1 Lab 4: Coordinated Movements in Webots

This laboratory requires the following equipment:

- C programming tools (gcc, make)
- Webots simulation software
- Webots User Guide
- Webots Reference Manual
- Matlab

Depending on your programming skills, the laboratory duration might require an effort of up to five hours although assistance will be provided in the computer rooms during a time window of three hours. We encourage you to take notes during the course of this laboratory to aid in the exams. A solution will be posted a few days after the lab session.

Office hours

Additional assistance outside the lab period (office hours) can be requested using the dis-ta@groupes.epfl.ch mailing list.

1.1 Information

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

- The notation $\mathbf{S_x}$ means that the question can be solved using only additional simulation or an experimental manipulation.
- The notation $\mathbf{Q_x}$ means that the question can be answered theoretically, without any simulation or an experimental manipulation.
- The notation $\mathbf{I_x}$ means that the problem has to be solved by implementing a piece of code and performing a simulation or an experimental manipulation.
- The notation $\mathbf{B_x}$ means that the question is optional and should be answered if you have enough time at your disposal.

To prepare yourself for the exam and to allow you for better time planning during the exercise session, we show an indicative number of points for each exercise between parentheses. The combined total number of points for the laboratory is 100.

2 Flocking Using Reynolds' Rules

For some applications (e.g., lawn-mowing, vacuum cleaning, security patrols, search and rescue in hazardous environment) it might be useful to have a multi-robot system moving as a spatially compact group towards a target position. Figure 1 shows an example of this type of behavior.

In the paper entitled "Flocks, Herds, and Schools: A Distributed Behavioral Model", by Craig W. Reynolds, a successful simulation of flocking is described. Reynolds lists three rules which he determined are needed for realistic flocking:

1. Cohesion: attempt to stay close to nearby flockmates,
2. Separation: avoid collisions with nearby flockmates,
3. Alignment: attempt to match velocity with nearby flockmates.

In this lab, you will explore the influence and importance of each of these rules.

2.1 Reynolds' rules weights

First you need to download the lab package from the course webpage on Moodle, following these steps:

- Download the *Lab04.tar.gz* file from the WEEK 5 section of the course's Moodle site, to the directory of your choice.
- Extract the file with this command: `tar xvzf Lab04.tar.gz`
- Launch Webots from a terminal by entering this command: `webots &`

S1 (2): Load *flocking_reynolds.wbt* in Webots. The robots are controlled by the *reynolds.c* controller. The objective of this controller is to implement the necessary functions to run the Reynolds' flocking behaviors. There is a supervisor (named *flocking_super.c*) which provides absolute positions to the robots using an emitter/receiver module.

Compile the controller and the supervisor and run the program (if Webots asks you to create a Makefile, click on OK). You may need to run the simulation in fast mode to understand the collective behavior of the robots. Each robot will try to get close to the other neighbors, in a flock, while moving forward in its own (individual) direction. *Note that there is only one unique controller that is run in all robots*. There is no leader or follower for this homogenous flock. Change the initial position of the robots to arbitrary locations and run the simulation to see if the behavior changes.

Note that, as it is, the controller does not implement the "alignment "rule. Do you note any weird behaviors of the flock related to this issue?

Q2 (1): The supervisor (*flocking_super.c*) sends the position of the robots to them through an emitter device. What part of the *reynolds.c* controller code receives the positions? What variable stores the location of the other neighboring robots in the controller code?

Q3 (2): What part of the code calculates the "cohesion" behavior? What part does the "separation" behavior?

S4 (3): Decrease the influence of "cohesion" in the controller by setting a smaller value (e.g., 1/3 of its current value) to its weight. Run the controller several times with different values of this weight and notice its effect.

S5 (3): Set the "cohesion" coefficient to its initial value and then change the value of "separation" weight to various values (e.g., even 10 times its default value). Run and notice its impact.

I6 (5): Set the "cohesion" coefficient to its initial value. Implement the "alignment" rule on the robot controller (use the "alignment" weight already provided in the code). Re-compile the controller and the supervisor and run the program. What improvements do you observe in the flock? Note the "flocking performance" metric that is printed out (you will need it for later).

2.2 The impact of flock size on the Reynolds' flocking behavior

S7 (2): *Set all the modified weights/thresholds to their initial values* and then add two more e-puck robots to the scenario. To add more e-puck robots follow these instructions:

1- Stop the simulation and revert the world.

2- Copy one e-puck (select it and then press ctrl+c) and paste (press ctrl+v) it in another location.

3- Select the newly added robot and (in the left menu of Webots) change its "DEF" values to "epuck4". The first added robot should be "epuck4", the second should be "epuck5" and so on.

4- Open the DEF tree of the newly added robot and change its "name" to the same value as you have entered for "DEF" ("epuck4", "epuck5", …)

5- Verify that the robots controller is set to "*reynolds*".

6- Save the world file with another name. Make sure simulation time is zero before saving.

7- In the "*reynolds.c*" and "*flocking_super.c*" files, modify the value of "FLOCK_SIZE" according to the number of robots you added.

Compile the controllers and run the simulation. Verify if the robots still perform the same flocking behaviors. *Hint:* If 2 robots try to get into the same place, try increasing the separation threshold.

S8 (5): Add a few more robots (3, 4 or even more) to the scenario and see if the flocking behavior is still the same. Do you observe any difference? What do you conclude?

2.3 The impact of neighborhood on the flock

In this flocking algorithm, each robot considers all the other robots in its local neighborhood. This implies that (based on the "cohesion" rule) all the robots tend to move towards the center of the flock. When the flock size is large, the robots will cluster in groups instead of being homogeneously spread throughout the flock. One solution for this problem is to consider only local neighbors in the "cohesion" rule.

I9 (5): Add one marginal threshold to the code such that in the cohesion rule, robots only flock with the neighbors which are closer than 40 cm. Run the simulation having about 8 to 10 robots and note the difference. *Hint: In the current code there is a part that calculates the center of the flock and stores it in a variable named "avg_loc". Change this part such that only the flockmates that are in close vicinity are considered in this calculation.*

2.4 The impact of obstacles on flocking

The controller code that you have been working with includes a Braitenberg obstacle avoidance behavior. In this part we study the impact of obstacles on the Reynolds' flocking method.

S10 (1): Reverse the modifications that you have done in **I9**. Locate the lines which implement the Braitenberg obstacle avoidance. How does it work?

S11 (3): Move the obstacles in the arena and put them in front of the path of flock. How does the flock react to the obstacle?

I12 (5): Now, instead of each robot wanting to go towards its own direction, we will implement a migratory urge towards a certain position, common to all the robots. The migration position is stored in variable `migr` on the code. Set MIGRATORY_URGE to 1, and implement the migratory urge in the specified place in the code. Compile and run the program.

S13 (3): Repeat **S11** with the modifications that you have done in **I12**. How is the flock reacting to the obstacle now? Why? What influence does the "cohesion" weight has in this case?

2.5 Flocking using odometry-based localization

The flocking controller which you have been working with so far uses accurate absolute positions of the robots provided by the Webots supervisor. In many real world scenarios there is no central station which can provide this information to the robots. Instead the robots can estimate their own locations and share this information among their teammates without needing a supervisor. In this part we will get familiar with the local location estimation and sharing in this flocking scenario.

Q14 (2): There is a function in the "*reynolds.c*" controller which calculates the absolute position of the robot using its odometry. Locate this function in the controller and try to understand how it works. Note also that the initial position of the robots is set in function *"initial_pos()"* and is sent by supervisor in *"send_init_poses()".*

I15 (6): Now we want to make the robots (instead of the supervisor) send their absolute positions to each other. First, in the supervisor code, find the part (2 lines) that broadcasts the positions and comment it out. Then, In the *main* function of the "*reynolds.c*", add some lines of code that allow each robot to send its local position (estimated by odometry) to the neighbors. Compile the controller and the supervisor and run the simulation several times. What is the main difference between the behaviors of the robots and the reported "flocking performance" compared to that of Section 2.2? What are the possible reasons?

Hint: use the same Webots function that you commented out in the supervisor code to communicate between robots.

2.6 Flocking using dedicated relative localization (range and bearing)

In Section 2.5, you experienced the issues raised due to inaccurate local position estimation of the robots. In this section, you will see how robots can use relative range and bearing in their flocking algorithm. Using infra-red emitter/receivers, e-pucks are able to estimate their distance and their bearing through exchanging messages and measuring their strength. (for more details refer to Webots manual document and look for wb_receiver_get_signal_strength() and wb_receiver_get_emitter_direction()).

I16 (2): Open *flocking_reynolds.wbt* and the *reynolds2.c* controller. Read the code and try to find out how the robots measure their relative position. Modify the code and print out the positions of the robots relative to robot e-puck0 (*hint: you need to uncomment one line*). Run the code for a few seconds and stop the simulation and compare the printed relative positions with the absolute positions of the robots which you can see in the "translation" vector of each robot in the "scene tree" of the Webots.

I17 (8): Using the gathered relative positions of the robots (variable `relative_pos`) compute the average speed and position of the robot neighborhood (variables `rel_avg_speed` and `rel_avg_loc`). Based on `relative_pos`, `rel_avg_speed` and `rel_avg_loc`, redo the Reynolds' rules using relative range measurements. Focus on the "cohesion" and "dispersion" rules, since the "alignment" rule might fail due to quantization problems of the simulation (we will work with MIGRATORY_URGE = 1 to compensate this fact). Compile the controller and run the program. **Note that there is no supervisor in this project**. You should see four robots flocking similar to the previous sections. *Hint: base your implementation*

on the Reynolds' rules implemented in the previous sections and note that relative_pos = pos_neighbor – pos_self.

S18 (4): To compute the flocking performance add a supervisor to your project and set its controller to "*performance_estimation*". To add a supervisor click on the '+' bottom in Webots and select "New Node" and then "Supervisor". Compile the supervisor's controller and run the simulation. Compare the flocking performance values of this section with the one you got in Section 2.5 and also Section 2.1. Which approach provides the worst performance? Which one is the best? What are the reasons?

B19 (2): Place a few obstacles in front of the path of flock. Is the flock robust to the obstacles?

3 Robust Formation Control: Leader-Follower Controllers

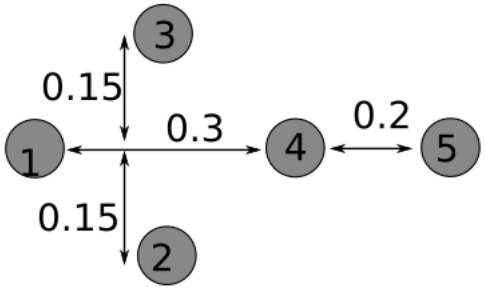
In this part, you will attempt to implement "leader-follower" formation movements using relative positions of robots. Open the *formation1.wbt* world and the *leader, follower* and *formation1_super* controllers (*formation1_super* is the controller for the world supervisor). The robots begin in a diamond formation with the leader at the head. Followers have the leader robot's relative range (distance between leader and follower) and bearing (relative angular offset of the leader's location in radians, i.e. 0 is straight ahead, $\pi/2$ is directly right, $-\pi/2$ is directly left) available to them.

S20 (3): Read the navigation algorithm for the follower robots and understand how the diamond formation is maintained, using the relative leader position and proximity sensors. It is assumed that the leader robot will never move backwards. Compile the *follower* and the *leader* controllers as well as the supervisor. Run the simulation a few times and see the behavior of the follower robots **while controlling the leader with your keyboard's arrow-keys**.

S21 (3): In the follower controller, there is a proportional controller to maintain the projected forward distance to the leader. Modify the value of the proportional coefficient of this controller, run and note the effect.

S22 (3): Test the robustness of this controller with the *leader_rand* controller for the leader; this controller applies random movements to the leader. In the *formation1_super* supervisor set *print_enable* to 1 (line 59). Compile and run for at least 30 minutes of simulated time. Use the error in position of each robot reported by *formation1_super* to determine how well the algorithm is working.

I23 (5): Consider the previous formation of four robots. Now we will add a fifth robot in the formation. The desired position of this robot in the formation is indicated in the figure below. Make the necessary modifications (in the world and in the code) in order to achieve the desired formation behavior.



4 Formation Control Using a Graph-Based Approach

Considering five holonomic agents, the goal is to design a graph-based control algorithm which leads them to the topological formation shown in Figure 2.

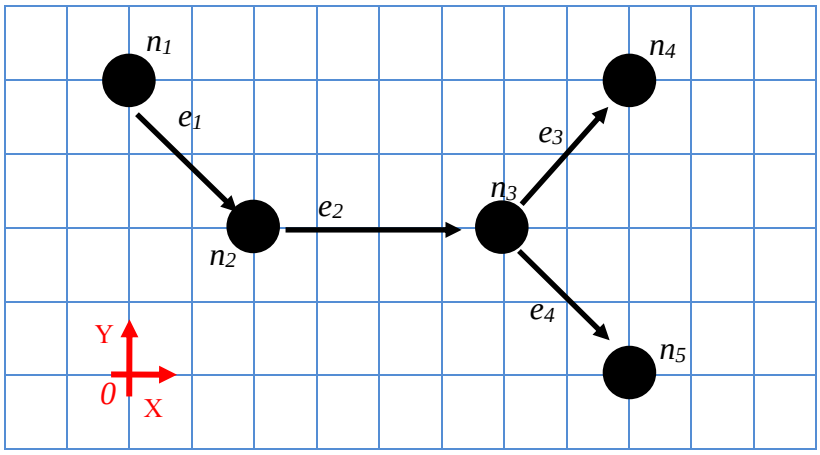


Figure 2. The desired topological graph of five agents.

Q24 (3): Write down the Laplacian matrix of the graph in Figure 2, assuming all weights equal to one.

Q25 (3): Write down the x- and y- biases of the agents in the graph relative to the origin of the coordinate system shown in the figure.

Q26 (3): Assuming that each node of this graph has a state defined by its Cartesian position (node n1 has (x1, y1); node n2 has (x2, y2); etc.), write a Laplacian control law that will accomplish the formation of Figure 2.

S27 (3): Open Matlab by typing "*matlab*" in a terminal. Open the script "*laplacian_formation.m*" in Matlab. Run the script. You will see five agents moving in the space converging to a formation. Verify that this formation is the one shown in Figure 2. How long does it take for the agents to get to their proper positions in the formation?

Q28 (2): Read the "*laplacian_formation.m*" script carefully and try to locate where the Laplacian matrix is computed. Locate where bias vectors are defined in the code and verify your answer to **Q25**.

Q29 (3): Try to find out how your differential equation in question **Q26** is implemented in discrete time. Write down the control law implemented in the code and compare it with your differential equation.

4.1 The impact of bias vectors on the formation

S30 (2): Set all the bias values to zero and run the code (*note*: the zero biases are already in the code but are commented out). You should observe that all agents converge to a rendezvous point. You may need to set the *Threshold* to a lower value to see the agents actually converge to one point.

I31 (5): Modify the bias vectors such that you achieve a diamond formation similar to the one in Part 3 of this laboratory. Observe trajectory of the agents. What is the impact of the bias vectors on the topology of the formation?

4.2 The impact of weights on the graph-based formation control

S32 (4): Change back the bias vectors to their initial values, and then set the weights of the edges all to 0.1. Run the script and observe the trajectory of the agents. How long does it take for the agents to reach to the desired formation? Repeat this experiment by setting all the edge weights to 5. What happens if you set them to even higher values? What do you conclude about the impact of edge weights on the graph-based formation control? *(Note: You may need to set the* Threshold *to a lower value when dealing with very low weights).*

4.3 Goal velocity + formation

So far the goal speed of the formation was set to zero. Now we want to have the group to get to the desired spatial formation as well as move with a constant speed.

I33 (4): Change back the edge weights to their initial values, and then set the goal speed (`VGx`, `VGy`) to (1,1). Run the script and observe the behavior of agents.

4.4 The impact of graph structure on the formation control

B34 (7): Add three more edges to the graph in Figure 2. Compute the "*Incidence*" matrix of the new graph. Modify the "I" matrix in the code and run the script. Is the achieved final formation topology different? Is the convergence faster than before? What do you conclude?