Practicum Assignment 2
(RO47004 / ME41105)
Bayesian filtering and state estimation

Ronald Ensing, Christian Muench and Dariu Gavrila

Intelligent Vehicles group
Delft University of Technology

December 8, 2020

Document Version 1

TU Delft

About the Assignment

You are expected to do the practicum assignments in student pairs, you receive both one grade (if you do not have a partner, post to the ”find a lab partner” Brightspace forum of this course). A maximum of one group can contain a single student, in case of an odd number of students (if you think this involves you, please contact the Practicum Coordinator before starting the assignment).

Please read through this whole document first such that you have an overview of what you need to do.

This assignment contains *Questions* and *Exercises*:

- You should address all of the questions in a 2-3 page report (excluding plots and figures). *Please provide separate answers for each Question in your report, using the same Question number as in this document.* Your answer should address all the issues raised in the Question, but typically should not be longer than a few lines.
- The Exercises are tasks for you to do, typically implementing a function or performing an experiment. Therefore, first study the relevant provided code before working on an exercise, as the code usually contains comments referring to each specific exercise. *If you do not fully understand the exercise, it may become more clear after reading the relevant code comments!* Do not directly address the exercises in your report. Instead, you should submit your solution code together with the report. Experimental results may be requested in accompanying questions.

You will be graded on:

1. Quality of your answers in the report: Did you answer the Questions correctly, and demonstrate understanding of the issue at hand? All Questions are weighted more-or-less equal.
2. Quality of your code: Does the code work as required? Only then you obtain full credit for the corresponding Question.

Submission

- Before you start, go to the course's Brightspace page, and enroll with your partner in a lab group (found under the 'Collaboration' page).
- To submit, upload **two** items on the Brightspace 'Assignments' page:

pdf attachment	A *pdf* with your report. Do not forget to add your student names and ids on the report.
zip attachment	A *zip* archive with your code for this assignment Do NOT add the data files! They are large and we have them already ...
deadline	**Tuesday, January 5th 2019, 23:59**

- Note that only a single submission is required for your group, and only your last group submission is kept by Brightspace. **You are responsible for submitting on time**. Do not wait till the last moment to submit your work, and *verify* that your files have been uploaded correctly. Connection problems and forgotten attachments are not valid excuses. The due deadline is automatically checked by Brightspace. If your submission is not on time,

points will be subtracted from your grade. Within 24 hours after the deadline, one point is subtracted. Between 24 and 48 hours, 2 points are subtracted, and so forth.

- You are only allowed to hand in work (code, report) performed by you and your practicum partner during this course this year. Passing other work as your group's work is considered *scientific fraud* (even if it consists a single line of code that you are "borrowing"). You are equally not permitted to share your work with anyone else outside of your group (including but not limited to WhatsApp, other social media, email, internet). This applies both to during the course and thereafter. Sharing your work will be considered as *abetting scientific fraud*. We have (and will use) manual and automatic means to detect possible instances of *scientific fraud* or *abetting scientific fraud*. We will always report them to the Examination Committee. Sanctions, if the suspicions materialize, could involve a failing grade for the course, an entry in the Academic Dishonesty registry of the Faculty up to expulsion from the MS program. It is not worth it!
- If code is submitted that was written with ill intent, e.g. to manipulate files in the user's home directory that were not specified by the task, you will also be reported to the Examination Committee.

Getting Assistance

The primary occasion to obtain help with this assignment is during the practicum contact hours of the course. An instructor and/or student assistant(s) will be present during the practicum to give you feedback and support. If you find errors, ambiguously phrased exercises, or have another question about this practicum assignment, please use the Brightspace practicum support forum. This way, all students can benefit from the questions and answers equally. If you cannot discuss your issue on the forum, please contact the Practicum Coordinator directly.

Practicum Coordinator: Ronald Ensing (R.M.Ensing@tudelft.nl)

Copyright Notice

Good Luck!

Bayesian filtering and state estimation

The previous lab assignment considered object detection from sensor measurements, in particular classifying pedestrians from video frames. This lab assignment we first focus on the next stage in the processing pipeline, namely combining object detections of multiple moving targets into a track representation. The vehicle in this exercise is equipped with a range sensor that obtains noisy distance measurements of the vehicle surroundings from which the true target tracks must be recovered. We shall use probabilistic filtering with Kalman Filters to predict and update estimated target positions, and to assign measurements to tracks. The second part of the lab then considers the related problem of vehicle self-localization, i.e. determine where is the vehicle within the (known) world. This too will be done using a probabilistic filter, but this time we will deal with non-linear vehicle dynamics and measurements using a particle filter.

1 Kalman filters and multi-object tracking

Exercises refer to code sections in `assignment_mot.py`.

We will start by defining a simple dynamical model for a moving object, such as a person or vehicle. In the context of tracking, such an object of interest is also called a *target*.

In our setup, a target's coordinates are expressed as 2D (x, y) coordinates. The target's state $\mathbf{x}_t = [x_t, y_t, \dot{x}_t, \dot{y}_t]^\top$ at time t describes its position and velocity $(\dot{x}_t, \dot{y}_t)$. We will use a fixed time interval of whole units, i.e. $t = 0, 1, 2, \cdots$ thus $\Delta t = 1$. Let the vector $\mathbf{z}_t = [z_t^x, z_t^y]^\top$ represent a positional measurement at that time instance t. Note that we do *not* directly measure the target's velocity.

We assume that targets generally move at constant velocity, but some deviations on this strict assumption are allowed by including process noise acting on both its position and velocity. This results in an additional random offset $\boldsymbol{\epsilon}_t = [\epsilon_t^x, \epsilon_t^y, \epsilon_t^{\dot{x}}, \epsilon_t^{\dot{y}}]$ at each time instance, and there is noise on the measurements as well, given by offset $\boldsymbol{\eta}_t = [\eta_t^x, \eta_t^y]$. Written as equations,

$$x_t = x_{t-1} + \dot{x}_{t-1}\Delta t + \epsilon_t^x \qquad y_t = y_{t-1} + \dot{y}_{t-1}\Delta t + \epsilon_t^y \tag{1}$$

$$\dot{x}_t = \dot{x}_{t-1} + \epsilon_t^{\dot{x}} \qquad \dot{y}_t = \dot{y}_{t-1} + \epsilon_t^{\dot{y}} \tag{2}$$

$$z_t^x = x_t + \eta_t^x \qquad z_t^y = y_t + \eta_t^y. \tag{3}$$

The noise offsets are assumed to be *independent* and Normally distributed, namely,

$$\epsilon_t^x \sim \mathcal{N}(0, \sigma_{\text{pos}}^2) \qquad \epsilon_t^y \sim \mathcal{N}(0, \sigma_{\text{pos}}^2) \tag{4}$$

$$\epsilon_t^{\dot{x}} \sim \mathcal{N}(0, \sigma_{\text{vel}}^2) \qquad \epsilon_t^{\dot{y}} \sim \mathcal{N}(0, \sigma_{\text{vel}}^2) \tag{5}$$

$$\eta_t^x \sim \mathcal{N}(0, \sigma_{\text{obs}}^2) \qquad \eta_t^y \sim \mathcal{N}(0, \sigma_{\text{obs}}^2) \tag{6}$$

where σ_{pos}^2, σ_{vel}^2 and σ_{obs}^2 are the variances of the noise distributions (noise for x and y are equally distributed). To track the target, we first reformulate these assumptions about the dynamics as a single Linear Dynamical System, see Appendix A.

Question 1.1. Formulate the above dynamics using Equations (10) and (11) as a single LDS with 4D state $\mathbf{x}_t$ and 2D measurements $\mathbf{z}_t$. This entails that you have to determine matrices F, H, $\Sigma_{\mathbf{x}}$, and $\Sigma_{\mathbf{z}}$. Express $\Sigma_{\mathbf{x}}$, and $\Sigma_{\mathbf{z}}$ in terms of σ_{pos}^2, σ_{vel}^2 and σ_{obs}^2, and note the

tip in the Appendix on covariance matrices for independently distributed random variables. What four matrices did you find?

Once we have formulated our problem as a LDS, the (linear) Kalman Filter (KF) can be used to estimate the target's true position, and maintain an uncertainty estimate. We shall implement the Kalman Filter using this LDS.

Exercise 1.1. Add the missing LDS matrices that you found in the previous question to the KF setup in `kf.py` under `__init__`. Values for the noise parameters σ^2_{pos}, σ^2_{vel} and σ^2_{obs} are given in that file too. The code block in the assignment script performs some checks on your solution.

Before moving to a situation with measurements, we focus on studying the effect of the Kalman predict step on a state distribution. Let's assume a simple initial state distribution, namely $P(\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_0|\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$ with $\boldsymbol{\mu}_0 = [0, 0, 0, 0]^\top$ and $\boldsymbol{\Sigma}_0 = I$ (where I means the identity matrix).

Exercise 1.2. Implement the predict step in `kf.py` in the `pred_step` function. The equations of the Kalman Filter predict step can be found in Appendix B. The code in the assignment script should show you that repeated application of the predict step will increase the initial uncertainty in both x and y direction.

The function that visualizes the distribution takes as argument the two dimensions from the state vector for which the distribution is plotted.

Exercise 1.3. Instead of dimensions 0 and 1, now visualize how the predict step affected the uncertainty on state dimensions 0 and 2, being x_t and $\dot{x}_t$. Then, also study the uncertainty for state dimensions 0 and 3, being x_t and $\dot{y}_t$.

Question 1.2. Include the final plot for dimensions 0 & 2, and for 0 & 3 in your report. Explain the direction of the shapes that you see: Are x_t and $\dot{x}_t$ (not?) correlated? How would you intuitively explain this? Are x_t and $\dot{y}_t$ (not?) correlated? How would you intuitively explain this?

Single target Kalman Filter We now turn towards a simulation where there is a single pedestrian moving in front of the vehicle. In our test setup demonstrated in Figure 1, the vehicle is not moving, and has a $90°$ front-facing distance sensor which measures proximity to nearby objects, such as pedestrians. Our goal is to use the sensor to monitor the presence of any pedestrians, and if detected, follow their path in front of the vehicle, and express their estimated positions as a sequence of groundplane coordinates.

Exercise 1.4. Run the code block with simulation code, which generates simulated measurements. A figure shows a matrix with all measurement over the T time steps. New random measurements can be made by altering the seed in `np.random.seed`. You can also change the variable `scenario_version` from 1 to 2 to make the simulated pedestrian stop half way.

Exercise 1.5. For both scenario versions ('crossing' and 'stopping'), also run the next code block which animates the simulated situation. Try to see if you can visualize a single frame from the animation by changing the time steps in the animation for-loop. Also try to speed-up or slow-down the animation with the `plt.pause` command. Understanding how

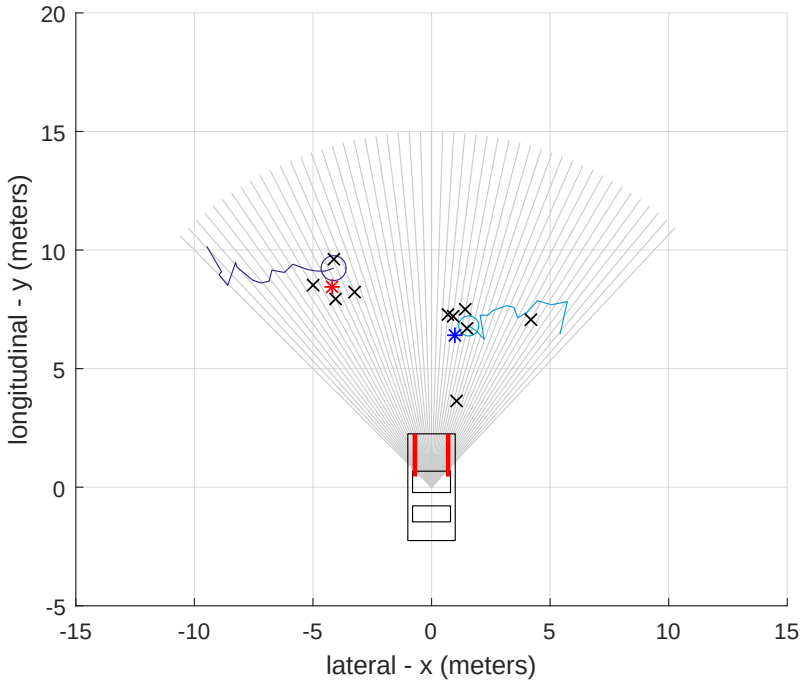

Figure 1: Multi-object tracking in the groundplane, seen in a top-down view. The vehicle's sensor is located at the origin of the coordinate system. The sensor measures distances up to 15 meters along R rays over a $90°$ angle. In the shown time instance, a pedestrian is crossing in front of the vehicle (true target positions shown as blue star). The $\times$ marks indicate measured distances along the sensor rays. The solid blue circle shows the filtered 2D mean position of the target, and its variance at 1 standard deviation. The blue line shows the estimated position from past frames. The dotted blue circle shows the 1 standard deviation of where the filter expects the noisy observations to be.

the script works will help you debug and study the results in the following exercises.

Now let's see what your Kalman predict implementation does if we initialize it on top of the pedestrian's position, and predict its movements.

Exercise 1.6. Run the Kalman predict step on the simulated scenario.

Clearly, the uncertainty distribution only grows as the filter never receives new information on the target. We must utilize the measurements to reduce its uncertainty. The `run_single_kf` implements the minimal Kalman Filter algorithm. For each time step, it (1) converts the measured distances along the rays to 2D positions, (2) executes the KF predict step, and finally (3) executes the KF update step on usable measurements.

Exercise 1.7. Implement the update step in the `update_step` function in `kf.py`. The equations of the Kalman Filter update step can also be found in Appendix B.

Your implementation should now be able to track the pedestrian reasonably well!

Exercise 1.8. Experiment with increasing and decreasing process noise Σ_x and observation noise Σ_y, for instance by scaling all its elements by 0.01 or 100 (see the commented line in the assignment script just after calling `Kf()`).

Question 1.3. When does the track become more smooth, when the process noise and/or observation noise covariance is large or small? What for effect does this have on the pedestrian 'stopping' scenario?

Single target with outliers Now we consider that the sensor measurements also include false positives, which are random and therefore do not provide any evidence on the target's position. We refer to such measurements as *outliers*.

Exercise 1.9. Run your current KF implementation on the same scenario but with the new measurements that contain outliers.

We can try to remove outliers by pre-processing the range measurements. For instance, we can compare the distance measurement of each sensor ray to that of the adjacent rays, and remove large differences. Hence, this is a form of *spatial filtering* (not to be confused with the *temporal filtering* of the KF). For this example, we shall use a median filter for spatial filtering. For instance, a 3-rd order median filter compares each distance measurement to its two surrounding measurements, and takes the median value of all three.

Exercise 1.10. A spatial median filter implementation is provided in the `preprocess_measurements` function. Try out different filter orders in the function, namely 1-st order, 3-rd order, 7-th order, and see what happens to the measurements in the measurement matrix visualization.

Question 1.4. Which order works best for the current setup? Add a matrix visualization with all the pre-processed sensor measurement in your report.

A different approach is not to alter the measurements themselves, but to try to determine which measurements are from the tracked target, and which should be discarded. This is done through comparing the measurements to the filtered position of the KF. If the distance between the two positions is below a certain *"gating"* threshold, the measurement can be used in the update step.

Exercise 1.11. Implement this gating strategy in `test_gating_score`: compute the Euclidean distance between the measurement, and the expected position given by the Kalman Filter. Accept the measurement if the distance is below some threshold (in meters). Try to get a feeling of what threshold works good by looking at the tracking results.

Question 1.5. What happens if the gating threshold is too large (e.g. 10 meters)? What happens if the gating threshold is too small (e.g. 1 meters)? Which gating threshold did you select?

Two known targets Now we move to the next scenario where there are two moving targets, for which there are again two scenario version: 'crossing' and 'stopping'.

Exercise 1.12. Generate measurements for the two target scenario, and plot the measurements over all time steps in a single figure. Do this for both scenario versions.

We assume that we know where the targets start, and initialize a KF for each. The function `run_multiple_kfs` executes the predict and update steps for both KFs, using *data association* at each time step to decide which measurements belong to which KF.

Exercise 1.13. In `associate_measurements_to_tracks`, finish the code to assign measurements to tracks. To do this, we compare a measurement's gating scores for all Kalman Filters, and assign it to the best scoring KF (if it exceeds the gating threshold).

Multi-object tracking In general, we do not know how many targets there are (or if there are any targets at all). Our tracker therefore needs to initialize and remove KFs when it is appropriate. The provided function `run_multi_object_tracker` extends `run_multiple_kfs` with track management code. It starts a new KF for unused observations, and terminates it when it has had no updates for too many iterations.

Exercise 1.14. Run your multi-object tracker on the scenario. Try to create an optimal multi-object tracker by pre-processing and/or adjusting your gating threshold, and/or the track termination threshold. You probably won't get it to work perfectly, this is inherently a hard problem!

Question 1.6. What settings did you find to work best? How many tracks were created?

Question 1.7. To answer the previous question, did you try out several variations of the simulation by altering the seed of the random number generator `np.random.seed` ? Motivate your choice: why is it important to try/not try variations of the scenario when optimizing an intelligent vehicle's tracker?

2 Vehicle self-localization

Exercises refer to code sections in `assignment_self_localization.py`.

After tracking other objects in the first part of the assignment, we now turn to localizing the vehicle itself. Consider the situation where we have a description of the static environment. One such description is an *occupancy map*, also known as *occupancy grid*. In our example, we use a simple binary occupancy grid which divides the 2D groundplane into square cells. Each cell can either be occupied (i.e. contain an obstacle), or not occupied (i.e. free space). The grid can be used in the sensor model to determine what distance we would measure at a particular location in a particular direction.

The vehicle is equipped with the same range sensor as in the previous exercise, but now has $R = 32$ rays and full $360°$ coverage. Thus, the sensor can reliably measure the distance to the closest obstacle in a large number of directions around the vehicle, see Figure 2. Apart from the measurements, we also know the vehicle's *control input* at each time step, which is given as a 2D vector $\mathbf{u}_t = [v_t, \omega_t]^\top$. Here v_t is the vehicle's velocity (in meters per second), and ω_t its turning rate (in radians per second).

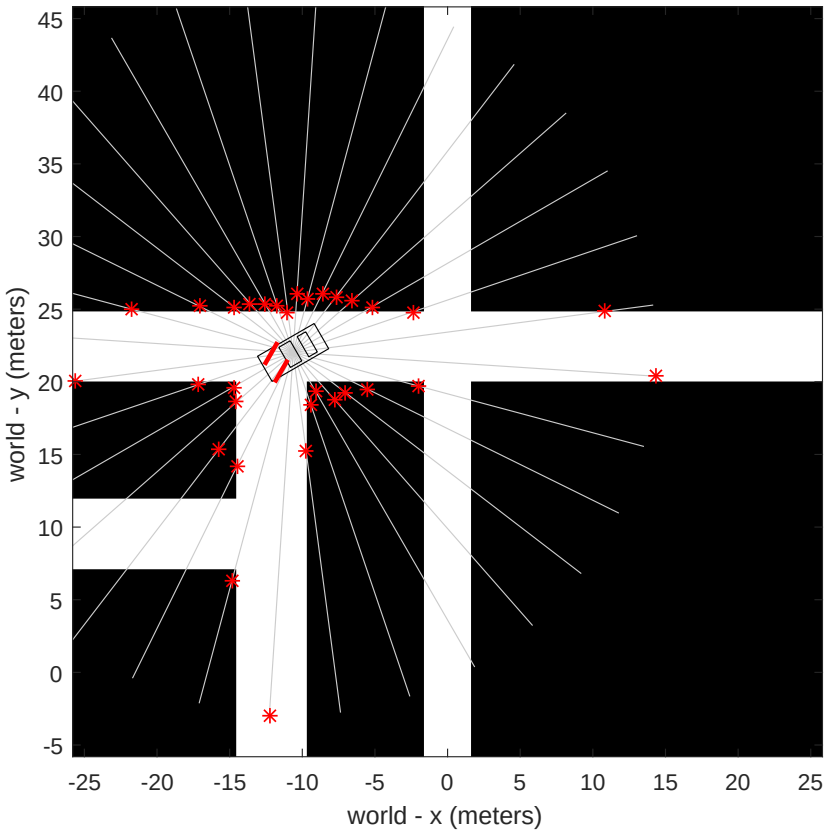

Figure 2: Vehicle self-localization problem. The vehicle moves through a binary grid world, represented by an occupancy map (here black regions define occupied space, white regions show free space). The vehicle has a range sensor with $R = 32$ rays and 360° coverage. The red stars show the R measured distances at the current time step.

Exercise 2.1. Run the self-localization scenario setup and visualization code. You will see the vehicle drive around some street blocks, while measuring its environment.

In the following exercise, you will implement a Particle Filter to track the vehicle itself through its environment using its measurements only.

The particle filter allows you to deal with the non-linear dynamics of the vehicle and sensor model. Additionally, it can represent more complex state distributions than a the single Gaussian of the Kalman Filter. It does this by keeping a set of N particles $S_t = \{\mathbf{x}_t^{(1)}, \cdots \mathbf{x}_t^{(N)}\}$ which represent possible vehicle states at time t. Note that in this notation $\mathbf{x}^{(i)}$ represents the i-th particle. You can think of the particles as hypotheses on the true vehicle state. In our case, each particle $\mathbf{x} = [x, y, \theta]^\top$ will consist of a 2D position (x, y) plus orientation angle θ to describe the forward direction of the vehicle. Please read Appendix C.

PF predict step We will use simple non-linear dynamics f to propagate the particles,

$$f(\mathbf{x}_t^{(i)}, \mathbf{u}_t, \Delta t) = \mathbf{x}_t^{(i)} + \begin{pmatrix} v_t\ \Delta t\ \sin(\theta_t^{(i)}) \\ v_t\ \Delta t\ \cos(\theta_t^{(i)}) \\ \omega_t \Delta_t \end{pmatrix}. \tag{7}$$

These dynamics define the how the particle would move in an 'ideal' deterministic world, i.e. $f(\mathbf{x}_t^{(i)}, \mathbf{u}_t, \Delta t)$ is the *expected* particle state at $\mathbf{x}_{t+1}^{(i)}$. Then, Gaussian noise is added to this predicted state of each particle to account for the uncertainty in the motion, see Equation (25).

Question 2.1. Using Equation (7), write out the expected state $\mathbf{x}_2 = [x_2, y_2, \theta_2]^\top$ as a function of the initial state $\mathbf{x}_0 = [x_0, y_0, \theta_0]^\top$, the control inputs $\mathbf{u}_0$ and $\mathbf{u}_1$ and Δt (assumed to be constant over all frames). Note that $\mathbf{x}_2 = f(f(\mathbf{x}_0, \mathbf{u}_0, \Delta t), \mathbf{u}_1, \Delta t)$, and that we do not consider any Gaussian noise in this question. Give your answer as the three equations, namely for x_2, y_2 and θ_2. Which of these equations is/are linear, and which non-linear?

Exercise 2.2. Complete the PF predict step in `pf_predict_step` by propagating the particles with the dynamics in Equation (7), and then adding the Gaussian noise sampled from $\mathcal{N}(0, \Sigma_x)$. The covariance Σ_x is given in the `pf_predict_step` function.

The assignment code block initializes $N = 1000$ samples at the initial vehicle state (position+orientation), and applies your prediction step several times. It visualizes the position (not orientation) of the particles with a scatter plot. You should see that the particles create a non-linear curved distribution. You can try changing the velocity and steering angle parameters used in the prediction. Notice that the used particle dynamics do *not* take the obstacles from the map into account.

PF update step Next, we will use the measurement to evaluate how likely each particle is. Before implementing code, we empirically study what the measurements would be at a few interesting predefined particle states. In the assignment code, you can select one of several interesting particles $\mathbf{x}$. The code block also shows how to create a virtual sensor located at the particle's position and orientation, and use it to create an ideal 'expected' distance measurement vector $\hat{\boldsymbol{z}} = h(\mathbf{x})$ of its environment.

Exercise 2.3. Run the code block for the 7 different particle positions, and visually compare the expected distance measurements of the particle to the actual measurements obtained at the true vehicle position. Notice that several particles result in very similar measurements.

Question 2.2. Which of the 7 provided particle locations results in a very similar measurement pattern to the actual measurement pattern? Why do the measurement patterns of those particles look like the actual measurements, and why are those of the other particles so different (look at the visualized vehicle state, its surroundings, and sensor measurements)?

The measurement model computes the probability of observing the given measurements at a particular particle state. But instead of normal probabilities, we will use log-probabilities, read Appendix D. Adapting Equation 26, the log measurement likelihood becomes

$$\log P(\mathbf{z}_t|\mathbf{x}_t^{(i)}) = \alpha + \sum_{r=1}^{R} -(z_r - \hat{z}_r^{(i)})^2/2\sigma^2 \tag{8}$$

where α is a certain constant.

Question 2.3. Prove that this equation follows from Equation (26) (you should use the definition of the Gaussian distribution p.d.f.). How does this show that α is constant for all particles $\mathbf{x}_t^{(i)}$?

The likelihood defines the *particle's weight*, but since the weights are eventually renormalized to probabilities that sum to 1, we only need a weight up to a certain constant factor. In the log space, this means that we can drop the α term, i.e.

$$\log w^{(i)} = \sum_{r=1}^{R} -(z_r - \hat{z}_r^{(i)})^2/2\sigma^2. \tag{9}$$

Exercise 2.4. Complete the `map_measurement_loglik` function using Equation (9). Note that the particles in Exercise 2.3 that you found to have similar measurements to the true position should have high probability.

Exercise 2.5. Now implement the PF update step in `pf_update_step`. The update step should evaluate the measurement log-likelihood for each particle, normalize these a probability distribution, and use these probabilities to resample the particles (with replacement).

Complete Particle Filter Next we will combine the predict and update step to run the full particle filtering algorithm on the test scenario. We will start with the situation where the vehicle's initial position and orientation are known.

Exercise 2.6. Run your particle filter on the full scenario. Try your filter with at least $N = 5$, $N = 10$, and $N = 100$ particles, but feel free to try other values. Repeat each experiment several times to reduce the effect of chance in your evaluation. Note that the magenta dots represent the predicted particles, and the greed dots the resampled ones. As a summary of all particles, the mean and covariance are computed, and shown as a blue 2D Gaussian distribution. See Figure 3.

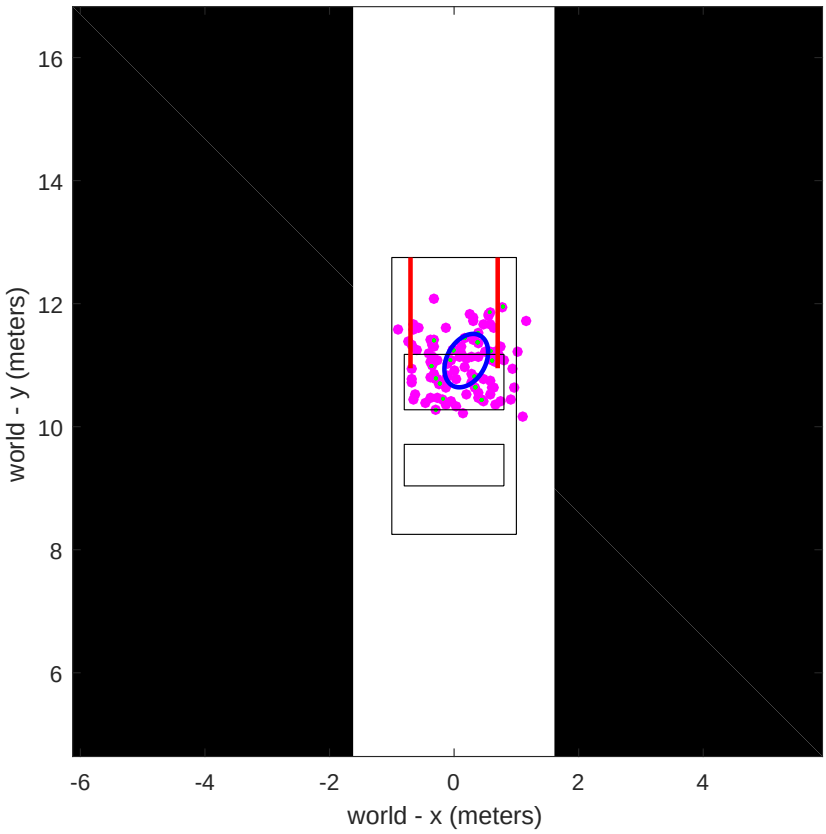

Figure 3: Particle filter with accurate estimate of the vehicle position (closeup on vehicle). The magenta dots show the location of the prediction particles, the green dots which particle state have been resampled (maybe multiple times). The blue 2D gauss shows the overall mean and covariance among the resampled particles.

Question 2.4. What benefit(s) do you notice of having few particles? What benefit(s) do you notice of having many particles? How many particles do you (approximately) need to reliably keep track of the vehicle?

Now let us consider that we do not know where the vehicle is (or maybe we lost track and need to reinitialize).

Exercise 2.7. This time, initialize all particles randomly at the unoccupied cells. You can do this by setting `INITIAL_POSITION_KNOWN = False` at the start of the code block for Exercise 2.6. Run the Particle Filter on the whole sequence multiple times as before, and try out various amount of N. You may want to try out values larger then $N = 100$.

Question 2.5. How does the number of particles correlate with the risk that the filter does not recover the vehicle position by the end of the sequence? How many particles are approximately needed to reliably track the vehicle up to the end of the sequence? Describe your experiments on which you base your conclusions (e.g. which values of N you tried, how many times you repeated each run, etc.).

You may have noticed that if all the particles in the filter have lost track of the actual vehicle state, it will not recover anymore. One strategy to tackle this situation is to randomly reinitialize part of the particles.

Exercise 2.8. Adjust the PF code such that before each update step a percentage of the N particles are replaced by randomly reinitialized particles, see `frac_reinit` in the assignment code block. You can use `pf_init_freespace` to create new initial samples. Try out reinitialization percentages of 5%, 25%, 50%, and 75%, on the situation where the initial position is not known. Do this again with $N = 20$ and $N = 100$ particles multiple times.

Question 2.6. What happens if you reinitialize a very low fraction of the particles? What happens if you reinitialize a very large fraction of the particles? For the same reinitialization factor, is there a benefit or downside for $N = 20$ versus $N = 100$? Motivate your answer.

Appendix

A Linear Dynamical System

A Linear Dynamical System (LDS) describes a sequence of states $\mathbf{x}_1, \mathbf{x}_2, \cdots$, where $\mathbf{x}_t$ is the state at time t, and each state is an N-dimensional vector. The transition from one state to the next can be written as a linear transformation of the previous state $\mathbf{x}_{t-1}$, and some additional Normally distributed noise $\boldsymbol{\epsilon}_t$. The state is called *latent* in the sense that it cannot be observed directly. In an LDS, a measurement is an M-dimensional vector $\mathbf{z}_t$. A measurement is related to the true latent state $\mathbf{x}_t$ through a linear transformation, but is again some Normally distributed measurement 'noise'.

The full system can therefore be written as

$$\mathbf{x}_t = F\mathbf{x}_{t-1} + \boldsymbol{\epsilon}_t \qquad \text{where} \qquad \boldsymbol{\epsilon}_t \sim \mathcal{N}(0, \Sigma_{\mathbf{x}}) \tag{10}$$

$$\mathbf{z}_t = H\mathbf{x}_t + \boldsymbol{\eta}_t \qquad \text{where} \qquad \boldsymbol{\eta}_t \sim \mathcal{N}(0, \Sigma_{\mathbf{z}}) \tag{11}$$

where F is the $N \times N$ transition matrix, and, H is the $M \times N$ observation matrix. Also, $\Sigma_{\mathbf{x}}$ is the $N \times N$ covariance matrix of the process noise, and $\Sigma_{\mathbf{z}}$ the $M \times M$ covariance matrix of the observation noise. $\mathcal{N}(\mu, \Sigma)$ indicates a Multivariate Normal (i.e. Gaussian) distribution with mean μ and covariance Σ.

Note that if all N elements $\boldsymbol{\epsilon} = [\epsilon_1, \cdots, \epsilon_N]$ are *independently* distributed, and element ϵ_i have variance σ_i^2, i.e. each $\epsilon_i \sim \mathcal{N}(0, \sigma_i^2)$, then $\Sigma_{\mathbf{x}}$ is a diagonal,

$$\Sigma_{\mathbf{x}} = \begin{bmatrix} \sigma_1^2, & 0, & \cdots, & 0 \\ 0, & \sigma_2^2, & \cdots, & 0 \\ & & & \\ 0, & 0, & \ddots, & 0 \\ 0, & 0, & \cdots, & \sigma_N^2 \end{bmatrix}. \tag{12}$$

B Kalman Filter

The Kalman Filter assumes that a process can be described as a Linear Dynamical System (see Appendix A). It maintains a posterior distribution over the process state, given all observations until the current time step. This state distribution is always represented by a Multivariate Normal distribution. Just as other Bayesian filters, it consists of the following steps:

Initialize To initiate the Kalman Filter at $t = 0$, we need to define the initial state distribution:

$$P(\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_0 \mid \boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0) \qquad \text{initial distribution} \tag{13}$$

This step can be followed with an *update* step if there are any initial observations. For subsequent time instances ($t > 0$), the initialization is replaced by a *predict* step (followed by an *update*).

Predict step The predict step uses the filtered state distribution $P(\mathbf{x}_{t-1}|\mathbf{z}_{0:t-1})$ of the previous time instance to create a predicted state distribution $P(\mathbf{x}_t|\mathbf{z}_{0:t-1})$ for the new time instance.

These Multivariate Normal distributions are parameterized as

$$P(\mathbf{x}_{t-1}|\mathbf{z}_{0:t-1}) = \mathcal{N}(\mathbf{x}_{t-1} \mid \boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1}) \quad \text{posterior from previous time step} \tag{14}$$

$$P(\mathbf{x}_t|\mathbf{z}_{0:t-1}) = \mathcal{N}(\mathbf{x}_t \mid \widehat{\boldsymbol{\mu}}_t, \widehat{\boldsymbol{\Sigma}}_t) \quad \text{prediction for new time step} \tag{15}$$

The predict step computes the new parameters as

$$\widehat{\boldsymbol{\mu}}_t = F\boldsymbol{\mu}_{t-1} \quad \text{predicted mean} \tag{16}$$

$$\widehat{\boldsymbol{\Sigma}}_t = F\boldsymbol{\Sigma}_{t-1}F^\top + \Sigma_{\mathbf{x}} \quad \text{predicted covariance} \tag{17}$$

Update step The update step incorporates evidence from measurements into the state distribution. On each time step, it can be called *zero or more times*, namely once for each available independent measurement $\mathbf{z}_t$.

For a single update with $\mathbf{z}_t$, we consider the distribution before the update the Bayesian *prior*, and the resulting distribution the *posterior*, which are parameterized as

$$P(\mathbf{x}_t|\mathbf{z}_{0:t-1}) = \mathcal{N}(\mathbf{x}_t \mid \widehat{\boldsymbol{\mu}}_t, \widehat{\boldsymbol{\Sigma}}_t) \quad \text{prior of } \mathbf{z}_t \text{ (from prediction or earlier updates)} \tag{18}$$

$$P(\mathbf{x}_t|\mathbf{z}_{0:t}) = \mathcal{N}(\mathbf{x}_t \mid \boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t) \quad \text{updated (posterior of } \mathbf{z}_t) \tag{19}$$

The parameters of the updated distribution are computed as

$$e_t = \mathbf{z}_t - H\widehat{\boldsymbol{\mu}}_t \quad \text{innovation (residual error)} \tag{20}$$

$$\mathbf{S}_t = H\widehat{\boldsymbol{\Sigma}}_t H^\top + \Sigma_{\mathbf{z}} \quad \text{innovation covariance} \tag{21}$$

$$K_t = \widehat{\boldsymbol{\Sigma}}_t H^\top \mathbf{S}_t^{-1} \quad \text{Kalman gain} \tag{22}$$

$$\boldsymbol{\mu}_t = \widehat{\boldsymbol{\mu}}_t + K_t e_t \quad \text{updated mean} \tag{23}$$

$$\boldsymbol{\Sigma}_t = (I - K_t H)\widehat{\boldsymbol{\Sigma}}_t \quad \text{updated covariance} \tag{24}$$

The updated state distribution after all measurements can then serve as the input to the predict step at the next time instance $t + 1$.

C Particle Filtering

While Kalman Filtering represents the state distribution using multivariate Gaussian distributions, a Particle Filter does not assume a specific parametric form of the distribution. Instead, the distribution is represented by a set of *samples*. The prediction and update step manipulate the set of samples, such that it remains representative for the state distribution as time progresses and more measurements are obtained. Benefits include the unconstrained form of the distribution, support of non-linear motion and observation models, and the simple algorithmic implementation. A downside is that it has inherent randomness, and the number of particles may be too limited to represent the shape of the true distribution.

Let $S_t = \{\mathbf{x}_1^1, \cdots, \mathbf{x}_t^N\}$ be the set of N particles at time t. The particle filter consists of the following steps:

Initialize Create the initial set $S_0 = \{\mathbf{x}_0^1, \cdots, \mathbf{x}_0^N\}$ with N particles by sampling N times from the initial distribution $P(\mathbf{x}_0)$. Depending on the setup, the initial distribution may be very specific (i.e. we know where the object is), or evenly spread out over the space (we have to recover where it is).

Predict The predict step creates a set $\widehat{S}_{t+1}$ from the current particle set S_t. It does this by applying a possibly non-linear dynamical model to move each particles forward, given the control input, and time difference Δt between current time step t and $t+1$. For the 2D space + orientation representation, a simple non-linear dynamical model $f(\mathbf{x}_t^{(i)}, \mathbf{u}_t, \Delta t)$ is

$$\mathbf{x}_{t+1}^{(i)} = f(\mathbf{x}_t^{(i)}, \mathbf{u}_t, \Delta t) + \epsilon_t^{(i)} \qquad \text{where } \epsilon_t^{(i)} \sim \mathcal{N}(0, \mathbf{\Sigma}_x) \tag{25}$$

which is an alternative way to formulate $P(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t) = \mathcal{N}(\mathbf{x}_{t+1}|f(\mathbf{x}_t, \mathbf{u}_t, \Delta t), \mathbf{\Sigma}_x)$. Similar to the Kalman Filter, $\epsilon_t^{(i)}$ is additional zero-mean Gaussian noise on the state. The predict step can therefore be split into two steps: first, apply the deterministic non-linear dynamics to move the particle forward using control input $\mathbf{u}_t$. Second, add some randomly sampled Gaussian noise to each particle.

Update In the update step, a new set $S_{t+1} = \{\mathbf{x}_{t+1}^{(1)}, \cdots, \mathbf{x}_{t+1}^{(N)}\}$ of N particles is constructed by sampling N times (with replacement) particles from the predicted set $\widehat{S}_{t+1}$. The probability $p^{(i)}$ that a particle i in $\widehat{S}_{t+1}$ is sampled is proportional to the *particle weight* $w^{(i)}$, i.e. $p^{(i)} = w^{(i)} / \sum_i w^{(i)}$. To determine this weight we compare the observed R-dimensional measurement vector to the expected R-dimensional measurement of each particle, i.e. the observation likelihood $P(\mathbf{z}|\mathbf{x}^{(i)})$ is used as the particle weight.

Let $h(\mathbf{x}_t^{(i)}) = \hat{\boldsymbol{z}}^{(i)} = [\hat{z}_1^{(i)}, \cdots, \hat{z}_R^{(i)}]^\top$ be the expected measurement at the i-th particle $\mathbf{x}_t^{(i)}$ in $\widehat{S}_{t+1}$. Here $h()$ is the non-linear observation model that maps the particle's state to the R-dimensional measurement vector (e.g. the expected distances along R sensor rays using an occupancy map). We place a zero-mean Gaussians with variance σ_x^2 on the R errors between the expected and observed values. In other words, the measurement likelihood is large when the error over all dimensions is small. The complete likelihood can thus be written as,

$$P(\mathbf{z}|\mathbf{x}^{(i)}) = P(\mathbf{z}|h(\mathbf{x}^{(i)})) = \prod_{r=1}^{R} P(z_r|\hat{z}_r^{(i)}) = \prod_{r=1}^{R} \mathcal{N}(z_r - \hat{z}_r^{(i)}|0, \sigma^2). \tag{26}$$

We use these likelihood as the non-zero particle weights. In other words, particles whose expected measurement is very similar to the actual measurement will have a high weight, which in return increases their chance of being present in the new resampled set S_{t+1}.

D Using log-probabilities in computations

Often in numerical computations concerning many small probabilities it is advantageous to use log-probabilities instead.

First, log probabilities are much less prone to suffer from the rounding errors that the computer makes to internally represent the numbers. In some cases, normal probabilities can get so small that they become indistinguishable after the rounding errors.

Another benefit is that we can avoid many calls to `exp()` and `log()` that occur in the pdfs of many distributions, e.g. the Gaussian, which are relatively slow to compute. Note that multiplying terms of normal probabilities results in adding terms of log-probabilities.

Special care has to be taken however when eventually converting the log-probabilities back to normal probabilities, e.g. when normalizing.