# Homework 3: Markov Localization

Don Miner

May 1, 2007

This document describes the results of implementing a robot that performs Markov Localization in a small world (Figure 1). The robot can move North, South, East and West and has sensors that detects obstacles in each of these directions.

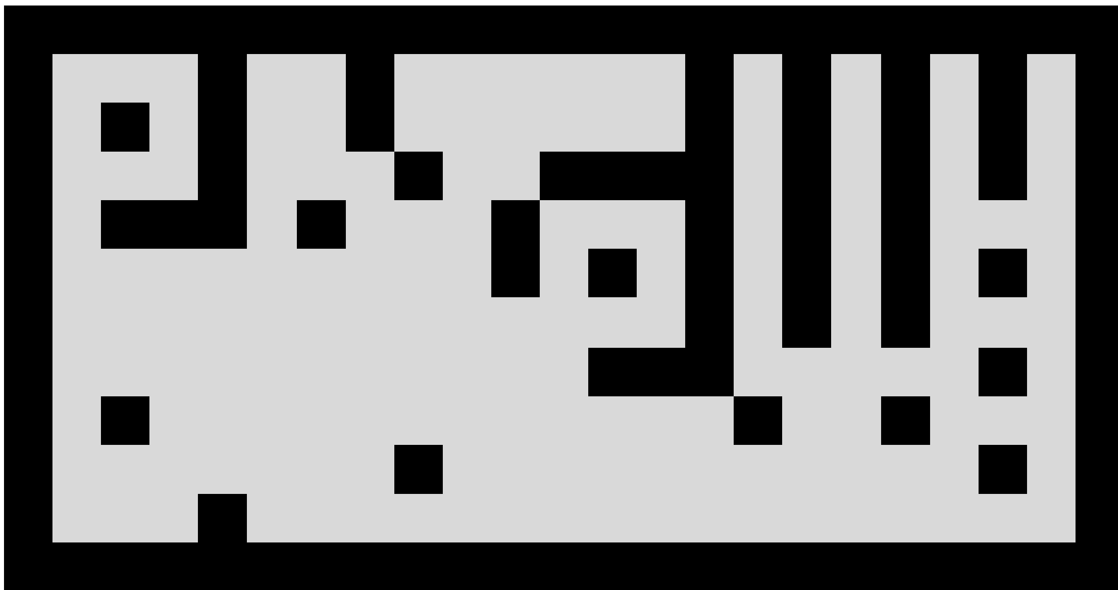


Figure 1: The world our robot inhabits.

This complete work can be downloaded and viewed at:
`http://www.coral-lab.org/~don/projects/markov/`

# 1 Sample Run

The following figures show a sample run in our environment using used $P = .15$ and $Q = .15$. Each image is an iteration of a `SEE` and `ACT` operation. The darkest spots are the walls and the lighter spots are the open areas. The grey areas demonstrate the probability that the robot is at that location. A darker area denotes a higher probability for that location.

A video of this run is available at:
`http://www.coral-lab.org/~don/projects/markov/markov1.mov`

The robot in this run starts at the bottom right corner of the top left room. At iteration 3, the robot has just moved left. Notice how in Figure 2 the distributions are strong where there are obstacles to the North and to the South. Also notice in Figure 2 that there is higher probability that the robot is in the bottom row. This is because the robot has moved South and if the robot was already on the South wall, it would have stayed put.

Figure 2: Iteration 3

In Figures 3 and 4, the robot continues to move South and the probability is getting stronger.

Figure 3: Iteration 5

Figure 4: Iteration 7

In the middle of the simulation, the robot is moved around the center of an open area. Since `SEE` returns "no obstacle" in all four directions in this open space, the robot must depend on its odometer. This is what makes the distribution spread out in Figures 5 and 6.

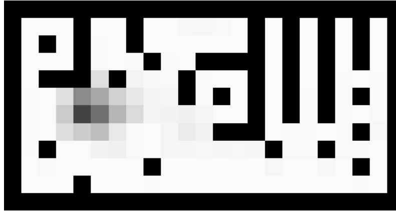


Figure 5: Iteration 16

Figure 6: Iteration 19

Then, the robot moves towards the South. Notice how the probability converges against the wall.

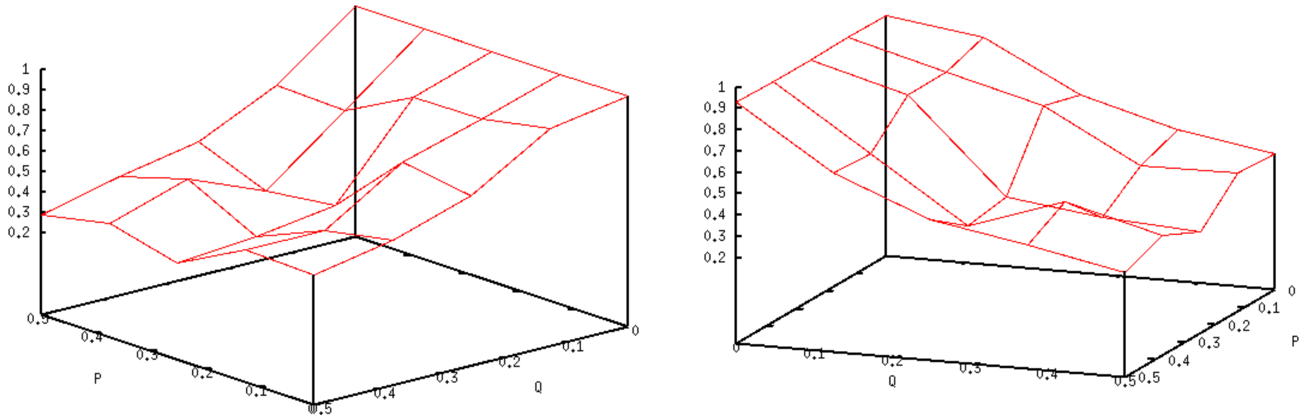
Figure 7: Iteration 23

# 2 Experiments

I performed 25 `ACT` and 25 `SEE` in sequence for different values of $P$ and $Q$. The scoring algorithm is a crude estimate of convergence: it simply sums the top five probabilities in the map. This may not be the best way to numerically describe convergence but the results are interesting and meaningful. The following chart and graphs show the results:

SCORES

| P \ Q | 0.000 | 0.125 | 0.250 | 0.375 | 0.500 |
|---|---|---|---|---|---|
| 0.000 | 0.928 | 0.864 | 0.631 | 0.512 | 0.434 |
| 0.125 | 0.925 | 0.804 | 0.684 | 0.446 | 0.450 |
| 0.250 | 0.921 | 0.797 | 0.362 | 0.306 | 0.273 |
| 0.375 | 0.925 | 0.624 | 0.325 | 0.477 | 0.358 |
| 0.500 | 0.929 | 0.636 | 0.456 | 0.378 | 0.290 |




Analyzing this data gives a few interesting conclusions:

- As expected, the score is low when $Q$ is high and/or $P$ is high.
- Having perfect sensors ($Q = 0$) yields the same performance no matter what $P$ is. The probable position in all these cases converged to the correct spot after only a few steps. Then, even if the odometer was completely off, it still quickly recovered every time step.
- It appears that $Q$ has more of an effect on the score than $P$.
- Some of the higher scores (e.g. $P = 0.375, Q = .375, Score = .477$) are due to an incorrect convergence. I am sure with more test cases these outliers would be flattened out. This case points out that faulty convergence is a serious problem when using bad sensors and bad odometers.

# 3 Code

The robot and the environment were implemented in Python.

```
#!/usr/bin/env python


from sys import argv
from random import random
from random import shuffle
# Enumerate directions
NORTH = 0
SOUTH = 1
WEST = 2
EAST = 3
DIRECTIONS = [NORTH, SOUTH, WEST, EAST]
DIRECTIONS_TEXT = {'n' : NORTH, 's' : SOUTH, 'w' : WEST, 'e' : EAST}

def format_prob(num):
   if num < .001:
      return '.000'

   return str(num)[1:5].ljust(4,'0')



class Robot:
   def __init__(self, motor_noise, sensor_noise, map_path, initial_position,  end_location):
      self.motor_noise = motor_noise
      self.sensor_noise = sensor_noise
      self.map = self.load_map(map_path)
      self.map_rows = len(self.map)
      self.map_cols = len(self.map[0])
      self.num_free_squares = self.count_free_squares()

      self.prob_map =  self.init_probability_map(1.0 / self.num_free_squares)
      self.path_map = self.init_path_map(end_location)

      self.position = initial_position[:]
      self.end_loc = end_location[:]

      self.cur_img = 0

   def prob_ACT_SEE_loop(self):
      self.prob_ACT()
      self.SEE()

      self.write_next_img()

      if self.position == self.end_loc:
         return True
      else:
         return False

   # updates the map
   def SEE(self):
      robby.check_map()

      sensor_readings = self.sense()

      new_map = self.init_probability_map(0)

      # used to normalize data
      total_prob = 0
      for row in range(self.map_rows):
         for col in range(self.map_cols):
            # based on my sensor readings, chances i am actually here

            hypothetical_readings = self.sense(perfect=True, position=[row,col])

            num_right = sum([ (sensor_readings[dir] == hypothetical_readings[dir] ) \
               for dir in range(4)])

            # p(l|i) = p(i|l) * p(l) / p(i)
            # p(l|i) is the new probability
            # p(i|l) is the probability we are getting this reading at this position
               # (1-Q)^4 - chance that all 4 sensors are right
               # 4 * (1-Q)^3 * Q - chance that 3 sensors are right
               # 6 * (1-Q)^2 * Q^2 - chance that 2 sensors are right
               # 4 * (1-Q) * Q^3 - chance that 1 sensor is right
               # Q^4 - chance that all 4 sensors are wrong

            # p(l) is the probability we are at this location
            # p(i) is the probability we got this sensor reading (always .5)

            old_prob = self.prob_map[row][col] # p(l)
            new_prob = self.prob_map[row][col]

            # p(i|l)
            if num_right == 4:
               new_prob *= (1-self.sensor_noise)**4
```

```
            elif num_right == 3:
                new_prob *= 4 * (1-self.sensor_noise)**3 * self.sensor_noise
            elif num_right == 2:
                new_prob *= 6 * (1-self.sensor_noise)**2 * self.sensor_noise**2
            elif num_right == 1:
                new_prob *= 4 * (1-self.sensor_noise) * self.sensor_noise**3
            elif num_right == 0:
                new_prob *= self.sensor_noise**4

            new_map[row][col] = new_prob

            total_prob += new_prob

    # Normalize data (p(i))
    for row in range(self.map_rows):
        for col in range(self.map_cols):
            new_map[row][col] /= total_prob

    self.prob_map = new_map

    robby.check_map()


def ACT(self, direction):

    robby.check_map()

    if direction not in DIRECTIONS:
        raise ValueError, 'invalid direction'

    self.move(direction)

    new_map = self.init_probability_map(0)

    for row in range(self.map_rows):
        for col in range(self.map_cols):
            # if I moved from here...
            # chances are I would be...

            # if there is as obstacle to the blocking us, we aint movin
            if self.sense(direction, True, [row, col]):
                new_map[row][col] += self.prob_map[row][col]
                continue

            if direction == NORTH:
                # if there is an obstacle two squares infront of us,
                # we will definitely move only 1
                if self.sense(direction, True, [row - 1, col]):
                    new_map[row - 1][col] += self.prob_map[row][col]

                # otherwise, we could move one square or two
                else:
                    new_map[row - 1][col] += self.prob_map[row][col] * (1 - self.motor_noise)

                    new_map[row - 2][col] += self.prob_map[row][col] * self.motor_noise

            if direction == SOUTH:
                # if there is an obstacle two squares infront of us,
                # we will definitely move only 1
                if self.sense(direction, True, [row + 1, col]):
                    new_map[row + 1][col] += self.prob_map[row][col]

                # otherwise, we could move one square or two
                else:
                    new_map[row + 1][col] += self.prob_map[row][col] * (1 - self.motor_noise)

                    new_map[row + 2][col] += self.prob_map[row][col] * self.motor_noise

            if direction == WEST:
                # if there is an obstacle two squares infront of us,
                # we will definitely move only 1
                if self.sense(direction, True, [row, col - 1]):
                    new_map[row][col - 1] += self.prob_map[row][col]

                # otherwise, we could move one square or two
                else:
                    new_map[row][col - 1] += self.prob_map[row][col] * (1 - self.motor_noise)

                    new_map[row][col - 2] += self.prob_map[row][col] * self.motor_noise

            if direction == EAST:
                # if there is an obstacle two squares infront of us,
                # we will definitely move only 1
                if self.sense(direction, True, [row, col + 1]):
                    new_map[row][col + 1] += self.prob_map[row][col]

                # otherwise, we could move one square or two
                else:
                    new_map[row][col + 1] += self.prob_map[row][col] * (1 - self.motor_noise)

                    new_map[row][col + 2] += self.prob_map[row][col] * self.motor_noise
```

```
    self.prob_map = new_map

    robby.check_map()


def count_free_squares(self):
    count = 0
    for row in range(self.map_rows):
        for col in range(self.map_cols):
            if self.map[row][col] == False: count += 1

    return count

def init_probability_map(self, weight):
    out_map = []
    for row in range(self.map_rows):
        out_map.append([])
        for col in range(self.map_cols):
            if not self.map[row][col] == True:
                out_map[row].append(weight)
            else:
                out_map[row].append(0)
    return out_map

def init_path_map(self, end_loc):
    LARGE = 64000

    out_map = []
    for row in range(self.map_rows):
        out_map.append([])
        for col in range(self.map_cols):
            out_map[row].append(LARGE)

    Q = [(0, end_loc)]

    while(len(Q) > 0):
        value, pos = Q[0]
        Q = Q[1:]

        # check for negative index
        if pos[0] < 0 or pos[1] < 0:
            continue

        # check for out of bounds index
        try:
            out_map[pos[0]][pos[1]]
        except IndexError:
            continue

        # check to see this node has not been changed
        if out_map[pos[0]][pos[1]] != LARGE:
            continue

        # don't care if this guy is a wall
        if self.is_obstacle(pos):
            continue

        # set this spot to the current value
        out_map[pos[0]][pos[1]] = value

        # enqueue adjacent nodes
        Q.append((value + 1, [pos[0] + 1, pos[1]] ))
        Q.append((value + 1, [pos[0] - 1, pos[1]] ))
        Q.append((value + 1, [pos[0], pos[1] + 1] ))
        Q.append((value + 1, [pos[0], pos[1] - 1] ))

    return out_map

def load_map(self, file_path):
    return [ [ bool(int(val)) for val in row.split() ] \
        for row in open(file_path, 'r').readlines() ]

def move(self, direction, second=False):
    if direction not in DIRECTIONS:
        raise ValueError, 'invalid direction'

    if direction == NORTH and not self.sense(NORTH, perfect=True):
        self.position[0] -= 1
    elif direction == SOUTH and not self.sense(SOUTH, perfect=True):
        self.position[0] += 1
    elif direction == WEST and not self.sense(WEST, perfect=True):
        self.position[1] -= 1
    elif direction == EAST and not self.sense(EAST, perfect=True):
        self.position[1] += 1

    if not second and random() < self.motor_noise:
        self.move(direction, True)

def prob_ACT(self):
    prob = [0, 0, 0, 0]

    for row in range(self.map_rows):
        for col in range(self.map_cols):
```

```
         # assume border is lined with walls
         if self.is_on_border([row,col]):
            continue

         # what direction SHOULD i move from here?
         results = []
         for dir in DIRECTIONS:
            if dir == NORTH:
               results.append((self.path_map[row - 1][col], dir))
            elif dir == SOUTH:
               results.append((self.path_map[row + 1][col], dir))
            elif dir == WEST:
               results.append((self.path_map[row][col - 1], dir))
            elif dir == EAST:
               results.append((self.path_map[row][col + 1], dir))

         shuffle(results)
         results.sort()

         prob[results[0][1]] += self.prob_map[row][col]

   self.ACT(prob.index(max(prob)))


def sense(self, direction=None, perfect=False, position=None):
   if direction == None:
      return [ self.sense(dir, perfect, position) for dir in DIRECTIONS ]

   if direction not in DIRECTIONS:
      raise ValueError, 'invalid direction'

   loc = self.position
   if position:
      loc = position

   try:
      if direction == NORTH:
         reading = self.map[loc[0] - 1][loc[1]]
      elif direction == SOUTH:
         reading = self.map[loc[0] + 1][loc[1]]
      elif direction == WEST:
         reading = self.map[loc[0]][loc[1] - 1]
      elif direction == EAST:
         reading = self.map[loc[0]][loc[1] + 1]
   except IndexError:
      return True

   if not perfect and random() < self.sensor_noise:
      reading = not reading

   return reading

def is_on_border(self, position):
   if position[0] in [0, self.map_rows-1] or position[1] in [0, self.map_cols-1]:
      return True
   return False

def is_obstacle(self, position):
   return self.map[position[0]][position[1]]

def check_map(self):
   total_prob = 1
   for row in range(self.map_rows):
      for col in range(self.map_cols):
         total_prob -= self.prob_map[row][col]

   if total_prob > .0001 or total_prob < -.0001:
      print 'Warning! inconsistent map?', total_prob

def __str__(self):
   out_str = ""

   for row in range(self.map_rows):
      for col in range(self.map_cols):
         if self.position == [row, col]:
            out_str += '  X  '
         else:
            out_str += '  ' + str(int(self.map[row][col])) + '  '

      out_str += '\n'


   out_str += '\n'

   for row in range(self.map_rows):
      for col in range(self.map_cols):
         if self.map[row][col]:
            out_str += '**** '
         else:
            out_str +=  format_prob(self.prob_map[row][col]) + ' '

      out_str += '\n'
```

```
      return out_str

   def write_next_img(self):
      self.write_img("output" + str(self.cur_img) + ".pgm")
      self.cur_img += 1

   def write_img(self, img_file_path):
      out_file = open(img_file_path, 'w')

      GRID_SIZE = 18

      header = \
"""P5
#LEAD Technologies Inc. V1.01
""" + str(self.map_cols * GRID_SIZE) + " " + str(self.map_rows * GRID_SIZE) + """
255
"""
      out_file.write(header)

      for row in range(self.map_rows):
         for i in range(GRID_SIZE):
            for col in range(self.map_cols):
               for j in range(GRID_SIZE):
                  if self.map[row][col]:
                     out_file.write(chr(0))
                  else:
                     out_file.write(chr(int(min(250, \
                        (1.0001 - self.prob_map[row][col]**.25) * 250)) ))

      out_file.close()

robby = Robot(float(raw_input("motor noise (0.0-0.99): ")), \
 float(raw_input("sensor noise (0.0-0.99): ")), \
 raw_input("map file: "), \
 [ int(raw_input("initial row: ")), int(raw_input("initial column: ")) ],
 [ int(raw_input("final row: ")), int(raw_input("final column: ")) ])

while True:
   print robby

   command = raw_input("> ").strip()

   if not command: continue

   command = command.split()

   if command[0] in ["move", "mv", "m"]:
      robby.ACT(DIRECTIONS_TEXT[command[1]])

   elif command[0] in DIRECTIONS_TEXT.keys():
      robby.ACT(DIRECTIONS_TEXT[command[0]])

   elif command[0] in ["see", "look", "l", "."]:
      robby.SEE()

   elif command[0] in ["i", "image"]:
      robby.write_img(command[1])

   elif command[0] in ["q", "quit", "exit"]:
      break

   elif command[0] in ["p", "prob"]:
      robby.prob_ACT()

   elif command[0] in ["loop"]:
      while(not robby.prob_ACT_SEE_loop()): pass
```