Homework 2

This homework requires the following software:

- C development tool (gcc, make etc.)
- Matlab, with the "Statistics and Machine Learning Toolbox" installed
- Webots R2021a [1], [2]

Information

In the following text you will find several exercises and questions.

- The notation **S** means that the question can be solved using only additional simulation.
- The notation **Q** means that the question can be answered theoretically, without any simulation.
- The notation **I** means that the problem has to be solved by implementing a piece of code and performing a simulation.
- The notation **B** means that the question is optional and should be answered if you have enough time at your disposal.

Outline

This homework includes questions about signal processing topics and embedded systems that have been covered in the course up to date. In Part 1, you are going to develop a control strategy for an e-puck robot to exit a maze through the help of road signs. In Part 2, you will implement and improve the odometry of an e-puck robot.

Collaboration policy

This homework aims also to foster the collaboration capabilities of students to work as a team. Teams have been defined by considering student preferences and eventually approved centrally by the Head TA of the course. Therefore, by default, the submitted deliverables will be considered a product of the team and students belonging to the same team will receive the same grade.

While intra-team collaboration is promoted, any inter-team collaboration is not allowed and, in case of evidence of copies between teams, there will be severe penalties for all parties involved.

In case of issues within a group in terms of contribution fairness or unenrollment of a group member from the course, please contact the Head TA as soon as possible in order to find a solution (e.g., splitting the team or grading separately the individual contributions).

In order to cope with odd student numbers and the potential unpredictable collaboration issues mentioned above, we are forced to consider team sizes different from two while maintaining the same assignment for all teams. In case of teams consisting of three students, we will apply a malus on the graded team performance, while for single students we will apply a bonus on the graded team performance. For these exceptional cases, the bonus and malus values will be communicated directly by the Head TA in timely fashion.

Getting Started

The questions in this homework can be solved more easily if you review the corresponding labs (7 to 9 mainly). Before starting please take a glance at the assignments, solutions and provided code for those labs.

To start with this homework, you will need to download the material available on Moodle. Download material_hw2.tar.gz or material_hw2.zip and extract it in your home directory (you can type tar xvfz material_hw2.tar.gz). Now start Matlab and change your "Current directory" to be material_hw2/part01/matlab. For each part, there will be templates for you to write your code.

Submitting your answers

You need to submit a report and all your implemented code for this homework via Moodle. The report including results, explanations and plots should be in .pdf format and named as SIS_21-22_Homework-2_ Group- GroupNo.pdf. The code submission should include the folders for part01 and part02 zipped into the file named SIS_21-22_Homework-2_Group-GroupNo.zip. Please use the templates scripts and functions given inside material_hw2.zip to implement your code, do not use any separate .m file or function. Include all codes you have written (i.e. plotting, configuring etc.). Due date for the homework will be announced via Moodle.

e-puck Robot

The *e-puck* is a fully open source, miniature mobile robot designed at EPFL for teaching purposes. The robot in reality is driven by differential wheel stepper motors while in simulation by motors combined with encoders. For the scope of this project, the supported sensors are wheel encoders, eight infra-red sensors for proximity measurement, and a built-in camera. The detailed information about the simulated e-puck can be found here [3]. For clarification, the main characteristics are given in Table 1.

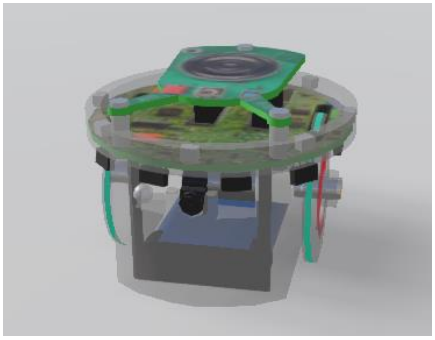

Figure 2. Simulated e-puck

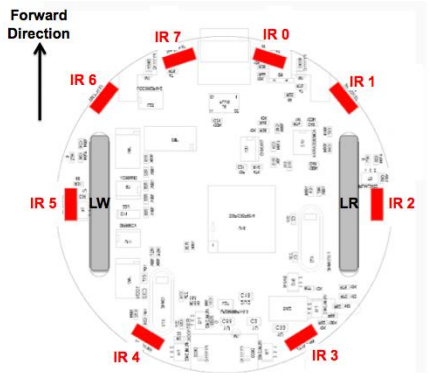

Figure 1. Sensor placements

Characteristics	**Values**
Diameter	71 mm
Height	50 mm
Wheel radius	20.5 mm
Axle Length (2 x l)	52 mm
Weight	0.16 kg
Max. forward/backward speed	0.25 m/s
Max. rotation speed	6.28 rad/s

Table 1. E-puck characteristics

On the simulated e-puck robot, every sensor and actuator is represented by a *device* with the same characteristics as described in Table 1. In Webots, you can access these devices by their names. These names are listed on Table 2.

Device	**Name**
Motors	‘left wheel motor’ & ‘right wheel motor’
Position sensors	‘left wheel sensor’ & ‘right wheel sensor’
Proximity sensors	‘ps0’ to ‘ps7’
Camera	‘camera

Table 2. Device names

Part I: Road Sign Maze – 60 pts

A screenshot taken from the environment used in this part is displayed in Figure 3. At the end of Part I, your e-puck will be able to autonomously navigate from the start position to the finish position, which you will demonstrate during the demo of the last week of the semester.

The e-puck's initial position in x and z axes (world coordinate frame) is given as [-0.875, -0.25] m, the initial orientation with respect to y axis is π rad. Following road signs are present in the world:

Turn right – horizontal stripes

Turn left – vertical stripes

Turn back – black wall

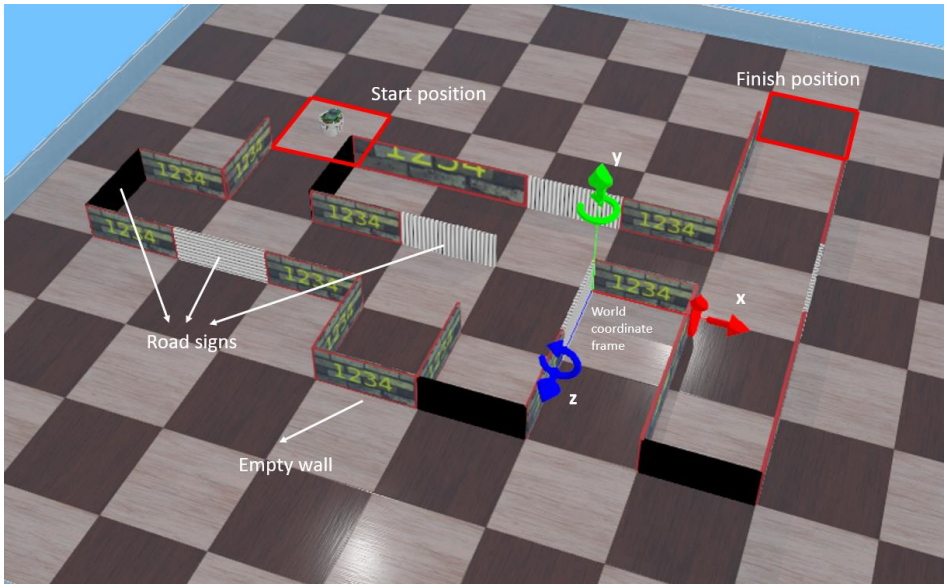

Figure 3. Environment for road sign recognition project

To start, open the Webots world *part1/webots/worlds/sandbox.wbt*. This world includes the keyboard controller you saw in the labs to allow you to manually control the robot.

1. Getting started – (S, 2 pts)

Investigate at what distance the road sign is detectable with the proximity sensors by driving the e-puck with the keyboard (same shortcuts as in Lab 7 with the additional control 'P' to take and save an image) take a couple of pictures at this distance and save them to .jpg files. Report one picture as well as the distance (robot center to wall) at which it was taken in your report.

Hint: the picture is saved in the webots/controller/sandbox directory and overwritten every time you take a new picture.

Hint: the coordinates of the robot center as well as the wall to calculate the distance can be found in the 'translation' field of the scene tree in Webots.

2. Matlab image process – 11 pts

Open the file *part1/matlab/analyze_image_with_FFT.m* in matlab. Our goal is to "read" the road signs using fft analysis.

2.1. (I, 2pt) Load the given (example) image *example_image.jpg* using `imread`. Convert the image into gray-scale using `rgb2gray` and display it using `imshow`. Report the resulting image here.

2.2. Develop an algorithm to detect the stripe direction and black image.
Hint: think about columns and rows instead of individual pixels.
Hint: also change lines 18 & 19 according to your strategy
Hint: if needed, try increasing the distance at which the images are captured.

2.2.1. (I, 3 pts) Develop an algorithm to decide if a picture contains horizontal stripes. Implement it in the marked part. Include the resulting plot for a horizontal picture, as well as the algorithm (paste the relevant matlab code) in your report.

2.2.2. (I, 3 pts) Develop an algorithm to decide if a picture contains vertical stripes. Implement it in the marked part. Include the resulting plot for a vertical picture, as well as the algorithm (paste the relevant matlab code) in your report.

2.2.3. (I, 3 pts) Develop an algorithm to decide if a picture contains black images. Implement it in the marked part. Include the resulting plot for a black picture, as well as the algorithm (paste the relevant matlab code) in your report.

3. Behavior-based controller strategy – 14 pts

Open the Webots world *part1/webots/worlds/maze.wbt*. This world contains the same environment as before. However, the robot is using a different controller.
Design a behavior-based control strategy to move the e-puck in the environment and implement this strategy in a Webots controller.

3.1. (Q, 6 pts) Fill in your behavior-based controller strategy in the file *BB_form.pdf* including both behaviors and transitions. Be as concise as possible.
Available basic behaviors are:
- Go forward
- Go backwards
- Turn left
- Turn right
- Turn around
- Take a picture
- Analyze picture

Hint: If you are missing states in the diagram, consider grouping basic behaviors together (e.g., group all turns, or group actions always taken in the same sequence without any conditions).
Hint: If a transition is always happening without condition, add 'true' as condition. If a transition never happens, add 'false'.

3.2. (I, 8 pts) Implement your behavior-based controller in the file *part1/webots/controllers/e-puck_controller/e-puck_controller.c*, lines 260 onwards and add your switch-case code to the report.

4. Basic behaviors – 9 pts

Design the basic behaviors in the e-puck controller file *part1/webots/controllers/e-puck_controller/e-puck_controller.c*
Hint: the skeleton functions exist already; you simply need to fill them in.

4.1. (I, 4 pts) Implement a controller to *go forward* in a straight line (staying in the middle of the corridor).
4.2. (Q, 1 pts) Describe your strategy to turn precisely 90°.
4.3. (I, 2 pts) Implement your strategy for both *turn left* and *turn right* functions
4.4. (I, 2 pts) Implement the *go backwards* function.

5. Image processing algorithm in C – 12 pts

Based on your analysis in MATLAB (Question 2), develop your image processing algorithm in C for the Webots controller.

5.1. (I, 10 pts) According to the logic of the algorithm developed in Matlab in question 3, modify the `analyzePictureBehavior` function to do the image processing in C. Add the code of the function to your report.
Hint: try to understand how the kiss_fft functions work based on the example provided in the code.
Hint: The comments in the function are supposed to help you, but if your algorithm doesn't need a step simply ignore them.
5.2. (Q, 2 pts) Summarize the changes necessary for the algorithm to work in C. Assume we want now to run this algorithm on real e-pucks. Do you expect any issues?

6. Algorithm validation – 12 pts

6.1. (S, 2 pts) Plot the trajectory of your robot exiting the maze in Matlab, using the ground truth recorded by the supervisor (and stored in */webots/controllers/supervisor/supervisor_logfile.txt*).

6.2. (S, 10 pts) Show off your controller at the DEMO! – Mandatory to validate all points achieved in Part 1. A partial demonstration of the tasks of this section will result in a partial allocation of the points of this question.

Part II: State Estimation and Sensor Fusion – 60 pts

Note this part is independent from Part I.

Here, you will implement a localization algorithm that relies on odometry and sensor fusion. You will implement an odometry-based localization algorithm, estimate the variance of the various measurement sources and fuse the data into a final state estimate using a Kalman filter. At the end of Part II, your e-puck will be able to localize accurately in the arena, which you will demonstrate during the demo of the last week of the semester.

Open the Webots world *part2/worlds/state_estimation.wbt*. The world is as follows, where an e-puck is located at the center of the world:

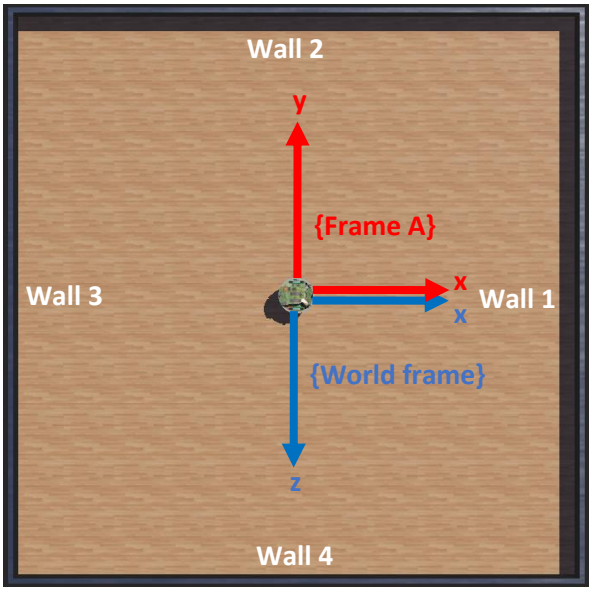

Figure 4. Webots world with frame definitions. The arena is 1m wide.

Everything you will do here has to be computed in the inertial frame A. This frame is aligned with the default World frame, but the axis definition is different, as illustrated in the figure above. It is therefore important that the robot always starts from this point (frame A's origin, with heading angle equal to zero, like in the figure above)!

The e-puck can use following sensors:

- Its wheel encoders
- A two-axis compass [10]
- Its distance sensors

The robot's controller is composed of the following source files (see *part2/controllers/controller*):

- *controller.c*: main controller file
- *odometry.c*: implement your odometry solution here
- *kalman.c*: implement your Kalman filter here
- *matrix.c*: contains methods to perform matrix operations (all provided)

7. Odometry – 9 pts

The provided controller is already programmed to let the robot perform a random walk in the arena and avoid obstacles. You will only take care of estimating its trajectory. **All computations must be performed in the Frame A**.

7.1. (I, 2 pts) Enable all relevant sensors and actuators in *controller.c*. To do so, complete the following functions:

- `controller_init_distance_sensors();`
- `controller_init_compass();`
- `controller_init_encoder();`

Hint: If this is done properly, the robot should drive randomly in the arena and avoid the walls.

7.2. (I, 4 pts) Implement an odometry-based localization algorithm in *odometry.c* in the function `odo_compute_encoders()`. We assume that the robot always starts at the origin of the inertial frame A (center of the arena). Note that the pose consists of three elements: x position, y position and θ orientation of the vehicle. Recompile the code for your changes to be applied.
Hint: Use wheel sensors to obtain the robot pose in the robot frame and convert it to the inertial frame A.
Hint: You can take inspiration from the odometry you implemented in Lab 8 here.

We will use the following metric to evaluate the precision of the pose estimate:

$$LM(n) = \frac{1}{N_f - N_i} \sum_{n=N_i}^{N_f} \|x[n] - \hat{x}[n]\|$$

where x is the ground truth, $\hat{x}$ is the estimated pose, N_i is the starting index and N_f is the final index. It has to be computed for each position x and y of the pose.

7.3. (S, 1 pt) Run the simulation to let the robot roam through the arena. This will generate a log file with the odometry data and ground truth. Open the file *plot_main.m* and run **Part A** to plot the trajectory. Report the generated plot.
Hint: You can run the simulation in fast mode to gather more data quickly.
*Hint: You should let the simulation run for **2-3 minutes** in simulated time to let the robot move enough.*

7.4. (I, 2 pts) Implement the metric in the files *metric_scalar.m* and *metric.m*. Follow the instructions in these functions. In the file *plot_main.m*, run **Part B** to plot the performance of the odometry. Report the plot and the achieved metric for the x and y state variables.

8. Covariance estimation – 6 pts

We would like to characterize the noise of our sensors to know how much we can rely on them to locate the robot. This will later allow us to fuse the information accordingly from several sensors to generate a more accurate state estimate. We will use following sensors or means of estimation:

- Compass
- Distance sensors
- Odometry

8.1. (S, 1 pt) We will study the compass first. Let the robot roam freely for a few seconds to collect some data. Note that the compass runs at 1 Hz. In Matlab, load the log file and plot the sensor data of the compass by running **Part C** of *plot_main.m*. Report the plot.

Hint: Make sure to collect enough samples, e.g.at least 30.

8.2. (I, 1 pt) Compute the standard deviation of compass heading error by completing the function in *compass_variance.m.* Run **Part D** of *plot_main.m* and report the obtained plot and standard deviation.
Note: since we can use the ground truth for the heading, we can let the robot move around and compare the measurements to the ground truth directly. This method is different to what would be typically done with a real compass. Indeed, a real compass would remain static at a known orientation while collecting data. However, since a ground truth is available here, we can easily gather data for any orientation and hence can let the robot roam freely while acquiring measurements.

8.3. (I, 2pts) We now want to characterize the distance sensors. To do so, place the robot in front of a wall so that one of its sensors is perpendicular to the wall and is able to detect it. In *controller.c*, set `MOVE` to `false` and recompile the controller. You can now run the simulation for a few seconds to collect data (the robot should not move!). Complete the function *distance_sensor_variance.m* and run **Part E** of *plot_main.m* to compute the standard deviation. Report the obtained plot and standard deviation.

We still have to characterize the odometry. This is not as straight forward as for the previous points. Indeed, we cannot let the robot sit still to collect data but need instead the robot to move around. Therefore, we need to know the ground truth, although this was optional for the compass.

8.4. (I, 2 pts) Set the `MOVE` value back to `true` in *controller.c* and recompile the controller to let the robot move around again. Let the simulation run for a couple minutes to collect some odometry data. To compute the standard deviation of the odometry's output, you will need to compare the odometry data to the ground truth for each pose variable (x, y and heading) and compute the standard deviation of the individual errors. To do so, complete the function *odometry_variance.m* that computes the variance for one variable at a time (it is hence called once for each pose variable from *plot_main.m*). Then run **Part F** to compute the standard deviations. Report the obtained plots and standard deviations.

9. Sensor fusion – 45 pts

We now want to compute a more reliable pose estimate that considers all measurements we have available. We will design a Kalman filter to this end. We recall below the structure of the filter. Note that steps 2 and 3 can be performed independently from steps 4 to 6.

1: **Algorithm Kalman_filter**($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$)**:**
2: $\bar{\mu}_t = A_t\ \mu_{t-1} + B_t\ u_t$
3: $\bar{\Sigma}_t = A_t\ \Sigma_{t-1}\ A_t^T + R_t$
4: $K_t = \bar{\Sigma}_t\ C_t^T (C_t\ \bar{\Sigma}_t\ C_t^T + Q_t)^{-1}$
5: $\mu_t = \bar{\mu}_t + K_t(z_t - C_t\ \bar{\mu}_t)$
6: $\Sigma_t = (I - K_t\ C_t)\ \bar{\Sigma}_t$
7: return μ_t, Σ_t

In our case, we will use the odometry to compute the prediction (steps 2,3), since it provides an estimate at each time step. Then, we will use both the compass and distance sensors to update the estimate (steps 4 to 6). The compass will let us update the estimated heading and the distance sensors will allow the robot to detect the walls. Since we assume the wall locations are known, we can update the estimated position of the robot when a wall is encountered.

It is important to notice that neither the compass, nor the distance sensors allow to measure a full pose and therefore only provide partial information. In other words, the update steps will only allow us to partially update the pose. Furthermore, the compass is slower than the other means of measurement, as it only produces a measurement each second, while the distance sensors only provide position information when a wall is encountered and therefore have no regular measurement frequency.

From now on, use the following estimated standard deviations for the prediction and update steps of the Kalman filter:

- Prediction:
  - Odometry:
    - position standard deviation: 0.06 [m]
    - heading standard deviation: 0.07 [rad]
- Update:
  - Compass:
    - heading standard deviation: 0.03 [rad]
  - Distance sensors:
    - position standard deviation: 0.001 [m]

We will use the following motion model, used in lab 8, with the velocities in the body frame expressed as:

$$v_B = \frac{(\Delta_{enc_r} + \Delta_{enc_l}) \cdot WHEEL\_RADIUS}{2 \cdot T}$$
$$\omega_B = \frac{(\Delta_{enc_r} - \Delta_{enc_l}) \cdot WHEEL\_RADIUS}{WHEEL\_AXIS \cdot T}$$

Hence the speeds in the inertial frame A are:

$$\begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix} = R_B^A \begin{bmatrix} v_B \\ 0 \\ \omega_B \end{bmatrix}$$

with

$$R_B^A = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Also, the model should be discretized using the Euler forward method:

$$v_x(t) = \dot{x}(t) = \frac{x_{t+1} - x_t}{T}$$

Let us define also the state μ_t and input u_t:

$$\mu_t = [x_t \; y_t \; \theta_t]^T$$
$$u_t = [v_B \; \omega_B]^T$$

9.1. (Q, 2 pts) Using the model described above, define all quantities involved in the prediction step at $t = 0$ (A_t, B_t, Σ_t, R_t).
Hint: If you need to define a covariance matrix, assume it is a diagonal matrix.
Hint: Assume we have perfect knowledge of the initial pose of the robot at t=0.

9.2. (Q, 1 pts) Define the quantities for the update step using the compass (z_t^c, C_t^c, Q_t^c).

9.3. (Q, 2 pts) Define the quantities, as in the previous question, for the update step based on the distance sensors ($z_t^x, C_t^x, Q_t^x, z_t^y, C_t^y, Q_t^y$). No need to provide any mathematical expression for the measurements z_t^x and z_t^y. Just explain in a few words what they represent.
Hint: You need to distinguish an update in x from an update in y, as these represent two distinct measurements.
Hint: Assume we measure the position of the robot directly. For instance, in the case where the wall 1 or 3 (see Figure 4) is detected by the distance sensors, we can deduce the position along the x-axis of the robot based on this detection (recall the arena has known dimensions). The same principle applies to the y-axis. You will implement this measurement method in question 9.7.

To implement the Kalman filter, we provide you with methods allowing to perform matrix operations declared in *matrix.h* and defined in *matrix.c*. A matrix is defined as a standard two-dimensional array, with its first index iterating over the rows and the second over the columns. For instance, to access the first row and second column of a matrix, you would call `m[0][1]`, with `m` declared as `double m[2][3];` for a 2 by 3 matrix. Please refer to the code directly for more information about the available methods and how to use them.

9.4. (I, 2 pts) In the file *kalman.c*, implement the function `kal_compute_input_u()` so that it converts the encoders increments into the input u_t and stores it in the variable `u`.

9.5. (I, 4 pts) Implement the prediction step of the Kalman filter in the file *kalman.c*. To do so, you need to declare the matrices A_t and Σ_t at lines 20 and 23, and to complete the function `kal_predict()`. Do not forget to compile your controller once you are done.

9.6. (S, 2 pts) Run the simulation and plot the estimated trajectory by running **Part H** of *main_plot.m*. Make sure at this point that the trajectory generated by the Kalman filter matches your previous odometry solution since they should produce equal or at least equivalent results. If you see a noticeable difference, verify your implementation. Report the obtained trajectory plot.

Before we can proceed with the implementation of the update step of the filter, we need to define how the robot can re-localize based on its distance measurements. We know that the arena is exactly 1 m wide and that the origin is at the center of the arena. This means each wall is placed at either +/- 0.5m from the center along the x- or y-axis respectively (c.f. Figure 4 for an illustration).

9.7. (I, 6 pts) Complete the function `compute_position_from_wall_detection()` in *controller.c* to implement a rule that lets the robot identify which wall is most likely seen when a distance sensor is triggered, and produce a localization along the axis that corresponds to the detected wall.
Hint: To make things simpler, you can assume only one wall is seen at a time. No need to try to handle detections in corners for instance. Also, only using the current estimated position in x

and y of the robot to identify which wall is most likely detected can be sufficient. You may not need to consider the current heading or which sensor was triggered.
Hint: You may want to consider the robot's body radius (assume it is roughly 0.03 m).

9.8. (I, 2 pts) Verify that your wall localization solution works as intended by giving `_ground_truth` as argument to the `compute_position_from_wall_detection()` function called in `main()` in *controller.c*. Compile your code, run the simulation and plot the wall detections by running **Part G** of *plot_main.m*. Report the generated plot. If something seems off, go back to your detection solution and verify your steps.
Note: Once you are satisfied with your solution, ***set the argument back to `_kalman` when calling `compute_position_from_wall_detection()`***.

We now have everything required to implement the update step of the Kalman filter.

9.9. (I, 6 pts) In *kalman.c*, define the C and Q matrices in each of the functions according to your answers to questions 9.2 and 9.3:
- `kal_update_x()`
- `kal_update_y()`
- `kal_update_heading()`

9.10. (I, 7 pts) In *kalman.c*, implement the function `kal_update()` that computes the update step.

9.11. (S, 1 pt) Run the Webots simulation and plot the estimated trajectories, and the localization metric in Matlab by running **Parts H** and **I** of *plot_main.m*. Compare the performance of the Kalman filter and the odometry. Report the obtained plots.

9.12. (S, 10 pt) Show off your Kalman filter at the DEMO by going through the same steps as in question 9.11! – Mandatory to validate all points achieved in Part 2. A partial demonstration of the tasks of this section will result in a partial allocation of the points of this question.

Resources

1. [1] Webots user guide, https://www.cyberbotics.com/doc/guide/index
2. [2] Webots reference manual, https://www.cyberbotics.com/doc/reference/index
3. [3] GCTronic' e-puck, https://cyberbotics.com/doc/guide/epuck
4. [4] Lecture notes (Week 8)
5. [5] Lecture notes and lab material (Week 9)
6. [6] Lecture notes and lab material (Week 10)
7. [7] Webots step function, https://cyberbotics.com/doc/guide/controller-programming#the-step-and-wb_robot_step-functions
8. [8] Webots motor, https://cyberbotics.com/doc/reference/motor
9. [9] Webots position sensor https://cyberbotics.com/doc/reference/positionsensor
10. [10] Webots compass https://cyberbotics.com/doc/reference/compass