Lab 8: Positioning Systems and Odometry

This laboratory requires the following equipment:

- Webots
- C compiler
- MATLAB

The laboratory duration is approximately 3 hours. Although this laboratory is not graded, we encourage you to take your own personal notes as the exam might leverage results acquired during this laboratory session. For any questions, please contact us at sis-ta@groupes.epfl.ch.

1.1 Information

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

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

1.2 Getting Started

Download lab08.zip available on moodle in your personal directory. Now, extract the lab archive.

1.3 General remarks and documentation

In this lab, you will continue working with the simulated e-puck robot which we introduced in Lab 7. Here you will test different absolute localization methods, based on an emulation of a Global Navigation Satellite System (GNSS) first and then on odometry. Additionally, two different motion models based on different proprioceptive sensing modalities (accelerometer and wheel encoders) will be introduced for odometry. Note that we assume in this lab noise-free sensors and actuators (see Table 1). Even though the simulation considers the friction and contact forces of the wheels with the ground, meaning the wheels can slip under certain conditions, we will assume perfect rolling in general for this lab. The goal is to make you understand the underlying deterministic error sources in the odometry models and how to reduce them.

Table 1 : Sensors parameters used in simulation

Sensor	**Resolution**	**Noise**
GPS	Inf [m]	No
Accelerometer	Inf [m/s^2]	No
Wheel encoders	2π / 1000 [rad]	No

1.4 Getting started with Webots

- Launch Webots by entering the following command in a terminal:
  ```
  webots --mode=pause &
  ```
- From the menu, select *File->Open World*, and choose the *odometry.wbt* file from the *material/worlds* directory.
- From the menu tree (left side of the window), select *DEF ROBOT1 Robot > controller > controller*. Then click on `edit` to open the c file in the text editor.
- Click on the `clean` button and then on the `build` button.

2 Localization with positioning systems

GNSSs such as the Global Positioning System (GPS) and Motion Capture Systems (MCSs) are widely used in robotics for achieving absolute localization either outdoor (GNSSs) or indoor (MCSs). They provide precise and accurate ways to measure the actual position and orientation of a device in the three-dimensional space. The main limitation of these techniques is that they are not available for a variety of robotic scenarios (e.g., GNSS-denied environments).

2.1 Simulated GNSS and reference frame

1. **(Q)**: How many independent variables are required to properly define the pose, i.e. position and orientation, of a body (e.g., that of a wheeled robot) in a 2D world?

2. **(Q)**: What about a body (e.g., that of an aerial robot) in the 3D world?

3. **(Q)**: Based on your previous answers, propose a possible set of states to define the pose of your robot in a 2D world and in a 3D world.
   Hint: Try to remember your physics courses on gyroscope and Euler angles.

4. **(Q)**: The code we use in this lab might seem a bit longer and more complex than that you are used to. Therefore, in order to ease your task, we already provided the whole structure of the code. You will only modify the `main`() function as well as some `init()` and `get()` functions. Take some time to read the `main`() in *controller.c*. What are the five main steps in the while loop and can you describe a bit their goal? Read the file *odometry.h* and the lines 36 to 76 in *controller.c*. What do contain the structures `simulation_t`, `measurement_t` and `pose_t`?

5. **(I)**: We now want to configure the simulation in order to use the GPS sensor which can be added on the e-puck. To do so, take some time to read the documentation of the GPS sensor provided on this page https://cyberbotics.com/doc/reference/gps.
   As reminder, to use a sensor in Webots (only once the sensor is part of the robot model!), you need to update the controller code as follows: include the related library, perform the initialization, and read the sensor values. Moreover, for easing your next tasks, the two functions `controller_init_encoders()` and `controller_get_encoders()` should be understood and used as a starting point.
   Hint: Follow the list below to guide you through these steps.
   Hint: The documentation of `memset()` and `memcpy()` is available under these links :
   https://www.tutorialspoint.com/c_standard_library/c_function_memset.htm
   https://www.tutorialspoint.com/c_standard_library/c_function_memcpy.htm

Include library:	
- Add the GPS library.	(Line 10)
Initialization in controller_init_gps() :	
- Get the GPS device from Webots	(Line 441)
- Enable the GPS	(Line 447)
Sensors reading in controller_get_gps() :	
- Get the GPS measurements from Webots	(Line 186-191)

Hint: At this point, if you run the code, your robot should do nothing.

6. **(I)**: In the part 2 and 3 of this lab, the same robot controller will be used. Therefore, before running, we also need to set up both motors and accelerometer. However, we are only interested in the GPS during this first part. Add the library for the accelerometer and complete the following functions (similar to what was done in question 5):
   - `controller_init_acc()`
   - `controller_init_motors()`
   - `controller_get_acc()`

   Note: Look at the documentation of both the accelerometer and motors.
   Accelerometer: https://cyberbotics.com/doc/reference/accelerometer.
   Motors: https://cyberbotics.com/doc/reference/motor .

   Hint: When you compile and run your code and something went wrong, you should have an error message in the terminal that tells you the line where you have a problem.

7. **(I)**: At the top of the file `controller.c`, set VERBOSE_GPS to true, to print the GPS values in the terminal. Then, compile your robot controller.

8. **(S)**: Run the simulation. **Click on the window of the world** (this will let Webots capture your keyboard inputs)**,** then use your keyboard to move the robot in the arena. Corresponding action and keyboard keys are listed in Table 2.

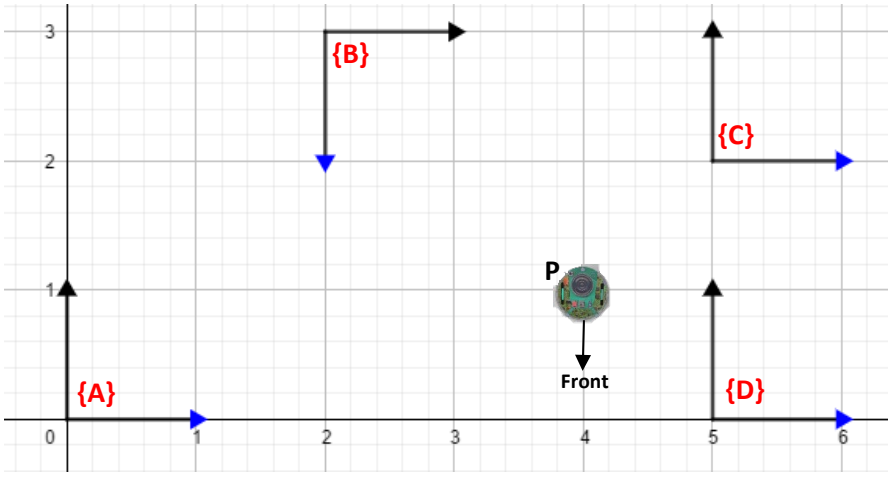
Table 2: Keyboard keys and corresponding actions

Keyboard key	Simulation actions
R	The robot starts to move
S	The robot stops
U	Increases the speed
D	Decreases the speed
Up Arrow	The robot moves forward
Down Arrow	The robot moves backward
Left Arrow	The robot turns left
Right Arrow	The robot turns right

Hint: You need to press R first for the robot to start moving.

9. **(Q)**: From the values obtained in the terminal, can you deduce in which direction point the x, y and z axes used by the GPS? Can you move the robot to the origin of the GPS coordinates (i.e. at least, to reach the point where two of the three values are zeros)?
   Hint: Look at line 194 to see the print command.

10. **(Q)**: Consider the following frames. The pose P of the e-puck expressed in {A} is $P_A = [4,1, -\pi/2]$. Which of the following statement is true?
    Hint: Assume the blue arrows represent the local x-axis. The heading is hence zero when the robot is aligned with this axis.



a) P_B = [2, -2, 0], P_C = [-1, 1, -π/2], P_D = [-1, 1, 3π/2]

b) P_B = [2, 2, 0], P_C = [-1, -1, -π/2], P_D = [-1, 1, -π/2]

c) P_B = [2, 2, π], P_C = [-1, -1, -π/2], P_D = [-1, 1, -π/2]

11. (**I**): The GPS does not provide any information about the orientation (heading) of the robot. One option is to use the delta position vector (difference between previous GPS values and actual ones) and to compute its angle.

- Modify the `controller_get_heading()` to compute the orientation of the robot.

Hint: Take care of using the right indexes of the GPS arrays. Assume that the z-axis is pointing up, x stays the same and use the right-hand rule to choose the sign of your y values. Use `atan2()` *to compute the angle.*

- Modify the `controller_get_pose()` to fill the pose structure with the GPS positions and heading. Use the reference frame detailed in the Figure 1 to fill the `pose_t` structure.

Note: The variable `_pose_origin` contains the position and orientation of the reference frame's origin A expressed in the world frame (same as GPS values) illustrated in *Figure 1*. Its fields are `x`, `y` and `heading`.

Set the VERBOSE_POSE to true and compile.

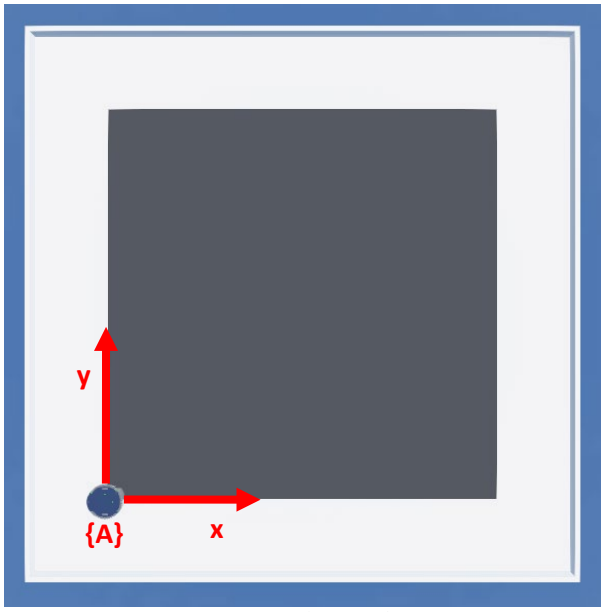


Figure 1 : Reference frame

12. **(S)**: Run the simulation by again moving the e-puck robot through the keyboard and try to follow the square. Does the pose behave as you expected? If not, try to correct your code.

13. **(Q)**: Is the heading angle still correct when you move your robot backward or rotate it on itself? Can you propose a sensor to directly get the orientation of the robot?

3 Odometry using an accelerometer

In this part, we are interested in investigating a low-cost, widely spread localization technique uniquely leveraging on-board proprioceptive sensing, namely odometry. The previous pose computed using the GPS is used as ground truth in the evaluation of the odometry. Here you will implement only a 1D odometry in the direction of the x-axis, according to the coordinate frame A.

14. (**S**): The controller leveraged in the previous part is again used here. Set VERBOSE_ACC to true and compile the robot's controller. Start the simulation on Webots and do not move the robot. You should see the values of the accelerometer in the terminal. Why those values are not zero, what these values mean (e.g., what force is involved)?

15. (**B**): We want now to focus on a thought experiment for a moment: imagine your e-puck is falling down (no friction with air), the accelerometer returns zero values for all the three axes. With regard to the previous question (static conditions), did you expect those values? What is an accelerometer really measuring (recall the schematic representation of accelerometer sensor mentioned in the lecture)?

16. (**S**): Uncomment, the lines (123 – 128 and 142) in `main()`. This will enforce the robot to stay static during five seconds. This time is used to estimate the bias of the accelerometer. The resulting mean values for the acceleration are stored in the table *_meas.acc_mean* in the file *controller.c*. Compile the robot controller and start the simulation. Let your robot move forward in the x-axis direction using the keyboard (press R). Stop it before it reaches the end of the arena (press S).
A file *data.csv* has been created inside the controller folder. This file contains the logs of the simulation. Run MATLAB code *plot_main.m* (only part A) to plot the values of the accelerometer. What is the frame of the accelerometer? Which indexes of the accelerometer array (*acc*) correspond to the x-, y- and z-axis of the frame A (see Figure 1)?

Let us define the following one-dimensional motion model (continuous time) in order to use the accelerometer data defined as a_x to estimate the position of our robot along the x-axis:

$$v_x = \int_0^T a_x dt$$

$$x = \int_0^T v_x dt$$

17. (**Q**): We need to discretize the model using the Euler Forward Method, where *T* is the sampling time of the simulation. Try to complete the following equations:

$$a(t) = \dot{v}(t) = \frac{v_{t+1} - v_t}{T}$$

$$v_{t+1} = \cdots$$

$$x_{t+1} = \cdots$$

18. (**S**): Look at the odometry equations leveraging the accelerometer implemented in the MATLAB function `odo_acc.m` and see if they match with the equations you defined in the previous question. Use the code *plot_main.m* (only part B), to plot the odometry. What happens if you remove the mean (bias) before the integration?

19. (**I**): Implement your equations for the odometry in Webots, within the function `odo_compute_acc()` of the file *odometry.c*. Run the simulation and let your robot run in a straight line. Analyze the resulting logs in MATLAB using *plot_main.m* (only part C). Do

you obtain similar results to those previously obtained in MATLAB? If not, try to correct your code or explain why.

4. Odometry using wheel encoders

In this part, we are interested in continuing our investigation with odometry. This time, we propose to take advantage of the type of locomotion of the e-puck, a differential wheeled vehicle, to design our 2D motion model. In fact, the simulated e-puck provides wheel encoders that return information on the actual position of its wheels. It is worth noticing that there is a main difference between the real e-puck and its simulated version. On the one hand, the real e-puck uses stepper motors, characterized by 1000 steps (increments) per wheel revolution; this type of motor is controlled in open-loop and provides information on the wheel position by itself. On the other hand, the simulated e-puck has one hinge joint (consider it as the rotating shaft) per wheel, to which a rotational motor and a position sensor are attached. The position sensors simulate wheel encoders, a sensing device type that in reality is combined with DC motors for closed-loop control; they are physically separated from the motors but anchored to the rotating shaft to obtain the same increment counting functionality.

20. (**S**): The same robot controller leveraged in Part 3 is used also for this part. Set VERBOSE_ENC to true and compile the robot's controller. Do a small run in the arena with the e-puck. What is the unit used by the wheel encoders and how to convert it in meters? Is it an absolute or a relative measurement of the wheel position?

21. (**Q**): We want to write the motion model (continuous time) for the odometry in the x-axis using the data provided by wheel encoders (consider Δ_{enc_r} and Δ_{enc_l} the difference between two encoder measurements from consecutive time steps for the right and left wheels respectively). To do this, the first step is to compute the forward and rotational speed in the (relative) coordinate frame B (body).

$$v_B = \cdots$$
$$\omega_B = \cdots$$

Hint: The wheel encoders are expressed in radiant, use the WHEEL_RADIUS to convert it into meters. The distance between the two wheels is stored in WHEEL_AXIS. The rotational speed is positive when the right speed is higher than the left one.

22. (**Q**): Compute the rotation matrix from the (relative) frame B to the (absolute) frame A. Try to complete the following equations.

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

$$R_B^A = \begin{bmatrix} \cdots & \cdots & 0 \\ \cdots & \cdots & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Hint: Use the orientation of the robot.

23. (**Q**): Using the previous equations, write the motion model (continuous time) of the robot in the frame A.

$$\dot{x} = \cdots$$
$$\dot{y} = \cdots$$
$$\dot{\theta} = \cdots$$

24. (**Q**): Based on the previous equations discretize your model using the Euler Forward Method, *T* is the sampling time of the simulation:

$$\dot{v}(t) = \frac{v_{t+1} - v_t}{T}$$
$$x_{t+1} = \cdots$$
$$y_{t+1} = \cdots$$
$$\theta_{t+1} = \cdots$$

25. (**S**): Do a complete run around the square with the e-puck. Then, run part D of *plot_main.m* to plot the odometry based on the wheel encoders (note that the odometry is already implemented for you in MATLAB).

26. (**S**): Look at the way the odometry is implemented in MATLAB in `odo_enc.m` and compare it to your answer to question 24. Try to reduce the deterministic errors by slightly changing the values of the *WHEEL_RADIUS* and *WHEEL_AXIS*. Your odometry curves should get closer to those obtained with the GPS.

27. (**I**): Implement your equations for the odometry in Webots in the function `odo_compute_enc()` in file *odometry.c*. Run the simulation and again follow the square. Compare the resulting logs on MATLAB using *plot_main.m* (only part E). As before, you can change the values of the *WHEEL_RADIUS* and *WHEEL_AXIS* to improve your odometry.

28. (**S**): Decrease the speed (press several times the D key) of the motors and let your robot carry out a complete tour of the square. Compare the logs on MATLAB code *plot_main.m* (only part E). What are your main observations?

29. (**S**): Increase the speed (press several times the U key) of the motors and let your robot carrying out complete tour of the square. Compare the logs on MATLAB code *plot_main.m* (only part E). What are your main observations?

30. (**Q**): Which strategies could we use to reduce the error in odometry in term of motor control? Think about the unmodeled dynamics of our motion models and when they occur.

31. (**B**): In the recommended reading material of week 9, the Chapter 5 of *Introduction to Autonomous Mobile Robot (2004, R. Siegwart* and *I. Nourbakhsh*) describes another motion model for differential wheeled robot. The equations (5.3 – 5.6) were implemented in MATLAB for you in `odo_enc_bonus.m`. You can run parts F and G of *plot_main.c* to test them. Try to reproduce them in Webots (you can create your own function in `odometry.c` to replace the default `odo_compute_encoders()` function). Does the odometry improve when compared to your previous implementation?