

Lab 8: Sensor Fusion for Localization

This laboratory requires the following equipment:

- C development tools (gcc, make, etc.)
- Webots 2020a installation

The laboratory duration is approximately 2 hours. Although this laboratory is not graded, we encourage you to take your own personal notes as the final exam might leverage results acquired during this laboratory session. For any questions, please contact us at sis-ta@groupes.epfl.ch.

1.1 Information

In this assignment, you will find several exercises and questions.

- The notation **Q** means that the question can be answered theoretically or with simple commands in the Linux operating system, without implementing or running any code.
- The notation **S** means that the question can be solved only by compiling and running a piece of code or an additional simulation.
- The notation **I** means that the problem has to be solved by implementing, possibly compiling, and running a piece of code.
- The notation **B** means that the question is optional (bonus) and should be answered if you have enough time at your disposal.

1.2 Getting Started (Short reminder)

To start with this lab, you will need to download the material available on Moodle. Download *lab08.zip* in your personal directory and extract the lab archive.

1.3 General remarks and documentation

In this lab, you will continue working on odometry with the e-puck robot, extending the techniques introduced in Lab 7. Here you will understand the importance of sensor fusion to obtain an accurate interpretation of the data, and how noise on these data can be considered in this process. The first part demonstrates how the e-puck can navigate using 1D dead reckoning under non-deterministic uncertainties. The second part shows different techniques to fusion odometry with feature-based localization in Matlab. The last part demonstrates how to leverage a Kalman filter in Webots to fuse odometry with feature-based localization.

2 Odometry with non-deterministic uncertainties

When dead reckoning is properly calibrated, the e-puck can track its initial position really well as you saw in Lab 07. Here we shall now add non-deterministic sensor noise as is present on real robots.

1. (Q): Open the world *featurenavigation.wbt*, compile the controllers and run the simulation. Try to understand the code and explain in your own words what happens. Try to come up with possible ways to improve the behavior under these conditions (do not implement them). What are their downsides?
2. (I): One possibility to correct the accumulating errors is feature-based navigation. We now want to use the front wall as a feature each time we are next to it. We will do this by developing a function that measures the distance to the wall (i.e., the feature) and adjust the robot's position such that it is the same every time the robot finishes going back and forth. Open the file *e-puck_feature.c* and look for the already defined function *feature_update*. Fill the contents of function *feature_update*, so that it does the described behavior.
Hint: The front wall is situated at 1 m from the initial position.

3 Sensor fusion with Kalman Filter in Matlab

The feature-based correction of the 1D odometry implemented in Part 2 is a very basic sensor fusion. Indeed, we used the information of both the wheel encoders as well as a distance sensor, effectively resulting in a fusion of the information. This implementation, however, is a relatively crude one, it assumes for example that the measurement of the back wall is free of non-deterministic noise which is obviously not possible. In this section we shall now gradually approach a more general approach to sensor fusion leveraging a well-known estimation technique called Kalman filter. For this, open the file *Kalman.m* in Matlab.

3. (Q) Try to understand what this script does.
4. (Q) Using linear propagation of covariance (c.f. slides week 9), calculate the noise on the position estimate (*est_odo_x*) after 5 timesteps (position noise at T=1 is equal to 0). What is the noise after 200 steps?
Hint: if $Q = a + b$ then $\delta Q = \sqrt{\delta a^2 + \delta b^2}$
5. (S) What happens if you change the noise level *localization_noise*? Is this factor taken into account in the position estimation?
6. (I) Since for most real applications the noise level is known, it is highly suboptimal to simply ignore this information. Applying the central limit theorem, a very simple option to take into account the noise levels of two variables when combining them has been proposed by Fraser and Potter in 1969 [1]:

$$x_{fused} = \frac{(\sigma_1^{-2}x_1 + \sigma_2^{-2}x_2)}{\sigma_1^{-2} + \sigma_2^{-2}}$$

For *est_fp_x*, replace the simple resetting of the position with this formula using the noise level calculated in Question 4 for the odometry estimate (200 steps correspond to the interval between features). Use $\sigma = \text{localization_noise}$ for the measurement noise.

Run the script and compare the resulting trajectories. What do you observe?

7. (S,4) What happens if the measurement noise is significantly higher than the odometry noise and vice-versa? What is the issue of this sensor fusion algorithm?
8. (Q) Another method to fuse measurements is the Kalman filter. Take a look at the formula (seen in class) in Figure 1. What are μ , Σ , u and z ? What is the purpose of K ?

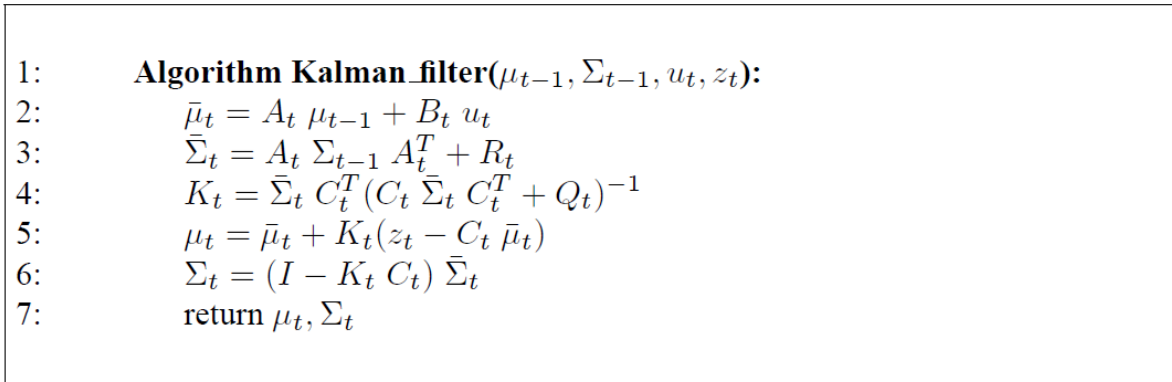


Figure 1: Kalman filter. Note that steps 2 and 3 (called prediction) can be done independent from steps 4 to 6 (update step).

9. (Q) Try to identify what each line (step) of the algorithm does. Which steps of the Kalman filter did we already implement previously in this lab (maybe in a slightly different way)?

Implement the different steps of the Kalman filter:

10. (I) Create the new state variable $\mu = [x]$. And implement the prediction step for μ such that the position value of μ increases by the odometry measure every timestep (i.e. every time the prediction is called). Run your code and verify (through printing) that μ increases as expected. *Hint: Since for now we don't have a robot model A we can set A to 1, without external input our estimation remains as it is. The odometry measure is added through u .*
11. (I) Create the noise state variable $\Sigma = [0]$. By initializing Σ in this way, what are we assuming?
12. (I) Implement the noise prediction update step. What is the purpose of R ? (Hint: line 3 of Figure 1).
13. (I) Implement now the correction step. This step is called every time a new measure is acquired to correct or update the prediction. Set the measurement noise covariance matrix (C) to *localization_noise*². Uncomment the plotting function to plot μ , run your code and compare the result with what you obtained before.
14. (I) We shall now consider the case where the robot speed v is considered as part of the state. Start by commenting out your previously implemented Kalman filter. Then create the new state variable $\mu = [x; v]$ with the noise state variable $\Sigma = [0, 0; 0, 0]$. Define A and B such that the position value of μ increases by *mov* every timestep. Adapt R for your code to execute and run your code and verify that μ and Σ increase as expected. *Hint: Be careful about the dimensions of your matrices.*

15. (I) Copy your previously implemented correction step and adapt it to the new two-variable state. Run your code and observe the result. Is there a major difference to the previous Kalman filter result? (Uncomment your previous implementation if you are not sure)
Hint: the correction happens only on x , leaving the speed part of μ untouched. This means that both z and $C\mu$ are 1 dimensional, whereas K needs to be in such a way that $K*[1 \text{ dim}]$ results in a vector of the same dimension as μ .*

4 Sensor fusion with Kalman Filter in C

In this exercise you will now use the Kalman Filter in C. The goal is to improve the odometry of a robot roaming an empty arena. For this open the Webots world *kalman.wbt*.

16. (Q) Open the controller code (both *e-puck_kalman.c* and *kalman.c*) and try to understand what it does. What does the supervisor (*supervisor.c*) do?
Hint: the robot controller code does not compile yet.
17. (I) Our state vector μ (mu) has now 6 dimensions: x , y , theta and their respective speeds. What dimensionality is needed for the matrix A ? How should A look like? On lines 20 and 22 in *kalman.c* fill in both matrices A as well as A_T which corresponds to the transposed matrix A .
18. (S) Now compile and run the simulation. After 4 minutes of simulated time the controllers terminate automatically. Now open the Matlab script *compare_trajectories.m* and run it. How well does the trajectories match compared to the one recorded by the supervisor?
19. (I) The function *feature_update* in *e-puck_kalman* is missing most of its functionality. Try to implement a feature-based update using the Kalman filter to increase the accuracy in the pose estimation. Start by implementing the detection of the wall, then you will need to estimate which wall was detected, which will allow you to then calculate the robot's current position in either x or y direction (depending on which wall is detected) using the wall's position ($\pm 0.5\text{m}$) and the distance sensor measurement. Print your estimation and run your code to check if your position estimation works.
*Hint: use *IR_to_distance* from *util.h**
20. (Q) This information has now to be added to the pose estimation through the *kal_update* function. However, *kal_update* takes 4 input values: x , y , s_x and s_y , whereas s_* corresponds to the standard deviation of the noise of a given measure. Assuming that on the e-puck we have measurement noise of $\sigma=100\%$ of the measured distance, what would be appropriate values for s_* ? What should be the value for the dimension not covered through a specific update?
21. (I) Implement the theoretical results from question 20 in the place marked in the function *feature_update*. **Change the name of the logfile in the *main* function to “kalman.txt”.** Compile and run your code. If you are not sure if your code works, use the *print_pose* function (c.f. *util.h*) to print your pose before and after the update step to check.
22. (S) Now run the Matlab script *compare_trajectories.m* again. Did the estimation improve? What happens if you change the values of s_* ? What happens if you change the R matrix in *kal_predict*? What can we conclude from this on the difficulty in implementing a Kalman filter?
23. (B) As you could see before, the trajectory depends highly on the precise estimation of the bearing angle. Try to adapt the implemented Kalman filter to accept also bearing values and try to estimate the bearing of the robot based on the sensor which detects the wall.
*Hint: For this you will need to modify most matrices in the *kal_update* function.*

5 References

- [1] D. Fraser and J. Potter, "The optimum linear smoother as a combination of two optimum linear filters," in *IEEE Transactions on Automatic Control*, vol. 14, no. 4, pp. 387-390, August 1969, doi: 10.1109/TAC.1969.1099196., url: <https://ieeexplore.ieee.org/document/1099196>