Hidden Markov Models with Forward Filtering - Project 3, EDAF70

Ola Johansson (jur10ojo@student.lu.se)
Niklas Lundström (tpi13nlu@student.lu.se)

March 2, 2018

1 Introduction

The problem at hand is the localization of a robot in a room, using an imperfect sensor. The task is performed in software where the world consists of a two-dimensional grid and movements are discrete steps in the grid. The sensor indicates the correct position in 10 % of cases, the closest neighbors in 5 % of cases (per neighbor), and the second neighbors in 2.5 % of cases (per neighbor). The remaining sensor values are blank i.e. the postion was not sensed. In addition to this the robot moves according to a set of rules. When not heading into a wall the robot has a 70 % chance of maintaining its course, the remaining probability is equaly distributed among the remaining adjacent cells. When heading into a wall the robot always changes course, and the probability of heading is equally distributed between the eligible adjacent neighbors.

Hidden Markov Models The basic approach to solving this problem will be that of *Hidden Markov Models* (HMM) and *Forward filtering*. The general idea behind HMM is *Markov Chains* which are used to represent states which are determined by a limited number of preceding states. In the case of HMM these states are not observable, they are hidden. Instead these states affect observation states from which the actual state can be determined using the laws of probability.

Forward Filtering Filtering, specifically forward filtering, calculates the probability of a state at time $t+1$, given all the evidence up until time $t+1$:

$$\boldsymbol{P}(\boldsymbol{X}_{t+1}|\boldsymbol{e}_{1:t+1}) = f(\boldsymbol{e}_{t+1}, \boldsymbol{P}(\boldsymbol{X}_t|\boldsymbol{e}_{1:t})) \tag{1}$$

This creates a recursive definition of the forward function f. This in principle means that the the prior distribution is projected forward to the next state and is combined with the new evidence $\boldsymbol{e}_{t+1}$.

2 Methods

From (1) it is possible to derive a more precise algorithm for the forward filtering:

$$\boldsymbol{P}(\boldsymbol{X}_{t+1}|\boldsymbol{e}_{1:t+1}) = \alpha \boldsymbol{P}(\boldsymbol{e}_{t+1}|\boldsymbol{X}_{t+1}) \sum \boldsymbol{P}(\boldsymbol{X}_{t+1}|\boldsymbol{x}_t)\boldsymbol{P}(\boldsymbol{x}_t|\boldsymbol{e}_{1:t}) \tag{2}$$

Where $\boldsymbol{X_{t+1}}$ is the sought state, $\boldsymbol{e_{t+1}}$ is the most recent observation. $\boldsymbol{x_t}$ is the prior state, and $\boldsymbol{e_{1:t}}$ is the accumulated evidence. This algorithm requires knowledge of the sensor model:

$$\boldsymbol{P}(\boldsymbol{e}_{t+1}|\boldsymbol{X}_{t+1}), \tag{3}$$

and the transition model:

$$\boldsymbol{P}(\boldsymbol{X}_{t+1}|\boldsymbol{x}_t). \tag{4}$$

Finally, an initial value for $\boldsymbol{P}(\boldsymbol{x}_t|\boldsymbol{e}_{1:t})$ for $t = 0$, is needed, which can simply be set to an equal distribution over all possible positions (in this case).

The sensor model is based on the specified behavior in the task. Which means that (3) is completely defined for any observation given any state. The definition of the sensor behavior entails that a sensor value of "nothing" will be more likely when the robot is close to a wall, due to the reduction in number of neighbors, and the redistribution of probability from these "illegal" neighbors, to "nothing". So desipte not receiving a sensor value for a given time, the algorithm will still be able to run, using a probability distribution favoring positions in the corners, and close to walls.

The transition model (4) is based on the robots possible movements. From each position the robot can move in the four directions: north, east, south, and west. It does so with different probabilities given different positions, as described in the introduction. This means that it is possible to generate data for all the possible next states from any given state, where a state defined as the robots position and heading.

Transition and Observation Matrices The probabilities discussed above can be stored in matrices which can then be used to perform basic algebraic operations in order to calculate the result of the forward filtering and obtain the desired probabilities. The transition model values can be stored in the transition matrix, which has size $(rows \cdot cols \cdot headings) \times (rows \cdot cols \cdot headings)$ where each field contains the probability to move from a specific position and heading, to another specific position and heading. The observation matrices on the other hand contain the data from the sensor model. Each observation matrix is a diagonal matrix with diagonal values representing the probability of a given observation given all the $(rows \cdot cols \cdot headings)$ possible states. In order to represent all the possible observations $(rows \cdot cols + 1)$ observation matrices are needed, one for each possible observation, including "nothing".

Matrix operations After calculating the values of these matrices and initializing the probability matrix which is referred to as *alpha*, it is a not complicated to calculate the new alpha value, using matrix operations based on (2):

$$\boldsymbol{f}_{1:t+1} = \alpha \mathbf{O}_{t+1} \mathbf{T}^{\intercal} \boldsymbol{f}_{1:t}, \tag{5}$$

where α is a normalizing factor, $\mathbf{O}_{t+1}$ is the observation matrix for $t+1$, and $\mathbf{T}$ is the transition matrix. This equation is implemented in the *Localizer.java* method *updateAlphaProb()*.

3 Results

The observation matrix for each time step and state can be seen in the application by clicking "Show sensor". In Figure 1 we can see an example of the observation matrix, and that it is consistent with the description of how the sensor should work in the introduction.

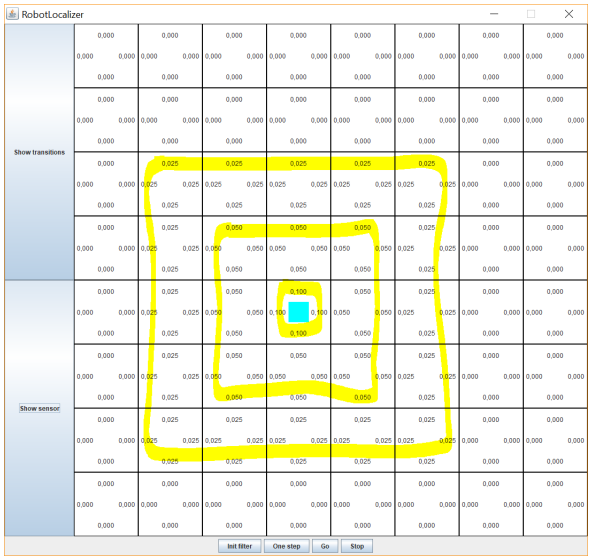

Figure 1: The observation matrix for the first time step and for $x = 5$, $y = 4$.

The transition matrix for $x = 2$, $y = 5$ can be seen in Figure 2. Observe that the transition matrix is the same for all t.

The probabilities after a few time steps can be seen in Figure 3, and after 200 time steps in Figure 4. The characteristic checker board pattern appears in Figure 5.

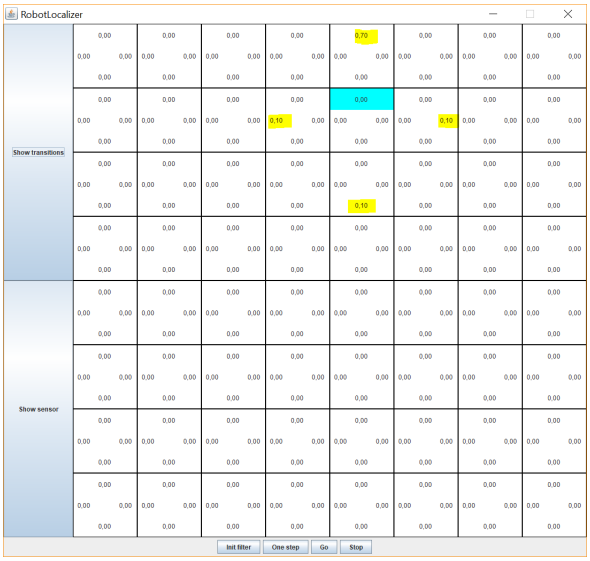

Figure 2: The transition matrix for $x = 2$, $y = 5$ and $head = north$.

Figure 3: The probabilities α after a few time steps.

4 Discussion

To evaluate the quality of our location estimations, we compare the estimated location and the real location, and calculate the difference using the Manhattan

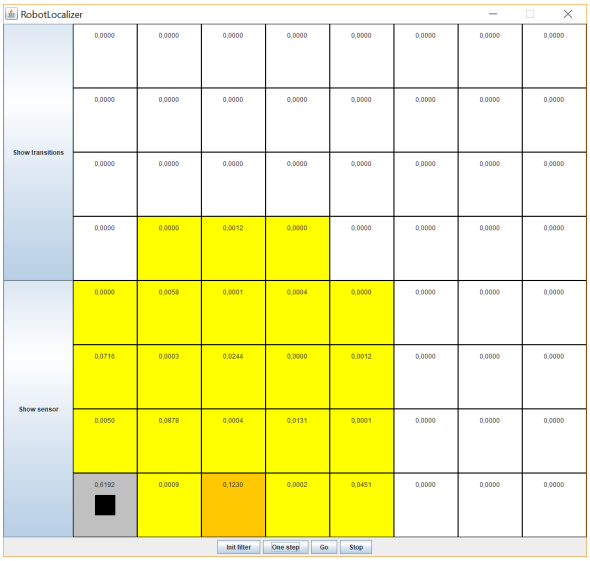

Figure 4: The probabilities α after 200 time steps.

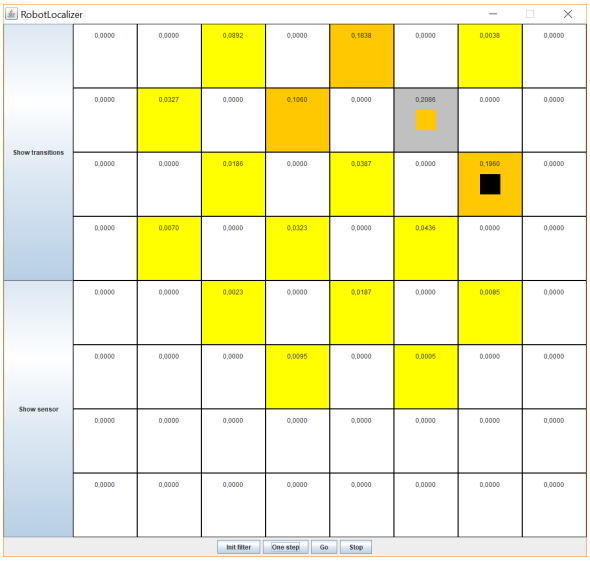

Figure 5: The probabilities α displays the characteristic checker board pattern.

norm. Due to the stochastic nature of this problem, we are estimating the expected value of the distance. In Figure 6 we can see the mean distances for $t = 1..10$ and the estimated standard deviations. The mean of all time steps $t = 1..10$ is 1.31.

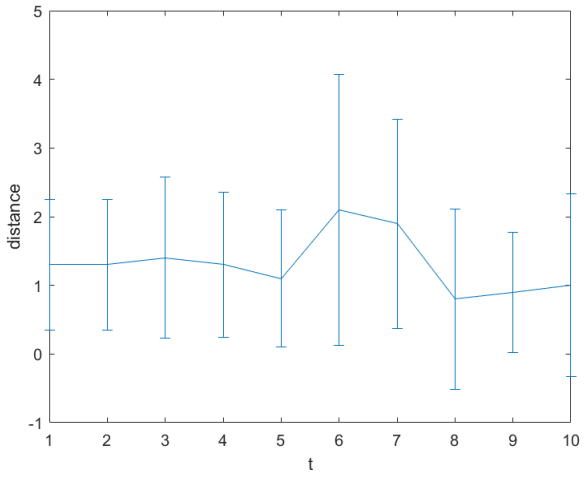

Figure 6: The mean distance between the real location and guessed location using the Manhattan norm, and the corresponding standard deviations.

As we can see, the variance is quite high. If we estimate the expected distance and variance for larger t, there is no big difference after the 10 first steps. We do know that the possibility that there is no sensor reading is quite high, and we can see when running the program that this is exactly when the distance is growing (as expected). This is also contributing that it is hard to reduce the variance over time and thus giving a more precise estimated location.

5 Implementation

The solution is implemented in Java using the supplied graphics package for visualization. In addition to the supplied code an additional Math library, Apache Commons Math3 3.6.1 (found at goo.gl/8junZT), is used to perform matrix operations using classes implementing the *RealMatrix* interface. The project is divided into three packages according to the *model-view-control* pattern. The HMM, forward filtering algorithm, and robot are implemented in the *model* package, in the *Robot*, and *Localizer* classes.

Robot The *Robot* class holds methods for moving the robot according to the defined behavior discusses above, and also methods for generating sensor values in a consistent manor. The robot knows its true position and heading.

Localizer The *Localizer* has a reference to the *Robot* and queries it for sensor values. Using these sensor vales the localizer has methods for calculating the probability of the robot being in any position. The localizer has public methods called by the *view* package to display probabilities in the simulation, and also display the contents of the transition and observation matrices.

method: getEmissProb(int senseX, int senseY) A method which is called iteratively in order to calculate the diagonals of each observation matrix. These values are stored as rows in a large matrix called *bigO*.

method: getTransProb(int x_tminus1, int y_tminus1, Heading heading_tminus1, int x_t, int y_t, Heading heading_t) A method which calculates the probability of moving from (x_tminus1, y_tminus1) in heading_tminus1, and ending up in (x_t, y_t) in heading_t. This method is also called iteratively in order to populate the transition matrix.

User Manual The application is packaged with all the required libraries in a runnable jar file. The jar can be run by executing the command: "java -jar HMM.jar" at path: /h/dk/x/jur10ojo/edaf70/HMM/HMMAssignmentAndViewerTool/AssignmentAndViewerTool/RobotLocalization, where the jar can be found. Executing the above command will start the application and display the GUI, with a default 8 x 8 world, from which the simulation can be run, as per separate, task instructions. Passing a command line argument in the form of an integer $x \in [5, 20]$ (depending on desired size and monitor real estate) will create a world which is $x * x$ in size. The command is then "java -jar HMM.jar x" where x can be replaced by integer as per above.

If one should desire to compile and run the project from scratch the necessary library/jar "commons-math3-3.6.1" can be found at the same path as the runnable jar. The main method is found in the control package in *Main.java*.