“Robot” localisation with HMM based forward-filtering

This task relies on the explanations for matrix based forward filtering operations according to section 14.3.1 of the book (15.3.1 in the previous, 3rd, edition, 14.3.1 in the 2nd edition).

The purpose of this assignment

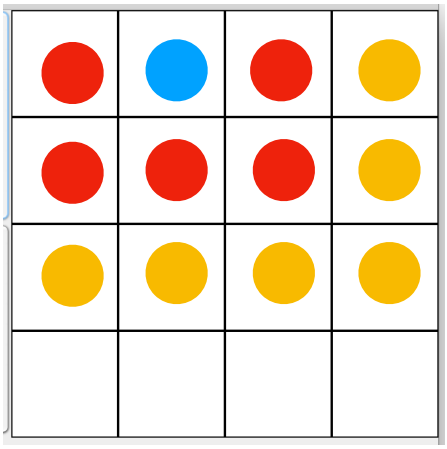
The idea with this assignment is to make you reflect about what you can do with a toy example implementation and how this is related to real world problems and applications. The idea is not to have you simply tick off a box when your implementation works, or to reach a specific number or other measure of quality. We expect you to read (the instructions, but also the theory behind the task in the book), discuss, and think for yourself. There is no absolute recipe to follow!

Task description

Please do not start out with the implementation, do the thinking and understanding first, it helps!

1) The scenario and overall problem to handle

You are supposed to track an agent (robot) (with forward filtering and smoothing based on an HMM) in an environment without any landmarks. Consider this agent to live *in an empty room*, represented by an *n x m* rectangular grid, that can move only along the rows and columns in the grid. The robot’s location is hidden; the only evidence available to you (the observer) is a noisy sensor that gives a direct, but not necessarily correct, approximation to the robot’s location. There are two sensor models for you to work with, but for both, the sensor gives approximations $S = (x',y')$ for the (true) location $L = (x, y)$, marked blue in the image, the directly surrounding fields *Ls* (red), or the “second surrounding ring” *Ls2* (yellow) according to the specifications below. Here, *n_Ls* is the number of directly surrounding fields for *L* (this can be 3, 5, or 8, depending on whether *L* is located in a corner, along a “wall”, or at least 1 step away from any “wall”) and *n_Ls2* is the number of secondary surrounding fields for *L* (this can be 5, 6, 7, 9, 11, or 16, depending on where *L* is located relative to “walls” and corners).



The sensor reports

- the true location *L* (blue) with probability 0.1 in both models,
- any of the $n\_Ls \in \{3, 5, 8\}$ existing surrounding fields *Ls* (red, here $n\_Ls = 5$) with a probability *p(Ls)* between 0.05 and 0.13 depending on the model and the number *n_Ls*,
- any of the $n\_Ls2 \in \{5, 6, 7, 9, 11, 16\}$ existing “secondary” surrounding fields *Ls2* (yellow, here $n\_Ls2 = 6$) with a probability *p(Ls2)* between 0.025 and 0.8 depending on the model and the number *n_Ls2*,
- "nothing"
  - with a non-uniform probability of $p(\text{“none”}) = 1.0 - 0.1 - n\_Ls*p(Ls) - n\_Ls2*p(Ls2)$ (sensor model NUF, non-uniform failure), or
  - with uniform probability over the entire grid $p(\text{“none”}) = 0.1$ (sensor model UF, uniform failure).

The robot moves according to the following strategy:

Pick random start heading h_0 among the four possible headings. For any new step pick new heading h_t+1 based on the current heading h_t according to:

P(h_t+1 = h_t | not encountering a wall) = 0.7
P(h_t+1 != h_t | not encountering a wall) = 0.3
P(h_t+1 = h_t | encountering a wall) = 0.0
P(h_t+1 != h_t | encountering a wall) = 1.0

It then moves in the direction h_t+1 by one direct step in the grid. This means essentially that a) it will always move one step and b) it can only move straight.

In case a new heading is to be found, the new one is randomly chosen from the possible ones (facing a wall somewhere along the wall leaves three, facing the wall in a corner leaves two options for where to turn).

2) Understanding the given models and viewer tool (Your TODO #1!)

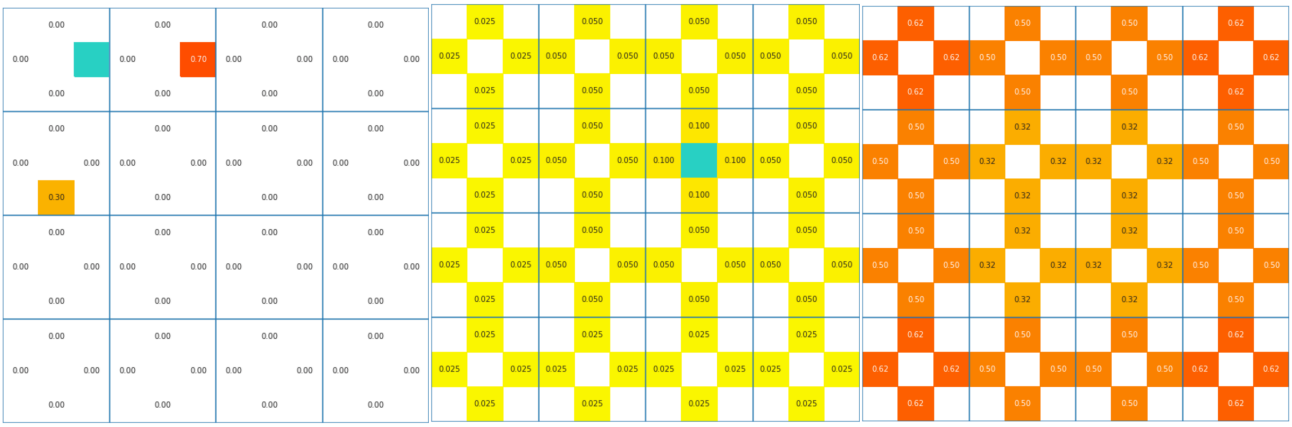
Essentially, you are supposed to implement a localisation approach for the robot using the models described above and a given simulation for the robot (movement and sensor readings). To do that, you are given a handout with the following files that implement the above described specifications for the robot (transition and observation models, as well as robot simulation) and the glue around them. Inspect these, consult the documentation ("Viewer Guide") and find your way through the handout, in particular work to understand the models in relation to the theory described in the book chapter and how you can make use of them later.
You are given:

- A Jupyter notebook (*TaskPart1*…) that can be used to run and visualise the application (see "Viewer Guide" for some description of the GUI in the second cell). Note that the two cells with code are independent of each other, the first shows how you can work with the models without the graphical interface, the second invokes the graphical interface.
- A viewer class (Dashboard) in which the visualisation is implemented (you do not need to change anything in there). The Dashboard makes use of another "controller" class (Localizer), which can control the localisation process for use with the visualisation and / or for test runs. In the handout version, it calls the stubb method *Filters.HMMFilter.filter(sensorReading)*, which basically does nothing (see below).
  The Localizer (as given) is not possible to be used directly for the comparisons of different setups / different sensor models, as it (as of now) cannot feed the same trajectory into two different sensor model runs.
- Four files containing the models that implement the above described scenario in form of a state model, and the transition and observation models for the HMM:
  - *StateModel.py* contains the overall information of how states (poses) are encoded and provides a number of methods to transform the different representations.
  - *ObservationModel_NUF.py* and *ObservationModel_UF.py* hold the observation models for the sensor with non-uniform failure (NUF) or uniform failure (UF) probability respectively and provide some methods each to access the probabilities stored in the respective matrices.
  - *TransitionModel.py* holds the transition model in form of the respective matrix.

Please consult the documentation INSIDE the model files for more details.
One task in the writing part (report) is to explain the models, hence you should be able to show that you have understood what they are good for and what the information in them actually means.

- A file *RobotSim.py* that contains the robot simulator (movement and sensor reading) class *RobotSim*. This class provides two methods, *move_once(transitionModel)* and *sense_in_current_state(observationModel)*, which you can use to simulate the robot movement and sensor readings after initialising the respective instance with the starting position (typically randomised).
- A file *Filters.py* that contains a class stubb *HMMFilter* and one called HMMSmoother for the filtering / smoothing methods. Note that the method *filter(sensorReading) in HMMFilter* is called from the *Localizer* for the visualisation.
- A second Jupyter Notebook for convenience, in which you can implement control code for tasks 2 and 3.

When starting to work with the handout, make sure you get the following results when visualising the transition model (left) and the sensor/observation model "0" (non-uniform failure) (centre and right) for a 4x4-grid with the GUI.



3) Implementation (Your TODO #2!)

Implement a localisation / tracking approach based on an HMM with a) simple **Forward Filtering** and b) **Forward-Backward Smoothing** with k = t-5 (a sequence length of 5) to track the robot (according to the matrix-vector notation suggested in section 14.3.1 of the course book and the respective lecture material, i.e. make use of the given models!).
This requires obviously a two-part implementation, as you first need to "simulate the robot" with its movement a) to have some ground truth to evaluate your tracking against, and b) to simulate a sensor reading from this robot for the HMM-based tracking algorithm.
You must make use of the given models for the robot simulation and the filtering / smoothing!

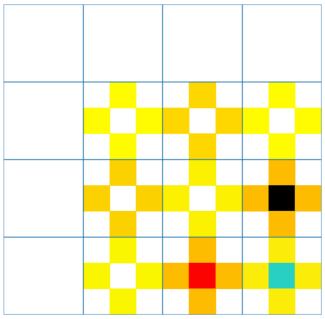
Your algorithm should basically loop over the following three steps, where steps 2 and 3 can have several instances for the comparing evaluations:

1. Move (simulated) robot to new pose using *move_once(...)* in *RobotSim.py;*
2. obtain (simulated) sensor reading(s) using *sense_in_current_state(...),* with the observation model you want to use. Note that it is possible to call

sense_in_current_state(...) as often as you like with different observation model instances, getting a sensor reading will not affect the true pose;

3. update the position estimate with either the forward- or smoothing approach based on the sensor reading(s) from step 2, using the known observation and transition models in your implementations of the forward filter and smoothing algorithms.

Note that a sensor reading of "nothing" normally means to do the forward step without update, i.e. it boils down to mere prediction in theory. However, here, you should not go the normal way, but always do a complete update.
Thus, even a "nothing" reading from the sensor should entail a proper prediction + update step!



If you use an 8x8 grid, the non-uniform-failure observation model, and forward filtering, you should observe something like 25% (probably more) of correct estimates rather quickly, roughly 100 steps should already get you there safely - note that this is only given as a *reference* for you. The average Manhattan distance should then be somewhere around 2.0 (probably below). If you test through the GUI, summed-up probabilities are sent to the viewer (i.e. all four state entries belonging to one position (grid cell) get the same value, which is the sum over the actual state probabilities in this cell). In this case, it is quite common to observe a "checker-board" pattern in the visualisation image with lower probabilities in every other position in the grid (in the image you see this also with the most likely position (red), the true position (black) and the sensed position (cyan)).

4) Evaluation (Your TODO #3!)

Evaluate your implementation as follows:
Run series of move-sense-update loops (500 steps should be enough also for the bigger grids) according to the comparisons described below. You can of course adapt the number of steps or the size of the grid if needed, but comment on it later in your report.
In terms of robot localisation it is often not relevant to know how often you are 100% correct with your estimate, but rather, how far "off" your estimate is from reality on average / how often. Thus, measure the **distance between true location and estimate by using the Manhattan distance** (how many robot steps off) and use this as the basis for comparisons, a good way is to use the average Manhattan distance.

Compare and provide **plots of the (average) Manhattan distance over time:**

1. **Forward Filtering** with non-uniform sensor failure on 8x8 grid **against**
   Sensor output only (non-uniform sensor failure, count sensor failures to get the average frequency, but do not count those steps into the avg Manhattan distance) on 8x8 grid
2. **Forward Filtering** with non-uniform sensor failure on 4x4 grid **against**
   Forward Filtering with uniform sensor failure on 4x4 grid
3. **Forward Filtering** with non-uniform sensor failure on 16x20 grid **against**
   Forward Filtering with uniform sensor failure on 16x20 grid
4. **Forward Filtering** with non-uniform sensor failure on 10x10 grid **against**
   Forward-Backward Smoothing with k = t-5 (five steps for b) and non-uniform sensor failure on 10x10 grid

NOTE: obviously, each pair-wise evaluation should be run based on the same true trajectory (cases 1, 2, 3) or same trajectory AND same sensor reading sequence (for case 4).

5) Peer review (Your TODO #4!)

Find a peer working group for a review and discussion when you have a working implementation (it does not need to be perfect, but it should not produce any blatant errors).
Note though that this is not a coding competition and that there are peers with different backgrounds in the course, so the code does not have to follow the harshest rules or conventions, but you should of course give constructive feedback if you see room for improvement, e.g. regarding efficiency!
Give (and receive) feedback, i.e. coordinate for an actual discussion with your peer.

Improve your implementation if necessary. After the peer review is done, it should not be necessary for any TA to debug your code!

6) Reading article (Your TODO #5!)

Read the article "Monte Carlo Localization: Efficient Position Estimation for Mobile Robots" by Dieter Fox et al., which you can find (for example) at http://robots.stanford.edu/papers/fox.aaai99.pdf.
It was published in the proceedings of the 16th AAAI Conference on Artificial Intelligence (AAAI-99) in 1999, and received the AAAI Classic AI Paper Award in 2017. Examples relating to this paper will be part of the lecture on Probabilistic Robotics.

7) Writing report (Your TODO #6!)

Write a brief (max 3 filled A4-pages excluding images) report that must follow the structure below and enclose all the listed points. Make use of the language check list below to avoid making the reader angry!

- ***Statement*** of who you did the ***peer review*** with, and (if so) who you worked with during the implementation phase (in addition to your partner).
- ***Summary*** of the ***task*** in your own words (this should not be a description of your implementation, but it should explain for a random reader what problem you are supposed to solve and with which methods you tackled the task).
- ***Explanation*** of the ***models*** given with the handouts, including an answer to this *question:* "***What is the actual difference in the two sensor/observation models, and how does this difference affect the localisation with forward filtering in different settings?"***. Discuss / explain especially the impact of the "no reading" result for the two observation models (also called sensor models)!
  You should base this discussion on your own inspection of the viewer, i.e. by visualising the different observation (sensor) models ("0", non-uniform failure, and "1", uniform failure) in the GUI cell of the visualisation notebook.
- ***Discussion*** of your ***results***, answering the *core question:* ***"How accurately can you track the robot with filtering (or smoothing), is it even worth doing that?"***.
- ***Discussion*** of the ***relation*** between your ***implementation*** and the work described in the ***article***, answering the *core question:* ***"Is the HMM approach as implemented suitable for solving the problem of robot localisation?"***. Give clear evidence / motivation for your statement.

If you experience difficulties with understanding the task, getting the implementation to work, etc, please, CONTACT me (Elin, elin_a.topp@cs.lth.se) or a TA well BEFORE the deadline!

Language check list:

- *Write entire sentences! Do this even though it seems now common practice in newspapers and other "official" texts to avoid this antique technique!*
  The second "sentence" of "The grass is wet. This because it rained until 15 minutes ago." or even "The grass is wet. Because it rained until 15 minutes ago." is not a sentence, as it is lacking a predicate (verb). Reading such constructs is tiring for the reader, who would be in search for an overlooked word or starts wondering whether the "." was accidentally placed instead of ",", which would have been correct in the second version, but would still not have helped in the first.

- *Check your grammar regarding the correct numerus of verb forms!*
  Rule of thumb: if the noun is in plural form (has an "s" in most cases[1]), the verb should NOT have an "s" or "es" and vice versa, as in "One bird flies, two birds fly". Respective mistakes should be even possible to catch with automated tools by now, at least the most blatant ones!

- *Decide on British or American English and stick to it!*
  British: colour, neighbour, visualise, e.g. ..., i.e. ...
  American: color, neighbor, visualize, e.g., ..., i.e., …

- *Keep track of possessive forms in comparison to plural forms and think about exceptions.*
  "The robot's arm was stuck because one of its motors broke."

- *Sort out "to" vs. "two" vs. "too", as well as "where" vs. "were"!*
  A typo here alters the meaning of the text (and yes, using the wrong form often is just a typo and just happens... but try to catch mix-ups), or makes it at least more difficult to understand.

- *Consider this to be a formal text!*
  Do not use short forms like "don't", "isn't", "we're", but use the complete expressions "do not", "is not", "we are" / "we were". OBS, exception: the negation of *can* is *cannot*, it is not *can not*.

[1] Some exceptions are: one criterion, two criteria; one child, two children; one fish, two fish