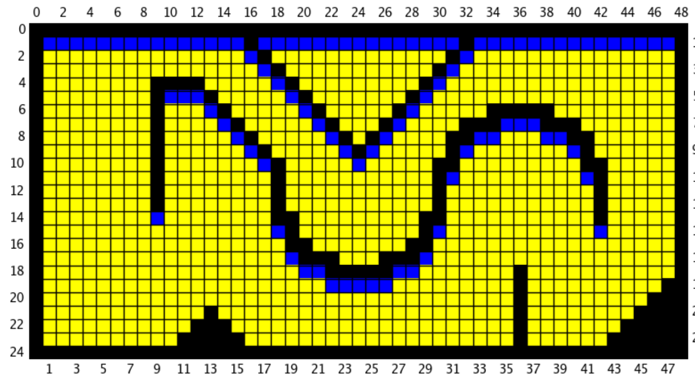
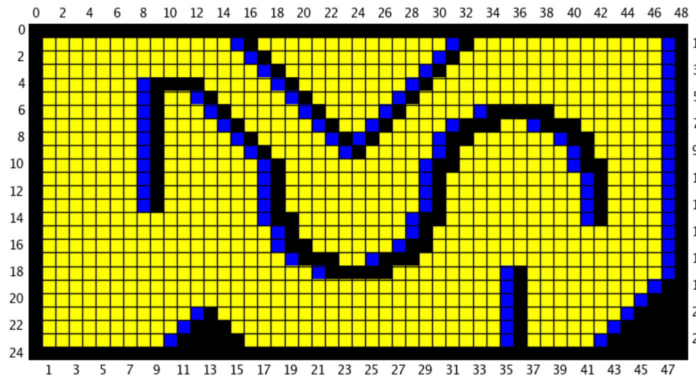
CS 133b **February 7, 2024**

Problem Set 5

Problem 1 (Basic Deterministic System) - 24 points:

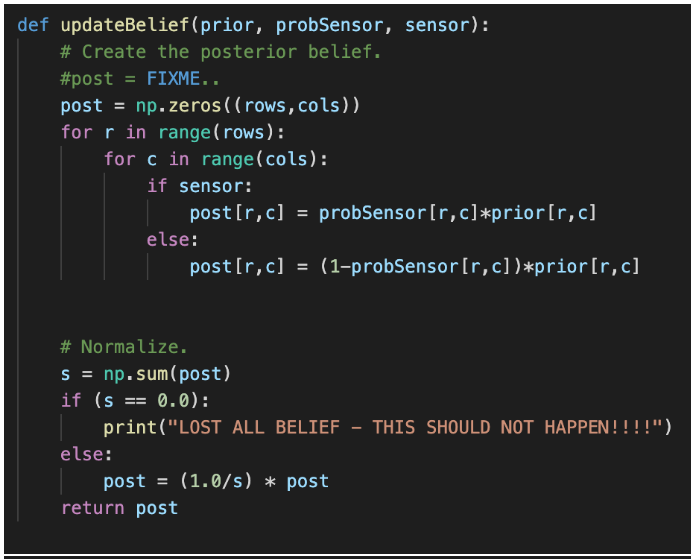




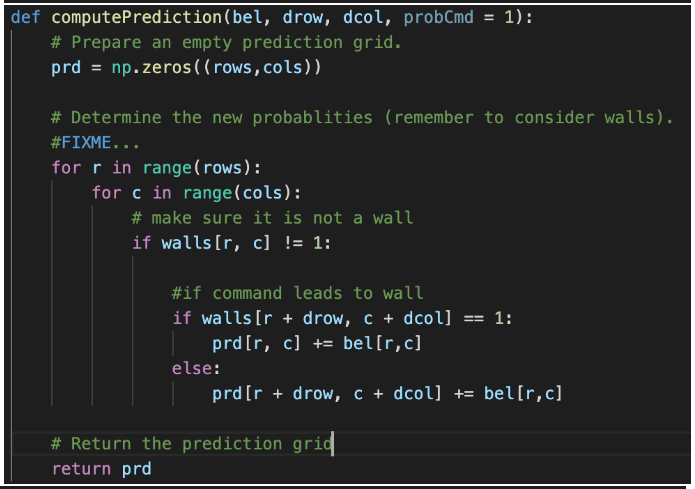
part (a)

Screenshots of the probability map for the up and right sensors are shown above (top figure corresponds to the up sensor). The code is shown on the next page.

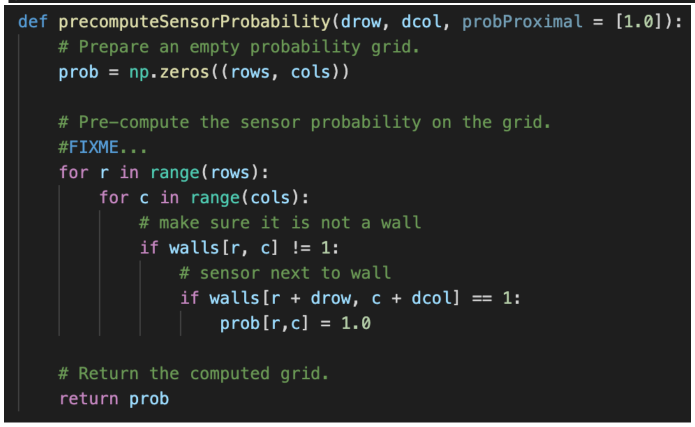
Code for problem 1



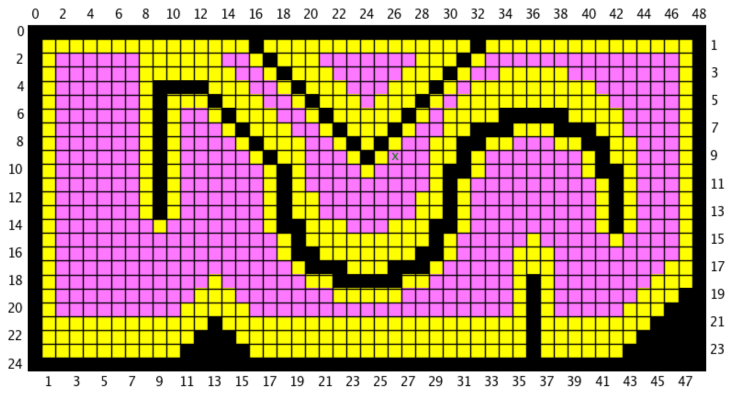
```python
def updateBelief(prior, probSensor, sensor):
    # Create the posterior belief.
    #post = FIXME..
    post = np.zeros((rows,cols))
    for r in range(rows):
        for c in range(cols):
            if sensor:
                post[r,c] = probSensor[r,c]*prior[r,c]
            else:
                post[r,c] = (1-probSensor[r,c])*prior[r,c]


    # Normalize.
    s = np.sum(post)
    if (s == 0.0):
        print("LOST ALL BELIEF - THIS SHOULD NOT HAPPEN!!!!")
    else:
        post = (1.0/s) * post
    return post
```

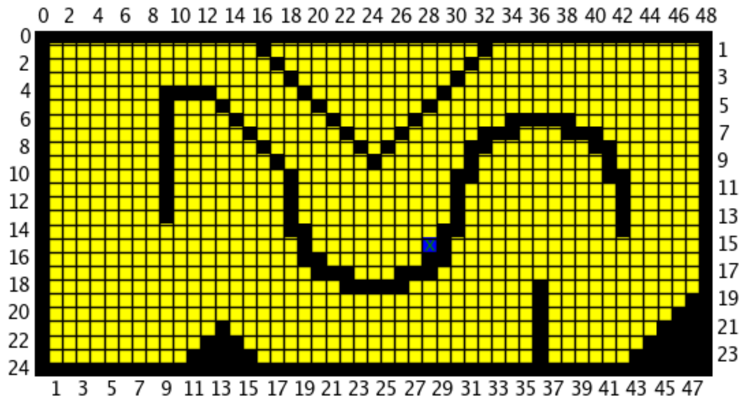


```python
def computePrediction(bel, drow, dcol, probCmd = 1):
    # Prepare an empty prediction grid.
    prd = np.zeros((rows,cols))

    # Determine the new probablities (remember to consider walls).
    #FIXME...
    for r in range(rows):
        for c in range(cols):
            # make sure it is not a wall
            if walls[r, c] != 1:

                #if command leads to wall
                if walls[r + drow, c + dcol] == 1:
                    prd[r, c] += bel[r,c]
                else:
                    prd[r + drow, c + dcol] += bel[r,c]

    # Return the prediction grid
    return prd
```



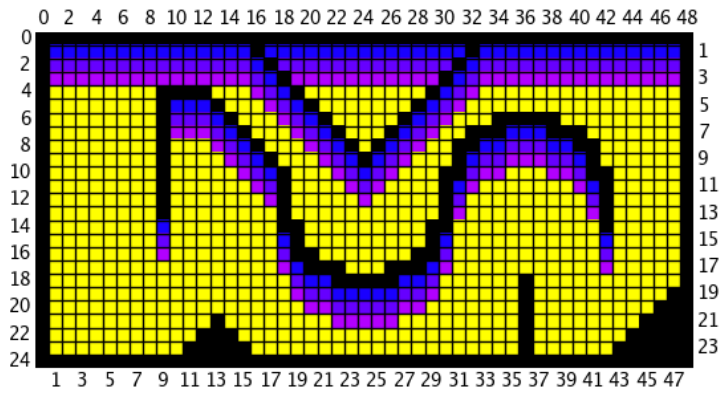
```python
def precomputeSensorProbability(drow, dcol, probProximal = [1.0]):
    # Prepare an empty probability grid.
    prob = np.zeros((rows, cols))

    # Pre-compute the sensor probability on the grid.
    #FIXME...
    for r in range(rows):
        for c in range(cols):
            # make sure it is not a wall
            if walls[r, c] != 1:
                # sensor next to wall
                if walls[r + drow, c + dcol] == 1:
                    prob[r,c] = 1.0

    # Return the computed grid.
    return prob
```

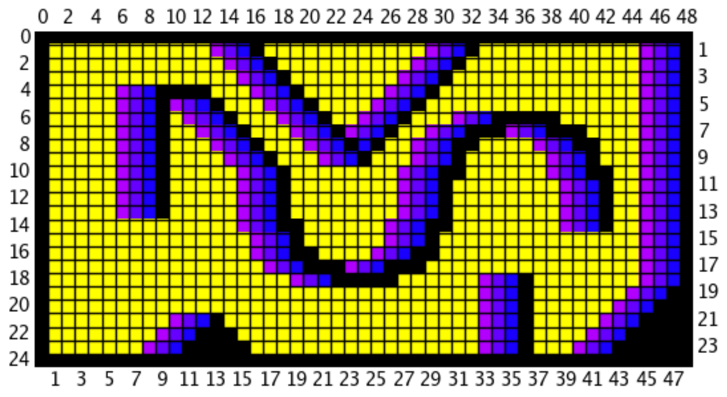




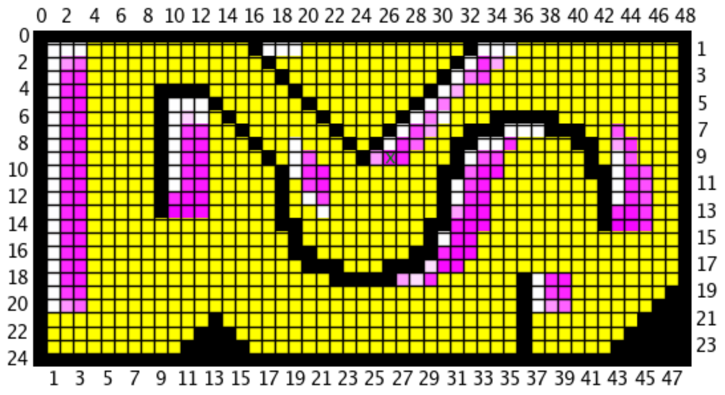
part (b)
The top figure corresponds to the robot starting at row 12 and column 26, and then moving up 3 times. The bottom figure corresponds to the robot moving right 3 times, then down once, then to the left once, and lastly down two times. This set of actions led to completely localizing the robot, the final position of the robot is row 15 and column 28. The code used for this is shown in the previous page (same as in 1a, with initial position robot set to (12, 26)).

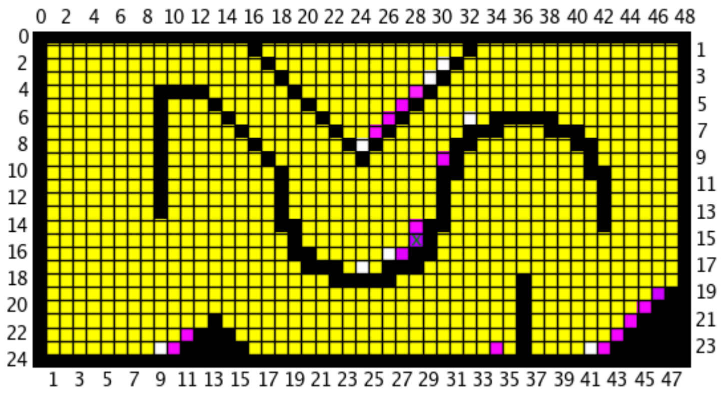
Problem 2 (Noisy Extended Sensor) - 12 points:





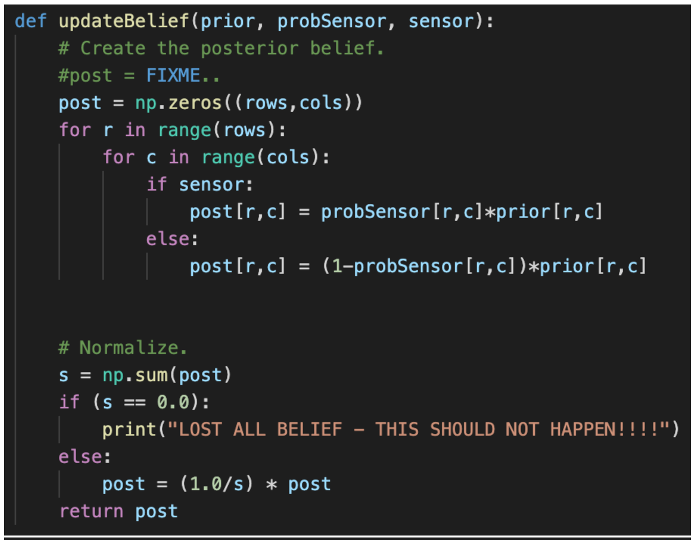
Screenshots of the probability map for the up and right sensors are shown above (top figure corresponds to the up sensor).



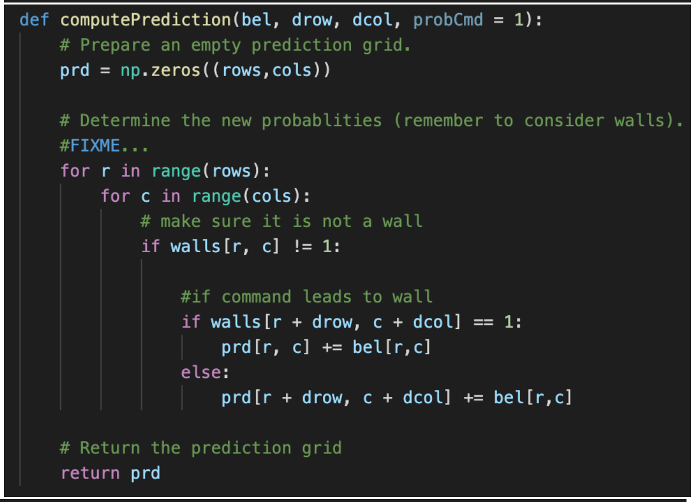


The belief grid when the robot moves 3 units up is shown above in the top figure. The belief grid of the extended sequence is shown above in the bottom figure (same extended sequence as used in problem 1b). As you can see, this set of actions does not completely converge to a single cell anymore as it did in 1b. This is because we are now less certain of what the sensors are reading, so a greater number of steps would be required to be able to completely converge. The code is shown on the next page, probProximal is set accordingly.

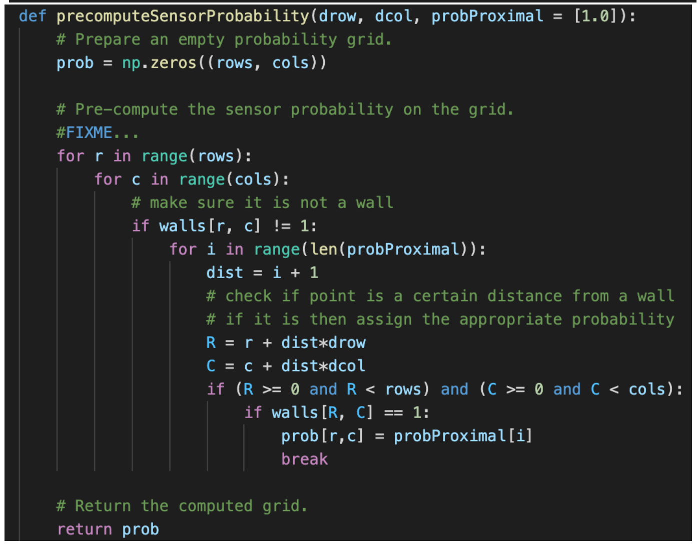
Code for problem 2



```python
def updateBelief(prior, probSensor, sensor):
    # Create the posterior belief.
    #post = FIXME..
    post = np.zeros((rows,cols))
    for r in range(rows):
        for c in range(cols):
            if sensor:
                post[r,c] = probSensor[r,c]*prior[r,c]
            else:
                post[r,c] = (1-probSensor[r,c])*prior[r,c]


    # Normalize.
    s = np.sum(post)
    if (s == 0.0):
        print("LOST ALL BELIEF - THIS SHOULD NOT HAPPEN!!!!")
    else:
        post = (1.0/s) * post
    return post
```

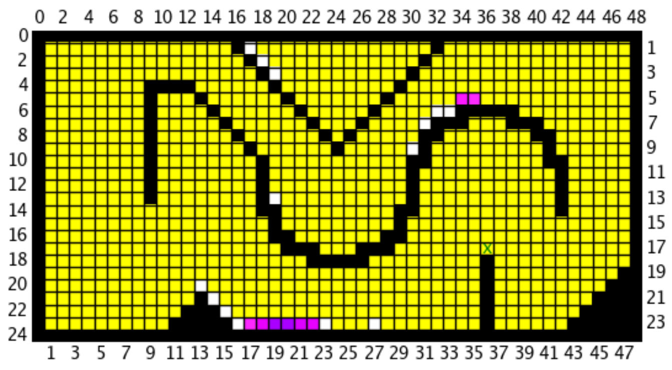


```python
def computePrediction(bel, drow, dcol, probCmd = 1):
    # Prepare an empty prediction grid.
    prd = np.zeros((rows,cols))

    # Determine the new probablities (remember to consider walls).
    #FIXME...
    for r in range(rows):
        for c in range(cols):
            # make sure it is not a wall
            if walls[r, c] != 1:

                #if command leads to wall
                if walls[r + drow, c + dcol] == 1:
                    prd[r, c] += bel[r,c]
                else:
                    prd[r + drow, c + dcol] += bel[r,c]

    # Return the prediction grid
    return prd
```

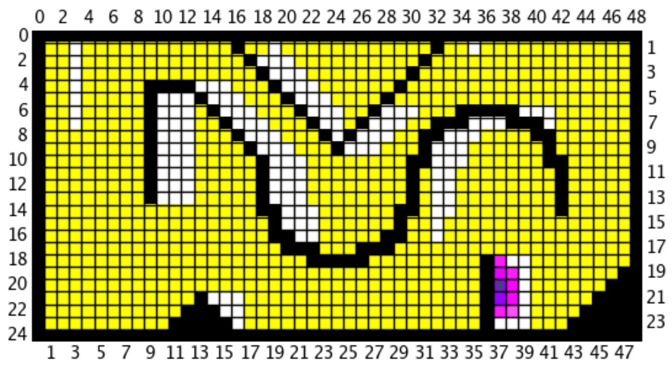


```python
def precomputeSensorProbability(drow, dcol, probProximal = [1.0]):
    # Prepare an empty probability grid.
    prob = np.zeros((rows, cols))

    # Pre-compute the sensor probability on the grid.
    #FIXME...
    for r in range(rows):
        for c in range(cols):
            # make sure it is not a wall
            if walls[r, c] != 1:
                for i in range(len(probProximal)):
                    dist = i + 1
                    # check if point is a certain distance from a wall
                    # if it is then assign the appropriate probability
                    R = r + dist*drow
                    C = c + dist*dcol
                    if (R >= 0 and R < rows) and (C >= 0 and C < cols):
                        if walls[R, C] == 1:
                            prob[r,c] = probProximal[i]
                            break

    # Return the computed grid.
    return prob
```

Problem 3 (Uncertain Command Execution) - 18 points:

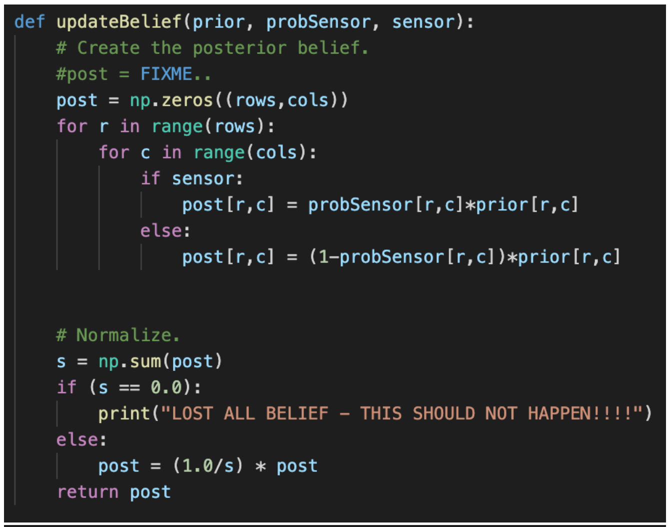


part (a)
Result of the belief grid after the 18 commands is shown above. As expected, the belief and the actual robot location are not in agreement. The code used is the same as in problem 2 aside from creating the the robot object like so: robot=Robot(walls, row=15, col=47, probProximal=[0.9,0.6,0.3],probCmd=0.8.)

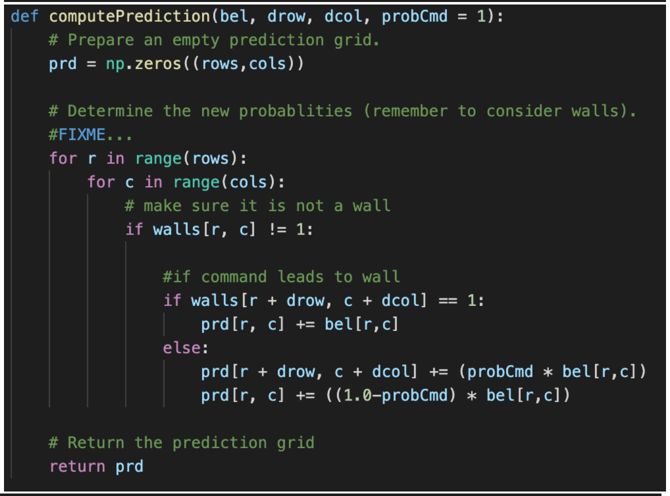


part (b)
Result of the belief grid after the 18 commands is shown above, with the algorithm prediction step fixed. The code is shown on the next page.

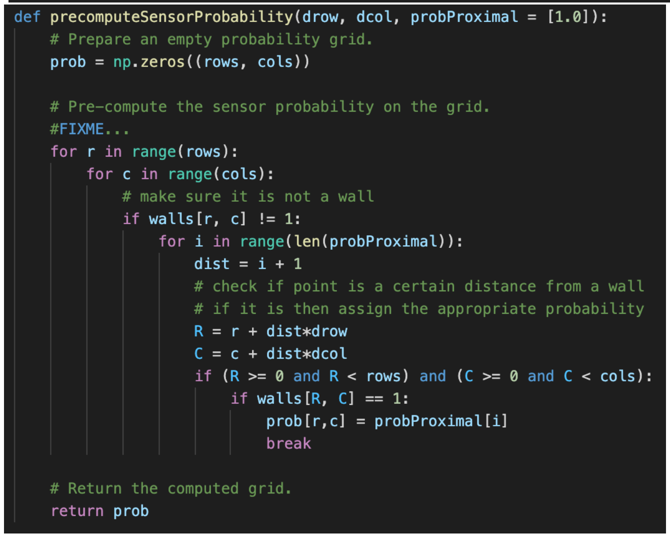
Code for problem 3b



```python
def updateBelief(prior, probSensor, sensor):
    # Create the posterior belief.
    #post = FIXME..
    post = np.zeros((rows,cols))
    for r in range(rows):
        for c in range(cols):
            if sensor:
                post[r,c] = probSensor[r,c]*prior[r,c]
            else:
                post[r,c] = (1-probSensor[r,c])*prior[r,c]


    # Normalize.
    s = np.sum(post)
    if (s == 0.0):
        print("LOST ALL BELIEF - THIS SHOULD NOT HAPPEN!!!!")
    else:
        post = (1.0/s) * post
    return post
```

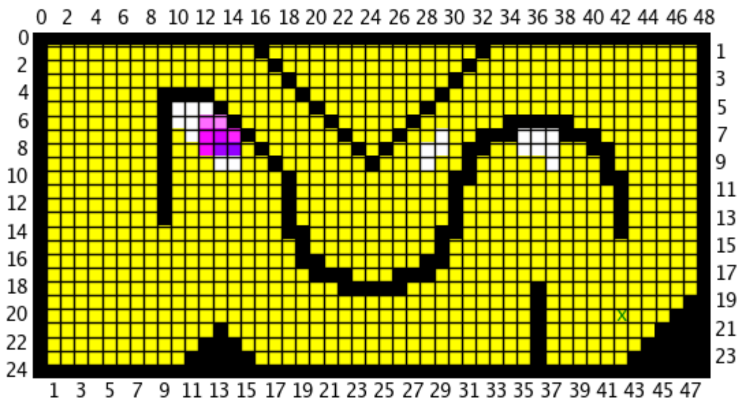


```python
def computePrediction(bel, drow, dcol, probCmd = 1):
    # Prepare an empty prediction grid.
    prd = np.zeros((rows,cols))

    # Determine the new probablities (remember to consider walls).
    #FIXME...
    for r in range(rows):
        for c in range(cols):
            # make sure it is not a wall
            if walls[r, c] != 1:

                #if command leads to wall
                if walls[r + drow, c + dcol] == 1:
                    prd[r, c] += bel[r,c]
                else:
                    prd[r + drow, c + dcol] += (probCmd * bel[r,c])
                    prd[r, c] += ((1.0-probCmd) * bel[r,c])

    # Return the prediction grid
    return prd
```

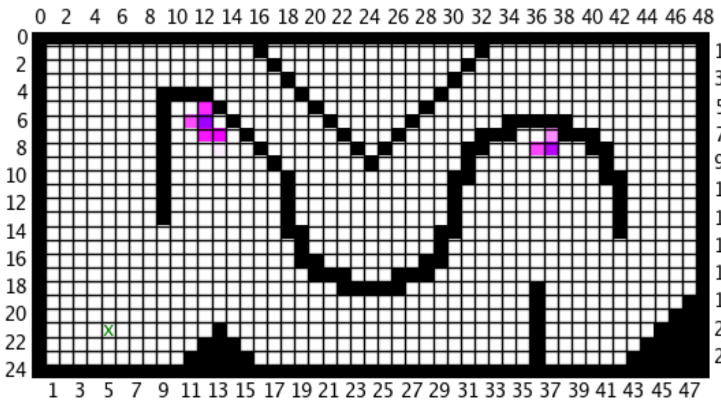


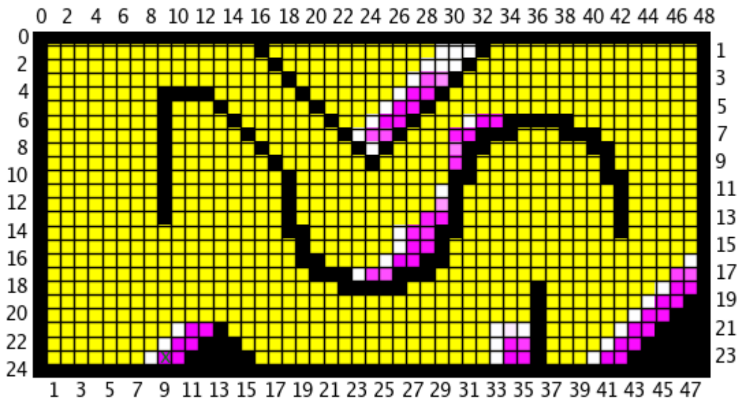
```python
def precomputeSensorProbability(drow, dcol, probProximal = [1.0]):
    # Prepare an empty probability grid.
    prob = np.zeros((rows, cols))

    # Pre-compute the sensor probability on the grid.
    #FIXME...
    for r in range(rows):
        for c in range(cols):
            # make sure it is not a wall
            if walls[r, c] != 1:
                for i in range(len(probProximal)):
                    dist = i + 1
                    # check if point is a certain distance from a wall
                    # if it is then assign the appropriate probability
                    R = r + dist*drow
                    C = c + dist*dcol
                    if (R >= 0 and R < rows) and (C >= 0 and C < cols):
                        if walls[R, C] == 1:
                            prob[r,c] = probProximal[i]
                            break

    # Return the computed grid.
    return prob
```

Problem 4 (Kidnapped Robots) - 14 points:



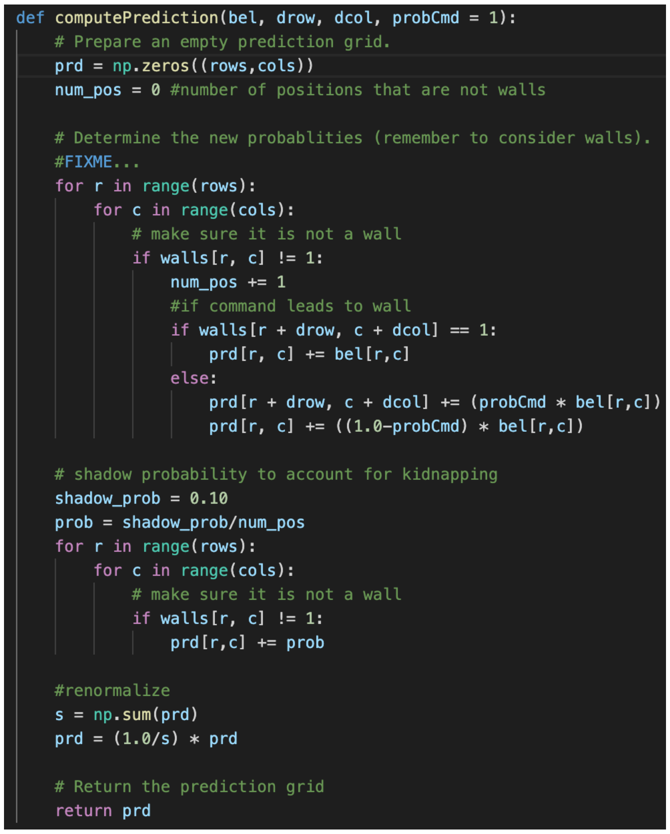
part (a)
Result of the belief grid right after the robot is kidnapped.The code is the same as the one used in problem 3.





part (b)
To fix the algorithm, the computePrediction function is changed such that a shadow probability is uniformly added to all cells/positions that are not walls. Normalization is done after uniformly adding the shadow probability to all cells/positions (that are not walls). The belief grid after the robot is teleported is shown above (the top figure). The bottom figure corresponds to the belief grid after 6 moves where made after the robot was teleported. As you can see, the robot has good idea of where it might be after about 6 moves. Although, the number of moves depend on how far away the robot is from a wall (the robot know where it is better when there are walls). The updated code is shown on the next page.

Code for problem 4b



```python
def computePrediction(bel, drow, dcol, probCmd = 1):
    # Prepare an empty prediction grid.
    prd = np.zeros((rows,cols))
    num_pos = 0 #number of positions that are not walls

    # Determine the new probablities (remember to consider walls).
    #FIXME...
    for r in range(rows):
        for c in range(cols):
            # make sure it is not a wall
            if walls[r, c] != 1:
                num_pos += 1
                #if command leads to wall
                if walls[r + drow, c + dcol] == 1:
                    prd[r, c] += bel[r,c]
                else:
                    prd[r + drow, c + dcol] += (probCmd * bel[r,c])
                    prd[r, c] += ((1.0-probCmd) * bel[r,c])

    # shadow probability to account for kidnapping
    shadow_prob = 0.10
    prob = shadow_prob/num_pos
    for r in range(rows):
        for c in range(cols):
            # make sure it is not a wall
            if walls[r, c] != 1:
                prd[r,c] += prob

    #renormalize
    s = np.sum(prd)
    prd = (1.0/s) * prd

    # Return the prediction grid
    return prd
```

Problem 5a (Kalman Filter)

t = 0 min
Initial values

$$\hat{x}_0 = 1000\ m$$

$$\sigma_0^2 = 10,000\ m^2$$

t = 1 min
Previous values (t=0 min) are shown below

$$\hat{x}_0 = 1000\ m$$

$$\sigma_0^2 = 10,000\ m^2$$

Prediction

$$\hat{x}_{pt} = \hat{x}_0 + 1000\ m = 2000\ m$$

$$\sigma_{pt}^2 = \sigma_0^2 + (30m.)^2 = 10,900m^2$$

No Measurement update

t = 2 min
Previous values (t=1 min) are shown below

$$\hat{x}_{t-1} = 2000\ m$$

$$\sigma_{t-1}^2 = 10,900m^2$$

Prediction

$$\hat{x}_{pt} = \hat{x}_{t-1} + 1000\ m = 3000\ m$$

$$\sigma_{pt}^2 = \sigma_{t-1}^2 + (30m.)^2 = 11,800m^2$$

Measurement update

$$z_t = 3100\ m$$

$$k = \frac{\sigma_{pt}^2}{\sigma_{pt}^2 + \sigma_w^2} = \frac{11,800}{11,800 + 100} = 0.9916$$

$$\hat{x}_t = \hat{x}_{pt} + k(z_t - \hat{x}_{pt}) = 3000 + 0.9916(3100 - 3000) = 3099.16\ m$$

$$\sigma_t^2 = \frac{\sigma_{pt}^2 \sigma_w^2}{\sigma_{pt}^2 + \sigma_w^2} = \frac{11,800 \cdot 100}{11,800 + 100} = 99.1597$$

t = 3 min
Previous values (t=2 min) are shown below

$$\hat{x}_{t-1} = 3099.16\ m$$

$$\sigma^2_{t-1} = 99.1597m^2$$

Prediction

$$\hat{x}_{pt} = \hat{x}_{t-1} + 1000\ m = 4099.16\ m$$

$$\sigma^2_{pt} = \sigma^2_{t-1} + (30m.)^2 = 999.1597\ m^2$$

Measurement update

$$z_t = 4050\ m$$

$$k = \frac{\sigma^2_{pt}}{\sigma^2_{pt} + \sigma^2_w} = \frac{999.1597}{999.1597 + 100} = 0.9090$$

$$\hat{x}_t = \hat{x}_{pt} + k(z_t - \hat{x}_{pt}) = 4099.16 + 0.9090(4050 - 4099.16) = 4054.4736\ m$$

$$\sigma^2_t = \frac{\sigma^2_{pt}\sigma^2_w}{\sigma^2_{pt} + \sigma^2_w} = \frac{999.1597 \cdot 100}{999.1597 + 100} = 90.9214\ m^2$$

t = 4 min
Previous values (t=3 min) are shown below

$$\hat{x}_{t-1} = 4054.4736\ m$$

$$\sigma^2_{t-1} = 90.9214\ m^2$$

Prediction

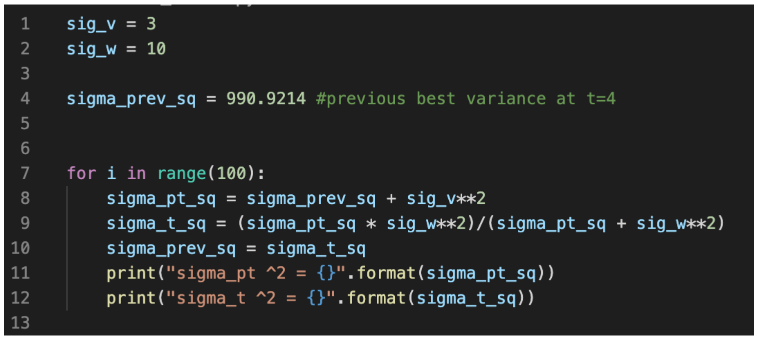
$$\hat{x}_{pt} = \hat{x}_{t-1} + 1000\ m = 5054.4736\ m$$

$$\sigma^2_{pt} = \sigma^2_{t-1} + (30m.)^2 = 990.9214\ m^2$$

No Measurement update

Problem 5b (Kalman Filter)

Using the value of $\sigma_{pt}^2 = 990.9214\ m^2$ for t=4 from part 5a, we get that at steady state the worst variance is 34.836 m^2 and the best variance is 25.836 m^2 assuming a measurement is received every minute after t=4 min. The code used is shown below:



```
sig_v = 3
sig_w = 10

sigma_prev_sq = 990.9214 #previous best variance at t=4


for i in range(100):
    sigma_pt_sq = sigma_prev_sq + sig_v**2
    sigma_t_sq = (sigma_pt_sq * sig_w**2)/(sigma_pt_sq + sig_w**2)
    sigma_prev_sq = sigma_t_sq
    print("sigma_pt ^2 = {}".format(sigma_pt_sq))
    print("sigma_t ^2 = {}".format(sigma_t_sq))
```

Problem 6 (Time Spent) - 4 points:

I spent about 5.5 hours on the set.