**Demetrius Johnson**

**UM-Dearborn CIS-479**

**Program 2 Report: Hidden Markov Model – Robot Localization**

**With Prof. Dr. Shengquan Wang**

**Summer II 2022**

**8-09-22**

Contents

[Transitional Probability 3](#_Toc110981175)

[Evidence Conditional Probability 3](#_Toc110981176)

[Filtering (from sensing) 3](#_Toc110981177)

[**Here, I demonstrate how to get the sum of all proportions of probabilities given an evidence:** 5](#_Toc110981178)

[**Here, I show a repeated process, but only this time we use our sum of proportions to get the probability for all locations:** 5](#_Toc110981179)

[Prediction (from movement) 6](#_Toc110981180)

[**Here, I demonstrate how I don’t add all probabilities for a location at one time, rather I sweep across the entire matrix and update probabilities as a location should absorb it given do a move from one location to another:** 6](#_Toc110981181)

[**And here, I demonstrate how I create a transition function that sweeps across the matrix allowing the proper location to absorb probability for all directions as shown in the function above, below I will show you just one of the functions for the sake of space and unnecessary redundancy:** 7](#_Toc110981182)

[Screenshots of solution output. Notice my last name and the time on the bottom right of each screenshot: 8](#_Toc110981183)

[Standard Test Case Screenshots: 9](#_Toc110981184)

[Initial Probabilities == uniform 9](#_Toc110981185)

[SENSE [0,0,0,0], MOVE N 10](#_Toc110981186)

[SENSE [1,0,0,0], MOVE N 11](#_Toc110981187)

[SENSE [1,0,0,0], MOVE W 12](#_Toc110981188)

[SENSE [1,0,0,0], MOVE N 13](#_Toc110981189)

[\*\*BONUS moving EAST and SOUTH screenshots\*\*: 14](#_Toc110981190)

[SENSE [0,0,0,0], MOVE S 14](#_Toc110981191)

[SENSE [1,0,0,0], MOVE N 15](#_Toc110981192)

[SENSE [0,0,0,0], MOVE E 16](#_Toc110981193)

# Transitional Probability

//moving transition probabilities:

const float forward\_probability = (float)0.8; //move forward in desired direction = 80%

const float left\_probability = (float)0.1; //drift left in undesired direction = 10%

const float right\_probability = (float)0.1; //drift right in undesired direction = 10%

# Evidence Conditional Probability

//obstacle sensor transition probabilities:

const float sensing\_OBJ\_correctly\_probability = (float)0.85; //correctly sense square as an obstacle = 85%

const float sensing\_OBJ\_incorrectly\_probability = (float)0.15; //false negative(no obstacle) - incorrectly sense square as NOT an obstacle (open square) = 15%

//open-square (non-obstacle) sensor transition probabilities:

const float sensing\_no\_OBJ\_correctly\_probability = (float)0.95; //correctly sense square as NOT an obstacle = 95%

const float sensing\_no\_OBJ\_incorrectly\_probability = (float)0.05; //false positive(is an obstacle) - incorrectly sense square as an obstacle = 5%

# Filtering (from sensing)

void updateProb\_sensing\_filtering(table& table\_struct, Location\* sensor\_evidence\_bit\_WNES) {

//Filtering: P(St|Z1=z1, …, Zt=zt) ∝ P(Zt=zt|St) P(St|Z1=z1, …, Zt-1=zt-1)

/\* note that Z\_t represents an evidence variable --> evidence from the neighbor sensor

Zt is composed of 4 random variables for four directions:

Zt = (ZW,t, ZN,t, ZE,t, ZS,t)

For each state St, conditionally independent:

P(Zt|St) = P(ZW,t|St) P(ZN,t|St) P(ZE,t|St) P(ZS,t|St)

note that:

P(Zt=zt|St)== likelihood

P(St|Z1=z1, …, Zt-1=zt-1) == prior

P(St|Z1=z1, …, Zt=zt) == posterior

\*/

float sum\_of\_proportions = 0; //sum of proportions == SUM for all States (S\_t) of [ P(Zt=zt|St) P(St|Z1=z1, …, Zt-1=zt-1) ].

//add all proportions to get total probability of evidence Z\_t given S\_t,

//then we can use this sum to get P(St|Z1=z1, …, Zt=zt) == [ P(Zt=zt|St) P(St|Z1=z1, …, Zt-1=zt-1) / sum\_of\_proportions ].

//This is the same thing as saying: [1 of the proportions / sum\_of\_propotions]...

//--> and we can do this for all states to find the probabiility the robot is at any given location given evidence(s) Z\_t.

Location neighbors\_WNES[4]; //use this to get the true neighbor based om the true map the robot has access to to compare it with the sensor evidence

for (int row = 0; row < table\_struct.numRows; row++) {

for (int col = 0; col < table\_struct.numCols; col++) {

if (is\_obstacleLocation(row, col)) //no calculation necessary for obstacle case --> robot knows it cannot be there

continue;

//check neighboring squares for obstacles so we can determine which sensor error transistion probability to use at each possible state S\_t

getNeighbors\_WNES(row, col, neighbors\_WNES);

//calculate likelihood --> P(Z\_t | S\_t) --> obstacle or open square probability at a given state S\_t = s\_t, based on evidence from sensor:

float sensorProb\_WNES[4];

for (int i = 0; i < 4; i++) {

//case: correctly sense an obstacle ==85%

if (sensor\_evidence\_bit\_WNES[i] == OBSTACLE && neighbors\_WNES[i] == OBSTACLE)

sensorProb\_WNES[i] = sensing\_OBJ\_correctly\_probability;

//case: incorrectly sense an obstacle as open square (false negative) ==15%

else if (sensor\_evidence\_bit\_WNES[i] == OPEN\_SQUARE && neighbors\_WNES[i] == OBSTACLE)

sensorProb\_WNES[i] = sensing\_OBJ\_incorrectly\_probability;

//case: correctly sense an open sqaure ==95%

else if (sensor\_evidence\_bit\_WNES[i] == OPEN\_SQUARE && neighbors\_WNES[i] == OPEN\_SQUARE)

sensorProb\_WNES[i] = sensing\_no\_OBJ\_correctly\_probability;

//case: incorrectly sense an open sqaure as an obstacle (false positive) ==5%

else if (sensor\_evidence\_bit\_WNES[i] == OBSTACLE && neighbors\_WNES[i] == OPEN\_SQUARE)

sensorProb\_WNES[i] = sensing\_no\_OBJ\_incorrectly\_probability;

}

//multiply all conditionall indepdent likelihoods to get total likelihood of a state S\_t given given sensor evidence Z\_t

float state\_totalLikelihood = 1;

for (int i = 0; i < 4; i++)

state\_totalLikelihood \*= sensorProb\_WNES[i];

//Now, multiply total likelihood by the prior and add that to the sum

sum\_of\_proportions += (state\_totalLikelihood \* table\_struct.tablePos\_locationProb\_prior[row][col]);

}

}

### **Here, I demonstrate how to get the sum of all proportions of probabilities given an evidence:**

//now, sum of proportions has been acquired, we now can normalize and find the posterior probabilitl for all states:

// P(St|Z1=z1, …, Zt=zt) == [ P(Zt=zt|St) P(St|Z1=z1, …, Zt-1=zt-1) / sum\_of\_proportions ]

for (int row = 0; row < table\_struct.numRows; row++) {

for (int col = 0; col < table\_struct.numCols; col++) {

if (is\_obstacleLocation(row, col)) //no calculation necessary for obstacle case --> robot knows it cannot be there

continue;

//check neighboring squares for obstacles so we can determine which sensor error transistion probability to use at each possible state S\_t

getNeighbors\_WNES(row, col, neighbors\_WNES);

//calculate likelihood --> P(Z\_t | S\_t) --> obstacle or open square probability at a given state S\_t = s\_t, based on evidence from sensor:

float sensorProb\_WNES[4];

for (int i = 0; i < 4; i++) {

//case: correctly sense an obstacle ==85%

if (sensor\_evidence\_bit\_WNES[i] == OBSTACLE && neighbors\_WNES[i] == OBSTACLE)

sensorProb\_WNES[i] = sensing\_OBJ\_correctly\_probability;

//case: incorrectly sense an obstacle as open square (false negative) ==15%

else if (sensor\_evidence\_bit\_WNES[i] == OPEN\_SQUARE && neighbors\_WNES[i] == OBSTACLE)

sensorProb\_WNES[i] = sensing\_OBJ\_incorrectly\_probability;

//case: correctly sense an open sqaure ==95%

else if (sensor\_evidence\_bit\_WNES[i] == OPEN\_SQUARE && neighbors\_WNES[i] == OPEN\_SQUARE)

sensorProb\_WNES[i] = sensing\_no\_OBJ\_correctly\_probability;

//case: incorrectly sense an open sqaure as an obstacle (false positive) ==5%

else if (sensor\_evidence\_bit\_WNES[i] == OBSTACLE && neighbors\_WNES[i] == OPEN\_SQUARE)

sensorProb\_WNES[i] = sensing\_no\_OBJ\_incorrectly\_probability;

}

//multiply all conditionall indepdent likelihoods to get total likelihood of a state S\_t given given sensor evidence Z\_t

float state\_totalLikelihood = 1;

for (int i = 0; i < 4; i++)

state\_totalLikelihood \*= sensorProb\_WNES[i];

### **Here, I show a repeated process, but only this time we use our sum of proportions to get the probability for all locations:**

//Now, multiply total likelihood by the prior

//Now, at this moment we have calculated one proportion; now,

//we normalize since we already have sum of all proportions, to get posterior for the current state at [row][col]

table\_struct.tablePos\_locationProb\_posterior[row][col] = (state\_totalLikelihood \* table\_struct.tablePos\_locationProb\_prior[row][col]) / sum\_of\_proportions;

}

}

}

# Prediction (from movement)

void updatProb\_moving\_prediction(table& table\_struct, Direction\_Cardinal move\_direction) {

//Prediction: P(St+1|Z1=z1, …, Zt=zt) = ∑sP(St+1|St=s) P(St|Z1=z1, …, Zt=zt)

//The prediction is essentially just the sum of all probabilities that you can get to state S\_t+1, given all possible prior states S\_t.

//first, reinitialize the posterior table, so that we can incrementally sweep across it and add the probability of getting to one state from one of the other possible states

for (int i = 0; i < table\_struct.numRows; i++) {

for (int j = 0; j < table\_struct.numCols; j++) {

if (is\_obstacleLocation(i, j))

continue;

else

table\_struct.tablePos\_locationProb\_posterior[i][j] = 0;

}

}

### **Here, I demonstrate how I don’t add all probabilities for a location at one time, rather I sweep across the entire matrix and update probabilities as a location should absorb it given do a move from one location to another:**

//now sweep across all locations adding any probability from neighbor states that it can be reached from - when done, then all probability for reaching a state S\_t+1 from all other possible states S\_t will be complete

for (int row = 0; row < table\_struct.numRows; row++) {

for (int col = 0; col < table\_struct.numCols; col++) {

if (is\_obstacleLocation(row, col)) //no need to check impossible cases since robot cannot be at an obstacle location

continue;

switch (move\_direction) {

case Direction\_Cardinal::WEST:

prob\_moving\_WEST(row, col, table\_struct);

break;

case Direction\_Cardinal::NORTH:

prob\_moving\_NORTH(row, col, table\_struct);

break;

case Direction\_Cardinal::EAST:

prob\_moving\_EAST(row, col, table\_struct);

break;

case Direction\_Cardinal::SOUTH:

prob\_moving\_SOUTH(row, col, table\_struct);

break;

}

}

}

}

### **And here, I demonstrate how I create a transition function that sweeps across the matrix allowing the proper location to absorb probability for all directions as shown in the function above, below I will show you just one of the functions for the sake of space and unnecessary redundancy:**

void prob\_moving\_WEST(int row, int col, table& table\_struct) {

//WEST == FORWARD-->(col - 1), LEFT-->(row + 1), RIGHT-->(row - 1):

//forward

//check wall bounce back case, otherwise use (col - 1)

if (is\_obstacleLocation(row, col - 1))

//current [row][col] location absorbs the probability

table\_struct.tablePos\_locationProb\_posterior[row][col] += (forward\_probability \* table\_struct.tablePos\_locationProb\_prior[row][col]);

else

//otherwise, forward location absorbs the probability

table\_struct.tablePos\_locationProb\_posterior[row][col - 1] += (forward\_probability \* table\_struct.tablePos\_locationProb\_prior[row][col]);

//left

//check bounce back case, otherwise use (row + 1)

if (is\_obstacleLocation(row + 1, col))

//current [row][col] location absorbs the probability

table\_struct.tablePos\_locationProb\_posterior[row][col] += (left\_probability \* table\_struct.tablePos\_locationProb\_prior[row][col]);

else

//otherwise, left location absorbs the probability

table\_struct.tablePos\_locationProb\_posterior[row + 1][col] += (left\_probability \* table\_struct.tablePos\_locationProb\_prior[row][col]);

//right

//check bounce back case, otherwise use (row - 1)

if (is\_obstacleLocation(row - 1, col))

table\_struct.tablePos\_locationProb\_posterior[row][col] += (right\_probability \* table\_struct.tablePos\_locationProb\_prior[row][col]);

else

//otherwise, right location absorbs the probability

table\_struct.tablePos\_locationProb\_posterior[row - 1][col] += (right\_probability \* table\_struct.tablePos\_locationProb\_prior[row][col]);

}

# Screenshots of solution output. Notice my last name and the time on the bottom right of each screenshot:

\*\*Notice how you are never 100% certain where the robot is, and also whenever the robot moves once the values converge, it is pretty easy to tell where it is and what square it moves too (with much greater certainty). This is especially demonstrated in my bonus test cases after the standard test cases show a convergence of where the robot most likely is. Also, if we were to just keep sensing and filtering, the result would be that we would become more and more certain – even with the error of the robot adding up. Over time, however, eventually all of the error over adds up (over many, many iterations) and will cause the probabilities to converge to a uniform distribution in this stochastic setup (especially if you moved the robot to every location over and over).

## Standard Test Case Screenshots:

### Initial Probabilities == uniform

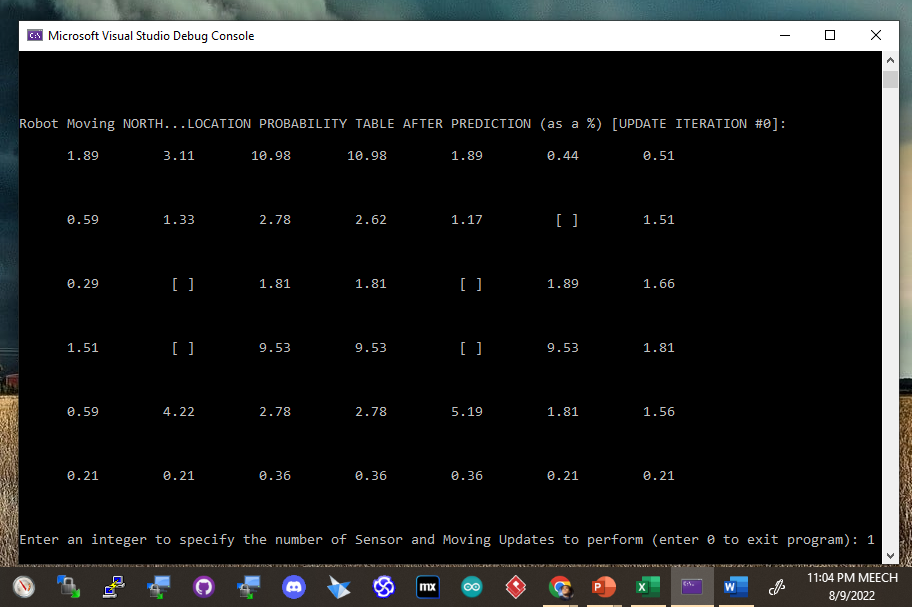
A screenshot of a computer screen

Description automatically generated

### SENSE [0,0,0,0], MOVE N

A screenshot of a computer

Description automatically generated



### SENSE [1,0,0,0], MOVE N

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

### SENSE [1,0,0,0], MOVE W

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

### SENSE [1,0,0,0], MOVE N

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## \*\*BONUS moving EAST and SOUTH screenshots\*\*:

### SENSE [0,0,0,0], MOVE S

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

### SENSE [1,0,0,0], MOVE N

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

### SENSE [0,0,0,0], MOVE E

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated