Homework

This homework requires the following software:

- C development tool (gcc, make etc.)
- Matlab, with the "Statistics and Machine Learning Toolbox" installed
- Webots R2022a [1], [2]
- Arduino IDE and DISAL Arduino kit

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.

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 localization of an e-puck robot. In Part 3, you will develop a fitness tracker using Arduino.

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**. Additionally, we will **use automatic tools to detect plagiarism for all deliverables (code and report) compared to what was submitted in previous editions of the course**. In case we detect any issue, there will be severe consequences, according to EPFL's plagiarism policy.

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 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 three while maintaining the same assignment for all teams. In case of teams consisting of two students (the only acceptable alternative for us), we will apply a small bonus on the graded team performance. For these exceptional cases, the bonus values will be communicated directly by the Head TA in a timely fashion.

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_22-23_Homework_ Group- GroupNo.pdf. The code submission should include the folders for part1, part2 and part3 zipped into the file named SIS_22-23_Homework_Group-GroupNo.zip. Please use the templates scripts and functions given inside material_hw.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.

Getting Started

The questions in this homework can be solved more easily if you review the corresponding labs (5 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_hw.tar.gz or material_hw.zip and extract it in your home directory (you can type tar xvfz material_hw.tar.gz). Now start Matlab and change your "Current directory" to be material_hw/part01/matlab. For each part, there will be templates for you to write your code.

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.

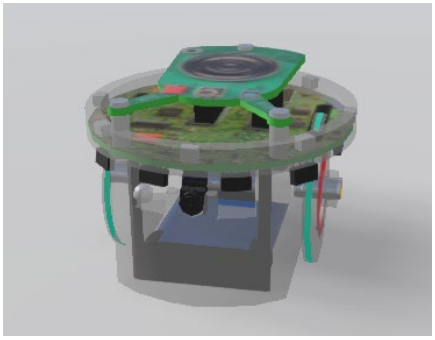

Figure 1. Simulated e-puck

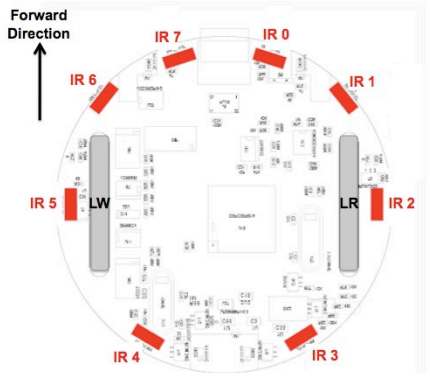

Figure 2. Sensor placements

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 – 40 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

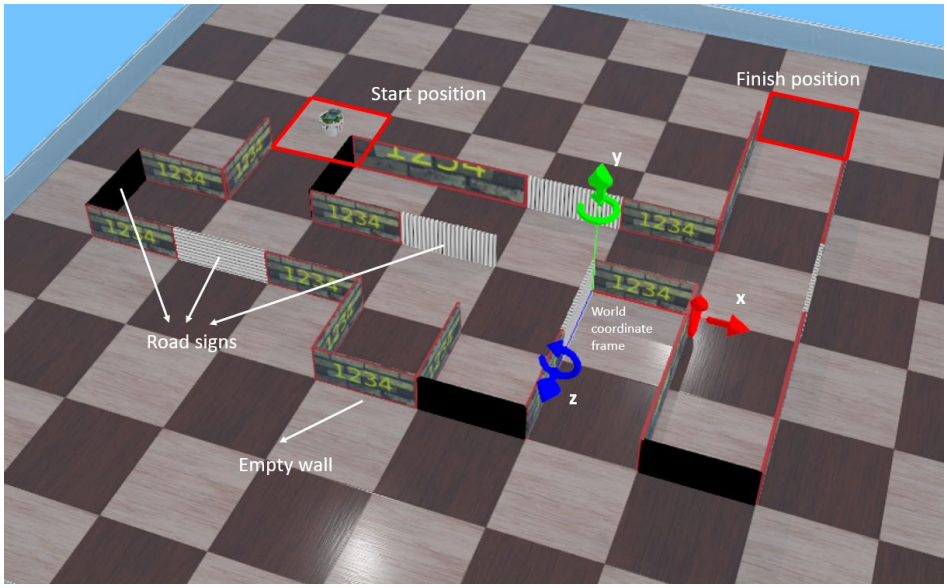

Figure 1. Environment for road sign recognition project

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.

Note: we will grade your answers based on correctness, completeness of implementation, critical reasoning, and clarity of presentation. We will take away points if it is not possible for us to understand what you implemented.

1. Getting started – (2 pts)

1.1 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.
Hint: if you right click on the e-puck in the scene tree and select "Show Robot Window", you can enable the distance sensors and visualize their values.

2. Matlab image processing – 7 pts

Open the file *part1/matlab/analyze_image_with_FFT.m* in matlab. Our goal is to "read" the road signs using FFT analysis. **Note: you must use FFT for your solution.**

2.1. I (1pt) 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 in your report.

2.2. I (6 pts) Develop an algorithm to decide if a picture contains horizontal or vertical stripes, or black images. Implement it in the marked part. Include the resulting plot for a horizontal, vertical, and black picture, as well as the algorithm (paste all the Matlab code you modified to implement this) in your report. Briefly explain your strategy in the report.
Hint: think about columns and rows instead of individual pixels.
Hint: also change lines 21&22 according to your strategy.
Hint: if needed, try increasing the distance at which the images are captured.

3. Behavior-based controller strategy – 5 pts

Open the Webots world *part1/webots/worlds/e-puck_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 (2 pts) Fill in your behavior-based controller strategy in the file *BB_form.pdf* including both behaviors and transitions.
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 (3 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 – 7 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 - goForwardBehavior(), goBackwardsBehavior(), turnRightBehavior(), turnLeftBehavior()****; you simply need to fill them in.*
NB: If you need to use the data coming from the e-puck's IR sensors, ***you must use the det_sensor_input() function****.*

4.1. I (2 pts) Implement a controller to *go forward* in a straight line (staying in the middle of the corridor). Briefly explain your strategy in the report.

4.2. Q (1 pts) Describe your strategy to turn precisely 90° in the report.

4.3. I (2 pts) Implement your strategy for both *turn left* and *turn right* functions. Briefly explain your strategy in the report

4.4. I (2 pts) Implement the *go backwards* function. Briefly explain your strategy in the report

5. Image processing algorithm in C – 8 pts

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

5.1. I (6 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 and briefly explain your strategy.
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 – 11 pts

6.1. S (1 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*). Show your plot in the report.

6.2. S (10 pts) Showcase your work on road sign recognition and answer questions at the final demo. In the final demo, you will have to show that your code works and convince us of your solution. You should be able to comment on the robustness of your algorithm and on its limitations.

Part II: State Estimation and Sensor Fusion – 50 pts

7. Odometry (12 points)

Open the Webots world *part2/worlds/arena.wbt*.

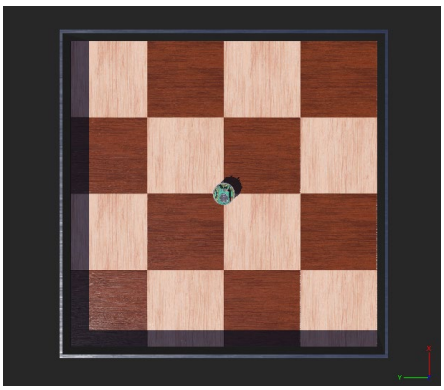

Figure 2: arena.wbt

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.cpp*: implement your Kalman filter here

You will use the following metric to evaluate the performance of the state estimation, which computes the average distance to the ground truth:

$$M = \frac{1}{N}\sum_{n=1}^{N} ||x[n| - \hat{x}[n]||$$

Where x represent the measured quantity (can be anything, scalar or vector), and n is the number of data points. Now, open Matlab and navigate to the correct folder path.

7.1 I (1 pts): Implement the metric computation in the file *compute_metric.m*. To do so, complete the two functions `metric_scalar` and `metric` at the end of *compute_metric.m* and verify your implementation by running **Part D** of *compute_metric.m*. Report your code.

We will refer to the odometry relying on both the accelerometer and the gyroscope as "*IMU based odometry*" (IMU stands for Inertial Measurement Unit). It is indeed common to find an accelerometer and a gyroscope packaged into a single device, commonly referred to as IMU.

The robot will estimate its pose as follows:

- 2D odometry (measuring the robot x,y-axis acceleration) with the accelerometer, using the current heading estimate to project the measured accelerations from robot frame to world frame.

- Heading odometry (measuring the robot heading angular velocity) with the gyroscope.

Set the flag `nav = STRAIGHT_LINE` (controller.c at line 25). This will let the robot drive straight for a short distance and then stop. *Note: the robot will start after 5s due to the calibration of the accelerometers.*

7.2 I (2 pts): Complete the two functions `controller_get_accelerometer()` and `controller_get_gyroscope()` to read the data from both the accelerometer and the gyroscope. You can verify that the measurements are occurring as intended by setting the flags `VERBOSE_ACC` and/or `VERBOSE_GYRO` to true to print the data into the console when the simulation is running. Set them back to false once you are done. Report your code here.

7.3 I (3 pts): Complete the function `odo_compute_acc` in *odometry.c* to implement the odometry based on the accelerometer (using x **and** y acceleration data). Report your equations (use matrix notation if/where appropriate for conciseness).

Hint: Do not forget to use the current heading estimate provided by the `_odo` variable.

7.4 S (2 pts): Run the simulation and stop it once the robot stops moving. Report the plot of the resulting trajectory by running **Part A** of *plot_main.m*. What happens to the estimated trajectory if you let the simulation run a bit longer after the robot stopped? Why?

7.5 S (1 pts): Set the robot's heading to -2.3562 rad and start the simulation as in Fig.5. Report the plot of the resulting trajectory by running **Part A** of *plot_main.m*. Does the odometry with the accelerometer work as intended?

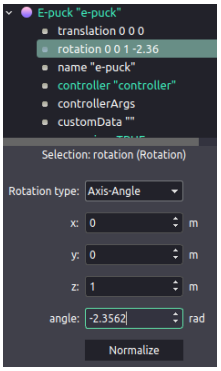

Figure 5: How to adjust the heading of the robot

Set the flag `nav = WAYPOINTS`. This will let the robot drive through a set of pre-defined waypoints around the arena. Note you can first test your solution on a much shorter trajectory by setting `nav = CURVE`, which will make the robot do a 90-degree rotation. However, **make sure to report your results with the `WAYPOINTS` option enabled in this section**.

7.6 I (2 pts): Complete the function `odo_compute_gyro` to implement the odometry based on the gyroscope measurements. Report your equations and the code you implemented.

7.7 Q (1 pts): Run the simulation until the robot stops moving and plot the estimated trajectory by running the script *plot_main.m*, **Part A**. Adjust the zoom level of the plot accordingly and report it. Report the performance of the IMU-based odometry by running **Part A** of *compute_metric.m* (report the generated plot as well). What performance do you obtain? Is the resulting trajectory satisfactory? Why do you obtain good or bad results here?

8. Sensor fusion (38 points)

So far you implemented odometry solutions that do not allow you to incorporate further information to improve the localization of the robot. We will now consider the Kalman filter algorithm to fuse several sources of information and improve the state estimate. We recall the algorithm here:

Algorithm **Kalman_filter**($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$):

1. $\hat{\mu}_t = A_t \mu_{t-1} + B_t u_t$
2. $\hat{\Sigma}_t = A_t \Sigma_{t-1} A_t^T + R_t$
3. $K_t = \hat{\Sigma}_t C_t^T (C_t \hat{\Sigma}_t C_t^T + Q_t)^{-1}$
4. $\mu_t = \hat{\mu}_t + K_t (z_t - C_t \hat{\mu}_t)$
5. $\Sigma_t = (I - K_t C_t) \hat{\Sigma}_t$

Return $\boldsymbol{\mu_t}$, $\boldsymbol{\Sigma_t}$

Figure 3: Kalman filter algorithm

We will use the following configuration of sensors:

- Prediction step: Accelerometer measurements used as estimate of the control input variable u.
- Update step: GPS (measures the x, y position of the robot) as measurement variable z.

For this part, you will use the following model $\mu = A\mu + Bu$, with the state μ containing the pose and velocities of the robot:

$$\mu = [x\ y\ \dot{x}\ \dot{y}]^T$$

$$u = [a_x\ a_y]^T$$

where a_x and a_y represent the acceleration measured by the accelerometer in x and y. The robot heading angle θ is not involved in the state estimation, but its value is updated separately with the result of gyroscope-based odometry. Note that **all quantities in the state estimate are expressed in the world frame**. Therefore, measurements performed in the body (robot) frame must be expressed into the world frame before being incorporated into the state estimate.

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

- Prediction:
  - Odometry:
    - Accelerometer noise 0.03 [m/s^2]
- Update:
  - GPS:
    - Position noise 0.01 [m]

You will use the C++ library Eigen for matrix operations. The library comes packaged within this project, so you do not need to install it yourself. We expose to you adapted matrix types for this homework, which you can find in *matrix.hpp* (look into *part2/libraries*). Have a look at the demonstration on how to manipulate matrices and vectors located in the `matrix_demo` function in *matrix.cpp*. To run this demo, set the flag `MATRIX_DEMO` to *true* in *kalman.cpp*, compile the controller, reset the simulation, and execute one simulation step. The controller will execute the `matrix_demo` function, which prints some matrices in the console and then exits. Note that only the file *kalman.cpp* can perform matrix operations. The controller is written in C and therefore cannot use C++ features.

8.1 Q (2 pts): Now that you have implemented the basic functionalities of the controller, look at the file *controller.c* and explain what steps are taking place in the main function.

8.2 Q (3 pts): Define and report the matrices A and B used as part of the model within the Kalman filter. Recall how the state and input are defined. What quantities of the state are modified by the prediction step? Which are not?

8.3 I (2 pts): Assuming the robot starts at the origin of the world and that its pose is assumed to be perfectly known at first, how should μ and Σ be initialised? In *kalman.cpp*, initialise the state μ and covariance matrix Σ in the function `kal_reset`. Do not forget to set the state according to the provided origin.

Set back the flag `nav = STRAIGHT_LINE`, and put `MATRIX_DEMO` to false. This will let the robot drive straight for a short distance and then stop.

8.4 Q (4 pts): Define and report the covariance matrices of predictionss noise R within the Kalman filter. Note that the accelerometer presents a standard deviation of **0.03 $[m/s^2]$** for all axes.
Hint: $Var(kx) = k^2\sigma_x^2, Covar(BX) = B\sum_X B^T$ *with* σ_x^2 *as variance of x, and* $\sum_X$ *as covariance of the X.*
Hint: Accelerometer noise is expressed in robot body frame

8.5 I (4 pts): Complete the `kal_predict` function to implement the prediction step of the Kalman filter based on your previous answers. Recall that the acceleration measurements occur in the robot frame, but the state estimate is expressed in the world frame. You have to implement the following steps:

- (1 pts) Declare the matrices A and B
- (1 pts) Declare and initialize the input vector u
- (1 pts) Declare the matrix R
- (1 pts) Implement the state and covariance prediction steps

8.6 Q (2 pts): Run the simulation and pause it once the robot stops moving. Report the plot of the resulting trajectory by running **Part B** of *plot_main.m*. How does the prediction step of the filter compare to the IMU-based odometry? Also make sure to test your prediction step as well by setting

the heading of the robot to -2.3562 rad and verify the prediction works also for this arbitrary heading (report the obtained plot as well).

8.7 Q (1 pts): Note that the data acquired from the accelerometer represent control input, so why do we use it as part of the prediction step and not as part of an update step like the GPS?

In general, to fuse the data from a sensor into the state estimate through an update step, the matrices C (design or state to measurement matrix) and Q (measurement uncertainty matrix) need to be defined, as well as the vector z that contains the measurement data. You will define these quantities for each measurement that has to be considered by the filter.

The GPS data $[G_x, G_y]$ can now be fused. Set the flag `nav = WAYPOINTS`. This will let the robot drive through a set of pre-defined waypoints.

8.8 I (1 pts): Complete the function `controller_get_gps` in *controller.c*. You can verify that the measurements are occurring as intended by setting the flags VERBOSE_GPS to true to print the data into the console. Set the flag back to false once you are done.

8.9 Q (3 pts): For the GPS update, define the matrices C and Q and vector z, where the measurement provided by the GPS corresponds to the x and y position of the robot. Note that the GPS presents a standard deviation of **0.01 [m]** for all axes.

8.10 I (1 pts): To implement the fusion of the GPS data, set the flag `FUSE_GPS` to true (in *controller.c*). Complete the `kal_update_gps` function in *kalman.cpp* by declaring the matrices C and Q and measurement vector z.

8.11 I (3 pts): Implement the generic `kal_update` function in *kalman.cpp*. This function implements the three equations of the update step of the Kalman filter algorithm.

8.12 S (1 pts): Run the simulation until the robot stops moving and plot the resulting trajectory by running **Part B** of *plot_main.m*. Report the obtained plot. Run **Part C** of *compute_metrics.m* and report the obtained plot and metrics as well.

8.13 Q (1 pts): Run **Part D** of *plot_main.m*. Comment on the observed covariances. Do you observe any unexpected behavior?

8.14 Q (10 pts): Now we will add the heading angle θ, and the angular velocity $\dot{\theta}$ into the system state, and the gyroscope reading v_θ as the system input, given that: $\mu' = \begin{bmatrix} x & y & \theta & \dot{x} & \dot{y} & \dot{\theta} \end{bmatrix}^T$ and $u' = [a_x \;\; a_y \;\; v_\theta]$. To update the estimation of the heading angle, the measurements of the compass C_θ is used, given that $z' = [G_x \; G_y \; C_\theta]$. Since now the system become nonlinear, an Extended Kalman Filter (EKF) is required, which uses the same steps for the state prediction and update as a Kalman filter, but with system transient matrix and measurement matrix calculated through their respective

Jacobian. We recall the definition of EKF here, with α_t and β_t represents the process and measurement noise.

System equation: $\bar{\mu}'_t = f(\mu'_{t-1}, u') + \alpha_t$

$$G_t = \frac{\partial f}{\partial \mu'}(\mu'_{t-1}, u') \quad L_t = \frac{\partial f}{\partial u'}(\mu'_{t-1}, u')$$

Measurement equation: $z'_t = g(\bar{\mu}'_t) + \beta_t$

$$H_t = \frac{\partial g}{\partial \mu'}(\bar{\mu}'_t)$$

Report the definition of system equation f and measurement equation g as well as G_t, L_t, H_t.

Part III: Fitness tracker with Arduino (30 pts)

In this part of the homework, you will use the Arduino kit to design your own fitness tracker. This part of the homework is open-ended. Please explain your reasoning **clearly and concisely**. You will have a chance to demo your fitness tracker at the end of the semester.

Note: we will grade your answers based on correctness, completeness of implementation, critical reasoning, and clarity of presentation. We will take away points if it is not possible for us to understand what you implemented.

9. Step Counter – 13 pts

The main feature you will implement for your fitness tracker is a step counter. Your step counter should be able to **count** the steps taken by a person holding the Arduino kit and **display** the number of steps on the Arduino's screen.

9.1 Q (4 pts): Describe the implementation of your step counter. Which sensor(s) will you use? How do you compute a step? How do you display the number of steps taken? Provide a pseudocode of your implementation and justify your choices. The pseudocode should highlight **all** aspects of your implementation.

9.2 I (6 pts) Open the file *part3/step_counter/step_counter.ino.* Implement the step counter based on your answer to the previous question. Comment your code and make sure that your implementation matches the pseudocode you provided above. *Note: we should be able to quickly see where you implemented each part of your code. Use comments and functions when necessary.*

9.3 Q (3 pts) Analyze the performance of your step counter. How robust is it? What are the limitations? How could you improve its performance? *Note: present some data and plots for this section to support your reasoning*

10. Additional Feature – 7pts

10.1 Q (2 pts) Envision an additional feature for your fitness tracker that uses the hardware provided. Be creative! In the report, briefly explain which feature you decided to implement and how you plan on implementing it. Provide a pseudocode of your additional feature and explain **all** aspects of it.

10.2 I (3pts) Implement your additional feature. First copy your step counter code in part3/fitness_tracker/fitness_tracker.ino, then add your feature. Highlight where the feature is added using comments in your code.

10.3 Q (2pts) Analyze the results of your additional feature. How robust is it? What are the limitations?
Note: present some data and plots for this section to support your reasoning

11. Arduino Demo – 10pts

Showcase your work and answer questions at the final demo.

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