Homework 1

This homework requires the following software:

- C development tool (gcc, make, etc.)
- Webots R2022a
- Matlab R2021a or later
- OS: Ubuntu 20.04

Information

Question types

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

In Part 1 of this homework, you will work on Ant Colony Optimization. In Part 2, you will work on navigation, state estimation and sensor fusion. In Part 3, you will implement flocking and formation algorithms for collective robot movements.

This homework comprises 120 points in total. The points are distributed as follows among the parts: Part 1 – 15 points, Part 2 – 60 points and Part 3 – 45 points.

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 to find a solution (e.g., splitting the team or grading separately the individual contributions).

Submitting your answers

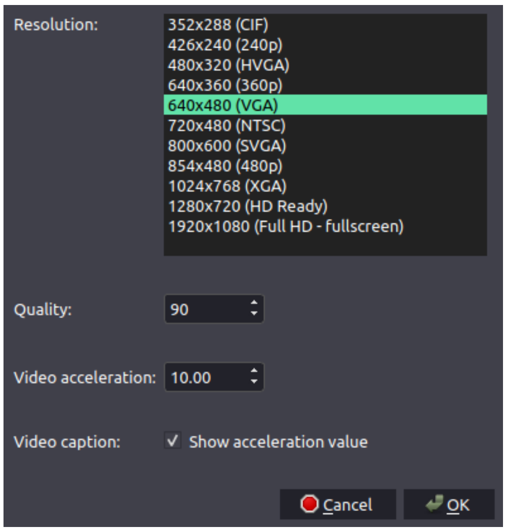
The deliverable for Homework 1 must be a zip archive named *DIS_21-22_hwk1_Group_X.zip*, where *X* is your group number. It must contain the following elements:

- The folders *part1*, *part2* and *part3* with your solutions implemented. **Submit the exact files that you were given as part of the assignment**. Any additional file you submit will be ignored.
- The videos *reynolds5_waypoints.mp4*, *reynolds10_waypoints.mp4*, *reynolds10_crossing.mp4* and *laplacian5_waypoints.mp4*.
- A report in pdf format named *DIS_21-22_ hwk1_Group_X_report.pdf*, where *X* is your group number. The report should contain the answers to the questions below. Try to keep the report short and to the point. **Precisely indicate which question you are answering throughout your report**.

Instructions for video recording

When asked to record a video of your simulation, use the following settings (also illustrated in the figure below):

- Choose **VGA resolution** (640x480)
- Set the **quality to 90**
- Set the video **acceleration value to 10**
- Enable the "**Show acceleration value**" field


Resolution:
352x288 (CIF)
426x240 (240p)
480x320 (HVGA)
640x360 (360p)
640x480 (VGA)
720x480 (NTSC)
800x600 (SVGA)
854x480 (480p)
1024x768 (XGA)
1280x720 (HD Ready)
1920x1080 (Full HD - fullscreen)

Quality: 90

Video acceleration: 10.00

Video caption: ✓ Show acceleration value

Make sure the robots are clearly visible and do not leave the field of view of the camera (you can reposition the viewpoint while recording if necessary, but try to position it properly beforehand). Please use the exact name specified in the corresponding questions to name your videos. Make sure to deselect any object in the scene when filming to avoid seeing its handles.

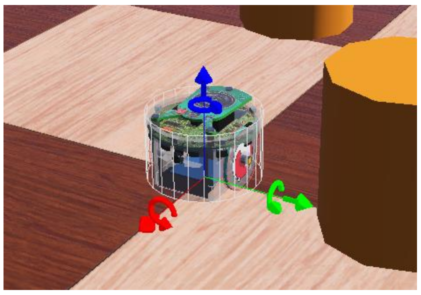


Figure 2: Bad (clicked on robot)

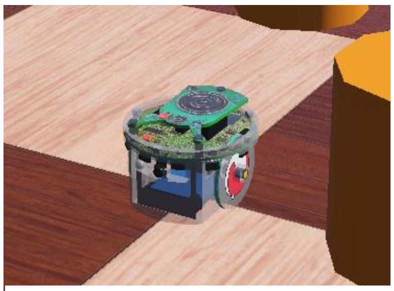


Figure 1: Good (clicked on the background to deselect everything)

Part 1 - Ant Colony Optimization (15 points)

In this section of the Homework, you will work on Ant Colony Optimization (ACO) applied to the Travelling Salesperson Problem (TSP), as seen in Lab 1. All the code related to this part can be found in the folder *part1*. To get started, open the directory *part1* in Matlab.

1.1. Q (2 pts): The file *run_tsp.m* allows you to run several simulations of TSP. Implement code that computes mean and standard deviation of the runs. Test it and report the values you obtain by running the code 10 times with the command `run_tsp(10,’ant’,cities)`. Recall from Lab 1 that you must first run the command `load(‘eli51.mat’).`

1.2. Q (1 pts): Now change the values of *alpha* and *beta* to *alpha=5* and *beta=10* in the `'ant'` case of the `tsp` function in the file *tsp.m*. Report the mean and standard deviation for 10 runs (use the command `run_tsp(10,’ant’,cities)`). What happens? Can you explain why?

1.3. I (4 pts): Revert to the original values of *alpha=beta=5*. The α (alpha) and β (beta) parameters can be modified to be a function of time rather than being constant values. The functionality can be designed to fine tune the impact of pheromones and heuristic information. The idea is to allow the ants to rely mostly on the heuristic information in the beginning when the pheromone trails are not very reliable yet. As time goes by and the pheromone trails form, the ants can rely more and more on the information provided by pheromone and eventually almost completely ignore the heuristic, to evaluate the quality of a path. Modify the alpha and beta parameters definition in the code such that they change linearly with time in the function `solvetsp_ant` in *tsp.m*. Report your solution and the mean and standard deviation values over 10 runs. Briefly explain your implementation and comment on the results you found.

1.4. I (3 pts): Keep the changes and add an evaporation rate of 10% for the pheromone trail. Where do you implement this change and how? Report the mean and standard deviation over 10 runs and comment on your result.

1.5. I (5 pts): A 2-opt local search systematically evaluates all permutations of a solution that can be achieved by exchanging two edges. Write the missing code to implement such permutation of edges in the function `local_search_2opt` in the file *tsp.m*. The skeleton of the function is already provided. Use the command `run_tsp(10,’ant’,cities,1)` to test your implementation. Report the mean and standard deviation over 10 runs and comment on your results.

Part 2 - Navigation and State Estimation (60 points)

In this section of the homework, you will work on navigation, state estimation and sensor fusion. All the code related to this part can be found in the folder *part2*. Open the *part2/matlab* folder in Matlab.

The robot you will use in this homework is the e-puck. You can find the complete documentation of the robot model here: https://cyberbotics.com/doc/guide/epuck

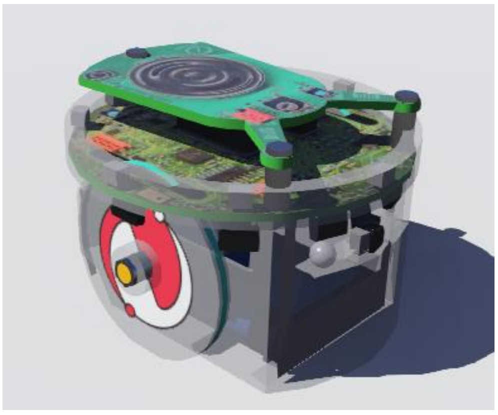


Figure 3: Simulated e-puck in Webots

Note that a GPS was added to the robot's model for the purpose of this homework.

The world frame follows the NWU (x-North, y-West, z-Up) convention and the robot's body frame is expressed as FLU (x-Forward, y-Left, z-Up). Hence, when the robot lays at the origin with its heading equal to zero, both the world frame and the body frame are perfectly aligned. This applies for all the remaining parts of this homework.

2.1. Navigation (5 points)

Start Webots and open the world *arena_obstacles.wbt*. Make sure to set `nav = BRAITENBERG` at the beginning of *controller.c* and compile the controller.

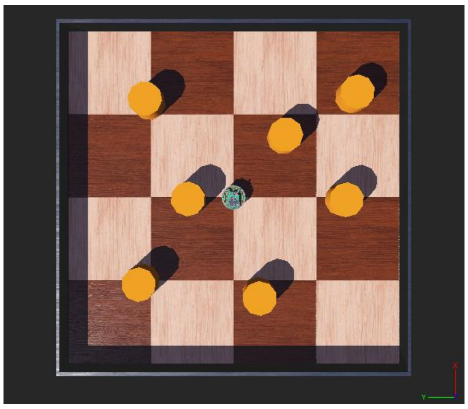


Figure 4: arena_obstacles.wbt

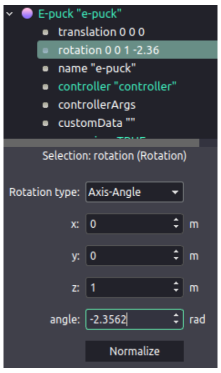
2.1.1. I (1 pts): Complete the function `controller_get_distance()` in *controller.c*, that reads the range values from the distance sensors. The sensors are already initialised for you.

2.1.2. I (2 pts): Implement a Braitenberg controller at the specified location in the function `controller_set_speed()` in the file *controller.c* and report the weights you selected along with a rationale motivating your choice.

2.1.3. S (1 pts): Let your robot roam through the arena and plot the obtained trajectory using the ground truth by running the script *plot_braitenberg.m*. How do you handle frontal obstacles, or more generally, symmetrical detections (e.g., a corner)?

As a general comment for part 2 of the homework, note that the robot can be placed and/or oriented arbitrarily in the arena before starting a simulation run, since the initial pose of the robot is used to initialise all state variables within the controller.

2.1.4. S (1 pts): Show that your controller can handle corners by applying a rotation in z to the robot's orientation of -2.3562 rad (-135°) before starting the simulation, so that the robot is at the center of the arena and faces the bottom right corner. Run the simulation and let the robot roam (the robot should move straight towards the corner as it will not encounter any obstacle) and then turn away from the corner. Report the resulting trajectory similarly to the previous question by running *plot_braitenberg.m* (you can stop the simulation once the robot has cleared the corner).



```
E-puck "e-puck"
    translation 0 0 0
    rotation 0 0 1 -2.36
    name "e-puck"
    controller "controller"
    controllerArgs
    customData ""
```

Selection: rotation (Rotation)

Rotation type: Axis-Angle

x: 0 m

y: 0 m

z: 1 m

angle: -2.3562 rad

Figure 5: How to adjust the heading of the robot

2.2. Odometry (15 points)

Open the world *arena.wbt* in Webots.

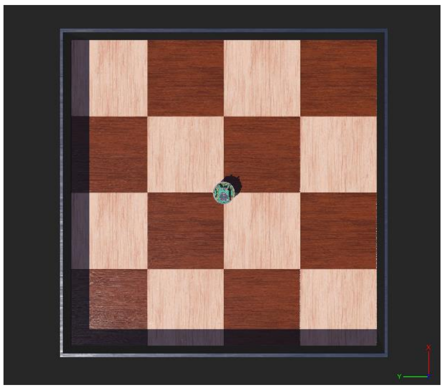


Figure 6: arena.wbt

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).

2.2.1. 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.

IMU (8 points)

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.
- 1D odometry (measuring the robot heading angular velocity) with the gyroscope.

Set the flag `nav = STRAIGHT_LINE`. This will let the robot drive straight for a short distance and then stop.

2.2.2. I (1 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.

2.2.3. I (2 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.

2.2.4. S (1 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?

2.2.5. S (1 pts): Similar to what you did when testing your Braitenberg controller, set the robot's heading to -2.3562 rad and start the simulation. Report the plot of the resulting trajectory by running **Part A** of *plot_main.m*. Does the odometry with the accelerometer work as intended?

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`. However, **make sure to report your results with the `WAYPOINTS` option enabled in this section**.

2.2.6. I (1 pts): Complete the function `odo_compute_gyro` to implement the odometry based on the gyroscope measurements. Report your equations and the code you implemented.

2.2.7. Q (2 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?

Wheel encoders (6 points)

2.2.8. I (1 pts): Complete the function `controller_get_encoder` to read the wheel angles. You can verify that the measurements are occurring as intended by setting the flag `VERBOSE_ENC` to true to print the data into the console. Set the flag back to false, once you are done.

2.2.9. I (3 pts): Complete the function `odo_compute_encoders` to implement the odometry based on the wheel encoders. Report your equations.

2.2.10. Q (2 pts): Run the simulation and plot the estimated trajectory by running the script *plot_main.m*, **Part A**. Report the performance of the encoder-based odometry by running **Part B** 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?

2.3. Sensor fusion (40 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:

```
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 $\mu_t, \Sigma_t$
```

Figure 7: Kalman filter algorithm

We will use the following configuration of sensors:

- Prediction step: Accelerometer measurements used as estimate of the control input u.
- Update step: Gyroscope (measures the heading rate), encoders (measure the body velocity in x and y as well as the heading rate), and GPS (measures the x, y position of the robot) as measurements 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:

$$\mu = \left[x\; y\; \theta\; \dot{x}\; \dot{y}\; \dot{\theta}\right]^T$$

and

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

where a_x and a_y represent the acceleration measured by the accelerometer in x and y, projected into the world frame. Note that **all quantities in the state estimate are expressed in the world (inertial) frame**. Therefore, measurements performed in the body (robot) frame must be expressed into the world frame before being incorporated into the state estimate.

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 as the controller is otherwise written in C and therefore cannot use C++ features.

2.3.1. Q (1 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.

2.3.2. 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?

2.3.3. Q (1 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?

2.3.4. Q (1 pts): Define and report the process noise matrix R. Recall that only the accelerometer is used for measuring the control input of the system. Be therefore especially mindful of which value you assign to the heading-related covariances. The accelerometer has a standard deviation of **0.003 [m/s²]** for all axes. Justify your choice of values for R.
Hint: As you saw in Lab 3, the timestep (16 ms) should be considered.

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

2.3.5. I (1 pts): 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.

2.3.6. I (6 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
- (2 pts) Declare and initialise the input vector u
- (1 pts) Initialise the matrix R
- (2 pts) Implement the state and covariance prediction steps

2.3.7. Q (1 pts): Run the simulation and pause it once the robot stops moving. Report the plot of the resulting trajectory by running **Part D** 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).

2.3.8. Q (1 pts): Note that the data acquired from the accelerometer represent measurements, so why do we use it as part of the prediction step and not as part of an update step like the other measurements (gyroscope, wheel encoders and GPS)?
Hint: You can refer to the selected state representation to answer this question.

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.

Set the flag `nav = WAYPOINTS`. This will let the robot drive through all waypoints. Run the simulation so that the robot goes through all waypoints once to collect some data and then pause the simulation.

2.3.9. I (1 pts): With the collected data, compute and report the standard deviation of the data generated by the gyroscope by completing **Part B** of *plot_main.m*. To this end, you can use the ground truth to compare the true heading rate against the ones that are measured.

Set the flag `nav = CURVE`. This will let the robot drive in a curve and will automatically pause the simulation once the robot turns 90°.

2.3.10. Q (3 pts): For the gyroscope update, define and report the matrices C and Q and vector z, where the value measured by the gyroscope is the heading rate (rotational speed around z-axis).

2.3.11. I (1 pts): To implement the fusion of the gyroscope data, set the flag `FUSE_GYRO` to true in *controller.c*. Based on your previous answers, complete the `kal_update_gyro` function in *kalman.cpp* by declaring the matrices C and Q and measurement vector z.

2.3.12. 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.
Hint: The arguments of the `kal_update` function are of type `MatX` or `VecX`, therefore arbitrary sized matrices or vectors can be given as arguments to this function. This means this function can be used for all the different updates considered in this homework without any modification.

2.3.13. Q (1 pts): Run the simulation until the robot stops moving and plot the resulting trajectory by running **Part D** of *plot_main.m*. Report and comment on the obtained plot.

The wheel encoder data can now be fused as well. Set the flag `nav = WAYPOINTS`. This will let the robot drive through a set of pre-defined waypoints.

It is first required to evaluate the noise of the speed estimate measured from the wheel encoders. To do this, let the robot go through all waypoints once and then stop the simulation.

2.3.14. I (2 pts): With the collected measurements, compute the standard deviation of the data (x, y velocities and body rate) generated by the encoders by completing **Part C** of *plot_main.m*. To this end, you can use the ground truth to compare the true velocities against the ones that are measured with the wheel encoders. To this end, use the pose estimate generated by the encoders from the odometry part and derive it to compute the resulting velocities. Explain your approach to compute the standard deviations.

2.3.15. Q (3 pts): For the encoder update, define the matrices C and Q and vector z, where the values measured by the encoder are the velocities in x and y and the heading rate (rotational speed around z-axis).

2.3.16. I (3 pts): To implement the fusion of the encoders' data, set the flag `FUSE_ENC` to true. Based on your previous answers, complete the `kal_update_enc` function in *kalman.cpp* by declaring the matrices C and Q and measurement vector z.

2.3.17. S (1 pts): Run the simulation (you should already have set `nav = WAYPOINTS`) until the robot stops moving and plot the resulting trajectory by running **Part D** of *plot_main.m*. Report the obtained plot. Run **Part C** of *compute_metrics.m* and report the obtained plot and metrics. Comment on the results.

Finally, we want to fuse the GPS data.

2.3.18. 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.

2.3.19. 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.

2.3.20. I (1 pts): To implement the fusion of the GPS data, set the flag `FUSE_GPS` to true. Based on your previous answers, complete the `kal_update_gps` function in *kalman.cpp* by declaring the matrices C and Q and measurement vector z.

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

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

Part 3 - Collective Movements (45 points)

In this part, you will implement collective movement algorithms in Webots with several e-pucks. You will first develop a Reynolds flocking solution for various flock sizes (from 5 up to 10 robots) and then implement a Laplacian-based controller for 5 robots. You will implement your solutions in the *collective.c* controller file and *flocking_super.c* supervisor file.

3.1. Flocking with Reynolds rules

Open the Webots file *obstacles.wbt*. It will look like so:

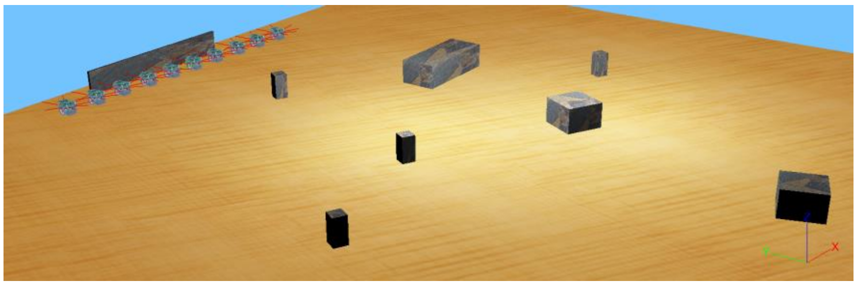


Figure 8: obstacles.wbt with 10 e-pucks

You will edit/work with the files

- *flock_params.h*: header files where several parameters are defined for the collective algorithms. If this file is modified, **you have to recompile both the robots' and the supervisor's controller** (hence *collective.c* and *flocking_super.c*).
- *collective.c*: This is the robots' controller files. All robots have the same controller.
- *flocking_super.c*: This is the supervisor controller. It localises the robots and broadcasts their locations with an emitter. Each robot has a receiver that it can use to receive information about the other robots of the flock.

Compile both the robots' (*collective*) and the supervisor's (*flocking_super*) controller. At this point, the robots will not move when running the simulation.

In the robot's controller *collective.c*, the pose of each robot of the flock is stored in the array `float loc[FLOCK_SIZE][3]`, which contains the x, y coordinate of a robot and its heading. `FLOCK_SIZE` is the number of robots in the flock.

3.1.1. Q (1 pt): Read through *collective.c* to understand how the controller is structured. What do the variables `robot_id_u`, `robot_id`, `loc`, `prev_loc`, `speed` and `migr` represent (declared at the beginning of the file)?

The way the robot's wheels are controlled is by setting the global variable `speed[robot_id]` with a desired 2D speed vector expressed in the world frame. This speed vector must then be converted into wheel speeds.

3.1.2. I (4 pts): Complete the function `compute_wheel_speeds` to translate the speed vector `speed[robot_id]` expressed in the global frame into wheel speeds. You can use the current heading of the robot stored in `loc[robot_id][2]`.
Hint: you can temporarily set the `speed[robot_id]` variable to an arbitrary constant value at the beginning of the function to test your solution, but do not forget to remove it once you are done!

3.1.3. I (3 pts): Implement a Braitenberg-based avoidance solution in the function `compute_braitenberg_speed`. You can re-use (and adapt) your Braitenberg solution implemented in Part 2. You need to set the `avoid` variable to true if avoidance is required.

We will first consider a flock of 5 robots. Make sure you set `FLOCK_SIZE` to 5 and `NUM_FLOCKS` to 1 in *flock_param.h*. We want to handle the case where the robots cannot interact with each other beyond a given range defined by `NEIGH_THRESHOLD`.

3.1.4. I (4 pts): In *collective.c*, in `reynolds_rules` function, compute the average speed and average location of the flock for both the whole flock or only for a neighborhood (whose radius is defined by `NEIGH_THRESHOLD`) around a given robot, depending on the value of `NEIGHBORHOOD` (either 1 or 0).

3.1.5. I (2 pts): Implement Reynolds flocking in the function `reynolds_rule`. Implement the 3 rules you saw in the course, namely: cohesion, separation, and alignment. At the end of the function, store the resulting velocity derived from the Reynolds rules into `speed[robot_id]`. Tune the Reynolds weights so that the flock behaves appropriately.

Currently, the robots will only stay in place and will not travel towards any objective. We need to add a migratory urge to push the flock in a given direction.

3.1.6. I (2 pts): Add a migratory urge at the end of the `reynolds_rules` function that considers the migration objective stored in the `migr` variable that shows that the flock is able to reach all waypoints sent by the supervisor (the simulation will automatically pause once all waypoints are reached).

3.1.7. I (4 pts): Implement the following performance metric in *flocking_super*.c in the function `compute_flocking_fitness` to evaluate the performance of your flocking algorithm when travelling between waypoints:

$$M_{fl}\,[t] = o[t] \cdot d[t] \cdot v[t], \qquad with\;\; o, d, t \in [0,1]$$

The orientation alignment of the robots $o[t]$ is measured as:

$$o[t] = \frac{1}{N}\left|\sum_{k=1}^{N} e^{i\psi_k[t]}\right|$$

Where ψ_k represents the absolute heading of robot k and N the number of robots in the flock. The distance between robots $d[t]$ is measured as:

$$d[t] = \left(1 + \frac{1}{N}\sum_{k=1}^{N} \left| dist(x_k[t], \bar{x}[t]) \; - \; rule1_{thres} \right| \right)^{-1}$$

Where x generally denotes an x, y location, $\bar{x}$ denotes the flock's average location and $rule1_{thres}$ is the distance threshold for the rule 1 (cohesion) of Reynolds' flocking. The velocity of the team towards the migration goal, v[t], is computed as:

$$v[t] = \frac{1}{v_{max}} max(proj_m\left(\frac{\bar{x}[t] - \bar{x}[t-1]}{\Delta T}\right), 0)$$

where $proj_m$ denotes the projection of the flock's velocity onto the vector linking the flock's center of mass to the migration goal, ΔT the sampling interval and v_{max} the maximal speed a robot can achieve. *Hint: Use the global variable `float loc[FLOCK_SIZE*NUM_FLOCKS][3]` declared at the top of flocking_super.c.*

3.1.8. Q (2pts): Run the simulation and let the flock go through each waypoint and record a video (name it *reynolds5_waypoints.mp4*). Plot the obtained fitness by running **Part A** of *plot_fitness.m* and report the obtained plots in your report. Report the time it takes the flock to go through all waypoints and comment on the results.

3.1.9. Q (2pts): Redo the same steps as in the previous question and record a video (name it *reynolds10_waypoints.mp4*), but with **10 robots** this time. To this end, make sure you set `FLOCK_SIZE` to 10 in *flock_params.h* (make sure to compile both the controller and the supervisor!). Plot the obtained fitness by running Part A of *plot_fitness.m* and report the obtained plots in the report. Report the time it takes to flock to go through all waypoints and comment on the results.

3.1.10. S (1 pts): Again, still using 10 robots, set `NEIGHBORHOOD` to 1. And let the flock go through all waypoints. Try the following neighborhood radiuses: 0.3, 0.4, 0.5, 0.6 [m] (set `NEIGH_THRESHOLD` accordingly). Feel free to explore further radiuses. Report the time it takes the flock to go through all waypoints for each radius.

3.1.11. Q (2 pts): Qualitatively comment on the time it takes the flock to go through all waypoints, based on the previous experiments you carried out (by varying the flock size and/or the neighborhood size). Can you explain what you observe?

So far, all obstacles encountered by the robots were static. Open the world *crossing.wbt* that opposes two groups of 5 robots. The two groups have to get across each other and reach the starting location of the opposite group.

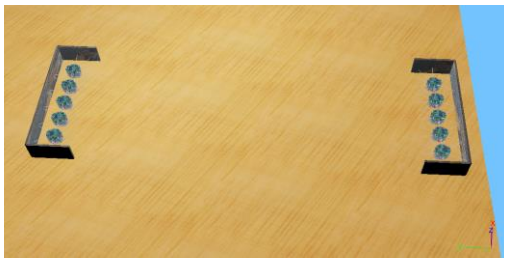


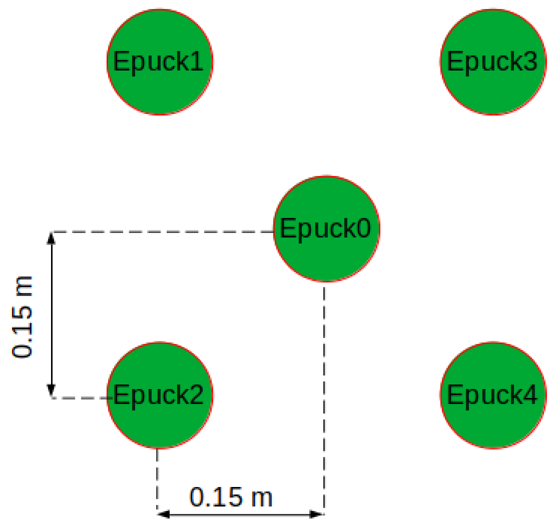
Figure 9: crossing.wbt

3.1.12. S (2 pts): Set the following constants in *flock_params.h*: `FLOCK_SIZE` to 5, `NUM_FLOCKS` to 2 and `NEIGHBORHOOD` back to 0. Record a video (name it *reynolds10_crossing.mp4*) of the two flocks crossing and qualitatively comment on how they perform. How does it compare to the static obstacle case? Do you see any noticeable difference? Comment on the performance of your controller and what you would do to improve its performance in this scenario.

3.2. Laplacian formation

We will now focus on developing a Laplacian controller for a formation of 5 e-pucks. For this, open again the *obstacle.wbt* world.

3.2.1. Q (2 pts): Go to the parameter file flock_param.h. Uncomment the line `#define LAPLACIAN`, which activates the Laplacian controller throughout the project. Make sure that the `FLOCK_SIZE` parameter is set to 5 and the `NUM_FLOCKS` parameter is set to 1. Now complete the Laplacian matrix and the bias matrix to create a formation that follows the schema below. Report and comment on the values of L and bias in the report.



3.2.2. I (6pts): In the file *collective.c*, go to the function `laplacian_formation`. Implement the Laplacian formation for the flock.
Hint: Use the variables `L`, `bias` and update the variable `speed`.

3.2.3. I (2pts): Add a migratory urge at the end of the function `laplacian_formation`. Record a video (*laplacian5_waypoints.mp4*) that shows that the formation can reach all waypoints sent by the supervisor and comment on your implementation of the Laplacian formation on the report. Report the time it takes to flock to go through all waypoints.

3.2.4. I (4 pts): In *flocking_super.c*, implement the following metric in the function `compute_formation_fitness` to evaluate the performance of your formation (see below for more details on how the metric is defined):

$$M_{fo}[t] = d[t] \cdot v[t], \qquad with\ d, v \in [0,1]$$

The overall metric is the average of all M_{fo} values for each time stamp t. Moreover, $d[t]$ is computed as:

$$d[t] = \left(1 + \frac{1}{N}\sum_{k=1}^{N} ||x_k - g_k||\right)^{-1}$$

Where N is the number of robots, x_k is the position of robot k and g_k is the target position of robot k in the formation. Note that this metric represents the error of the robots with respect to the central robot (epuck0).

Finally, $v[t]$ is computed as:

$$v[t] = \frac{||x[t] - x[t-1]||}{D_{max}}$$

Where D_{max} is the maximal distance possible per timestep, given the robot maximum speed v_{max} .

3.2.5. Q (2 pts): Save the metrics for a total run of the experiment (identified as all the time it takes the formation to reach all three migratory urge positions). Plot them by running part B of *plot_fitness.m* and report the plots in the report. Briefly comment on your results.