class Agent(object):
14     def __init__(self,env):
```

```
15          """set up the agent"""
16          self.env=env
17
18      def go(self,n):
19          """acts for n time steps"""
20          raise NotImplementedError("go") # abstract method
```

The environment implements a *do*(*action*) method where *action* is a variable-value dictionary. This returns a percept, which is also a variable-value dictionary. The use of dictionaries allows for structured actions and percepts.

Note that *Environment* is a subclass of *Displayable* so that it can use the *display* method described in Section 1.7.1.

agents.py — (continued)

```
22 from utilities import Displayable
23 class Environment(Displayable):
24     def initial_percepts(self):
25         """returns the initial percepts for the agent"""
26         raise NotImplementedError("initial_percepts") # abstract method
27
28     def do(self,action):
29         """does the action in the environment
30         returns the next percept """
31         raise NotImplementedError("do") # abstract method
```

2.2 Paper buying agent and environment

To run the demo, in folder "aipython", load "agents.py", using e.g., `ipython -i agents.py`, and copy and paste the commented-out commands at the bottom of that file. This requires Python 3 with `matplotlib`.

This is an implementation of the paper buying example.

2.2.1 The Environment

The environment state is given in terms of the *time* and the amount of paper in *stock*. It also remembers the in-stock history and the price history. The percepts are the price and the amount of paper in stock. The action of the agent is the number to buy.

Here we assume that the prices are obtained from the *prices* list plus a random integer in range $[0, max\_price\_addon)$ plus a linear "inflation". The agent cannot access the price model; it just observes the prices and the amount in stock.

agents.py — (continued)

```
33 class TP_env(Environment):
```

```
    prices = [234, 234, 234, 234, 255, 255, 275, 275, 211, 211, 211,
    234, 234, 234, 234, 199, 199, 275, 275, 234, 234, 234, 234, 255,
    255, 260, 260, 265, 265, 265, 265, 270, 270, 255, 255, 260, 260,
    265, 265, 150, 150, 265, 265, 270, 270, 255, 255, 260, 260, 265,
    265, 265, 265, 270, 270, 211, 211, 255, 255, 260, 260, 265, 265,
    260, 265, 270, 270, 205, 255, 255, 260, 260, 265, 265, 265, 265,
    270, 270]
    max_price_addon = 20 # maximum of random value added to get price

    def __init__(self):
        """paper buying agent"""
        self.time=0
        self.stock=20
        self.stock_history = [] # memory of the stock history
        self.price_history = [] # memory of the price history

    def initial_percepts(self):
        """return initial percepts"""
        self.stock_history.append(self.stock)
        price = self.prices[0]+random.randrange(self.max_price_addon)
        self.price_history.append(price)
        return {'price': price,
                'instock': self.stock}

    def do(self, action):
        """does action (buy) and returns percepts (price and instock)"""
        used = pick_from_dist({6:0.1, 5:0.1, 4:0.2, 3:0.3, 2:0.2, 1:0.1})
        bought = action['buy']
        self.stock = self.stock+bought-used
        self.stock_history.append(self.stock)
        self.time += 1
        price = (self.prices[self.time%len(self.prices)] # repeating pattern
                 +random.randrange(self.max_price_addon) # plus randomness
                 +self.time//2)                          # plus inflation
        self.price_history.append(price)
        return {'price': price,
                'instock': self.stock}
```

The *pick_from_dist* method takes in a *item* : *probability* dictionary, and returns one of the items in proportion to its probability.

agents.py — (continued)

```
def pick_from_dist(item_prob_dist):
    """ returns a value from a distribution.
    item_prob_dist is an item:probability dictionary, where the
        probabilities sum to 1.
    returns an item chosen in proportion to its probability
    """
    ranreal = random.random()
    for (it,prob) in item_prob_dist.items():
        if ranreal < prob:
```

```
81              return it
82          else:
83              ranreal -= prob
84      raise RuntimeError(str(item_prob_dist)+" is not a probability distribution")
```

2.2.2 The Agent

The agent does not have access to the price model but can only observe the current price and the amount in stock. It has to decide how much to buy.

The belief state of the agent is an estimate of the average price of the paper, and the total amount of money the agent has spent.

agents.py — (continued)

```
86  class TP_agent(Agent):
87      def __init__(self, env):
88          self.env = env
89          self.spent = 0
90          percepts = env.initial_percepts()
91          self.ave = self.last_price = percepts['price']
92          self.instock = percepts['instock']
93
94      def go(self, n):
95          """go for n time steps
96          """
97          for i in range(n):
98              if self.last_price < 0.9*self.ave and self.instock < 60:
99                  tobuy = 48
100             elif self.instock < 12:
101                 tobuy = 12
102             else:
103                 tobuy = 0
104             self.spent += tobuy*self.last_price
105             percepts = env.do({'buy': tobuy})
106             self.last_price = percepts['price']
107             self.ave = self.ave+(self.last_price-self.ave)*0.05
108             self.instock = percepts['instock']
```

Set up an environment and an agent. Uncomment the last lines to run the agent for 90 steps, and determine the average amount spent.

agents.py — (continued)

```
110 env = TP_env()
111 ag = TP_agent(env)
112 #ag.go(90)
113 #ag.spent/env.time ## average spent per time period
```

2.2.3 Plotting

The following plots the price and number in stock history:

agents.py — (continued)

```
115 import matplotlib.pyplot as plt
116
117 class Plot_prices(object):
118     """Set up the plot for history of price and number in stock"""
119     def __init__(self, ag,env):
120         self.ag = ag
121         self.env = env
122         plt.ion()
123         plt.xlabel("Time")
124         plt.ylabel("Number in stock.                                    Price.")
125
126     def plot_run(self):
127         """plot history of price and instock"""
128         num = len(env.stock_history)
129         plt.plot(range(num),env.stock_history,label="In stock")
130         plt.plot(range(num),env.price_history,label="Price")
131         #plt.legend(loc="upper left")
132         plt.draw()
133
134 # pl = Plot_prices(ag,env)
135 # ag.go(90); pl.plot_run()
```

2.3 Hierarchical Controller

To run the hierarchical controller, in folder "aipython", load "agentTop.py", using e.g., `ipython -i agentTop.py`, and copy and paste the commands near the bottom of that file. This requires Python 3 with `matplotlib`.

In this implementation, each layer, including the top layer, implements the environment class, because each layer is seen as an environment from the layer above.

We arbitrarily divide the environment and the body, so that the environment just defines the walls, and the body includes everything to do with the agent. Note that the named locations are part of the (top-level of the) agent, not part of the environment, although they could have been.

2.3.1 Environment

The environment defines the walls.

agentEnv.py — Agent environment

```
11 import math
12 from agents import Environment
13
14 class Rob_env(Environment):
```

```
    def __init__(self,walls = {}):
        """walls is a set of line segments
              where each line segment is of the form ((x0,y0),(x1,y1))
        """
        self.walls = walls
```

2.3.2 Body

The body defines everything about the agent body.

agentEnv.py — (continued)

```
import math
from agents import Environment
import matplotlib.pyplot as plt
import time

class Rob_body(Environment):
    def __init__(self, env, init_pos=(0,0,90)):
        """ env is the current environment
        init_pos is a triple of (x-position, y-position, direction)
            direction is in degrees; 0 is to right, 90 is straight-up, etc
        """
        self.env = env
        self.rob_x, self.rob_y, self.rob_dir = init_pos
        self.turning_angle = 18 # degrees that a left makes
        self.whisker_length = 6 # length of the whisker
        self.whisker_angle = 30 # angle of whisker relative to robot
        self.crashed = False
        # The following control how it is plotted
        self.plotting = True    # whether the trace is being plotted
        self.sleep_time = 0.05  # time between actions (for real-time plotting)
        # The following are data structures maintained:
        self.history = [(self.rob_x, self.rob_y)] # history of (x,y) positions
        self.wall_history = []  # history of hitting the wall

    def percepts(self):
        return {'rob_x_pos':self.rob_x, 'rob_y_pos':self.rob_y,
                'rob_dir':self.rob_dir, 'whisker':self.whisker() , 'crashed':self.crashed}
    initial_percepts = percepts # use percept function for initial percepts too

    def do(self,action):
        """ action is {'steer':direction}
        direction is 'left', 'right' or 'straight'
        """
        if self.crashed:
            return self.percepts()
        direction = action['steer']
        compass_deriv = {'left':1,'straight':0,'right':-1}[direction]*self.turning_angle
        self.rob_dir = (self.rob_dir + compass_deriv +360)%360 # make in range [0,360)
        rob_x_new = self.rob_x + math.cos(self.rob_dir*math.pi/180)
```

```
60          rob_y_new = self.rob_y + math.sin(self.rob_dir*math.pi/180)
61          path = ((self.rob_x,self.rob_y),(rob_x_new,rob_y_new))
62          if any(line_segments_intersect(path,wall) for wall in self.env.walls):
63              self.crashed = True
64              if self.plotting:
65                  plt.plot([self.rob_x],[self.rob_y],"r*",markersize=20.0)
66                  plt.draw()
67          self.rob_x, self.rob_y = rob_x_new, rob_y_new
68          self.history.append((self.rob_x, self.rob_y))
69          if self.plotting and not self.crashed:
70              plt.plot([self.rob_x],[self.rob_y],"go")
71              plt.draw()
72              plt.pause(self.sleep_time)
73          return self.percepts()
```

This detects if the whisker and the wall intersect. It's value is returned as a percept.

agentEnv.py — (continued)

```
75      def whisker(self):
76          """returns true whenever the whisker sensor intersects with a wall
77          """
78          whisk_ang_world = (self.rob_dir-self.whisker_angle)*math.pi/180
79              # angle in radians in world coordinates
80          wx = self.rob_x + self.whisker_length * math.cos(whisk_ang_world)
81          wy = self.rob_y + self.whisker_length * math.sin(whisk_ang_world)
82          whisker_line = ((self.rob_x,self.rob_y),(wx,wy))
83          hit = any(line_segments_intersect(whisker_line,wall)
84                    for wall in self.env.walls)
85          if hit:
86              self.wall_history.append((self.rob_x, self.rob_y))
87              if self.plotting:
88                  plt.plot([self.rob_x],[self.rob_y],"ro")
89                  plt.draw()
90          return hit
91
92  def line_segments_intersect(linea,lineb):
93      """returns true if the line segments, linea and lineb intersect.
94      A line segment is represented as a pair of points.
95      A point is represented as a (x,y) pair.
96      """
97      ((x0a,y0a),(x1a,y1a)) = linea
98      ((x0b,y0b),(x1b,y1b)) = lineb
99      da, db = x1a-x0a, x1b-x0b
100     ea, eb = y1a-y0a, y1b-y0b
101     denom = db*ea-eb*da
102     if denom==0:  # line segments are parallel
103         return False
104     cb = (da*(y0b-y0a)-ea*(x0b-x0a))/denom # position along line b
105     if cb<0 or cb>1:
106         return False
```

```
107     ca = (db*(y0b-y0a)-eb*(x0b-x0a))/denom # position along line a
108     return 0<=ca<=1
109 
110 # Test cases:
111 # assert line_segments_intersect(((0,0),(1,1)),((1,0),(0,1)))
112 # assert not line_segments_intersect(((0,0),(1,1)),((1,0),(0.6,0.4)))
113 # assert line_segments_intersect(((0,0),(1,1)),((1,0),(0.4,0.6)))
```

2.3.3 Middle Layer

The middle layer acts like both a controller (for the environment layer) and an environment for the upper layer. It has to tell the environment how to steer. Thus it calls $env.do(\cdot)$. It also is told the position to go to and the timeout. Thus it also has to implement $do(\cdot)$.

agentMiddle.py — Middle Layer

```
11 from agents import Environment
12 import math
13 
14 class Rob_middle_layer(Environment):
15     def __init__(self,env):
16         self.env=env
17         self.percepts = env.initial_percepts()
18         self.straight_angle = 11 # angle that is close enough to straight ahead
19         self.close_threshold = 2 # distance that is close enough to arrived
20         self.close_threshold_squared = self.close_threshold**2 # just compute it once
21 
22     def initial_percepts(self):
23         return {}
24 
25     def do(self, action):
26         """action is {'go_to':target_pos,'timeout':timeout}
27         target_pos is (x,y) pair
28         timeout is the number of steps to try
29         returns {'arrived':True} when arrived is true
30             or {'arrived':False} if it reached the timeout
31         """
32         if 'timeout' in action:
33             remaining = action['timeout']
34         else:
35             remaining = -1  # will never reach 0
36         target_pos = action['go_to']
37         arrived = self.close_enough(target_pos)
38         while not arrived and remaining != 0:
39             self.percepts = self.env.do({"steer":self.steer(target_pos)})
40             remaining -= 1
41             arrived = self.close_enough(target_pos)
42         return {'arrived':arrived}
```

This determines how to steer depending on whether the goal is to the right or the left of where the robot is facing.

agentMiddle.py — (continued)

```
44      def steer(self,target_pos):
45          if self.percepts['whisker']:
46              self.display(3,'whisker on', self.percepts)
47              return "left"
48          else:
49              gx,gy = target_pos
50              rx,ry = self.percepts['rob_x_pos'],self.percepts['rob_y_pos']
51              goal_dir = math.acos((gx-rx)/math.sqrt((gx-rx)*(gx-rx)
52                                                     +(gy-ry)*(gy-ry)))*180/math.pi
53              if ry>gy:
54                  goal_dir = -goal_dir
55              goal_from_rob = (goal_dir - self.percepts['rob_dir']+540)%360-180
56              assert -180 < goal_from_rob <= 180
57              if goal_from_rob > self.straight_angle:
58                  return "left"
59              elif goal_from_rob < -self.straight_angle:
60                  return "right"
61              else:
62                  return "straight"
63
64      def close_enough(self,target_pos):
65          gx,gy = target_pos
66          rx,ry = self.percepts['rob_x_pos'],self.percepts['rob_y_pos']
67          return (gx-rx)**2 + (gy-ry)**2 <= self.close_threshold_squared
```

2.3.4 Top Layer

The top layer treats the middle layer as its environment. Note that the top layer is an environment for us to tell it what to visit.

agentTop.py — Top Layer

```
11  from agentMiddle import Rob_middle_layer
12  from agents import Environment
13
14  class Rob_top_layer(Environment):
15      def __init__(self, middle, timeout=200, locations = {'mail':(-5,10),
16                           'o103':(50,10), 'o109':(100,10),'storage':(101,51)} ):
17          """middle is the middle layer
18          timeout is the number of steps the middle layer goes before giving up
19          locations is a loc:pos dictionary
20              where loc is a named location, and pos is an (x,y) position.
21          """
22          self.middle = middle
23          self.timeout = timeout # number of steps before the middle layer should give up
24          self.locations = locations
25
```

```
26      def do(self,plan):
27          """carry out actions.
28          actions is of the form {'visit':list_of_locations}
29          It visits the locations in turn.
30          """
31          to_do = plan['visit']
32          for loc in to_do:
33              position = self.locations[loc]
34              arrived = self.middle.do({'go_to':position, 'timeout':self.timeout})
35              self.display(1,"Arrived at",loc,arrived)
```

2.3.5 Plotting

The following is used to plot the locations, the walls and (eventually) the movement of the robot. It can either plot the movement if the robot as it is going (with the default *env.plotting* = *True*), or not plot it as it is going (setting *env.plotting* = *False*; in this case the trace can be plotted using *pl.plot_run*()).

agentTop.py — (continued)

```
37  import matplotlib.pyplot as plt
38
39  class Plot_env(object):
40      def __init__(self, body,top):
41          """sets up the plot
42          """
43          self.body = body
44          plt.ion()
45          plt.clf()
46          plt.axes().set_aspect('equal')
47          for wall in body.env.walls:
48              ((x0,y0),(x1,y1)) = wall
49              plt.plot([x0,x1],[y0,y1],"-k",linewidth=3)
50          for loc in top.locations:
51              (x,y) = top.locations[loc]
52              plt.plot([x],[y],"k<")
53              plt.text(x+1.0,y+0.5,loc) # print the label above and to the right
54          plt.plot([body.rob_x],[body.rob_y],"go")
55          plt.draw()
56
57      def plot_run(self):
58          """plots the history after the agent has finished.
59          This is typically only used if body.plotting==False
60          """
61          xs,ys = zip(*self.body.history)
62          plt.plot(xs,ys,"go")
63          wxs,wys = zip(*self.body.wall_history)
64          plt.plot(wxs,wys,"ro")
65          #plt.draw()
```

The following code plots the agent as it acts in the world:

agentTop.py — (continued)

```
67 from agentEnv import Rob_body, Rob_env
68
69 env = Rob_env({((20,0),(30,20)), ((70,-5),(70,25))})
70 body = Rob_body(env)
71 middle = Rob_middle_layer(body)
72 top = Rob_top_layer(middle)
73
74 # try:
75 # pl=Plot_env(body,top)
76 # top.do({'visit':['o109','storage','o109','o103']})
77 # You can directly control the middle layer:
78 # middle.do({'go_to':(30,-10), 'timeout':200})
79 # Can you make it crash?
```

Exercise 2.1 The following code implements a robot trap. Write a controller that can escape the "trap" and get to the goal. See textbook for hints.

agentTop.py — (continued)

```
81 # Robot Trap for which the current controller cannot escape:
82 trap_env = Rob_env({((10,-21),(10,0)), ((10,10),(10,31)), ((30,-10),(30,0)),
83                   ((30,10),(30,20)), ((50,-21),(50,31)), ((10,-21),(50,-21)),
84                   ((10,0),(30,0)), ((10,10),(30,10)), ((10,31),(50,31))})
85 trap_body = Rob_body(trap_env,init_pos=(-1,0,90))
86 trap_middle = Rob_middle_layer(trap_body)
87 trap_top = Rob_top_layer(trap_middle,locations={'goal':(71,0)})
88
89 # Robot trap exercise:
90 # pl=Plot_env(trap_body,trap_top)
91 # trap_top.do({'visit':['goal']})
```