Robot Localization Python Implementation CIS 479 Project 2 Report

Team: Paul Murariu and Dominic Baughman

Date: 11/5/2023

8:05 PM
11/5/2023

Initial Probabilities

(0,0): 2.50%	(0,1): 2.50%	(0,2): 2.50%	(0,3): 2.50%	(0,4): 2.50%	(0,5): 2.50%	(0,6): 2.50%
(1,0): 2.50%	(1,1): 0.00%	(1,2): 2.50%	(1,3): 0.00%	(1,4): 2.50%	(1,5): 0.00%	(1,6): 2.50%
(2,0): 2.50%	(2,1): 2.50%	(2,2): 2.50%	(2,3): 2.50%	(2,4): 2.50%	(2,5): 0.00%	(2,6): 2.50%
(3,0): 2.50%	(3,1): 2.50%	(3,2): 2.50%	(3,3): 0.00%	(3,4): 2.50%	(3,5): 2.50%	(3,6): 2.50%
(4,0): 2.50%	(4,1): 0.00%	(4,2): 2.50%	(4,3): 2.50%	(4,4): 2.50%	(4,5): 2.50%	(4,6): 2.50%
(5,0): 2.50%	(5,1): 0.00%	(5,2): 2.50%	(5,3): 0.00%	(5,4): 2.50%	(5,5): 0.00%	(5,6): 2.50%
(6,0): 2.50%	(6,1): 2.50%	(6,2): 2.50%	(6,3): 2.50%	(6,4): 2.50%	(6,5): 2.50%	(6,6): 2.50%

8:07 PM
11/5/2023

Filtering after Evidence [0, 1, 0, 0]

maze after action

(0,0): 2.01%	(0,1): 2.01%	(0,2): 19.08%	(0,3): 2.01%	(0,4): 19.08%	(0,5): 2.01%	(0,6): 2.01%
(1,0): 0.01%	(1,1): 0.00%	(1,2): 0.01%	(1,3): 0.00%	(1,4): 0.01%	(1,5): 0.00%	(1,6): 0.01%
(2,0): 0.11%	(2,1): 19.08%	(2,2): 1.06%	(2,3): 2.01%	(2,4): 0.11%	(2,5): 0.00%	(2,6): 0.01%
(3,0): 0.11%	(3,1): 0.11%	(3,2): 0.11%	(3,3): 0.00%	(3,4): 0.11%	(3,5): 19.08%	(3,6): 0.11%
(4,0): 0.01%	(4,1): 0.00%	(4,2): 0.11%	(4,3): 2.01%	(4,4): 1.06%	(4,5): 0.11%	(4,6): 0.11%
(5,0): 0.01%	(5,1): 0.00%	(5,2): 0.01%	(5,3): 0.00%	(5,4): 0.01%	(5,5): 0.00%	(5,6): 0.01%
(6,0): 0.01%	(6,1): 2.01%	(6,2): 0.11%	(6,3): 2.01%	(6,4): 0.11%	(6,5): 2.01%	(6,6): 0.01%

8:08 PM
11/5/2023

Prediction after Action E

maze after action

(0,0): 2.01%	(0,1): 2.01%	(0,2): 19.08%	(0,3): 2.01%	(0,4): 19.08%	(0,5): 2.01%	(0,6): 2.01%
(1,0): 0.01%	(1,1): 0.00%	(1,2): 0.01%	(1,3): 0.00%	(1,4): 0.01%	(1,5): 0.00%	(1,6): 0.01%
(2,0): 0.11%	(2,1): 19.08%	(2,2): 1.06%	(2,3): 2.01%	(2,4): 0.11%	(2,5): 0.00%	(2,6): 0.01%
(3,0): 0.11%	(3,1): 0.11%	(3,2): 0.11%	(3,3): 0.00%	(3,4): 0.11%	(3,5): 19.08%	(3,6): 0.11%
(4,0): 0.01%	(4,1): 0.00%	(4,2): 0.11%	(4,3): 2.01%	(4,4): 1.06%	(4,5): 0.11%	(4,6): 0.11%
(5,0): 0.01%	(5,1): 0.00%	(5,2): 0.01%	(5,3): 0.00%	(5,4): 0.01%	(5,5): 0.00%	(5,6): 0.01%
(6,0): 0.01%	(6,1): 2.01%	(6,2): 0.11%	(6,3): 2.01%	(6,4): 0.11%	(6,5): 2.01%	(6,6): 0.01%

8:12 PM
11/5/2023

Filtering after Evidence [0, 0, 0, 0]

maze after action

(0,0): 2.01%	(0,1): 2.01%	(0,2): 19.08%	(0,3): 2.01%	(0,4): 19.08%	(0,5): 2.01%	(0,6): 2.01%
(1,0): 0.01%	(1,1): 0.00%	(1,2): 0.01%	(1,3): 0.00%	(1,4): 0.01%	(1,5): 0.00%	(1,6): 0.01%
(2,0): 0.11%	(2,1): 19.08%	(2,2): 1.06%	(2,3): 2.01%	(2,4): 0.11%	(2,5): 0.00%	(2,6): 0.01%
(3,0): 0.11%	(3,1): 0.11%	(3,2): 0.11%	(3,3): 0.00%	(3,4): 0.11%	(3,5): 19.08%	(3,6): 0.11%
(4,0): 0.01%	(4,1): 0.00%	(4,2): 0.11%	(4,3): 2.01%	(4,4): 1.06%	(4,5): 0.11%	(4,6): 0.11%
(5,0): 0.01%	(5,1): 0.00%	(5,2): 0.01%	(5,3): 0.00%	(5,4): 0.01%	(5,5): 0.00%	(5,6): 0.01%
(6,0): 0.01%	(6,1): 2.01%	(6,2): 0.11%	(6,3): 2.01%	(6,4): 0.11%	(6,5): 2.01%	(6,6): 0.01%

8:13 PM
11/5/2023

Prediction after Action N

maze after action

(0,0): 2.01%	(0,1): 2.01%	(0,2): 19.08%	(0,3): 2.01%	(0,4): 19.08%	(0,5): 2.01%	(0,6): 2.01%
(1,0): 0.01%	(1,1): 0.00%	(1,2): 0.01%	(1,3): 0.00%	(1,4): 0.01%	(1,5): 0.00%	(1,6): 0.01%
(2,0): 0.11%	(2,1): 19.08%	(2,2): 1.06%	(2,3): 2.01%	(2,4): 0.11%	(2,5): 0.00%	(2,6): 0.01%
(3,0): 0.11%	(3,1): 0.11%	(3,2): 0.11%	(3,3): 0.00%	(3,4): 0.11%	(3,5): 19.08%	(3,6): 0.11%
(4,0): 0.01%	(4,1): 0.00%	(4,2): 0.11%	(4,3): 2.01%	(4,4): 1.06%	(4,5): 0.11%	(4,6): 0.11%
(5,0): 0.01%	(5,1): 0.00%	(5,2): 0.01%	(5,3): 0.00%	(5,4): 0.01%	(5,5): 0.00%	(5,6): 0.01%
(6,0): 0.01%	(6,1): 2.01%	(6,2): 0.11%	(6,3): 2.01%	(6,4): 0.11%	(6,5): 2.01%	(6,6): 0.01%

Filtering after Evidence [1, 0, 0, 1]

8:14 PM
11/5/2023

```
maze after action
(0,0): 2.01%   (0,1): 2.01%   (0,2): 19.08%   (0,3): 2.01%   (0,4): 19.08%   (0,5): 2.01%   (0,6): 2.01%
(1,0): 0.01%   (1,1): 0.00%   (1,2): 0.01%   (1,3): 0.00%   (1,4): 0.01%   (1,5): 0.00%   (1,6): 0.01%
(2,0): 0.11%   (2,1): 19.08%   (2,2): 1.06%   (2,3): 2.01%   (2,4): 0.11%   (2,5): 0.00%   (2,6): 0.01%
(3,0): 0.11%   (3,1): 0.11%   (3,2): 0.11%   (3,3): 0.00%   (3,4): 0.11%   (3,5): 19.08%   (3,6): 0.11%
(4,0): 0.01%   (4,1): 0.00%   (4,2): 0.11%   (4,3): 2.01%   (4,4): 1.06%   (4,5): 0.11%   (4,6): 0.11%
(5,0): 0.01%   (5,1): 0.00%   (5,2): 0.01%   (5,3): 0.00%   (5,4): 0.01%   (5,5): 0.00%   (5,6): 0.01%
(6,0): 0.01%   (6,1): 2.01%   (6,2): 0.11%   (6,3): 2.01%   (6,4): 0.11%   (6,5): 2.01%   (6,6): 0.01%
```

Prediction after Action N

8:15 PM
11/5/2023

```
maze after action
(0,0): 2.01%   (0,1): 2.01%   (0,2): 19.08%   (0,3): 2.01%   (0,4): 19.08%   (0,5): 2.01%   (0,6): 2.01%
(1,0): 0.01%   (1,1): 0.00%   (1,2): 0.01%   (1,3): 0.00%   (1,4): 0.01%   (1,5): 0.00%   (1,6): 0.01%
(2,0): 0.11%   (2,1): 19.08%   (2,2): 1.06%   (2,3): 2.01%   (2,4): 0.11%   (2,5): 0.00%   (2,6): 0.01%
(3,0): 0.11%   (3,1): 0.11%   (3,2): 0.11%   (3,3): 0.00%   (3,4): 0.11%   (3,5): 19.08%   (3,6): 0.11%
(4,0): 0.01%   (4,1): 0.00%   (4,2): 0.11%   (4,3): 2.01%   (4,4): 1.06%   (4,5): 0.11%   (4,6): 0.11%
(5,0): 0.01%   (5,1): 0.00%   (5,2): 0.01%   (5,3): 0.00%   (5,4): 0.01%   (5,5): 0.00%   (5,6): 0.01%
(6,0): 0.01%   (6,1): 2.01%   (6,2): 0.11%   (6,3): 2.01%   (6,4): 0.11%   (6,5): 2.01%   (6,6): 0.01%
```

Filtering after Evidence [0, 0, 1, 0]

8:16 PM
11/5/2023

```
maze after action
(0,0): 2.01%   (0,1): 2.01%   (0,2): 19.08%   (0,3): 2.01%   (0,4): 19.08%   (0,5): 2.01%   (0,6): 2.01%
(1,0): 0.01%   (1,1): 0.00%   (1,2): 0.01%   (1,3): 0.00%   (1,4): 0.01%   (1,5): 0.00%   (1,6): 0.01%
(2,0): 0.11%   (2,1): 19.08%   (2,2): 1.06%   (2,3): 2.01%   (2,4): 0.11%   (2,5): 0.00%   (2,6): 0.01%
(3,0): 0.11%   (3,1): 0.11%   (3,2): 0.11%   (3,3): 0.00%   (3,4): 0.11%   (3,5): 19.08%   (3,6): 0.11%
(4,0): 0.01%   (4,1): 0.00%   (4,2): 0.11%   (4,3): 2.01%   (4,4): 1.06%   (4,5): 0.11%   (4,6): 0.11%
(5,0): 0.01%   (5,1): 0.00%   (5,2): 0.01%   (5,3): 0.00%   (5,4): 0.01%   (5,5): 0.00%   (5,6): 0.01%
(6,0): 0.01%   (6,1): 2.01%   (6,2): 0.11%   (6,3): 2.01%   (6,4): 0.11%   (6,5): 2.01%   (6,6): 0.01%
```

Prediction after Action W

8:17 PM
11/5/2023

```
maze after action
(0,0): 2.01%   (0,1): 2.01%   (0,2): 19.08%   (0,3): 2.01%   (0,4): 19.08%   (0,5): 2.01%   (0,6): 2.01%
(1,0): 0.01%   (1,1): 0.00%   (1,2): 0.01%   (1,3): 0.00%   (1,4): 0.01%   (1,5): 0.00%   (1,6): 0.01%
(2,0): 0.11%   (2,1): 19.08%   (2,2): 1.06%   (2,3): 2.01%   (2,4): 0.11%   (2,5): 0.00%   (2,6): 0.01%
(3,0): 0.11%   (3,1): 0.11%   (3,2): 0.11%   (3,3): 0.00%   (3,4): 0.11%   (3,5): 19.08%   (3,6): 0.11%
(4,0): 0.01%   (4,1): 0.00%   (4,2): 0.11%   (4,3): 2.01%   (4,4): 1.06%   (4,5): 0.11%   (4,6): 0.11%
(5,0): 0.01%   (5,1): 0.00%   (5,2): 0.01%   (5,3): 0.00%   (5,4): 0.01%   (5,5): 0.00%   (5,6): 0.01%
(6,0): 0.01%   (6,1): 2.01%   (6,2): 0.11%   (6,3): 2.01%   (6,4): 0.11%   (6,5): 2.01%   (6,6): 0.01%
```

Filtering after Evidence [0, 1, 0, 1]

8:18 PM
11/5/2023

```
maze after action
(0,0): 2.01%   (0,1): 2.01%   (0,2): 19.08%   (0,3): 2.01%   (0,4): 19.08%   (0,5): 2.01%   (0,6): 2.01%
(1,0): 0.01%   (1,1): 0.00%   (1,2): 0.01%   (1,3): 0.00%   (1,4): 0.01%   (1,5): 0.00%   (1,6): 0.01%
(2,0): 0.11%   (2,1): 19.08%   (2,2): 1.06%   (2,3): 2.01%   (2,4): 0.11%   (2,5): 0.00%   (2,6): 0.01%
(3,0): 0.11%   (3,1): 0.11%   (3,2): 0.11%   (3,3): 0.00%   (3,4): 0.11%   (3,5): 19.08%   (3,6): 0.11%
(4,0): 0.01%   (4,1): 0.00%   (4,2): 0.11%   (4,3): 2.01%   (4,4): 1.06%   (4,5): 0.11%   (4,6): 0.11%
(5,0): 0.01%   (5,1): 0.00%   (5,2): 0.01%   (5,3): 0.00%   (5,4): 0.01%   (5,5): 0.00%   (5,6): 0.01%
(6,0): 0.01%   (6,1): 2.01%   (6,2): 0.11%   (6,3): 2.01%   (6,4): 0.11%   (6,5): 2.01%   (6,6): 0.01%
```

Programmed by pmurariu and baughboy

SOURCE CODE

```
#Robot Localization done by pmurariu and baughboy
# [W,N,E,S]
# Correct Obstacle 0.90
# Correct Open Space 0.95
# Incorrect Open Space 0.10
# Incorrect Obstacle 0.05
import numpy as nump
maze = [
    [0,0,0,0,0,0,0],
    [0,1,0,1,0,1,0],
    [0,0,0,0,0,1,0],
    [0,0,0,1,0,0,0],
    [0,1,0,0,0,0,0],
    [0,1,0,1,0,1,0],
    [0,0,0,0,0,0,0],
    ]

#defining sensing probabiities
correct_obstacle = 0.90
correct_open_space = 0.95
incorrect_open_space = 0.10
incorrect_obstacle = 0.05

#defining the possible drift \
possible_drift = {
    'straight': 0.75,
    'left': 0.15,
    'right' : 0.10
    }

#making copy of the maze with location probabilties
rows, cols = len(maze), len(maze[0])
initial_probability = 0.025
```

```
prob_maze = nump.full((rows, cols), initial_probability)

def prediction(distance, action):
    new_distance = nump.zeros((7, 7), nump.float64) #array of zeroes
    for spaces in range(rows): #iterating
        for (state, prob) in transitional_prob(spaces, action):
#iterate though spaces we can travel to
            #add on term for total probability
            new_distance[state[0], state[1]] += prob *
distance[spaces[0], spaces[1]]
    return new_distance #update distribution

def transitional_prob(state, action):
    #go in intended direction
    drift_straight = transition(state, action)

    #left drift
    drift_left = transition(state, (action - 1) % 4)

    #drift right
    drift_right = transition(state, (action + 1) % 4)

    # return the 3 directions
    return ((drift_straight, possible_drift),
            (drift_left, possible_drift),
            (drift_right, possible_drift))

def transition(curr_state, action):
    global maze
    if (action == 0): #west
        new_location = (curr_state[0], curr_state[1] - 1)
    elif (action == 1): #north
        new_location = (curr_state[0] - 1, curr_state[1])
    elif (action == 2): #east
        new_location = (curr_state[0], curr_state[1] + 1)
    elif (action == 3): #south
        new_location = (curr_state[0] + 1, curr_state[1])

    if (new_location in maze or  # moving into obstacles
```

```
        new_location[0] < 0 or new_location[0] >= 7 or  #above or
below maze
        new_location[1] < 0 or new_location[1] >= 7):  #left or
right of maze
        return (curr_state[0], curr_state[1])  #stay where you are
    else:
        return new_location

#handles all of the moving and updating the prob maze after
def moving (move_direction):
    global prob_maze
    temp_prob_maze = nump.zeros_like(prob_maze)
    for row in range(rows):
        for col in range(cols):
            if maze[row][col] == 1:
                temp_prob_maze [row][col] = prob_maze[row][col]
            else:
                for row_change, col_change in [(0, 1), (1, 0), (0,
-1), (-1, 0)]:
                    #finding the new positions
                    new_pos_row, new_pos_col = row + row_change, col
+ col_change
                    #checking to see if the robot will bounce or not
(not moving is the same as bouncing for all intents and purposes)
                    if 0 <= new_pos_row < rows and 0 <= new_pos_col
< cols:
                        #updating probabilties and checking for
drift
                        if move_direction == 'E':
                            #checking for straight move
                            temp_prob_maze[new_pos_row][new_pos_col]
+= prob_maze[row][col] * possible_drift['straight']
                            #checking for the left drifft
                            left_drift_row = row - row_change
                            left_drift_col = col - col_change

temp_prob_maze[left_drift_row][left_drift_col] +=
prob_maze[row][col] * possible_drift['left']
                            #checking for the right drift
                            right_drift_row = row + row_change
```

```
                            right_drift_col = col + row_change

temp_prob_maze[right_drift_row][right_drift_col] +=
prob_maze[row][col] * possible_drift['right']
                        if move_direction == 'N':
                            #checking for straight move
                            temp_prob_maze[new_pos_row][new_pos_col]
+= prob_maze[row][col] * possible_drift['straight']
                            #checking for the left drifft
                            left_drift_row = row - row_change
                            left_drift_col = col - col_change

temp_prob_maze[left_drift_row][left_drift_col] +=
prob_maze[row][col] * possible_drift['left']
                            #checking for the right drift
                            right_drift_row = row + row_change
                            right_drift_col = col + row_change

temp_prob_maze[right_drift_row][right_drift_col] +=
prob_maze[row][col] * possible_drift['right']
                        if move_direction == 'W':
                            #checking for straight move
                            temp_prob_maze[new_pos_row][new_pos_col]
+= prob_maze[row][col] * possible_drift['straight']
                            #checking for the left drifft
                            left_drift_row = row - row_change
                            left_drift_col = col - col_change

temp_prob_maze[left_drift_row][left_drift_col] +=
prob_maze[row][col] * possible_drift['left']
                            #checking for the right drift
                            right_drift_row = row + row_change
                            right_drift_col = col + row_change

temp_prob_maze[right_drift_row][right_drift_col] +=
prob_maze[row][col] * possible_drift['right']
                        if move_direction == 'S':
                            #checking for straight move
                            temp_prob_maze[new_pos_row][new_pos_col]
+= prob_maze[row][col] * possible_drift['straight']
```

```
                            #checking for the left drifft
                            left_drift_row = row - row_change
                            left_drift_col = col - col_change

temp_prob_maze[left_drift_row][left_drift_col] +=
prob_maze[row][col] * possible_drift['left']
                            #checking for the right drift
                            right_drift_row = row + row_change
                            right_drift_col = col + row_change

temp_prob_maze[right_drift_row][right_drift_col] +=
prob_maze[row][col] * possible_drift['right']
    #replaces the values in the main prob maze with the tepm one
    prob_maze[:] = temp_prob_maze

def sensing(row, col):
    global maze
    direction = ["West", "North", "East","South"]
    result = []

    for dir in direction:
        if dir == "West":
            r,c = row, col - 1
        elif dir == "North":
            r,c = row - 1, col
        elif dir == "East":
            r,c = row, col + 1
        else:
            r,c = row +1, col

        if ( c < 0 or c>=7 or r >= 7 or r < 0):
         result.append(1)
        else:
         result.append(maze[r][c])
    return result

def filtering(visual, next_action):
    global prob_maze, rows, cols
    global correct_obstacle, incorrect_obstacle, correct_open_space,
incorrect_open_space
```

```
    #make a new prob maze to store updated probabilities
    new_prob_maze = [[0 for _ in range(cols)] for _ in range(rows)]

    for row in range(rows):
        for col in range(cols):
            if maze[row][col] == 0:
             current_prob = prob_maze[row][col]
             result = sensing(row, col)
             updated_prob = current_prob

             for v, r in zip(visual, result):
                if v == r:  #the visual matches the maze layout
                    updated_prob *= correct_obstacle if v == 1 else
correct_open_space
                else:  #the visual doesnt match the maze layout
                    updated_prob *= incorrect_obstacle if v == 1
else incorrect_open_space

             new_prob_maze[row][col] = updated_prob
            else:
                continue

    new_prob_maze /= nump.sum(new_prob_maze)
    return new_prob_maze, "predict"

def maze_print(maze, prob_maze):
    for row in range(7):
        for col in range(7):
            if maze[row][col] == 1:  # Check if there is a wall
                print(f"({row},{col}): {0.00:.2f}%", end="\t")
            else:
                print(f"({row},{col}): {prob_maze[row][col] *
100:.2f}%", end="\t")
        print()  # make a new row


#start of the maze
print ("Initial Probabilities")
maze_print(maze, prob_maze)
```

```python
#list of actions to be performed as given by project instructions
actions_list = [
    #sensing
    ([0, 1, 0, 0], None),
    #prediction
    ([0, 0, 0, 0], 'E'),
    #sensing
    ([0, 0, 0, 0], None),
    #prediction
    ([0, 0, 0, 0], 'N'),
    #sensing
    ([1, 0, 0, 1], None),
    #move north
    ([0, 0, 0, 0], 'N'),
    #sensing
    ([0, 1, 0, 0], None),
    #moving west
    ([0, 0, 0, 0], 'W'),
    #sensing
    ([0, 1, 0, 1], None)
    ]

next_action = "filter"
#doing the actions
for visual, direction in actions_list:
    if next_action == "filter":
     prob_maze, next_action = filtering(visual, next_action)
    elif next_action == "predict":
        #make predict function
        #moving(direction)
        print("maze after action")
    maze_print(maze, prob_maze)
    print()

print("Programmed by pmurariu and baughboy")
```

Transitional Probability

Transitional Probability happens in three functions which are def transitional_prob, def transition and def moving. In def transitional_prob, the situation where the robot moves error-free and directly in the planned direction is represented by the drift_straight variable. Nevertheless, there can be a drift to the left or right and imperfect robot movements. To take this into consideration, the drift_left variable computes the state that results from a leftward drift by deducting one from the action and then use %4 to make sure the outcome is within the permissible action range. Similarly, drift_right applies %4 and adds 1 to the operation to account for a drift to the right. These computations acknowledge the cyclical nature of directional decisions; for example, a left drift from the west (activity 0) should appropriately lead to the south (action 3), while a right drift from the south should lead back to west.

In def transition, it determines the new location after the robot takes one step in the designated direction based on the action argument. It verifies the validity of the new location. If a location moves into a cell that is part of the maze's structure (an obstacle, indicated by a 1 in the maze array), it is deemed invalid because it is outside the maze's bounds, which are indicated by the indices being less than 0 or greater than or equal to the maze's size, which in this case is 7x7. The function returns the robot's current status, indicating that it should remain where it is if the new position is invalid (as in, outside the maze bounds or an obstacle). The function returns the new location as a tuple of (row, column) if it is legitimate (not an obstacle and inside the maze).

In def moving, once a move operation has been made, this function computes the probability distribution over the new locations. It considers the potential for drifting, with specific probabilities provided in possible_drift, to travel straight, left, or right. It uses these drift probabilities ('straight', 'left', and 'right') in conjunction with the current probabilities in prob_maze to compute temp_prob_maze, which yields a new distribution that takes the robot's movement uncertainty into account.

Evidence CP

Evidence CP is a part of the filtering function. It goes through each maze cell in a loop. It uses the sensing function to gather sensor measurements for every cell. Next, depending on how closely the actual sensor measurements—which come from sensing—match the expected measurements—which come from visual inspection—it changes the probability that the robot is in each cell. The likelihood is increased by a greater probability (correct_obstacle which is 0.90 or correct_open_space which is 0.95) if the sensor measurement agrees with the expected outcome (for example, both indicate an obstruction). This is because it is expected that the sensor reading will be accurate the majority of the time. The probability is increased by a lower probability (incorrect_obstacle which is 0.05 or incorrect_open_space which is 0.10) if the sensor measurement does not match the expected result (for example, the sensor identifies an obstruction where there should be open space). This is because the sensor reading is less likely to be correct

Filtering

Before adding the new sensor data, it iterates over each cell in the prob_maze, which represents the probability distribution of the robot's position. Then, since we only update probabilities for open spaces and not walls (maze[row][col] == 1), it determines whether the cell is an open space (maze[row][col] == 0). In order to obtain the anticipated sensor values for the current cell based on the maze structure, it then invokes the sensing function. It then makes a comparison between the visual sensor values and the expected readings (result). The current probability of the cell is multiplied by the likelihood of a correct reading (correct_obstacle or correct_open_space) if they match, and by the likelihood of an inaccurate reading (incorrect_obstacle or incorrect_open_space) if they don't. The new_prob_maze is updated using the new probability for the cell. It normalizes the new_prob_maze after updating every cell, ensuring that all probabilities add up to 1, which is a prerequisite for probability distributions. The revised probability distribution is returned by the function.

Prediction

Initially, new_distance = np.zeros((7, 7), dtype=np.float64) generates a 7x7 array that is entirely composed of zeros. Since no probabilities have yet been calculated, this array, which reflects the probability distribution after the action has been taken, is initially set to zero. Next, we have the nested for-loops for row in range(7): for col in range(7): and they run over every cell in the 7x7 grid, assuming that the grid contains all feasible positions for the robot. The next function call is called transitional_prob((row, col), action), which accepts a position (row, col) and an action and returns a list of tuples. A new state (a location in the format new_row, new_col)) and the probability (prob) of getting there from the present state when the action is carried out are contained in each tuple. Next, we have the line that is critical: new_distance[state[0], state[1]] += prob * distance[row, col]. By multiplying the chance of moving to that state (prob) by the current likelihood of being in the original state (distance[row, col]), it updates the probability of the robot being in a specific new state (state). Lastly, we have return new_distance: this indicates that the updated probability distribution new_distance is returned following the iteration over all places and the calculation of all new probabilities.