Lab 8: Positioning Systems and Odometry

This laboratory requires the following equipment:

- Webots 2023a
- C compiler
- MATLAB R2021b

The laboratory duration is approximately 3 hours. Although this laboratory is not graded, we encourage you to take your notes as the final exam might leverage results acquired during this laboratory session. For any questions, please get in touch with 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 code execution.
- 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.

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 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 the 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 *E-puck "e-puck" > 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. Please ignore "unused function/variable" warnings.

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 a device's actual position and orientation in the three-dimensional space. The main limitation of these techniques is that they are not available for various robotic scenarios (e.g., GNSS-denied environments).

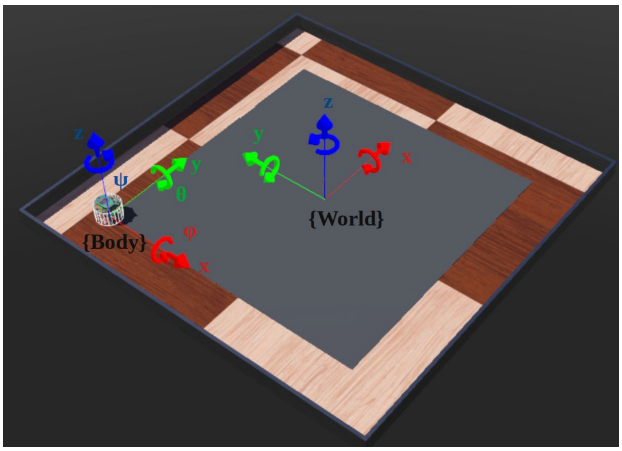

***Figure 1**. e-puck robot's body and world coordinate frames and pose variables*

Fig 1 demonstrates two important coordinate frames used in this lab: Body-fixed {Body} and World-fixed {World}. To properly define the pose of the robot (or body), we need six independent variables, i.e., the position and orientation of a body (with respect to the World frame) in a 3D world. There are six variables: three for positions x, y, and z, and three for the orientation φ, θ, ψ (roll, pitch, and yaw Euler angles). However, e-puck only moves in a 2D plane. As a result, we only need three of them, x,y, and ψ, for the planar pose definition.

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

2. **(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.

3. **(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. The 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 must press R first for the robot to start moving.

4. **(Q)**: From the values obtained in the terminal, can you deduce in which direction point the x, y, and z axes are used by the GPS with respect to {World} frame? 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 195 to see the print command.

As a preparation for the next exercise, consider the following frames. The pose *P* of the e-puck expressed in {E} is $P_E = [4,1, -\pi/2]$. Can you guess the pose P of the e-puck expressed in coordinate frames {B}, {C} and {D}? Think about it and then check the answer.

Hint: Assume the blue arrows represent the local x-axis. The heading is hence zero when the robot is aligned with this axis.

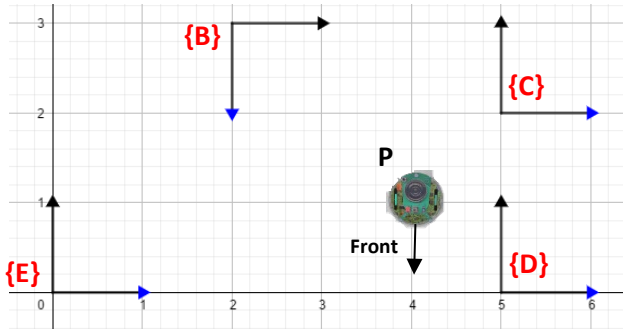

***Figure 2:** e-puck and various coordinate frames*

$P_B = [2,\ 2, 0]$, $P_C = [-1, -1, -\pi/2]$, $P_D = [-1, 1, -\pi/2]$

5. **(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 as can be seen in `controller_get_heading()`.

   Uncomment the indicated lines in the function `controller_get_pose()` to fill the pose structure with the GPS positions and heading. The pose should be written in `pose_t` structure with respect to the reference frame {A}, which coincides with the {Body} frame at the beginning of the motion, detailed in Figure 3.

   Note: The variable `_pose_origin` contains the position and orientation of the reference frame {A} expressed in the {World} coordinate frame illustrated in Figure 1. Its fields are `x`, `y`, and `heading`. z

   Set the VERBOSE_POSE to true and compile.

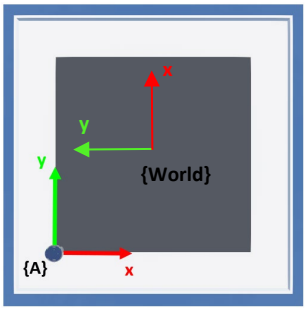

***Figure 3** : Reference Frames {A} and {World}*

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

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

3 Odometry using an accelerometer

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

8. (**S**): The controller leveraged in the last 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 are those values not zero? What do these values mean (e.g., what force is involved)?

9. (**B**): We want now to focus on a thought experiment for a moment: imagine your e-puck is falling (no friction with air), and the accelerometer returns zero values for all three axes. What quantity is an accelerometer measuring (recall the schematic representation of the accelerometer sensor mentioned in the lecture)?

10. (**S**): Uncomment lines (122 – 127 and 141) in `main()`. This will enforce the robot to stay static for 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 3)?

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

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

We need to discretize the model using the Euler Forward Method, where *T* is the sampling time of the simulation. As a result, we obtain the following equations:

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

$$v_{t+1} = v_t + T \cdot a_x$$

$$x_{t+1} = x_t + T \cdot v_t$$

11. (**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 don't remove the mean (bias) before the integration? Compare two scenarios.

12. (**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. The simulated e-puck provides wheel encoders that return information on the actual position of its wheels. It is worth noting that this is the 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.

13. (**S**): The same robot controller leveraged in Part 3 is also used 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 unit is used by the wheel encoders and how can it be converted into meters? Is it an absolute or a relative measurement of the wheel position?

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 {Body}.

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

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

Note: The wheel encoders are expressed in radiant, we 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.

We also need to compute the rotation matrix from the (relative) frame {Body} to the (absolute) frame {A} for the odometry calculation from encoders.

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

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

Using the previous equations, we can write the robot's motion model (continuous time) in the frame {A}.

$$\dot{x} = v_B \cdot \cos(\psi)$$

$$\dot{y} = v_B \cdot \sin(\psi)$$

$$\dot{\psi} = \omega_B$$

Let's discretize the continuous model using the Euler Forward Method, *T* is the sampling time of the simulation:

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

$$x_{t+1} = x_t + T \cdot v_B \cdot \cos(\psi)$$

$$y_{t+1} = y_t + T \cdot v_B \cdot \sin(\psi)$$

$$\psi_{t+1} = \psi_{t+1} + T \cdot \omega_B$$

14. (**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).

15. (**S**): Look at the way the odometry is implemented in MATLAB in `odo_enc.m`. 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.

16. (**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 in *odometry.c*.

17. (**B**): 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?

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

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

20. (**B**): In the recommended reading material of week 10, Chapter 5 of *Introduction to Autonomous Mobile Robot (2004, R. Siegwart* and *I. Nourbakhsh*) describes another motion model for a 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 reproducing them in Webots (you can create your function in `odometry.c` to replace the default `odo_compute_encoders()` function). Does the odometry improve when compared to your previous implementation?