Minimizing Effect of Noisy Sensors on Agent Localization Using Hidden Markov Models

Jacob Barna
Computer Science Department
University of Nebraska-Omaha
Omaha, NE USA
jbarna@unomaha.edu

I. Abstract

Consumer electronics prices have been trending downward as the number of features and their “smart” capabilities have simultaneously been trending upward. The affordability is driven by the use of inexpensive, mass-produced parts in their construction. These inexpensive parts can introduce faults and errors in system behavior, which can render a system useless if not accounted for.

This paper examines the use of hidden Markov models, a temporal probabilistic model, as a technique to mask faulty sensor data. The example task explored in this paper is the task of agent localization. Given an agent in a partially observable environment with a non-deterministic move action and a sensor error rate, the agent must model and maintain a belief state about its location.

Another primary objective of this research is to create an interactive web-based simulator for such an agent. This simulator will allow users to construct a grid environment, change sensor error rates, and examine the changes in the belief state over time.

Using the simulator, the effects of the error rate on the belief state of the agent will be examined. The principal question to be answered is how the error rate affects location accuracy, the Manhattan distance between the true location of the agent and the location that the agent believes it is in.

II. Introduction

Intelligent agents in partially-observable environments rely on information about the environment (percepts) to make decisions [9, pp. 34]. For instance, in a grid environment, an agent can use information about obstacles in its path to reason where it is located. In a grid with no obstacles, an agent that receives that perceives boundaries to the North and West *must* be in the upper left corner of the grid. This type of logical reasoning has roots in classical philosophy, but was more directly explored by John McCarthy, who suggested that “a program has common sense if it automatically deduces for itself a sufficiently wide class of immediate consequences of anything it is told and what it already knows” [7, pp. 2]. This led to a great deal of work in logic-based artificial intelligence.

This type of logic fails when uncertainty is added to the problem. If a program can be “told” something that is inaccurate, how can it be asked to make deductions with the information? In our grid world, what happens if the agent is unable to detect obstacles and boundaries with 100% accuracy? This is a realistic problem as sensors used in automated systems are often intrinsically noisy. This is a significant challenge because the agent must now reason about not only what it is currently perceiving, but what it has perceived in the past.

One approach lies in the use of probabilistic temporal models with which an agent can observe the changes in its precepts over time and use this knowledge to model the current belief state – a set of possible environments with a calculated probability. One such model that uses this approach is a hidden Markov model. A hidden Markov model is defined by two properties: the observer cannot see the state of the process, and the hidden process follows the Markov assumption. The Markov assumption simplifies the problem by assuming that “the state at some time encapsulates all we need to know about the history of the process in order to predict the future of the process” [3, pp. 1].

In this paper, we will implement a hidden Markov model to generate the belief state of an agent engaged in the *localization problem;* modeling the belief state of the agent with respect to its current location [9, pp. 145]. This agent will be placed in a $M \ x \ N$ grid environment with some number of occupied spaces (obstacles). As will be shown, the agent can use knowledge about its environment (boundaries and obstacles) along with evidence collected over time to mask sensor faults and determine with some accuracy its location. We will also show the effect that varying sensor error rates have on the accuracy of the belief state. We will also examine some of the weaknesses of the model by contrasting the result of two experiments: localizing in a maze-like environment and localizing an agent in a grid that consists of one row and sixty-four columns.

The rest of the paper is outlined as follows. First, an overview of temporal probabilistic models, and more specifically, hidden Markov models will be given. Next is a description of the implementation of a web-based simulator using hidden Markov models as the underlying mechanism for updating the belief state based on the environment and sensor readings. This is followed by the description of two experiments conducted in which agent localization performance is measured over 40 observations at different sensor error rates. The performance is averaged over seven test runs with each different sensor error rate provided. Finally, conclusions will be drawn, and future work discussed.

III. Background

A. Probabilistic Reasoning over Time

Many agents use logic to describe *possible* belief states. For example, suppose we have an agent in a partially observable environment. This agent knows the locations of boundaries and obstacles. Further suppose that the agent's sensors are always accurate. In this case, agent localization (in most cases) is an easy problem, even if the agent initially is unaware of its location and has a non-deterministic move action. After each move, the accurate sensor information can be used to update the belief state to only those locations that would show the same sensor readings.

Knowing that sensors can be noisy and even incorrect, the localization problem becomes more difficult as the agent is unsure if it can trust its sensors. This *uncertainty* can be modeled using temporal probabilistic models, in which the evidence gathered over time can be used in shaping the belief state of the agent. In the models studied in this paper, time is treated as a series of time slices, denoted by t. At each time slice, the belief state, in our example, the locations that the agent thinks it could be occupying, can be updated by evidence at time t. In this paper, and elsewhere, $\boldsymbol{X}_t$ is used to denote the belief state of the agent. The notation $\boldsymbol{E}_t = \boldsymbol{e}_t$ denotes the observed evidence at time t, in our example, these are observation from sensors.

In temporal probabilistic models, there are two mechanisms used to update the belief state, the transition model and the sensor model. The transition model expresses how the current state is dependent on prior state. The transition model is the distribution $\boldsymbol{P}(\boldsymbol{X}_t \mid \boldsymbol{X}_{t-1})$. One may notice that this distribution uses only 1 prior time slice to determine the current state. This is due to the *Markov assumption* that the future depends only on a fixed number of prior states. In our example, we are using a *first-order Markov process* because we assume that only one prior time slice matters.

The sensor model, on the other hand, expresses how current evidence being observed depends on the actual state. Like the transition model, the sensor model we use also takes advantage of the *Markov assumption*, in which we assume that the current evidence depends only on the current state. The sensor model is denoted as $\boldsymbol{P}(\boldsymbol{E}_t \mid \boldsymbol{X}_t)$.

The last requirement for our temporal probabilistic model is an initial belief state, $\boldsymbol{P}(\boldsymbol{X}_0)$.

Now that the definition of the transition model, sensor model, and initial belief state is known, we can turn to inference in temporal probabilistic models. In this paper, the only inference task discussed is *filtering*, also known as *state estimation*, which is the mechanism by which our agent will update its belief state based on its current belief state and observed evidence. The filtering algorithm was developed for continuous variables by Weiner, and adapted for discrete time by Kolmogorov [4, 11]. The well-known filtering algorithm is a recursive algorithm that takes the output of filtering up to time t and updates it with the current observed evidence, or percept. The equation used to filter is shown below. Note that α is a normalizing constant that is used to make the probabilities sum to 1. Next to the normalizing constant is the factor from the sensor model, and in the summation, the left term is from the transition model. $\boldsymbol{P}(\boldsymbol{x}_t \mid \boldsymbol{e}_{1:t})$ is the output of the filtering operation up to time t, and is also referred to as the forward message, and denoted as $\boldsymbol{f}_{1:t}$. The output of the equation below is the updated belief state, in our example, a probability that the agent is located in each open square:

$$\boldsymbol{P}(\boldsymbol{X}_{t+1} \mid \boldsymbol{e}_{1:t+1}) \\ = \alpha \boldsymbol{P}(\boldsymbol{e}_{t+1} \mid \boldsymbol{X}_{t+1}) \sum_{x_t} \boldsymbol{P}(\boldsymbol{X}_{t+1} \mid \boldsymbol{x}_t) \boldsymbol{P}(\boldsymbol{x}_t \mid \boldsymbol{e}_{1:t})$$

B. Hidden Markov Models

A *hidden Markov model* (HMM) is a temporal probabilistic model as described in the section above. The principal advantage of this model is that the state is described as a single discrete random variable, and it allows for the matrix implementation described below first proposed by

C. Matrix Implementation

There is a matrix implementation of the forward-backward algorithm introduced by Baum [1]. This can be used to calculate the filtered estimate of probability. The matrix implementation uses two matrices. $\mathbb{T}$, the transition matrix, and $\mathbb{O}_t$, the sensor matrix at time t. Given S states, the matrix $\mathbb{T}$ is an $S\ x\ S$ matrix that describes the probability of transitioning from state i to state j. It is defined as $\boldsymbol{P}(\boldsymbol{X}_{t+1} = j \mid \boldsymbol{X}_t = i) = \mathbb{T}_{ij}$.

The sensor matrix $\mathbb{O}_t$ is also an $S\ x\ S$ matrix and it describes the probability that each of the S states, denoted as i could generate the observed evidence. It is defined as $\boldsymbol{P}(\boldsymbol{E}_t = \boldsymbol{e}_t \mid \boldsymbol{X}_t = i)$.

Using the initial belief state, $\boldsymbol{P}(\boldsymbol{X}_0)$ to start the process as $\boldsymbol{f}_{1:0}$, the belief state can be updated through filtering using the following equation, where $\mathbb{T}^{T}_{i,j} = \mathbb{T}_{j,i}$:

$$\boldsymbol{f}_{1:t+1} = \alpha \mathbb{O}_{t+1} \mathbb{T}^{\mathrm{T}} \boldsymbol{f}_{1:t}$$

The new forward message $\boldsymbol{f}_{1:t+1}$ is the filtered belief state incorporating the new evidence. In our example, this is a vector of probabilities that the agent is in each open square.

IV. Implementation and Results

A. Description

The project was implemented using Visual Studio Code, Angular 7, TypeScript, and Canvas graphics. The code is stored in GitHub at the following URL: https://github.com/jacob-barna/MarkovSensorLocation.

Implementation of a hidden Markov Model

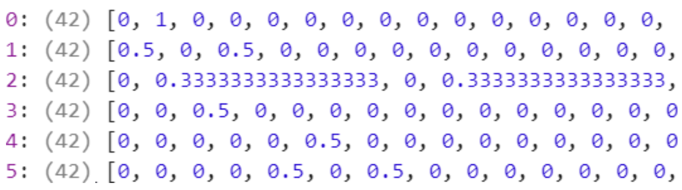
A hidden Markov model (HMM) has been implemented to model the belief state of an intelligent agent. The model of the belief state consists of one discrete variable which describes the possible locations that the agent could occupy, along with their respective probabilities, in an $m\ x\ n$ grid with obstacles. The model will be used to examine the evolution of the belief state of the agent as the agent receives more sensor readings.

As previously discussed, the HMM consists of an initial belief state, a transition model, and a sensor model. The initial belief state $\boldsymbol{P}(\boldsymbol{X}_0)$ in this implementation assigns an equal probability to all open squares in the grid. That is, if there are n open squares, then the probability that the agent is on any square is $1/n$.

The transition model is defined next. Assume that the set $NEIGHBORS$ is defined as all open squares adjacent to the current square. Then the robot can move to any square in $NEIGHBORS$. Let $N(i)$ be the size of the $NEIGHBORS$ set for square i then the transition matrix is defined as:

$$\boldsymbol{P}(\boldsymbol{X}_{t+1} = j \mid \boldsymbol{X}_t = i) = \mathbb{T}_{ij} = (\tfrac{1}{N(i)} \ if \ j \in NEIGHBORS(i) \ else \ 0)$$

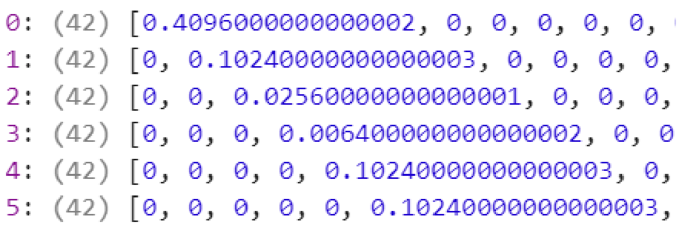
The screenshot below shows the transition matrix for the grid shown in figure 4 below. If we number the squares from top-left to bottom-right starting with 0, one can see that square 0 has obstacles on all sides except to the East. This is shown in row 0 of figure 1 because the only possible transition for the agent is to move to square 1, thus $\mathbb{T}_{0,1} = 1$. Similarly, an agent in square 2 can move with equal probability to square 1, square 3, and square 13 making $\mathbb{T}_{2,1} = \mathbb{T}_{2,3} = \mathbb{T}_{2,13} = \frac{1}{3}$.



```
0: (42) [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
1: (42) [0.5, 0, 0.5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
2: (42) [0, 0.3333333333333333, 0, 0.3333333333333333,
3: (42) [0, 0, 0.5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
4: (42) [0, 0, 0, 0, 0, 0.5, 0, 0, 0, 0, 0, 0, 0, 0, 0
5: (42) [0, 0, 0, 0, 0.5, 0, 0.5, 0, 0, 0, 0, 0, 0, 0,
```

Figure 1 - A portion of the 42x42 transition matrix for the grid shown in figure 4.

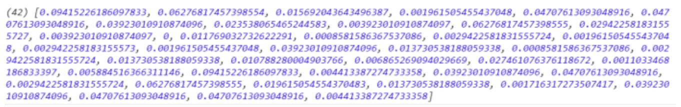
For the sensor model, let d_{it} be the number of directions in which the current sensor reading differs from the true reading. Then the sensor model is defined as $\boldsymbol{P}(\boldsymbol{E}_t = \boldsymbol{e}_t \mid \boldsymbol{X}_t = i) = \mathbb{O}_{ii} = (1 - \epsilon)^{4-d_{it}}\epsilon^{d_{it}}$. Recall that the sensor model is intended to show the probability that a given state can generate evidence. It is then noteworthy to point out that $(1 - \epsilon)^{4-d_{it}}$ is the probability that the sensor is right in $4 - d_{it}$ directions whereas $\epsilon^{d_{it}}$ is the probability of getting d_{it} bits wrong. A screenshot of a portion of a sensor matrix for a $42 \ x \ 42$sensor matrix is shown below. The probability in $\mathbb{O}_{0,0}$ shown in the screenshot corresponds to the probability that the reading NSW is received by the sensor when NSW is the true reading, given a sensor error rate of $\epsilon = 0.2$. In this case, $d_{it} = 0$ because the sensor does not differ from the true reading, making the value in $\mathbb{O}_{0,0} = (1 - 0.2)^4 0.2^0 = 0.8^4 = 0.4096$. Note that the value computed here differs slightly from the screenshot due to the use of JavaScript floating point numbers. This small imprecision is not addressed in this paper, although libraries exist which could be used to address this issue [5].



```
0: (42) [0.4096000000000002, 0, 0, 0, 0, 0,
1: (42) [0, 0.10240000000000003, 0, 0, 0, 0,
2: (42) [0, 0, 0.025600000000000001, 0, 0, 0,
3: (42) [0, 0, 0, 0.006400000000000002, 0, 0
4: (42) [0, 0, 0, 0, 0.10240000000000003, 0,
5: (42) [0, 0, 0, 0, 0, 0.10240000000000003,
```

Figure 2 - Portion of a 42 x 42 sensor matrix, note the probabilities of the sensor readings for state i are on the coordinate (i,i)

As mentioned prior, knowing the structure of the transition and sensor matrices $\mathbb{T}$ and $\mathbb{O}$, the filtered belief state can then be obtained by $\boldsymbol{f}_{1:t+1} = \alpha\mathbb{O}_{t+1}\mathbb{T}^{\mathrm{T}}\boldsymbol{f}_{1:t}$. $\mathbb{T}^T$ is the transpose of the matrix. The simulated agent uses this equation to update its belief state based on the current precept. The updated belief state is modeled as a S-element vector:



```
(42) [0.09415226186097833, 0.06276817457398554, 0.015692043643496387, 0.001961505455437048, 0.04707613093048916, 0.047
07613093048916, 0.03923010910874096, 0.023538065465244583, 0.003923010910874097, 0.06276817457398555, 0.02942258183155
5727, 0.003923010910874097, 0, 0.011769032732622291, 0.0008581586367537086, 0.0029422581831555724, 0.00196150545543704
8, 0.002942258183155573, 0.001961505455437048, 0.03923010910874096, 0.013730538188059338, 0.0008581586367537086, 0.002
9422581831555724, 0.013730538188059338, 0.010788280004903766, 0.006865269094029669, 0.027461076376118672, 0.0011033468
186833397, 0.005884516366311146, 0.09415226186097833, 0.004413387274733358, 0.03923010910874096, 0.04707613093048916,
0.0029422581831555724, 0.06276817457398555, 0.019615054554370483, 0.013730538188059338, 0.001716317273507417, 0.039230
10910874096, 0.04707613093048916, 0.04707613093048916, 0.004413387274733358]
```

Figure 3 - a 42-element matrix representing the belief state (also known as forward message) of the agent in an environment with 42 possible states

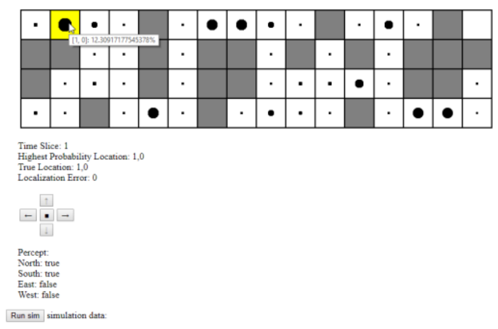
Simulated Agent and Environment

A model-based reflex agent has also been implemented. The agent uses the HMM developed to update its internal belief state based on the current sensor readings and the recent history of the sensor readings. Sensor readings consist of 4 bits that signify whether an obstacle is detected in the North, South, East, or West direction. This agent interacts with an $m \ x \ n$ grid environment that that contains some number of non-occupiable squares (obstacles).

Analysis Tools

One goal of this research is to clearly visualize the evolution of our model-based reflex agent's belief state over time given varying sensor rates and a varying number of discrete sensor readings (time slices).

In pursuance of this goal, a UI has been developed consisting of five parts: i.) a graphical representation of the belief state of the agent, ii.) diagnostic information, iii.) navigation controls, iv.) percept readings, and v.) a simulation feature. Each of these numbered parts is shown in the screenshot below, which shows the probability distribution $\boldsymbol{P}(\boldsymbol{X}_1 | \boldsymbol{E}_1 = NSW)$.



Time Slice: 1
Highest Probability Location: 1,0
True Location: 1,0
Localization Error: 0

Percept:
North: true
South: true
East: false
West: false

simulation data:

Figure 4 - Graphical representation of belief state at time t=1 with the probability distribution P(X_1 | E_1=NSW)

The graphical representation of the belief state, as shown in figure 4, consists of several features. Obstacles are represented as grey squares in contrast to occupiable spaces, which are white squares. The belief state of the agent is shown as a circle within each occupiable square, with the radius of the circle corresponding to the probability that the agent believes it is within the square. The precise probability is displayed by a tooltip shown when a coordinate is hovered over. This representation also displays the true state. This is displayed as a convenience to the user as the true state is unknown by the agent and the underlying hidden Markov model.

Graphics are rendered using a Canvas 2d Rendering Context. The Canvas context is wrapped in an angular component with the input bindings necessary to render the grid, namely:

- the dimensions of the grid
- the number of obstacles (and the placement of the obstacles)
- the location of the agent
- the belief state of the agent

The diagnostic information section displays data that changes as the robot navigates and consequently collects more percepts. The time is incremented with each observation and the most likely location is displayed next to the true location. In addition, the localization error, in our model, the Manhattan distance is displayed. Given perceived location is point $\mathrm{p} = (\mathrm{p1}, \mathrm{p2})$ and the true location is point $\mathrm{q} = (\mathrm{q1}, \mathrm{q2})$, then the Manhattan distance is defined as: $|\mathrm{p}_1 - q_1| + |p_2 - q_2|$ [2].

The navigation section consists of four arrow keys, each of which can be pressed to move the agent in the indicated direction. In addition to moving the agent, the navigation buttons increment the time slice. At each time slice, the agent gathers another percept with randomly generated error and updates the belief state using the hidden Markov model implementation. Lastly, the grid is re-rendered with the updated belief state.

The analysis tools are also equipped to show the sensor readings at each step. At each compass direction, the sensor reading is displayed and can be contrasted with the true sensor readings visible on the grid. For example, in figure 4, the true sensor readings match the sensor readings received by the agent. Comparing the sensor readings and their variance from the true readings show how the agent updates its belief state. For example, if the readings in figure 4 had been North: true, South: True, East: false, and West: true, the agent would have assigned more probability to the upper left corner of the grid, as that grid square has a higher probability of generating these sensor readings.

The last analysis tool is a simulation runner. This consists of a button which activates a simulation, and a section to output the data which can be copied into a spreadsheet for analysis. The simulation runner just causes the agent to move in the same predefined path on every simulation run while the runner collects the localization error at each time slice to be output on the screen at the completion of the simulation.

B. *System Architecture*

The system is a Single Page Application (SPA) written using the Angular JavaScript framework and TypeScript. Two Angular components have been created along with supporting Typescript classes and enumerated types. The overall system architecture is shown in figure 5.

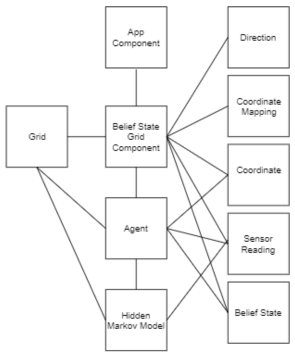


Figure 5 - Implementation Classes

1) App Component

App Component is the entry point of the Single Page Application. This component exists solely to host the Belief State Grid Component, which is the main component in the application.

2) Belief State Grid Component

The Belief State Grid Component is the main component in the system. This component ties several parts of the system together and orchestrates interaction between the remaining parts of the application to perform several duties:

- Rendering the belief state of the agent as a grid,
- Agent navigation, which consists of several sub-tasks
    - Tracking time,
    - Updating the belief state,
    - Re-rendering the grid with the updated belief state
- Rendering diagnostic information,
- Running simulations

Rendering the belief state of the grid consists is achieved using an HTML canvas element. The belief state grid accepts inputs for: i.) number of rows, ii.) number of columns, and iii.) the location of obstacles and renders the appropriate grid. The component also tracks the true location of the agent and renders the true location as a yellow grid square to provide an easily visible indication of the quality of the agent's belief state. The component also tracks mouse movement and renders a tooltip with the coordinate and probability that the agent is in the location over which the mouse is hovering.

The Belief State Grid Component has an Agent as a private property of the component. This Agent contains a belief state and a mechanism by which to update its belief state. The Belief State Grid Component uses these mechanisms in Agent navigation. When an agent move is requested, the grid uses the current true location of the agent and an input "Direction" enumerated type parameter to determine if a move in the requested direction is legal. If the move is legal, then the true location of the agent is updated. The time is also incremented one step in the future, and the Agent belief state is updated. Finally, the grid is re-rendered using the new belief state and true location of the agent. The diagnostic information, i.e. sensor localization error, is also updated.

As already mentioned, there are several diagnostics available at each time slice: the time slice itself, the highest probability location, the true location, and the localization error. These are displayed by the grid using two-way data binding provided by the Angular framework. In other words, as the belief state is updated, the Belief State Grid Component simply updates the values stored in variables used in the diagnostic section, and the Angular framework renders the changes on the UI.

The last implementation detail of the Belief State Grid Component is the simulation feature. This feature is implemented by performing a scripted set of 40 moves and outputting the Manhattan distance between the highest probability location and the true location of the agent. The UI control that enables a simulation to run is just an HTML button. The click handler for this button runs the scripted set of moves and the screen is updated with the localization error at each step by two-way data binding. These step-wise localization error values are easily copied to other tools--such as Excel--for analysis.

3) Agent

The Agent class is responsible for holding and updating the belief state of the agent. To construct an agent, the caller must provide and error rate parameter as well as information about the environment.

The mechanism by which the agent updates the belief state is a hidden Markov model, which is kept as a private class property and initialized in the constructor. The Agent class also initializes the belief state of the agent when the agent comes on-line. The initial belief state is that the agent can be in any open square with equal probability.
The Agent class also exposes a method to get a percept, or generate sensor readings, based on the true location of the agent and the error rate given in the constructor. The agent will return the wrong sensor reading in a given direction with probability equal to the error rate and return a "SensorReading" object to the caller.

Another public method available to callers is the "update" method which takes a single "SensorReading" parameter. In this implementation, the reading is passed to the hidden Markov model to update the belief state. The hidden Markov model returns a vector of size n, where n is the number of occupiable spaces, so the Agent class must first transform this vector to a "BeliefState" object, which is an object that contains a coordinate and a probability. This object is used by the Belief State Grid Component to draw the grid and tooltips used to represent the belief state visually.

4) Hidden Markov Model

The hidden Markov model is the mechanism by which the Agent class updates its internal belief state. Because there exist other means of updating belief state, such as Dynamic Bayesian Networks, which would allow the system "to learn much larger models much faster, without requiring manual segmentation of the training data" [10, pp. 1], the agent is designed to make it easy to swap to other inference models without impacting other code. This would be done by replacing the hidden Markov model with another model in the constructor and in the update step. All other system code would remain untouched.

The hidden Markov model holds as private properties the transition matrix and sensor matrix represented as matrices using the Math.js library [6]. The Math.js library API is used to reduce the implementation time as it supports multiplication and transpose operators, as well as an easy way to create diagonal matrices. Each of these API methods are required in the implementation of the hidden Markov model.

Upon construction, the hidden Markov model initializes the transition matrix based on the environment (grid) passed into the constructor. This is achieved by iterating through occupiable coordinates in the grid and assigning equal probability to each occupiable neighboring grid square. In addition, the initial belief state, $\boldsymbol{P}(\boldsymbol{X}_0)$, is initialized as an n-element vector with each element assigned the probability of $1/n$. These two steps are prerequisite in enabling the agent to update the belief state.

The hidden Markov model exposes only one public method which updates the belief state based on a "SensorReading." This sensor reading is the percept of the agent at the current time slice, which, as described above, can be faulty. The method initializes the sensor matrix, then computes the forward message:

$$\boldsymbol{f}_{1:t+1} = \alpha \mathbb{O}_{t+1} \mathbb{T}^{\mathrm{T}} \boldsymbol{f}_{1:t}$$

Recall that the $\mathbb{O}$ matrix is the sensor matrix and the $\mathbb{T}$ matrix is the transition matrix, and that $\boldsymbol{f}_{1:t}$ at time 0 is the initial belief state $\boldsymbol{P}(\boldsymbol{X}_0)$, which is initialized in the constructor. The steps in updating the belief state then follow from the equation. The steps are: i.) initialize the sensor matrix at the time slice $t+1$ using the observation just received by the agent, ii.) multiply this sensor matrix by the transpose of the forward message at time slice t, initially $\boldsymbol{P}(\boldsymbol{X}_0)$, and iii.) normalizing the vector so that the probabilities sum to 1.

Recall that the sensor matrix is equivalent to the probability of receiving the given sensor reading based on the current state: $\boldsymbol{P}(\boldsymbol{E}_t = \boldsymbol{e}_t \mid \boldsymbol{X}_t = i) = \mathbb{O}_{ii} = (1 - \epsilon)^{4-d_{it}} \epsilon^{d_{it}}$, where ϵ is the error rate, and that d_{it} can be thought of the Hamming distance between the perceived sensor readings and the true sensor readings if the sensor readings are represented as bits. Given the binary string representing the true sensor reading α and the binary string representing the perceived sensor reading β, then the Hamming distance is $\alpha\ XOR\ \beta$. Finally recall that for an n states, the sensor matrix is of size $n\ x\ n$. The steps in initializing the sensor matrix follow from these definitions. First the true sensor reading of each occupiable coordinate is converted into vector form. Each vector entry is then mapped to the value $(1 - \epsilon)^{4-d_{it}} \epsilon^{d_{it}}$. Finally, the vector is used to create an $n\ x\ n$ diagonal matrix using the Math.js diagonal method – which, given an n element vector, returns a matrix with the vector on the diagonal.

Continuing with the calculation of the forward message, the transpose of the transition matrix is calculated using the Math.js transpose method which returns a matrix with the values of the transition matrix reflected over its main diagonal. This transposed matrix is multiplied with the sensor matrix calculated above. This results in an n element vector representing the belief state before normalization.

The final step in calculating the forward message is to normalize the vector resulting from the multiplication described in the preceding paragraph. This is achieved by summing the vector values into s and then mapping each vector element with value v to the value v/s. As mentioned in a prior section, in this implementation of the hidden Markov model, these operations are performed using JavaScript floats, and the resulting normalized vector can sometimes add up to slightly over or under 1. For any production use of a hidden Markov model, this imprecision should be addressed.

The final step in updating the belief state is to update the forward message held as a private property by the agent. This represents the agent's filtered belief state at the current time slice which will be needed if the agent is asked to calculate the filtered estimate of the next time slice.

5) Grid

The Grid class is used to represent the environment that the agent is contained within. Upon construction, the grid requires that the number of columns, the number of rows, and the coordinates of any obstacles are provided. The public properties exposed by this class are the number of columns, the number of rows, the location of the obstacles, and the occupiable coordinates. These properties are used by the other components for rendering the grid and to quickly convert the environment to vector form for use in inference tasks.

The grid also exposes the public methods that given a coordinate, return true or false if the coordinate: is an obstacle, is in the first row, is in the last row, is in the first column, or is in the last column. These methods are used in determining if a requested move is legal as well as in determining the true sensor reading of the square that the agent occupies. This latter use is integral to the calculation of the sensor matrix described in the hidden Markov model implementation. Rather than forcing the hidden Markov model to determine the true environment sensor reading, another public method is exposed. This method will return the true environment sensor reading (using the helper methods described above) given any grid coordinate.

The Grid class also exposes a public method which returns the occupiable neighbors of a given coordinate. This is used by the hidden Markov model to create the transition matrix based on the grid environment passed to the model.

6) Direction

Direction is an enumerated type used in navigation created for code readability. It is used as a parameter or for equality checks in various methods dealing with navigation. The values that can be specified using this enumerated type are: Up, Down, Left, Right.

7) Coordinate Mapping

This interface is used to map the browser coordinates of the mouse pointer to a grid coordinate. This is used in handling mouse move events. When the mouse is moved into a grid coordinate, this mapping is referenced to display the correct tooltip with the grid coordinate and probability that the agent believes it occupies that grid coordinate.

8) Coordinate

Coordinate is simply a custom type designed to represent an (x, y) coordinate, whether that be a mouse coordinate, or a grid coordinate. This type is used throughout the system.

9) Sensor Reading

This is a class representing the true or perceived sensor readings. Each direction, North, South, East, and West is represented as a Boolean value. This class is used to pass information about the percepts of the agent and the environment between the agent, grid, and hidden Markov model.

10) Belief State

This class represents the belief state of the agent for a coordinate. It exists merely to tie a coordinate with a probability. An array of Belief State objects represents the internal belief state of the agent. This class is used in the agent as well as in the grid, where it is used as the source of data in the Belief State Grid Component which renders the belief state visually.

C. *Experiment Design*

To show the usefulness of the application developed in this paper, two experiments were designed to show the effectiveness of the hidden Markov model for agent localization in two different environments. For each environment, the localization error is averaged over 7 runs for varying error rates. Table 1 shows the error rates and number of runs that the localization error is averaged over.

Table 1 - Experiment Design

Environment 1	
Error Rate	Number of Runs
5%	7
10%	7
20%	7
Environment 2	
Error Rate	Number of Runs
5%	7
10%	7
20%	7

1) Environment 1

The environment of the first experiment is a grid with 16 columns and 4 rows. Of this grid, 42 squares are occupiable. The remaining squares are obstacles placed so that the resulting grid resembles a maze.

A set path of 40 moves is to be used in each run of the experiment. The path is shown in figure 6. Note that the color of the path line changes when the agent must reverse and loop back on its prior path.

The initial placement of the agent is in the top left corner of the grid, highlighted in yellow in figure 6.

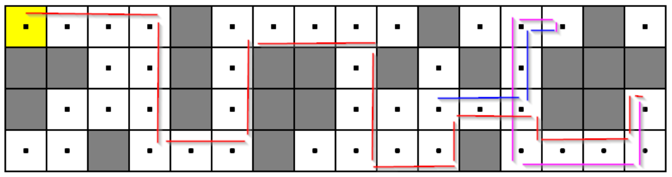


Figure 6 - Experiment 1-Environment and Path

2) Environment 2

The second experiment is conducted in a grid with 64 columns and 1 row.

A set path is used on each run. This path begins with moving right 3 squares, left 3 squares, right 12 squares, and continues navigating a similarly defined path for a total of 40 moves. A partial representation of this grid is shown in figure 7 without path illustration.

The initial placement of the agent in this experiment follows experiment 1 in that the agent is initially placed in the top left square of the grid.

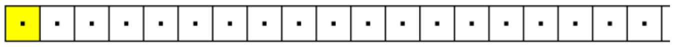


Figure 7- Experiment 2 - Environment (partial representation)

D. *Experiment Results*

The results of experiment 1 are shown in table 2 and graphically in figure 8.

With an error rate of 20%, the Manhattan distance varies between 2 and 8 on each time slice. With sensor error rate of 10%, the localization error starts around 3 and drops quickly to less than 2 for the remainder of the experiment. With sensor error rate of 5%, the localization error again spikes at around a Manhattan distance of 3 and then falls to close to 0 for the remainder of the experiment.

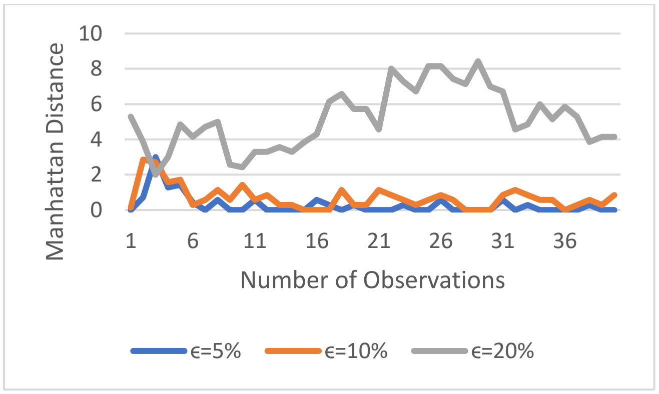


Figure 8 - Experiment 1 Results

The results of experiment 2 are shown in table 3 and graphically in figure 9. The graphical representation shows that in environment 2, the performance of the hidden Markov model was particularly poor, with the sensor rate staying almost constant during the entire experiment for any error rate.

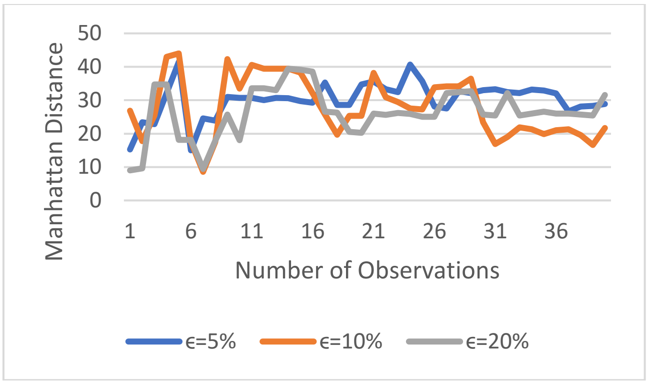


Figure 9- Experiment 2 Results

Table 2 - Results of Experiment 1

Avg. Manhattan Distance			
Time Slice	Error Rate 5%	Error Rate 10%	Error Rate 20%
1	0.0000	0.1429	0.3810
2	0.7143	2.8571	1.8571
3	3.0000	2.7143	2.9048
4	1.2857	1.5714	2.2857
5	1.4286	1.7143	2.7143
6	0.4286	0.2857	2.2381
7	0.0000	0.5714	2.5238
8	0.5714	1.1429	3.2381
9	0.0000	0.5714	3.1905
10	0.0000	1.4286	3.8095
11	0.5714	0.5714	4.0476
12	0.0000	0.8571	4.2857
13	0.0000	0.2857	4.4286
14	0.0000	0.2857	4.7619
15	0.0000	0.0000	5.0000
16	0.5714	0.0000	5.5238
17	0.2857	0.0000	5.7619
18	0.0000	1.1429	6.3810
19	0.2857	0.2857	6.5238
20	0.0000	0.2857	6.7619
21	0.0000	1.1429	7.3810
22	0.0000	0.8571	7.6190
23	0.2857	0.5714	7.9524
24	0.0000	0.2857	8.0952
25	0.0000	0.5714	8.5238
26	0.5714	0.8571	9.1429
27	0.0000	0.5714	9.1905
28	0.0000	0.0000	9.3333
29	0.0000	0.0000	9.6667
30	0.0000	0.0000	10.0000
31	0.5714	0.8571	10.8095
32	0.0000	1.1429	11.0476
33	0.2857	0.8571	11.3810
34	0.0000	0.5714	11.5238
35	0.0000	0.5714	11.8571
36	0.0000	0.0000	12.0000
37	0.0000	0.2857	12.4286
38	0.2857	0.5714	12.9524
39	0.0000	0.2857	13.0952
40	0.0000	0.8571	13.6190

Table 3 - Results of Experiment 2

Avg. Manhattan Distance			
Time Slice	Error Rate 5%	Error Rate 10%	Error Rate 20%
1	15.2857	26.8571	9.0000
2	23.4286	17.7143	9.5714
3	22.8571	25.4286	34.7143
4	32.1429	43.0000	34.7143
5	41.4286	44.0000	18.1429
6	15.0000	17.5714	18.1429
7	24.5714	8.5714	9.4286
8	23.8571	17.4286	17.8571
9	31.0000	42.2857	25.7143
10	30.7143	33.4286	18.0000
11	30.7143	40.5714	33.5714
12	30.0000	39.4286	33.5714
13	30.7143	39.4286	33.0000
14	30.5714	39.4286	39.4286
15	29.7143	38.2857	39.1429
16	29.2857	32.1429	38.5714
17	35.2857	25.7143	26.5714
18	28.5714	19.7143	26.2857
19	28.5714	25.2857	20.5714
20	34.7143	25.2857	20.2857
21	35.5714	38.1429	26.0000
22	33.2857	30.8571	25.5714
23	32.4286	29.4286	26.1429
24	40.7143	27.5714	25.8571
25	35.7143	27.2857	25.0000
26	28.1429	33.8571	25.0000
27	27.5714	34.1429	32.1429
28	32.7143	34.1429	32.4286
29	32.1429	36.4286	32.7143
30	33.0000	23.4286	25.7143
31	33.2857	16.8571	25.4286
32	32.4286	19.0000	32.1429
33	32.1429	21.8571	25.4286
34	33.1429	21.2857	26.0000
35	32.8571	19.8571	26.5714
36	32.0000	21.0000	26.0000
37	26.7143	21.2857	26.0000
38	28.1429	19.5714	25.7143
39	28.2857	16.5714	25.4286
40	28.8571	21.7143	31.5714

E. Discussion

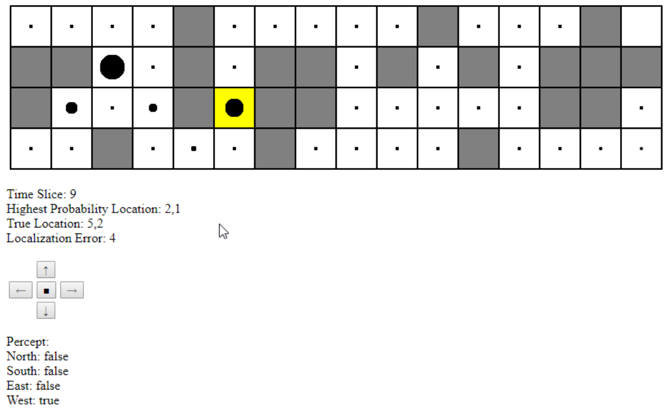
1) Experiment 1

It is evident in figure 8 that with error rate of 5-10%, the agent tracks the true location rather well with localization error staying less than 2 after the first few observations. With an error rate of 20%, the agent did not fare so well.

This can be partially explained by the error rate. The agent can only expect to have all sensor reading correct 40.96% of the time. This can be seen by using the formula for the sensor matrix and setting the Hamming distance, d_{it}, between the perceived reading and the true reading to 0. Recall that the sensor matrix formula is, $(1 \ - \epsilon)^{4-d_{it}}\epsilon^{d_{it}}$.

When $d_{it} = 0$ and $\epsilon = 0.2$, we have $(1 - 0.2)^4 0.2^0 = 0.4096$. Despite this severe limitation, the agent still maintains an average Manhattan distance of 5.175 when the average distance at each time slice is averaged over all runs.

The relatively poor performance can also be seen more clearly by the example shown in figure 10. In this situation, the agent has received a sensor reading that the only obstacle lies to the West when there are obstacles to the East and West. This leads the agent to believe, based on the sensor readings it received in the past (the forward message), as well as the current sensor and transition models, that it is in location (2, 1). Note that this location has an obstacle to the West, but no obstacle to the East, which matches the sensor reading for this time slice.


Time Slice: 9
Highest Probability Location: 2,1
True Location: 5,2
Localization Error: 4

Percept:
North: false
South: false
East: false
West: true

Figure 10- P(X_9 | e_1:9) – a faulty sensor reading has caused the agent to estimate a location with Manhattan distance of 4 from the true location

Another interesting observation from Experiment 1 is that the starting location appears to have the effect of increasing the accuracy of the early updates to the belief state. This may be due to the geography of the starting location. Note that there are relatively few grid cells that have an obstacle on the North, South, and West. If the agent receives an accurate sensor reading in the starting location, which will happen with a probability of 40.96%, then the agent starts out with accurate belief states in the early stages of the experiment.

1) Experiment 2

From figure 9, it is clear that the agent in Experiment 2 was not able to work out its location with any degree of accuracy. This is because the sensor readings are not providing any useful information [8, pp. 32, 9, pp. 145]. The topography of this environment has only two interesting points – if we are in either the first or last cell, we have the reading NSW or NSE. All other cells will generate the reading NS. This means that the agent can never shrink the pool of probable locations unless it believes it is in two of the 64 occupiable grid cells.
The results of experiment 2 could likely be made worse by starting the agent in a location somewhere in the middle of the corridor.

F. Future Work

Future work will concentrate on implementing the Viterbi algorithm, an algorithm which can find the most likely sequence of locations that lead the agent to the current location. This allows the agent to form a belief about the location it started from.

In support of implementing the Viterbi algorithm, the starting location of the agent should also be randomized. An alternative to randomizing the starting location could be to allow a user to specify the original location. This would also lead to more interesting experiments, such as determining how much affect starting location has on the early updates of the belief state and the performance of the hidden Markov model overall.

V. Conclusion

In this paper we have implemented an interactive, web-based simulator for agent localization. This simulator shows a graphical representation of the belief state of an agent and allows a user to study how topology of the environment, movement within the environment, and sensor error rates (and resulting readings) can affect the evolution of the belief state of the agent.

The simulator implemented in this paper supported the main goal of the paper to study the effects of sensor error rate on agent localization. In this paper, and the simulator, we studied an agent in a partially-observable environment using the temporal probabilistic hidden Markov model as the underlying mechanism to provide filtered estimates based on sensor observations. As can be seen by the results, the hidden Markov model is rather effective with moderate sensor error rates. In addition, the model is also able to provide fair estimates in the face of high error rates, such as 20%, where at least one of the four sensor readings are wrong 59.04% of the time.

The simulator also showed some of the limitations of the hidden Markov model. Given an environment where the topology provides very little information, such as a $1\ x\ 64$ grid, the hidden Markov model is very ineffective in agent localization.

We also can note that starting location in combination with the topology of the environment may influence the accuracy of the perceived location. Analysis of this effect is left to future work.

References

[1] Baum, Leonard E.. "An Inequality and Associated Maximization Technique in Statistical Estimation for Probabilistic Functions of Markov Processes." Paper presented at the meeting of the Inequalities III: Proceedings of the Third Symposium on Inequalities, University of California, Los Angeles, 1972.

[2] Black, Paul. 2019. "Manhattan Distance." n.d. Accessed April 15, 2019. https://xlinux.nist.gov/dads/HTML/manhattanDistance.html.

[3] Ghahramani, Zoubin. 2002. “Hidden Markov Models.” In , 9–42. River Edge, NJ, USA: World Scientific Publishing Co., Inc. http://dl.acm.org/citation.cfm?id=505741.505743.

[4] Kolmogorov, Andrei Nikolaevitch. "Stationary sequences in Hilbert space." Bull. Math. Univ. Moscow 2, no. 6 (1941): 1-40.

[5] M, Michael. (2012) 2019. *A JavaScript Library for Arbitrary-Precision Decimal and Non-Decimal Arithmetic : MikeMcl/Bignumber.Js*. JavaScript. https://github.com/MikeMcl/bignumber.js.

[6] “Math.Js | an Extensive Math Library for JavaScript and Node.Js.” n.d. Accessed April 19, 2019. https://mathjs.org/.

[7] McCarthy, John. 1995. “Computation & Intelligence.” In , edited by George F. Luger, 479–492. Menlo Park, CA, USA: American Association for Artificial Intelligence. http://dl.acm.org/citation.cfm?id=216000.216028.

[8] Precup, Doina. 2017. “Lecture 9: Hidden Markov Models” n.d. Accessed April 19, 2019. https://www.cs.mcgill.ca/~dprecup/courses/ML/Lectures/.

[9] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009, pp. 34, pp. 145.

[10] Theocharous, G., K. Murphy, and L. P. Kaelbling. 2004. “Representing Hierarchical POMDPs as DBNs for Multi-Scale Robot Localization.” In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA ’04. 2004*, 1:1045-1051 Vol.1. https://doi.org/10.1109/ROBOT.2004.1307288.

[11] Wiener, Norbert. 1977. *Time Series*. 5. print. MIT 9. Cambridge/Mass: MIT Pr.