Hidden Markov models

Fischer Dominik, Rutrle Tomáš

***Abstract*—This text validates a solution of robot position localization problem using the Hidden Markov models framework. Three prediction algorithms, namely the filtering, smoothing and Viterbi algorithms are used for robot position prediction based only on inaccurate measurements and their performance is compared against each other. Said comparison is performed using an own metric based on scaled Manhattan distance as well as a simple hit/miss criterion. Furthermore a number of possible improvements to the used algorithms were implemented and are summarized in this text as well. First, the naive implementation using `for` loops was replaced with matrix algebra which resulted in a significant decrease in computation time in both filtering and smoothing algorithms. Then the issue of underflow which arises when large products of small numbers are computed, is addressed by extending the algorithm with scaling. Similarily, for a better numerical stability of the Viterbi algorithm the log probabilities are used. Finally, the Baum–Welch algorithm for estimating the parameters of HMM is described.**

I. Introduction

Robot localization is a problem that can be solved in numerous ways and offers a wide range of possible approaches and viewpoints. The rather standard way to tackle this problem is to rely on external means of localization using i.e. GPS or built–in position reference sources such as odometry or even relative position sensors.

However, situations can occur where it is either impossible to use any of the aforementioned localization means, due to the lack GPS signal, inaccessibility of the area and other ambient conditions, or because it can be simply too expensive to equip the robot with precise localization sensor. In such cases a different, non–standard way of localization is required.

The Hidden Markov Models (HMM) based algorithms can be used in the discussed localization problem and pose an alternative to the standard means of localization which can work even with highly inaccurate sensors, that only provide the robot with *near* or *far* information. The mentioned algorithms use the model of the system as the transition and emission probabilities to predict the future position.

The viability of using the HMM for robot localization is studied in this text which is organized as follows. First, the accuracy metric together with other influences on the performance of the localization algorithms are presented. Then in Section IV, possible improvements to the algorithms are presented, namely their implementation using matrix algebra, scaling as protection against underflow issues, Viterbi algorithm with log probabilities and the Baum–Welch algorithm. Finally, Section V sums up the main results and concludes this text.

II. Methodology

Our team relied mainly on the Microsoft Teams for communication and Overleaf for sharing the report source codes. Each team–member kept his Python source codes separately and worked on his tasks and the final form of the codes was assembled first before the submission.

Task	Author
Experimental protocol	Tomáš Rutrle
HMM with matrices	Dominik Fischer
Scaling	Tomáš Rutrle
Viterbi with log probabilities	Tomáš Rutrle
Baum–Welch algorithm	Dominik Fischer

TABLE I: Workload distribution among our team.

Table I shows the distribution of tasks among our team.

III. Experimental protocol

As a first step of this semestral project we have to develop a methodology which will allow us to estimate the accuracy of the three implemented HMM algorithms.
Firstly we have to realize that both the Viterbi and the Forward / Forward-backward algorithms are probabilistic methods. Therefore no matter which metric we end up using, to determine its accuracy a sufficiently large number of measurements will have to be carried out in order to draw any conclusions. Furthermore the performance of the algorithm is not only given by the algorithm itself but by the environment as well. We will have to carry out further experiments to determine the impact of the environment layout on the individual algorithms and determine whether some surpass the others for some specific layouts.

A. Accuracy metric

To try and logically compare the precision of the predictions of the individual algorithms we first have to look at the outputs of those algorithms. The Viterbi method takes the believed robot position in the previous step together with the sensory observation and returns next position which it believes to be most probably the new robot position. The Forward and Forward-backward algorithms on the other hand do not specify one selected position but assign a probability of the robot position to all the possible states in the environment.
With this in mind the first metric we came up with was to measure the $L1$ norm between the predicted and real robot position and accumulate this error over the whole simulation. For the Forward / Forward-backward algorithms we can then choose how many of the most probable prediction we want to take into consideration and scale the errors proportionally to the probabilities of the individual states. Let us therefore formulate the error for the Viterbi algorithm as follows:

$$E_{vit} = \sum_{i=1}^{n} \|X_{i,pred} - X_{i,true}\|_1 \tag{1}$$

Where n is the number of simulation steps and $X_{i,..}$ predicted and true states. Similarly for the other two algorithms:

$$E_{fb} = \sum_{i=1}^{n} \left(\sum_{j=1}^{k} \| X_{i,j,pred} - X_{i,j,true} \|_1 \; p_{i,j,norm} \right) \quad (2)$$

$$p_{i,j,norm} = \frac{p_{i,j}}{p_{i,1} + ... + p_{i,k}} \quad (3)$$

Where k is the number of highest probability states we take into account and $p_{i,j,norm}$ are the individual scaling factors.

The following figure 1 showcases this metric for a 50 step long simulation on the *rect_6x10_obstacles* map where the simulation was repeated 30 times. Furthermore several values of k have been tried to determine its impact on the accuracy of the Forward and Forward-backward algorithms. Interestingly enough, with the increase of k the average performance of both of the Forward and Forward-backward algorithms seems to have slightly worsened. This validates the approach of simply choosing the highest probable state without taking the other states into consideration.

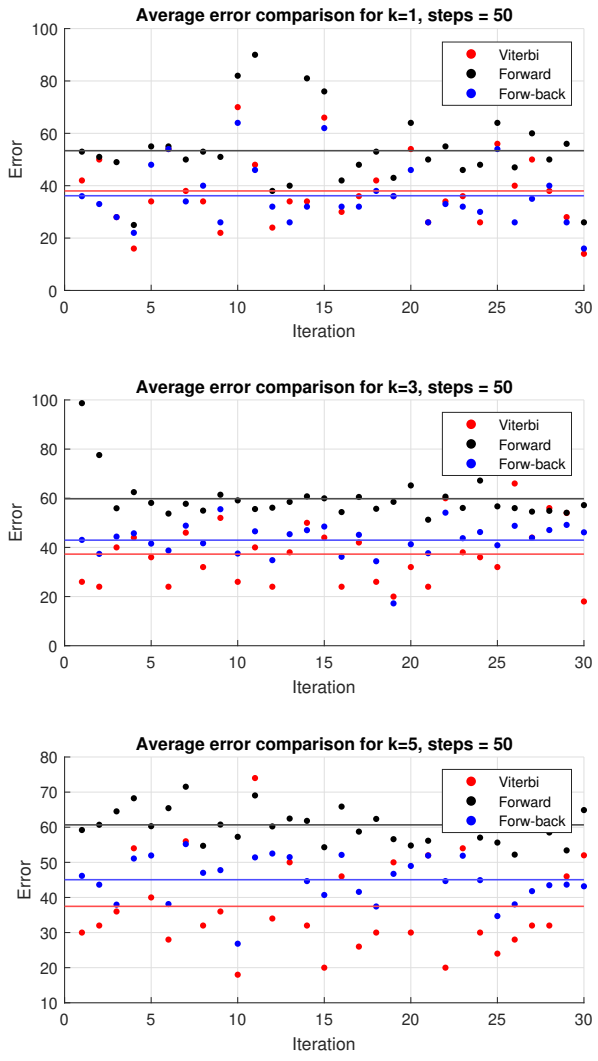

Fig. 1: Influence of k on the average error of the position estimator

The fact that best results are obtained by selecting the top one candidate of the Forward(-backward) algorithms leads to another potential metric and that is simply by computing the hit and miss score. The following figure 2 compares the hit rates of the individual predictors, where the experiment has been carried out in the same fashion as previously. That is 30 independent simulations, containing 50 steps each. The relative results correspond to the findings from the previous figure 1 for $k = 1$. Those were, that the Forward-backward algorithm performs almost identically as the Viterbi algorithm, whereas the simplest Forward predictor lags behind the first two.

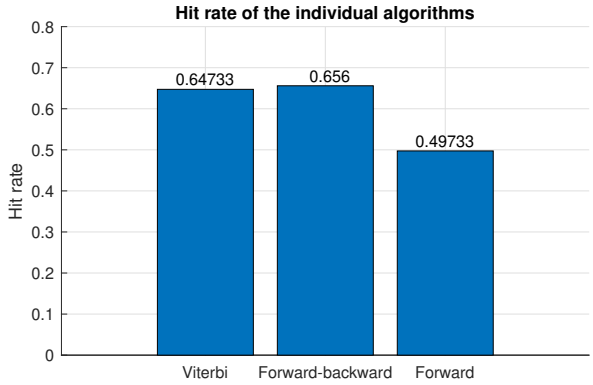

Fig. 2: Hit / attempts ratios for 1500 attempts

B. The influence of environment

As mentioned in the introduction of this section, another factor which may play role in the measured accuracy is the layout of the environment. Up until this point all measurements have been carried out in the *rect_6x10_obstacles* map shown in the following picture. This map was chosen because it consists of open spaces as well as closed corridors.

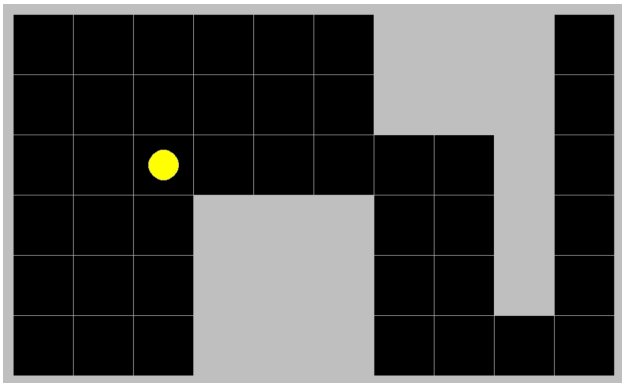

Fig. 3: The *rect_6x10_obstacles* map

But how would the algorithms perform in different environments? Let us try and repeat the accuracy experiment on two more, structurally different maps shown in the following

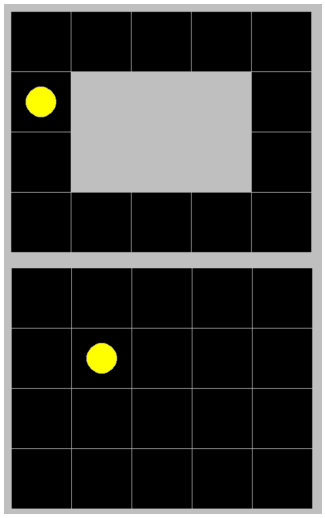

Fig. 4: Different types of possible environments

figure 4. In regards to their characteristics we will call the first one *the corridor* and the second one *wasteland*.

With the experiment carried out in the same fashion as in the previous section, that is 30 independent 50 step long simulations, we can see the results in form of hit rates in the following bar figures 5.

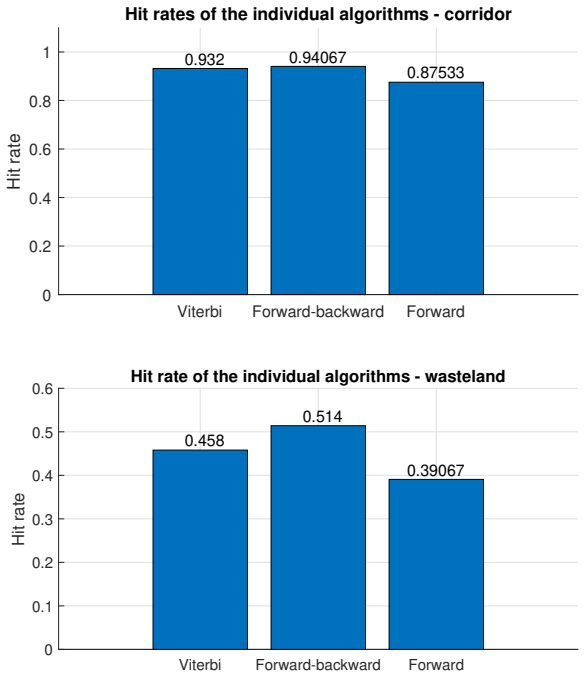

Fig. 5: Hit rate results for the *the corridor* and *wasteland* environments

Unsurprisingly the individual accuracies are much higher for the simpler *corridor* map since the movement of the robot is only restricted to two directions at all times, whereas the open *wasteland* environment offers three or four directions of movement for most of its states. Furthermore we can conclude that although the environment affects the overall accuracies of the algorithms their relative accuracies remain, for the most part, unaffected. That is to say, that the Viterbi and Forward-backward algorithms always seem to be more accurate than the simple Forward algorithm. Although we can notice the Viterbi algorithm performing a little worse than the Forward-backward method in open spaces when compared to the more restricted types of maps.

IV. Optional parts

In this section the proposed solution of a couple of optional parts is described, namely the *Implementation of HMM using matrices* in IV-A, *Scaling* in IV-B and with *Viterbi algorithm with log probabilites* deals the Subsection IV-C. Finally, the Baum–Welch algorithm is described in Subsection IV-D.

A. Implementation of HMM using matrices

The initial, naive implementation of the prediction algorithms works well, however it still allows for improvements especially in terms of efficiency.

The nature of the performed computations makes them suitable for substituting the `for` loops used in naive implementation with matrix multiplication which is noticeably faster and more efficient.

Let us first define matrices representing the used HMM model which will be used to facilitate the computations. Let $\mathbf{A}$ and $\mathbf{B}$ be the transition probability matrix and the observation probability matrix respectively. Furthermore it holds that

$$\mathbf{A} = [a_{ij}] = [P(X_t = x_j | X_{t-1} = x_i)] \tag{4}$$

and

$$\mathbf{B} = [b_{ij}] = [P(E_t = e_j | X_t = x_i)]\,, \tag{5}$$

where X and E are the sets of all states and observations respectively and $x \in X$ and $e \in E$. These matrices can be obtained using the `hmm.get_transition_matrix()` and `hmm.get_observation_matrix()` methods.

The obvious way to validate whether the implementation is correct is to try to replicate the results obtained by the naive implementation. The comparison of the former implementation's results with the matrix–based versions is in Tables II, III and IV. A noticeable improvement in terms of computation time is apparent for the filtering and smoothing algorithms. On the other hand, the matrix–based Viterbi performs slightly poorer as far as computation time is considered. The performed test can be recreated by running the `test_matrix_implementation.py` script.

A quick comment on how we measure how much do the results of individual implementations differ is in place. When considering the filtering and smoothing algorithms, the resulting probabilities for each individual states were compared and the largest difference is presented in the last column of Tables II and III. The Viterbi algorithm however does not provide us with probabilities, only with the most likely prediction. Therefore we computed how many times the final predictions differ and summed these "misses" together. It is worth noting, that when two states have the same probabilities, the Viterbi

Simulation steps	Naive filtering	Matrix filtering	Maximum error
5	0.0683 s	0.0222 s	0.0
10	0.1396 s	0.0232 s	0.0
50	0.6932 s	0.0262 s	$5.551e^{-17}$
100	1.4761 s	0.0304 s	$2.082e^{-17}$

TABLE II: Computation time and difference for individual implementations of the filtering algorithm.

Simulation steps	Naive smoothing	Matrix smoothing	Max. error
5	0.2516 s	0.0445 s	0.0
10	0.5071 s	0.0464 s	$2.711e^{-20}$
50	2.7670 s	0.0604 s	$3.186e^{-58}$
100	5.2470 s	0.0583 s	$1.110e^{-16}$

TABLE III: Computation time and difference for individual implementations of the smoothing algorithm.

algorithm chooses one arbitrarily which is a potential cause for the couple of misses detected.

The description of necessary changes to each individual prediction algorithm follows.

1) Prediction: The task of simply predicting new state $\tilde{\mathbf{v}}$, given the current state can be easily computed using the transmission matrix (4) as

$$\tilde{\mathbf{v}} = \mathbf{A}\mathbf{v}, \tag{6}$$

where $\mathbf{v}_i = P(x_i)$ is the previous vector of probabilities of individual states.

2) Forward algorithm: The forward algorithm consists of calling a forward update for each observation in a given sequence E. A single forward update is the predicting the probabilities of individual next states given a particular observation $e_i \in E$.

It is first necessary to introduce a new matrix $\mathbf{O}$ which allows for a compact multiplication of individual states' and current observation's probabilities. With a slight abuse of Python's syntax we can write the matrix $\mathbf{O}$ as

$$\mathbf{O} = \mathrm{diag}(\mathbf{B}(:, i)). \tag{7}$$

Equipped with (7) we can now express the forward update as

$$\tilde{\mathbf{v}} = \mathbf{O}(\mathbf{A}\mathbf{v}), \tag{8}$$

where again, $\mathbf{v}$ are the prior probabilities of individual states and $\tilde{\mathbf{v}}$ are the updated probabilities that take a current observation into consideration.

3) Backward algorithm: Backward algorithm is similar to the Forward algorithm in the sense that it performs an update for each observation e_i in a sequence E.

Simulation steps	Naive Viterbi	Matrix Viterbi	Num. of misses
5	0.0838 s	0.1091 s	0
10	0.1392 s	0.2179 s	0
50	0.7180 s	1.1510 s	1
100	1.3697 s	2.1719 s	3

TABLE IV: Computation time and difference for individual implementations of the Viterbi algorithm.

Again, we can take advantage of the introduced matrix $\mathbf{O}$ from eq. (7) and compute the backward update as

$$\tilde{\mathbf{v}} = \mathbf{A}^\top(\mathbf{O}\mathbf{v}). \tag{9}$$

4) Forward–backward algorithm: The forward–backward algorithm itself was subject to just minor changes since it only combines the forward and backward algorithms described in subsections IV-A2 and IV-A3. In a nutshell the only difference is that it uses the aforedescribed matrix based computations in the individual forward and backward runs.

5) Viterbi algorithm: There is slightly more the to Viterbi algorithm than just one matrix equation that will be able to replace the naive implementation. Similarly as with the previous algorithms, Viterbi sequentially processes a sequence of observations and performs an update which can be broken down into the following steps.

First, the forward step is performed. In other words, the transition probabilities from individual states are computed, this can be mathematically expressed as

$$\mathbf{T} = \mathbf{A} \odot \mathbf{v}, \tag{10}$$

where $\odot$ denotes element–wise multiplication, $\mathbf{v}$ is the vector of prior probabilities and $\mathbf{A}$ is the transition matrix introduced in (4).

The computation of the best predecessor follows. We now need to extract the information about what preceding state is most likely to lead into each individual state. This is equivalent to getting the largest element (transition probability) in each column of $\mathbf{T}$. Again, abusing the Python syntax, let $\mathbf{p}_{max}$ be a vector such that

$$\mathbf{p}_{max,i} = \max(\mathbf{T}(:, i)). \tag{11}$$

Then we can find the best predecessor of current state as $X(\mathrm{argmax}(\mathbf{p}_{\max}))$, where X is the set of all states.

Having all that we can finally compute the resulting probability distribution $\tilde{\mathbf{v}}$ over all states given current observation as

$$\tilde{\mathbf{v}} = \mathbf{O}\mathbf{p}_{max}, \tag{12}$$

where again $\mathbf{O}$ represents the current observation.

B. Scaling to help with underflow issues

One of the issues when using the vanilla Forward(backward) algorithm arises for longer simulations. As per the definition of the Forward algorithm, at each step the predicted probabilities of each state are acquired as follows

$$P(X_t|e_1^t) = P(e_t|X_t) \sum_{x_{t-1}} P(X_t|x_{t-1}) P(x_{t-1}|e_1^{t-1}), \tag{13}$$

where the left part of the equation is the new estimate, first part of the right side is the sensor model and the sum consists of the transition model and the estimate from the previous step. Exactly this continuous multiplication between probabilities leads to decreasing estimation values and inevitably to underflow as the numbers become too small for the `double` type representation. The following figure 6 show the dependency of the highest state probability and the number of iterations.

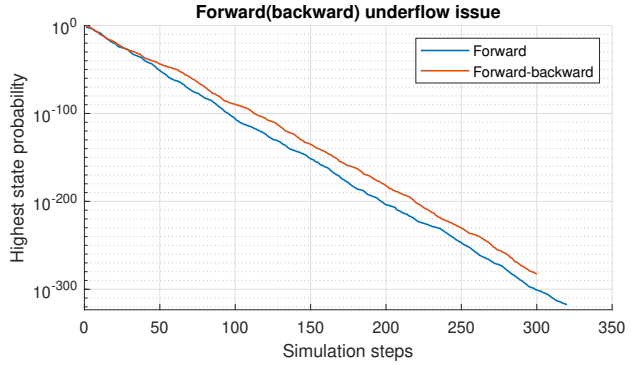

Fig. 6: The dependency of robot position probability on the number of simulation iterations

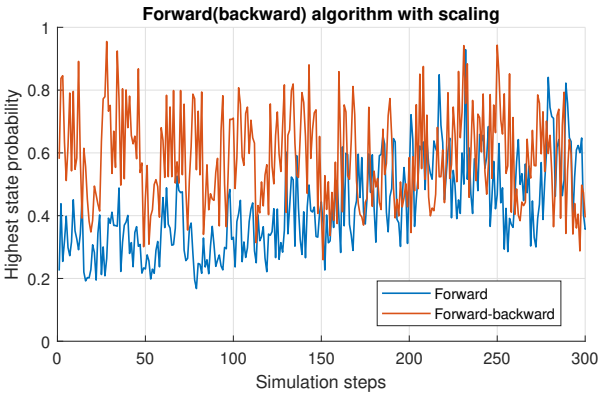

Fig. 7: The dependency of robot position probability on the number of simulation iterations with scaling introduced

Note

To ensure consistency all experiments throughout this and next sections were carried out in the *wasteland* map which was introduced in the first section as per fig. 4.

As we can see the algorithms will reliably underflow in between 300-350 iterations and are rendered useless afterwards. To battle this issue we will use scaling. For the forward algorithm this simply means to normalize the new estimate in every step. In words, the new estimate $P(X_t|e_1^t)$ consists of pairs of states and probabilities that the robot finds itself in this given state. By simply normalizing these probabilities in every step we prevent the diminishing of the future state probabilities. That is, for every step we calculate the scaling factor

$$c_t = 1/\sum_{x \in X} p_t(x) \tag{14}$$

where $p_t(x)$ is the probability for given state in given time (step) t. We then use this factor to normalize the individual probabilities as follows

$$\hat{p}_t(x) = c_t p_t(x). \tag{15}$$

If we now look at the way we define the Forward-backward algorithm

$$P(e_{k+1}^t|X_k) = \sum_{x_{k+1}} P(e_{k+1}|x_{k+1}) P(e_{k+2}^t|x_{k+1}) P(x_{x+1}|X_k), \tag{16}$$

we can see that it is similar to the Forward algorithm but improved by the recursion step $P(e_{k+2}^t|x_{k+1})$. To deal with the underflow issue for the Forward-backward algorithm we compute the forward pass first and store the normalization factors c_t. We then use them in the backward pass where similarly as in (13) we end up with the pairs of states and their probabilities where we use the beforehand acquired factors to normalize these probabilities. The following figure 7 shows the highest state probability with introduced scaling and we can see that the probability indeed does not diminish as in the previous case.

But to determine whether the introduced scaling mechanic indeed results in a better state predictor we need to compare the performance and even the individual guesses of the Forward-backward algorithm with and without scaling. The following figure 8 helps us visualize this. By integrating the hits of the individual algorithms as well as the information about matching predictions we can see that for the first cca 300 iterations both version predict the same states and therefore both of their accuracies are identical. But as expected after those first 300 iterations the bare-bone Forward(backward) algorithm underflows and fails to make any more successful predictions whereas the number of successful predictions of the scaled version grows further. We can thus conclude that the introduced scaling mechanic indeed improves the performance of the algorithms.

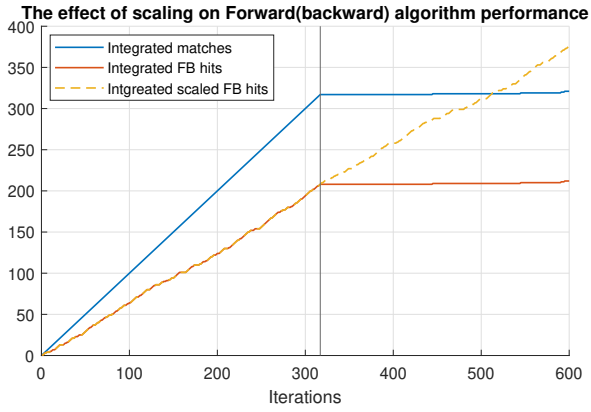

Fig. 8: Effect of scaling on the the Forward(backward) algorithm

C. Viterbi algorithm using log probabilities

Although the inner workings of the Viterbi algorithm and the Forward(backward) algorithms are different the issue of underflowing probabilities is still present. Firstly, to initialize

the algorithm, the Forward algorithm with scaling is used. Then, working from the final state the Viterbi algorithm tries to identify the most likely sequence of states by calculating the following max message at times $t \geq 2$

$$m_t(X_t) = P(e_t|X_t)\max_{x_{t-1}} P(X_t|x_{t-1})m_{t-1}(x_{t-1}), \quad (17)$$

where $P(e_t|X_t)$ is the emission probability, $P(X_t|x_{t-1})$ transition probability and $m_{t-1}(x_{t-1})$ the probability of previous max message. We can see the evolution of the max message probability in the following figure 9. As expected the probabilities diminish even quicker than in the example of Forward(backward) algorithms.

To battle this issue we will use so called logarithmic sum approach. Simply put instead of calculating

$$p_{curr} = p_e \times p_t \times p_{prev} \quad (18)$$

we will use

$$p_{curr} = \log p_e + \log p_t + p_{prev}. \quad (19)$$

In case either the transition or emission probabilities equal zero, we can set the current probability to a large negative number, ideally $-\infty$.

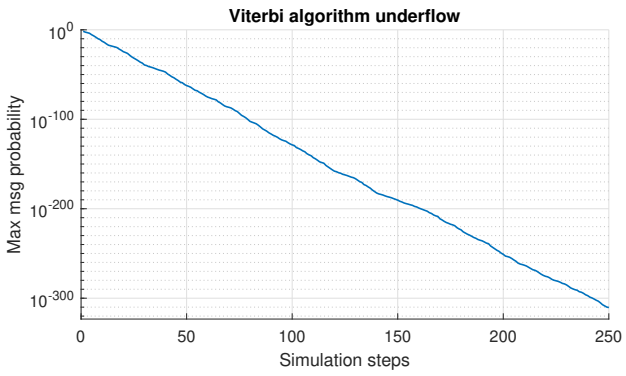

Fig. 9: The underflow of probabilities in the Viterbi algorithm

To verify the proper workings of this principle we can perform the same experiment as in the previous section - that is to measure the predictions for a long simulations and observe how the hitrate evolves as well how similarly the two algorithms predict the states. As we can see in the following figure 10 up until the point of the inevitable underflow of the original Viterbi algorithm, both of the variants predict the states identically and have thus identical hit rates. After the underflow occurs the non-logarithmic Viterbi has only a minimal chance of guessing the correct state as it simply returns any random state. On the other hand the logarithmic Viterbi algorithm continues to predict the states with similar success rate as before.

Note

The Viterbi and logarithmic Viterbi algorithms do not have to strictly match in their predictions 100% of the time. Different predictions can occur in instances when multiple maximum probability messages are calculated. From these messages any message can be selected as the final one and this can lead to rare miss-matches between these two algorithms.

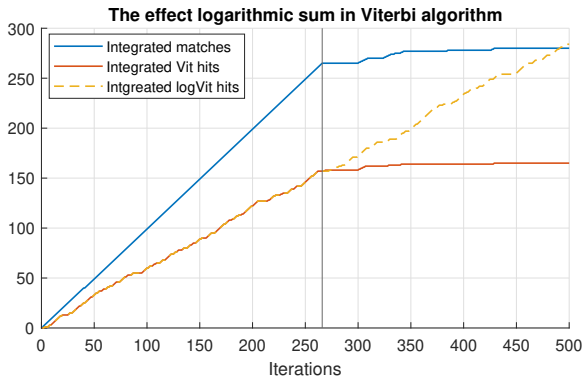

Fig. 10: Performance comparison of vanilla and logarithmic Viterbi algorithms

D. Baum–Welch algorithm

The Baum–Welch algorithm is an expectation–maximization algorithm for solving the learning problem in HMM. In other words, it allows for estimating the parameters of an unknown HMM (transition and emission probabilities etc.), just from the available sequence of observations.

This iterative algorithm itself uses the forward–backward algorithm and can be broken into two successive steps that are performed until the algorithm converges or until the maximal number of iterations is reached. These two steps are the following. First forward–backward algorithm is performed and the latent variables $\xi_t(i,j)$, $\gamma_i(t)$ are computed. Then these are used for updating the transition and observation matrices **A** and **B**. [1].

The aforementioned latent variables ξ and γ describe individual probabilities, that are of use when computing the new matrices **A** and **B**. The forward–backward algorithm leaves us with two sets of probabilities, namely the *forward* probabilities α and *backward* probabilities β, where $\alpha_t(i)$ gives the probability that the agent finds himself in state i at time t given the observations up to time t, i.e.

$$\alpha_t(i) = P(x_i|e_1^t) \quad (20)$$

and $\beta_t(i)$ is the probability that the agent is in state i at time t given all the observations beginning at $t+1$ and going into the future

$$\beta_t(i) = P(x_i|e_{t+1}^N). \quad (21)$$

Having the *forward* and *backward* probabilities, we can now compute γ and ξ. ξ gives the probability that the agent is in state i at time t and that he will be in state j at $t+1$. This can be computed as

$$\xi_t(i,j) = \frac{\alpha_t(i)\mathbf{A}_{i,j}\mathbf{B}_{j,O_{t+1}}\beta_t(j)}{\sum_{i=1}^{M}\sum_{j=1}^{M}\alpha_t(i)\mathbf{A}_{i,j}\mathbf{B}_{j,O_{t+1}}\beta_t(j)}, \quad (22)$$

Iteration	$\|\|\mathbf{A} - \mathbf{A}_{est}\|\|_F$	$\|\|\mathbf{B} - \mathbf{B}_{est}\|\|_F$
1	3.926	3.717
2	3.937	3.841
3	3.957	3.840
4	4.016	3.806
5	4.206	3.813
6	4.618	3.796
7	4.970	3.802
8	5.136	3.793
9	5.296	3.799
10	5.740	3.794

TABLE V: Results of the Baum–Welch algorithm in terms of Frobenius distance of estimated and true HMM matrices.

where O_{t+1} denotes the observation at time $t+1$ and M is the number of states.

γ encapsulates the combined probability of α and β that is, how likely is the agent in state i at time t given the whole observation sequence and can be obtained as

$$\gamma_t(i) = \sum_{j=1}^{M} \xi_t(i,j). \tag{23}$$

Then follows the second phase of the algorithm, namely the updating of matrices **A** and **B**. This is done according to the following relations.

$$\mathbf{A}_{i,j} = \frac{\sum_{t=1}^{T-1} \xi_t i,j}{\sum_{t=1}^{T-1} \gamma_t(i)}, \tag{24}$$

$$\mathbf{B}_{i,j} = \frac{\sum_{t=1}^{T} 1_{O_t=j} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}, \tag{25}$$

where $1_{O_t=j}$ is the indicator function

$$1_{y_t=v_k} = \begin{cases} 1, & \text{if} y_t = v_k, \\ 0, & \text{otherwise.} \end{cases} \tag{26}$$

Put into words, the transition matrix is updated by the ratio of cases when state i lead to state j and the total expected number of transition from state i and the observation matrix's elements become the ratio of the number of given observations in state j to the total expected number of times that the agent was in state i [2].

The algorithm can be described with the following pseudo–code 1.

The implemented algorithm was evaluated in the following way. A trajectory of 50 states (and thus 50 observations) was generated and the estimated parameters were compared to the real parameters which were used when generating the sequence. In each iteration of the algorithm the Frobenius norm between the true and estimated emission and transition matrices was computed as well as the likelihood, that the given HMM will generate the considered trajectory. The results are summarized in Tables V and VI.

Achieved results in Tables V and VI show, that the algorithm is capable of estimating the the underlying HMM to a certain extent. From table VI one can notice, that in the beginning the estimated model was much less likely to generate the trajectory than the real model, but after couple of iterations the likelihoods got closer, meaning that the estimated parameters

Algorithm 1 Baum–Welch algorithm

$\pi \ldots$ prior belief,
$O \ldots$ sequence of observations
$\mathbf{A} \ldots$ transition matrix
$\mathbf{B} \ldots$ observation matrix
function BAUMWELCH(π, O, **A**, **B**)
  for $iteration = 1, 2, \ldots, n\_iters$ **do**
    # initialize the latent variables
    α = forward(π, O, **A**, **B**)
    β = backward(O, **A**, **B**)
    $\xi_t(i,j)$ = empty $\forall i, j, t$
    $\gamma_t(i)$ = empty $\forall i, t$
    # compute ξ and γ
    for $t = 1, 2, \ldots, |O| - 1$ **do**
      for $i = 1, 2, \ldots, M$ **do**
        for $j = 1, 2, \ldots, M$ **do**
          $\xi_t(i,j) = \alpha_t(i)\mathbf{A}_{i,j}\mathbf{B}_{j,O_{t+1}}\beta_{t+1}(j)$
        end for
      end for
      $\xi(i,j) = \xi(i,j) / \sum_{k,l} \xi_t(k,l)$
    end for
    $\gamma_t(i) = \sum_j \xi_t(i,j)$
    # update **A** and **B**
    A = $\sum_t \xi_t / \sum_t \gamma_t$
    B = $\sum_t 1_{O_t=j} \gamma_t / \sum_t \gamma_t$
    # update prior belief
    $\pi = \gamma_0$
  end for
end function

Iteration	Likelihood HMM$_{true}$	Likelihood HMM$_{est}$
1	1.07e^{-45}	2.45e^{-80}
2	1.06e^{-45}	5.01e^{-48}
3	1.10e^{-45}	4.22e^{-48}
4	1.09e^{-45}	3.41e^{-48}
5	1.12e^{-45}	1.62e^{-48}
6	1.16e^{-45}	1.87e^{-49}
7	1.20e^{-45}	4.65e^{-50}
8	1.17e^{-45}	7.18e^{-51}
9	1.11e^{-45}	3.35e^{-52}
10	1.09e^{-45}	1.13e^{-53}

TABLE VI: Results of the Baum–Welch algorithm in terms of likelihood.

resembled the original HMM more precisely. It is also worth noting that in this application the observation matrix **B** converges better than matrix **A** as can be inferred from V. To add up, the transition matrix seems to even slowly diverge from its true values, as the Frobenius norm keeps increasing slightly. This could be caused either by a bug in the implementation or by the nature of the problem, that the random states and observation point more towards a different parameter values.

V. Conclusion

The presented text dealt with how well the individual HMM–based prediction algorithm, namely the filtering, smoothing and Viterbi algorithms, will tackle the robot localization problem.

For this purpose two metrics were used, first the scaled Manhattan distance which proved to yield the best results for individual algorithms when considering only the most probable measurement. This lead us to the idea to try another simple metric, the hit/miss score. Both these measures showed that the Viterbi and Forward–backward algorithms with 0.65 and 0.66 hit rate respectively, perform comparably well and that they outperform the simple Forward algorithm with only 0.5 hit rate, as presented in Fig. 2. Similar conclusions can be drawn also from the first, Manhattan distance–based metric, where no matter the amount of number of considered fields, the Forward algorithm has always lagged behind the other two, as can be inferred from Fig. 1.

Similar results in terms of accuracy of individual algorithms were achieved even when we experimented with different maps, see Fig. 5. Therefore we conclude that the most successful algorithm is the Forward–backward algorithm which exhibited the highest hit rate across all performed experiments and on top of that showcased the best behaviour in terms of E_{fb} for $k = 1$, as is discussed in Subsection III-A.

After the evaluation part, a part where a couple of improvements to the used prediction algorithms were implemented. First, the inefficient `for` loops were replaced with matrix multiplication, which resulted in a substantial decrease in computation time in the filtering and smoothing algorithm, where the time needed was nearly constant regardless of how long the observation sequence was, as is presented in Tables II and III.

Then the underflow issue was addressed and two possible solutions for individual algorithms were implemented. With the forward and forward–backward algorithms, scaling was introduced to battle this issue. The employment of said algorithm successfully suppressed the effect of underflow, as can be seen in Fig. 7. The issue with underflowing can occur also with the Viterbi algorithm. The prevent this, log probabilities were introduced instead of scaling in this instance, as per Subsection IV-C. The so–called third problem of HMM–learning was solved with the implemented Baum–Welch algorithm as per Subsection IV-D. The implementation is probably not flawless, as the transition matrix seems to be diverging from its true values, but still the algorithm is able to predict such values, that are similarly likely to generate the same observation sequence as the original underlying HMM as is backed up by results in Tables V and VI. This can serve as a basic sanity check.

To conclude, the framework of HMM provides an interesting tool even for problems such as robot localization in a maze, which performs reasonably well despite the lack of accurate sensors and random movements of the robot. However where there are conditions and means for precise localization and sensors employment, a more accurate results will without a doubt be achieved through more standard ways of localization.

REFERENCES

[1] HINNO, Risto: *Baum–Welch algorithm. What's under the hood of Hidden Markov Models training*, medium.com, 2021, [available at: https://medium.com/mlearning-ai/baum-welch-algorithm-4d4514cf9dbe].

[2] WIKIPEDIA: *Baum–Welch algorithm*, Wikipedia, the Free Encyclopedia, [available at: https://en.wikipedia.org/wiki/Baum%E2%80%93Welch_algorithm#Algorithm].