Artificial Intelligence - EDAP01
Assignment 3 - Spring 2021
Probabilistic Reasoning over Time

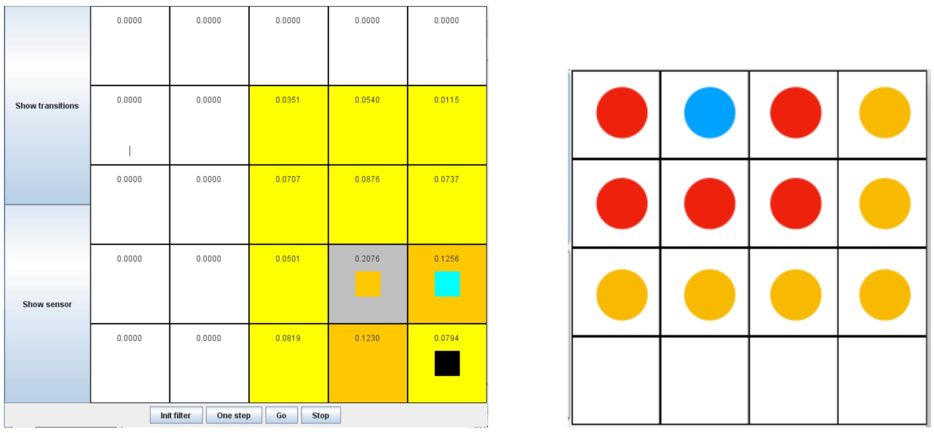
Robot localisation using Hidden Markov Model

Hicham Mohamad (hi8826mo-s)

March 14, 2021

1 Introduction

In this assignment we will work on an Artificial Intelligence project that involves **robot localisation** in the **grid world**, based on the **forward filtering** algorithm with a **Hidden Markov Model** (HMM). The localiser we consider will be implemented in Java. This task is based on task 15.9 in the course book [?] and on the explanations for matrix based forward filtering operations according to section 15.3.1 of the book. At the end, in section **??** we study and summarize the paper *Monte Carlo Localization: Efficient Position Estimation for Mobile Robots* by [Fox, Burgard, Dellaert, Thrun] [?], which introduce a new algorithm for mobile robot localisation called **Monte Carlo localization** (MCL).



0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0351	0.0540	0.0115
0.0000	0.0000	0.0707	0.0876	0.0737
0.0000	0.0000	0.0501	0.2076	0.1256
0.0000	0.0000	0.0819	0.1230	0.0794

(a) (b)

Figure 1: a) ***Visualization****: One state of the output* 5×5 *checker board after running the implemented HMM localizer, obtained by stepping in the filtering process. In this visualization is illustrated pattern of different colour where the darker colours stand for higher probabilities estimated using the implemented HMM forward algorithm. On the other hand,* ***black*** *refers to the true location and* ***cyan*** *marks the current sensor reading, and consequently the colour* ***white*** *means impossible, that is* $p = 0$*. Here it is worth noting that our guessed position in grey computed using the filtering algorithm is distanced 2 squares from the true location in black. So in this way, we say that the estimation error is 2 using Manhattan distance. b)* ***Surrounding fields for the true location****: in this example we can see the true location L in blue, the directly surrounding squares in red where* $n\_Ls = 5$ *and the secondary surrounding ones in yellow where* $n\_Ls2 = 6$*. This classification of field is used in the sensor model shown in Table* **??** *below.*

1.1 Objectives

In this assignment, it is provided a Java-based tool for **visualisation** of the transition model, the sensor model and the estimation, as shown in Figure **??**. This Java tool is designed according to the Model-View-Control (MVC) pattern. The main task in this assignment is to construct a **HMM robot localiser** class that implement the `EstimatorInterface` in the model. Then *we need to investigate how accurately we can track the robot's path in the grid world.*

1.2 User guide and Feedback

To run the localizer, we need to run the program `main.java` in the submitted folder `robotLocHandout/src/control`. The implementation is found in the class `HMM_RobotLocaliser.java` which it includes all the necessary functions (without using the other classes in the skeleton) for making the simulation work nicely.

For the relevant matrix operations, the java library `SimpleMatrix` were downloaded and used from EJML in [**?**]. That's why, IF YOU DON'T USE ECLIPSE, for using an **external library** we need to build and run the Java project from command line in Linux in the following way:

```
javac -cp "ejml-simpleMatrix-libs/*:." control/Main.java

java -cp "ejml-simpleMatrix-libs/*:." control/Main
```

Do the following command lines for running the implemented class `EvaluateEstimates` for evaluating the correct estimates and the Manhattan distance. It is in the folder "robotLocalisation/src/model".

```
javac -cp "ejml-simpleMatrix-libs/*:." model/EvaluateEstimates.java

java -cp "ejml-simpleMatrix-libs/*:." model/EvaluateEstimates
```

2 Preliminaries: Temporal Probabilistic reasoning and Markov localisation

Here we outline the basic Markov localization algorithm. The central idea of Markov localization is to represent the robot's belief *Bel(l)* by a **probability distribution** over all possible positions in the environment, and use **Bayes rule** and convolution to update the belief whenever the robot senses or moves.

The distribution *Bel(l)* expresses the robot's belief for being at position l. Initially, *Bel(l)* reflects the initial state of knowledge: if the robot knows its initial position, *Bel(l)* is centered on the correct position; if the robot does not know its initial position, *Bel(l)* is uniformly distributed to reflect the global uncertainty of the robot. As the robot operates, *Bel(l)* is incrementally refined.

In this way, Markov localization applies two different **probabilistic models** to update $Bel(l)$:

- an **action model** to incorporate movements of the robot into $Bel(l)$.
- a **perception model** to update the belief upon sensory input.

2.1 Uncertainty in Robotics

Robotics is the science of `perceiving and manipulating` the physical world through computer-controlled mechanical devices. Robotics systems have in common that they are are situated in the physical world, perceive their environments through sensors, and manipulate their environment through things that move [**?**]. `Localization` is the problem of estimating a robot's coordinates in an external reference frame from sensor data, using a map of the environment. Examples of successful robotic systems include mobile platforms for planetary exploration, robotics arms in assembly lines, cars that travel autonomously on highways, actuated arms that assist surgeons.

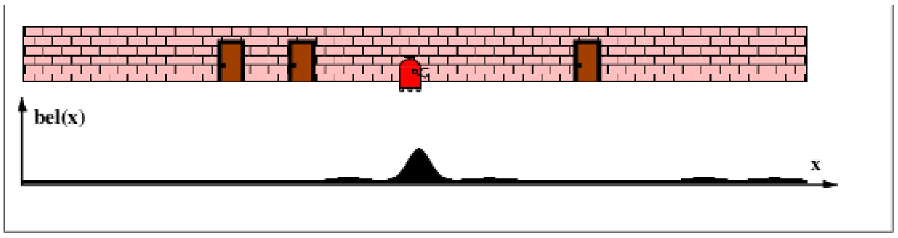


Figure 2: The basic idea of Markov localization: A mobile robot during global localization. Courtesy: [Thrun, Burgard, Fox] [?]

In **probabilistic reasoning over time**, agents/robots in partially observable environments must be able to keep track of the current state. This can be done when an agent maintains a **belief state** that represents *which states of the world are currently possible.* From the belief state and a `transition model`, the agent can predict how the world might evolve in the next time step. From the percepts observed and a `sensor model`, the agent can update the belief state.

In this way, a *temporal probability model* can be thought of as containing a `transition model` describing the state evolution and a `sensor model` describing the observation process (percepts). The principal inference tasks in temporal models are filtering, prediction, smoothing, and computing the most likely explanation. Thus, we look at dynamic processes where everything both state and observations depend on time.

2.2 Hidden Markov Model (HMM)

According to Markov assumption, **Markov property**, future states depend only on the current state, not on the events that occurred before it, see Figure **??**.

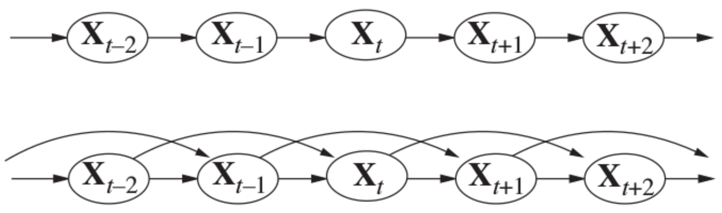


Figure 3: a) Bayesian network structure corresponding to a first-order Markov chain with state defined by the variables $\mathbf{X}_t$. b) A second-order Markov chain where $P(X_t|X_{t-2}, X_{t-1})$. Courtesy: AIMA Book [?]

Markov models seem extremely limiting. Even with a k-th step Markov model

$$p(X_{n+1}|X_n, X_{n-1}, X_{n-2}, ..., X_{n-k}) = p(X_{n+1}|X_n)$$

we are limited to k previous observations. Thus the idea is to introduce **latent variables** to permit a rich class of models. For each observation x_n, we introduce a corresponding latent variable z_n. We now assume that it is the latent variables that form a Markov chain. Assuming the latent variable to be discrete, we get the Hidden Markov model, as illustrated in Figure **??**.

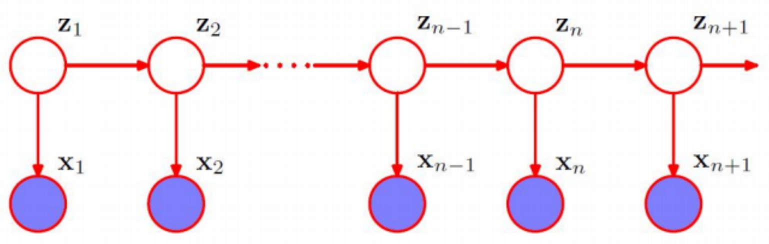


Figure 4: Illustration of the Hidden Markov Model (HMM): Underlying Markov chain over states Z and You observe outputs (effects) at each time step

An HMM is a **temporal probabilistic model** in which the state of the process is described by a **single discrete random variable**. The possible values of this variable are the possible states of the world. What happens if we have a model with two or more state variables? In this case, we can still fit it into the HMM framework by combining the variables into a single **megavariable** whose values are all possible tuples of values of the individual state variables.

In probabilistic robotics, the state transition probability and the measurement/observation probability together describe the **dynamical stochastic system** of the robot and its environment. The state at time t is stochastically dependent on the state at time $t-1$. Such a temporal generative model is also known as *hidden Markov model (HMM)* or dynamic Bayes network (DBN) which include hidden Markov models and Kalman filters as special cases.

2.2.1 Transition model and Markov assumption

The transition model specifies the probability distribution over the latest state variables, given the previous values, that is,

$$\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{0:t-1})$$

where the set $X_{0:t-1}$ is unbounded in size as t increases. We solve the problem by making a **Markov assumption**: *the current state depends on only a finite fixed number of previous states.* Hence, in a first-order Markov process, the transition model is the conditional distribution

$$\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{t-1})$$

2.2.2 Sensor model and Markov stationary process

Even with the Markov assumption there is still a problem: there are infinitely many possible values of t. Do we need to specify a different distribution for each time step? We avoid this problem by assuming that changes in the world state are caused by a **stationary process**: *a process of change that is governed by laws that do not themselves change over time.* Now with a sensor Markov assumption we get the sensor model, also called the observation model:

$$\mathbf{P}(\mathbf{E}_t|\mathbf{X}_t)$$

2.2.3 Filtering and prediction

In **partially observable environments**, which include the vast majority of real-world environments, maintaining one's belief state is a core function of any intelligent system. This function goes under various names, including **monitoring**, **filtering** and **state estimating**.

Filtering is the task of computing the **belief state**, i.e. the posterior distribution over the most recent state given all evidence up to this point, i.e. $\mathbf{P}(X_t|\mathbf{e}_{1:t})$, where

- $\mathbf{X}_t$ is the state at time t, taking on values from 1 to S, with S the number of possible states.
- $\mathbf{e}_t$ is the observation at time t

Filtering is also called **state estimation** and is what a rational agent does to keep track of the current state so that rational decisions can be made. *A useful filtering algorithm needs to maintain a current state estimate and update it*, rather than going back over the entire history of precepts for each update. In other words, given the result of filtering up to time t, the agent needs to compute the result for $t+1$ from the new evidence $\mathbf{e}_{t+1}$

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

This **recursive estimation** can be seen as composed of two parts: first, the current state distribution is projected forward from t to $t+1$, then it is updated using the evidence $\mathbf{e}_{t+1}$

$$\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) = \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}, \mathbf{e}_{t+1}))$$

using **Bayes' rule** we get

$$\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) = \alpha\ \mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1}, \mathbf{e}_{1:t})\ \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t})$$

by the sensor Markov assumption

$$\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) = \alpha\ \mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})\ \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}) \tag{1}$$

Here α is the **normalizing factor** used to make probabilities sum up to 1. The second term, $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t})$ represents a **one-step prediction** of the next state, whereas the first term updates this with the new evidence. We notice that $\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{1:t+1})$ is obtainable directly from the sensor model.

Now we can obtain the one-step prediction for the next state by *conditioning on the current state* $\mathbf{X}_t$, i.e.

$$\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) = \alpha\ \mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1}) \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t, \mathbf{e}_{1:t}) P(\mathbf{x}_t|\mathbf{e}_{1:t})$$

using Markov assumption we get the desired recursive formulation:

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

Within the summation, the first factor comes from the transition model and the second comes from the current state distribution. *We can think of the filtered estimate $\boldsymbol{P}(X_t|\boldsymbol{e}_{1:t})$ as a **message** $\boldsymbol{f}_{1:t}$ that is propagated forward along the sequence, modified by each transition and updated by each new observation.* The process is given by

$$\mathbf{f}_{1:t+1} = \alpha\ \texttt{FORWARD}(\mathbf{f}_{1:t}, \mathbf{e}_{t+1})$$

where `FORWARD` implements the update described in Equation **??** and the process begins with $\mathbf{f}_{1:0} = \mathbf{P}(\mathbf{X}_0)$

2.2.4 Simplified forward filtering process for the matrix formulation of HMMs

For an hidden Markov model HMM with a *single discrete* state variable X_t , we can give concrete form to the representations of the transition model, the sensor model, and the forward message. We let the state variable X_t have values denoted by integers 1, . . . , S, where S is the number of possible states.

The transition model $\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{t-1})$ is then expressed as $S \times S$ matrix $\mathbf{T}$:

$$\mathbf{T}_{ij} = P(X_t = j|X_{t-1} = i) \text{ in time step } t$$

That is, $\mathbf{T}_{ij}$ is the probability of a transition from state i to state j. The **sensor model** for the corresponding observations depending on the current state, i.e. $\mathbf{P}(\mathbf{e}_t|\mathbf{X}_t = i)$, is then expressed as $S \times S$ **diagonal matrix** O_t in time step t with

$$\begin{cases} \mathbf{O}_{ij} = P(e_t|X_t = i) \quad \text{for } i = j \\ \mathbf{O}_{ij} = 0 \quad \text{for } i \neq j \end{cases} \tag{3}$$

The reason behind that is because the value of the evidence variable E_t is known at time t (call it e_t), we need only specify, for each state, how likely it is that the state causes e_t to appear: we need $P(e_t|X_t = i)$ for each state i. For mathematical convenience we place these values into an $S \times S$ diagonal matrix, $\mathbf{O}_t$ whose ith diagonal entry is $P(e_t|X_t = i)$ and whose other entries are 0.

Now if we use column vectors to represent the forward messages, all the computations become simple **matrix-vector operations**. Thus, the forward filtering equation for the matrix formulation of HMMs becomes

$$\boldsymbol{f}_{1:t+1} = \alpha\ \boldsymbol{O}_{t+1}\ \boldsymbol{T}^T\ \boldsymbol{f}_{1:1} \tag{4}$$

where $\boldsymbol{f}$ is the **message** that is propagated forward along the sequence, modified by each **transition T** and updated by each new **observation O**. And here α is the **normalizing factor** used to make probabilities sum up to 1.

3 Problem description - Forward filtering

To navigate reliably in indoor environments, a mobile robot must know where it is. **Robot Localisation** is the problem of determining the position of a mobile robot from sensor data. As usual in designing an agent, it is reasonable to make the problem slightly more realistic by including a simple **probability model** for the robot's motion and by allowing for **noise** in the sensors. In addition, we need to consider a **belief** state variable X_t that in our assignment represents the location of the robot on the **discrete grid**, where the domain of this variable is the set of empty squares [**?**].

In this work, our **localisation problem** will be solved using the HMM-based **forward filtering** algorithm to track the robot in an environment with no landmarks. We consider a vacuum robot in an empty room, represented by an $n \times v$ rectangular grid. The robot's location is hidden; the only evidence available to the observer is a *noisy location sensor* that gives a direct but **vague approximation** to the robot's location.

Implementing this localisation problem as an HMM according to the **matrix-vector** notation in section 15.3.1 in [Russel and Norvig, 2010] [**?**] requires obviously two-part implementation:

- *Simulate the robot and its movements*, from which we also can simulate the sensor readings, to have some **ground truth** to evaluate the tracking against.
- The HMM-based tracking algorithm.

Specifically, our agent should predict the new step/pose from the belief state and the transition model shown in Table **??**. Then, after receiving percepts based on its actual position, from a noisy sensor according to the model shown in Table **??**, the agent will update its belief state/estimate using the sensor and transition models. This means computing the vector $\boldsymbol{f}$ in the **forward filtering** algorithm in matrix formulation, as explained in **??**

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

To summarise, this **forward filtering algorithm** should basically loop over the following three steps:

1. *Move (simulated) robot to new pose* according to movement model in Table **??**.
2. *Obtain (simulated) sensor reading* based on its actual position using the given sensor model shown in Table **??**.
3. *Update the position estimate*, vector $\boldsymbol{f}$ in the forward algorithm, based on the sensor reading from step 2, using the known sensor and transition models, that is expressed in the matrix formulation in **??**.

In this way, as we have mentioned previously, the goal in this assignment consist of constructing a **HMM robot localiser** class `HMM_RobotLocaliser` according to this forward filtering algorithm to track the robot. Then *we need to investigate how accurately we can track the robot's path in the grid world.*

3.1 Task environment specification - Models

As mentioned in the textbook [**?**], the precise definition of the **task environment** is acronymically given by the **PEAS** description. In the following we define the **P**erformance measure, the **E**nvironment, the agent's **A**ctuators and **S**ensors for the robot's task environment.

3.1.1 Performance

The performance measure can evaluate the behavior of the agent in an environment. In our grid world, we can evaluate the performance of this HMM localization task by computing the localisation/estimation **error**, defined as the **Manhattan distance** from the true location, i.e. the number of robot steps necessary to get from estimated position to true position.

3.1.2 Environment

Because our environment is modelled as a $n \times m$ rectangular grid, we assume a **state encoding** model based on triplets (x, y, h), i.e. together the position (x, y) = (row, column) and h the heading of the robot in the grid world, with $x = 0$ being the top row, $y = 0$ the leftmost column and h running around the compass as shown in Table **??**. A sample grid world of size 5 × 5 is shown in Figure **??**.

Compass directions h_t	North N	East E	South S	West W
Encoding	0	1	2	3

Table 1: Headings encoding.

3.1.3 Actuators - Movement policy

The robot moves in the picked direction h_{t+1} by one step and only straight, i.e. it can move forward (north), turn right (east), turn left (west) and move back (south). Here we assume the **transition model** where T_{ij} represents the probability of stepping from state i to state j depending on whether it faces a wall or not, as shown in Table **??**.

	Combinations based on h_t and wall facing	Probability
case 1	$h_t + 1 = h_t$ | not encountering a wall	0.7
case 2	$h_t + 1 \neq h_t$ | not encountering a wall	0.3
case 3	$h_t + 1 = h_t$ | encountering a wall	0.0
case 4	$h_t + 1 \neq h_t$ | encountering a wall	1.0

*Table 2: **The robot moving strategy**: Initially, it starts by picking **random start heading** h_0, then for any new step, the robot picks new heading h_{t+1} based on current heading h_t and on the case if facing a wall or not. In case a new heading is to be found, the new one is **randomly chosen from the possible ones**. In this way, when facing a wall somewhere along the wall we cancel one of the possible options of where to run and have 3 options left; and when facing the wall in a corner we should cancel two options and have 2 options left.*

3.1.4 Sensors - Sensor model

In general, sensors are the fundamental robot input for the process of **perception**. Here we consider the robot's location to be hidden and the possible perceptions can be only through a **noisy sensor** that gives a direct, but **vague approximation** to the robot's location. Assuming the true location is $L = (x, y)$, the sensor readings/approximations can be distinguished in three different levels L, Ls or $Ls2$, where Ls denotes any of the 8 squares in the **directly surrounding** ring, and $Ls2$ any of the 16 ones in **second surrounding ring**. Consequently, it is worth noting that being in a corner or somewhere close to the wall can be considered as a particular case of the sensor approximations where some squares of the ring stay actually outside the grid such that the sensor is more likely to produce nothing, i.e. $(-1, -1)$.

Sensor Reading	Probability	n	Ring probability
True location `L(x,y)`	0.1		
`Ls`	0.05	$\{3, 5, \mathbf{8}\}$	$8 * 0.05 = 0.4$
`Ls2`	0.025	$\{5, 6, 7, 9, 11, \mathbf{16}\}$	$16 * 0.025 = 0.4$
No reading	0.1		

*Table 3: Sensor model: According to the given specifications, as shown in Figure **??**, n_Ls is the number of directly surrounding fields for L and n_Ls2 is the number of secondary surrounding fields, depending on whether L is in corner, along a wall or at least 2 fields away from any wall. In this way, we have $n\_Ls \in \{3, 5, 8\}$, $n\_Ls2 \in \{5, 6, 7, 9, 11, 16\}$. This means that the sensor is more likely to produce NOTHING when the robot's true location is less than two steps from a wall or in a corner.*

Thus, in the case L is at least 2 fields away from any wall, if the robot is at this location then

- with probability 0.1, the sensor gives the **true location**,
- with probability 0.05 each, it reports the 8 **immediately surrounding locations** to $L = (x, y)$ with probability
$$n\_Ls * 0.05 = 8 * 0.05 = 0.4$$
- with probability 0.025 each, it reports the 16 locations that surround those 8 with probability
$$n\_Ls2 * 0.025 = 16 * 0.025 = 0.4$$
- and it reports "**no reading**" with the remaining probability
$$1.0 - 0.1 - n\_Ls * 0.05 - n\_Ls2 * 0.025 = 1.0 - 0.1 - 0.4 - 0.4 = 0.1$$

In this way, looking at the sensor approximations, we can notice that the sensor is more likely to produce "nothing" when the robot's true location is less than two steps from a wall or in a corner.

Since the filtering algorithm only knows sensor readings, which only represent positions in the grid, one sensor reading is equally likely for the four states (headings) that corresponds to one position in the grid. *Thus, the values within each "bundle" of four consecutive entries along the observation matrix should be the same.*

3.2 Transition and observation matrices

To model our robot, each state in the model consists of a **location-heading** pair based on triplets (x,y,h). For the **sensor readings** it assumes to receive $8 \cdot 8 = 64$ probabilities, one for each position (x, y), to have caused the reading r = (rX, rY) or r = (-1, -1), which means "nothing".
In this way, the number of **possible states** are thus

$$rows \cdot columns \cdot 4 = 256$$

and the number of **possible sensor readings** are thus

$$rows \cdot columns + 1 = 65$$

This means that we construct a **transition matrix** of dimensions:

$$(rows \cdot columns \cdot 4) \times (rows \cdot columns \cdot 4) = (8 \cdot 8 \cdot 4) \times (8 \cdot 8 \cdot 4) = 256 \times 256$$

and we need $(rows \cdot columns + 1) = 8 \cdot 8 + 1 = 65$ observation matrices, each being a **diagonal** matrix of the same dimensions as the transition matrix, in order for the matrix multiplication to work out. That's why it is reasonable to store the observation matrices in **vectors**, representing the diagonal of the respective matrix.

Evaluation of coding According to the rule of thumb for evaluation of coding, if we stack all $n \cdot m + 1 = 65$ *grid visualisations of the observation matrices* on top of each other, the sum over the probabilities in each "pose-stack" should be exactly 1.0 (with some rounding errors). The sum over all cells within one visualisation can in theory be anything between 0.0 and

$$(n \cdot m \cdot 4 + 1) \cdot 1.0 = 257.$$

On the other hand, we can evaluate the transition matrix T by checking if every row in T sums to 1.

4 Instructions about the implementation

The program for running the robot localiser is implemented in Java. In the following list, we describe the principal methods used in the class `HMM_RobotLocaliser` to solve the problem. This java program works together with the provided interface and main method.

Here we should mention that for the matrix operations, we had implemented first the needed routines but then for better performance the java library `SimpleMatrix` were downloaded and used from EJML in [**?**]. To evaluate the correct estimates the class `EvaluateEstimates` is implemented. With this class it is possible to measure the localisation error averaged over 20 runs by using the Manhattan distance, and to show the evaluated correct estimates as well.

1. `HMM_RobotLocaliser()`, constructor to create the localiser with the current location at (0,0) being the top row and the leftmost column in the grid world
2. `createSensorModel()` create the sensor vectors according to the noisy model.
3. `getOrXY()` returns the probability entry of the sensor matrices O.
4. `computeNullSensing()` compute the probability to give "nothing" reading for each cell and put that data in a matrix of the same dimensions as the grid. A sensor reading of "nothing" is slightly more likely to get when robot is close to a wall.
5. `createTransMatrix()` Create the transition matrix.
6. `getTProb()` returns the probability entry T_{ij} of the transition matrix T.

7. `getNewStepProb()` For any new step pick new heading $h_t + 1$ based on the current heading h_t according to the movement model. In case a new heading is to be found, the new one is randomly chosen from the possible ones.
8. `getCurrentReading()` returns the currently available sensor reading obtained for the true position after the simulation step. It returns null if the reading was ”nothing”.
9. `pickNewStep()` returns the new true state.
10. `pickOMatrix()` returns the corresponding observation matrix as a SimpleMatrix.
11. `getCurrentProb()` returns the currently estimated (summed) probability for the robot to be in position (x,y) in the grid.
12. `update()` should trigger one step of the estimation, i.e., true position, sensor reading and the probability distribution for the position estimate should be updated one step after the method has been called once.
13. `notFacingWall` Check if any direction does not point to a wall.
14. `toDiagonal()` generate diagonal matrix from vector.
15. `multiplyM()` return $c = a \cdot b$ and matrix-vector multiplication $y = A \cdot x$.
16. `transposeM()` return $B = A^T$

5 Results - Accuracy of tracking the robot's path

Initially, the performance of the localizer is measured by finding the correct estimates which results between 28% and 33%. However, as mentioned in the assignment text, in terms of robot localisation, it is more relevant to know how far the estimate is from reality on average. Thus, we consider the **Manhattan distance** to measure the distance between the true location and our estimate, i.e. the location/approximation error. By running the **world simulator** (movement and percepts in 5×5 grid) together with the implemented class `EvaluateEstimates`, the expected localization/approximation error, averaged over 20 runs, results to be between 1.2 and 2.

This result seems to be acceptable, because with the obtained average Manhattan distance we can conclude that the localizer is able to track/follow the robot's true location but without hitting it exactly. This distance error can be returned to the fact that in our model the next step heading is determined randomly in some cases.

On the other hand, when evaluating the guesses generated by the vague approximations of the sensor model shown in Table **??**, we see that the probability is high (about 0.4) when the robot is away from the walls. But comparing to the performance obtained by applying HHM filtering above, the accuracy of tracking the robot must be worse, especially when the sensor is more likely to produce ”nothing” when the robot's true location is less than two steps from a wall or in a corner.

6 Reading

In this section, we need to read and summarize the article *Monte Carlo Localization: Efficient Position Estimation for Mobile Robots* by [Dieter Fox et al] [**?**], which was published in 1999, and received the AAAI Classic AI Paper Award in 2017.

Summary

The article presents a new algorithm for mobile robot localization called Monte Carlo Localization (MCL), which is a version of Markov localization, computationally efficient and applies **sampling-based methods** for approximating probability distributions, in a way that places computation “ where needed.” First it is worth noting that the localization problem comes in two types: **position tracking** where the initial position is known, and **global localization** (also known as hijacked robot problem) where the initial position is unknown. Here we outline the basic characteristics of Monte Carlo localization algorithm in [**?**]

- MCL is a version of **sampling/importance re-sampling** (SIR). The key idea underlying this algorithm is to represent the posterior belief *Bel(l)* by a set of N weighted, **random samples** or **particles**
$$S = \{s_i \mid i = 1 \cdots N\}$$
A sample set constitutes a **discrete approximation** of a probability distribution, where samples are of the type
$$\langle\langle x, y, \theta\rangle, p\rangle$$
where $\langle x, y, \theta\rangle$ denote a robot position and $p \geq 0$ is a numerical **weighting factor**, analogous to a discrete probability.

- **Robot motion**: When the robot moves, MCL generates N new samples that approximate the robot's position after the motion command. `Each sample is generated by randomly drawing a sample from the previously computed sample set, with likelihood determined by their p-values.` The **p-value** of the new sample is N^{-1}.

- **Sensor readings** are incorporated by **re-weighting** the sample set, in a way that implements Bayes rule in Markov localization:
$$p \leftarrow \alpha\ P(s \mid l)$$
where $\langle l, p\rangle$ is a sample and s is the sensor measurement.

- An **adaptive sampling** scheme, which determines the number of samples on-the-fly, can be employed to trade-off computation and accuracy. As a result, MCL uses many samples during **global localization** when they are most needed, whereas the sample set size is small during **tracking**, when the position of the robot is approximately known.

- A nice property of the MCL algorithm is that it can **universally** approximate arbitrary probability distributions. In addition, MCL is an **online algorithm** and it lends itself nicely to an any-time implementation. Consequently, it can generate an answer at **any time** and the quality of the solution improves over time. The sampling step in MCL can also be terminated at any time.

Comparison and discussion

After characterizing the different capabilities of MCL, we need to compare it to grid-based Markov localization used in our implementation. Empirical results illustrate that MCL yields **improved accuracy** while requiring an order of magnitude **less computation** when compared to our implemented approach. It is also much easier to implement.

Grid-based HMM approach methods are powerful, but suffer from **excessive computational overhead** and a priori commitment to the **size** and **resolution** of the state space. In addition, the resolution and thereby also the precision at which they can represent the state has to be fixed beforehand. As shown there in [**?**], the accuracy increases with the resolution of the grid. Thus, I think we can choose or utilize both of them depending on the applications complexity and/or the efficiency required in them.

7 Appendix

References

[1] Stuart Russell and Peter Norvig, Artificial Intelligence: A Modern Approach, 3/e, Prentice Hall Series in Artificial Intelligence, 2010. ISBN-10: 0132071487 or 1292153962. Available: `https:\aima.cs.berkeley.edu`.

[2] Forward-Backward algorithm, Wikipedia: `https://en.wikipedia.org/wiki/Forward-backward_algorithm`.

[3] H. Mohamad, L. Anderberg, *Probabilistic Reasoning over Time*: EDAF70 Applied Artificial Intelligence Assignment 2 - Spring 2019.

[4] Java implementation of algorithms from Russell and Norvig's Artificial Intelligence - A Modern Approach 3rd Edition. Fixed-Lag-Smoothing algorithm: `https://github.com/aimacode/aima-java/blob/AIMA3e/aima-core/src/main/java/aima/core/probability/hmm/exact/FixedLagSmoothing.java`

[5] Sebastian Thrun, Wolfram Burgard, Dieter Fox, Probabilistic Robotics.

[6] Dieter Fox, Wolfram Burgard, Frank Dellaert, Sebastian Thrun, "Monte Carlo Localization: Efficient Position Estimation for Mobile Robots", *16th AAAI Conference on Artificial Intelligence (AAAI-99)* in 1999, and received the AAAI Classic AI Paper Award in 2017. `https://aaai.org/Papers/AAAI/1999/AAAI99-050.pdf`

[7] EJML - Class SimpleMatrix in java. Available: `http://ejml.org/javadoc/org/ejml/simple/SimpleMatrix.html`