1 Lab 3: Introduction to the e-puck Robot

This laboratory requires the following (the development tools are installed in GR B0 01 and GR C0 02 already):

- C development tools (gcc, make, etc.)
- C30 programming tools for the e-puck robot
- The "development tree" which is a set of files required by the C30 compiler
- One e-puck robot, one Bluetooth USB dongle, one battery
- Three paper "walls" and six stands
- The document "Introduction to the e-puck robot"

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. In case you would like to continue to work with hardware beyond the assisted exercise hours, talk to the TAs about returning the hardware later. 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.

1.1 Office hours

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

1.2 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 or homework exercises is 100.

.

1.3 General remarks and documentation

In this lab, you are working with the e-puck robot, a small mobile robot developed to perform desktop experiments. For more information about the e-puck, please read the document: "Introduction to the e-puck robot".

1.4 Controlling the e-puck robot

In this lab, we will control the e-puck robot by uploading a program. This is done by cross-compiling a program for the e-puck on your computer. Cross compilation is the process of compiling and building executable code for a platform other than the one on which the compiler is running. This can be useful for platforms upon which it is not feasible to do the compiling, such as microcontrollers that do not support an operating system. After creating the executable, you will then create a connection between your computer and the robot, and finally upload the program onto the robot's flash memory through the connection, using a wireless bootloader. The wireless bootloader code is already programmed on all the robots. The executable (a *.hex* file) runs directly on the e-puck. The procedure to do this will be explained in detail in the following paragraphs.

2 Lab: Getting familiar with the e-puck

2.1 Preparing software and hardware

Before starting the lab, we need to set up the experimental environment. Follow the instructions below to prepare hardware and software.

2.1.1 Preparing the software

You will need to download the e-puck development environment onto your computer. Copy these files in your home directory by doing the following:

a) First, create a directory for e-puck related files.
(example: `/home/user/myfiles`)

b) Now go to this directory and download (clone) the necessary files from the e-puck git repository by typing the following, in one line:

```
$ git clone https://disalgitn.epfl.ch/epuck/epuck.git
```

The download might take a few minutes.

c) Inside the *epuck/EpuckDevelopmentTree/* directory, you will find a *library/* and a *program/* directory. In order to use the e-puck library code that you just downloaded, it must be compiled. Do this by going to the *library/* directory and executing *make*:

```
$ cd epuck/EpuckDevelopmentTree/library
$ make clean
$ make
```

Your e-puck development environment is now ready.

2.1.2 Preparing the hardware

Prepare your hardware setup now:
1. Plug in the Bluetooth USB dongle into your desktop computer.
2. Place the battery in the robot with the serial number facing up (such that it is not visible when then battery is plugged in).
3. Switch the robot on.

4. Check your robot number written on a sticker at the front of the robot.
My robot number is : ……

In the remainder of this lab, we assume that your robot ID is 999. **Make sure to replace 999 with the number of your robot.**

2.2 Sensors and Odometry

Before doing the following exercises, you will need to extract the lab archive by executing **tar xvfz lab03.tar.gz** in a terminal.
We will start with some questions which should allow you to get a feeling of working with real hardware in general, and the e-puck in particular. To do this, you will upload the *sercom* program onto the e-puck, and run the *minicom* program on your computer. *Minicom* is a small terminal program to communicate with a device over a serial link. Since the e-puck Bluetooth protocol emulates a serial link, we can use this program to chat with the e-puck. Notice the small orange LED which indicates the Bluetooth connection being active.

Navigate to the directory lab03/ and upload *sercom.hex* onto the robot:

```
$ epuckupload -f sercom.hex 999
```

Note:Before executing the previous command, do not forget to get the proper access rights for the Bluetooth port in your terminal, by typing:

```
$ ssh localhost
```

And provide your user password.

As soon as dots appear on your screen (`$ .....`) press the blue reset button on the robot. This will start writing the program to the e-puck memory. Now stars will appear on your screen, indicating that the robot is being programmed (`$ .......**********`). When no more stars appear, the robot has finished programming. You will see the following message in your terminal:

```
[/dev/rfcomm999] Transmission of sercom.hex complete!
```

Now establish a communication channel with your e-puck and then start *minicom*.
The following line establishes a serial connection first (via `epuckconnect 999`) and then immediately launches *minicom* after the connection has been established:

```
$ epuckconnect 999 && minicom
```

Now press the reset button on the robot. Your terminal should display:

```
WELCOME to the SerCom protocol on e-Puck
the EPFL education robot type 'H' for help
```

With `Ctrl-A Z` (hold the `Ctrl` key while typing `A`, then release the `Ctrl` and `A` key and type `Z`) you get to the main menu of *minicom*. Configure your terminal to

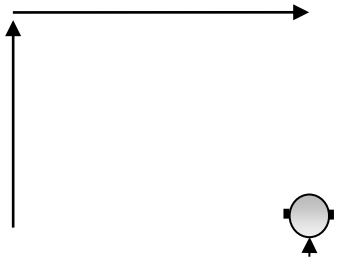
have a local echo, i.e. to display the characters you type, by typing `E <Enter>` in your minicom terminal.

S1 : Enter `H <Enter>` in your minicom terminal. Your robot will answer with all the commands it understands through the serial line.

Q2 (5): Type `N <Enter>` in your minicom terminal. Then put a finger in front of one of the IR sensors and type `N <Enter>` again. What are you measuring? When was the value higher?

Q3 (5): Put different objects (white sheet of paper, dark objects, etc.) at the same distance in front of the sensor and take a measurement for each. Explain why the values are different, even though the distance is the same.

Q4 (5): Although not endowed with optical encoders, the e-puck wheels are driven using stepper motors, which allow for adjusting the precise wheel position. A stepper motor (or step motor) is a brushless DC electric motor that divides a full rotation into a number of equal steps. The motor's position can then be commanded to move and hold at one of these steps without any feedback sensor.
Check the possible commands list. You can set the wheel speed using the d,x,x command, which sets the value by which an internal step counter increases per second.
Check the possible commands list. You can set the step counter for both motors to 0 using the command p,0,0.
Give a sequence of commands which you have to enter at the right time such that the robot approximately moves along the following line.



Note: the small red LED close to the on/off switch signals the battery level. In case it turns on when the wheels are rotating, this will indicate that your battery level is low and the robot might show unpredicted behaviors. Please change the battery.

S5: Stop the motors and execute the following commands

```
p,0,0
d,200,-200
```

While the e-puck is moving, press `Q <Enter>` from time to time.

Q6 (5): What do the values returned by Q mean? By how much does Q increase per second?

Q7 (5): The circumference of the e-puck wheel is about 13 cm. The stepper motor travels 1000 steps per wheel rotation. The distance between the two wheels is 53 mm. Compute the proper command for wheel speed such that the center of your e-puck travels on a full circle of 15.9 cm radius within 25.62 seconds.

S8: Execute the computed command.

Q9 (5): Try to increase the wheel speeds such that your e-puck travels the same circle faster.

S10: Execute the computed commands and observe what happens when you try to travel too fast and try to explain it.

You can now quit *minicom* by typing `Ctrl-A Q`.

2.3 Obstacle Avoidance

In the previous lab, we introduced the Braitenberg robot controller. In this section, you will run the Braitenberg obstacle avoidance algorithm on the e-puck itself.

S11: Within your lab03 directory, type the following commands to program your robot with the Braitenberg obstacle avoidance program:

```
cd obstacleavoidance/
epuckupload -f obstacleavoidance.hex 999
```

Do not forget to press the blue reset button on the e-puck as soon as the dots appear on the screen (stars indicate that program is being uploaded). Now, set up the arena, put the robot inside and observe it doing obstacle avoidance.

S12: Do the same for *obstacleavoidance_log*. This code differs in three ways from the previous algorithm:

- The output of the infrared sensors is passed through a log function. Since the infrared sensors output decreases exponentially with the distance, this makes the modified sensors output a linear function of the distance. The input values to the Braitenberg would then be a linear function of the distance to the obstacle.
- An average over 10 samples is computed in order to reduce measurement noise.
- The sensors are calibrated by taking a series of measurements at the beginning. Therefore, make sure that there is no obstacle around when you switch on (or reset) the e-puck.

Q13(5): What differences in obstacle avoidance behavior do you observe between the two algorithms? Consider in particular the smoothness of the behavior.

Q14(5): **Important: in the Makefile, make sure to write the correct path to your e-puck library. You can get the correct path by checking the "properties" of the desired folder.** Now open *obstacleavoidance_log/main.c* and modify the algorithm such that it averages over 1000 instead of 10 samples. Recompile

obstacle-avoidance_log.hex by typing `make clean` and then `make` in that folder and upload it. Does this improve the robot's behavior? Explain what happens.

I15(10): There are also other ways of programming an obstacle avoidance controller. Write a simple rule-based, i.e. if "condition_k" satisfied, then do "action_k", obstacle avoidance algorithm, based on the skeleton provided in *rulebased/main.c.* Once you have finished programming, compile and upload your program, and test it on the robot.

Q16(10): In your opinion, what are the advantages and disadvantages of the Braitenberg controller with respect to the rule-based controller? If you have time, test both your obstacle avoidance algorithms in a U-obstacle such as the one you saw in the previous lab. You can build a U-obstacle using extra white walls.

The Braitenberg principle can also be used to program an object following algorithm: instead of avoiding obstacles, the e-puck is supposed to follow an object (e.g., your hand) in front of it.

B17(10): Find a simple way of modifying the Braitenberg obstacle avoidance algorithm to produce object following behavior. We prepared the folder *objectfollowing* for you with a copy of the *obstacleavoidance* program.
Make the necessary modifications in *objectfollowing*, compile (`make clean; make`), upload (`epuckupload -f objectfollowing.hex 999`) and test your algorithm.

1.1 Dead-reckoning navigation

This exercise will demonstrate dead-reckoning on the e-puck and the effects of bad calibration.

Q18(5): Open the file *drnavigation/main.c* and try to understand how the code works. There are three functions which are relevant to the dead-reckoning navigation. `e_set_dr_position()` sets the position at which the robot thinks it is. By calling `e_set_dr_position(0,0,0)` the robot thinks it is at the origin of a local coordinate system pointing into the direction of the y-axis. `e_do_dr()` reads the number of steps each motor took since the last call of `e_do_dr()` and computes the new dead-reckoned position based on the previous position (determined during the last call of `e_do_dr()`) and the motor steps. `e_get_dr_position(&dr_x,&dr_y,&dr_theta)` retrieves the actual dead-reckoned position. First the motor speed is set to 300 for both motors so that the e-puck goes forward. Then the dead-reckoned position is continuously updated `e_do_dr()` and read until the robot has traveled the desired distance. Similarly the robot turns 180 degrees, travels the same distance in the opposite direction and turns another 180 degrees. At the end of the program's execution the robot should be in the starting position and orientation (="pose").

S19(5): Make sure the robot has enough space to travel 20 cm or more forward. Using a pencil mark the starting position on the table. Compile the program, upload and execute it.

S20(10):Notice how the robot does not really return very well to the starting position. This is because the *distance between the wheels* in centimeters and *steps traveled per wheel rotation* are not properly calibrated. *Hint:* you can compare the distance which the robot traveled by comparing it to the short side of an A4 paper which is about 20 cm.

Open the file *EpuckDevelopmentTree/library/motor_led/e_motors.h*. In line 56 and 57 you will find the two `#define` statements

```
#define STEPS_PER_M 4443
#define WHEEL_DISTANCE 0.060
```

where the two relevant values are defined. Looking at how the e-puck went compared to how it should have and looking at the effect of the wheel distance on the dead-reckoning computation (lecture notes), which of the following set of `#define` statements probably leads to the best result? Try to solve it by analyzing the dependencies between the `STEPS_PER_M` and `WHEEL_DISTANCE` value and the dead-reckoned distance traveled and angle turned. Verify it by modifying the code accordingly, recompiling and running the robot. Remember that every time you modify something in *EpuckDevelopmentTree/library/motor_led/e_motors.h* you need to rebuild the library by going into *EpuckDevelopmentTree/library/* and execute *make clean* and then *make*. Do this **before** recompile *drnavigation/main.c*.

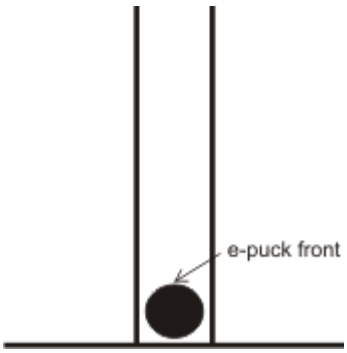
a.)
```
#define STEPS_PER_M 10237
#define WHEEL_DISTANCE 0.053
```
b.)
```
#define STEPS_PER_M 7767
#define WHEEL_DISTANCE 0.053
```
c.)
```
#define STEPS_PER_M 7767
#define WHEEL_DISTANCE 0.060
```

1.2 Feature-based navigation

When dead reckoning is properly calibrated, the e-puck is able to return to its initial position really well; however, after running for a while, it will eventually drift quite a bit, due to accumulation of unavoidable small errors. One way to make sure that we get back to the right position every time is by using very basic featured-based navigation.

The following experiment will demonstrate dead-reckoning with and without correction through feature observation. In order to better observe the effect of the drift we put the robot in a corridor and let it use its infrared sensors to not touch the walls. This way the robot will not drift in the cross-corridor direction and we can better observe the drift in the along-corridor direction.

S21(5): Set up the e-puck and three walls as shown in the figure below. The e-puck should have about 5 mm distance to each wall with the front of the robot (where the camera is) pointing to the exit of the corridor.



First you need to make sure that your dead-reckoning is properly calibrated. Open the file *EpuckDevelopmentTree/library/motor_led/e_motors.h* and set line 56 and 57 to

```
#define STEPS_PER_M 7767
#define WHEEL_DISTANCE 0.053
```

Make sure you recompile the library by executing `make` and then `make clean` in *EpuckDevelopmentTree/library/*. Compile and upload the file *featurenavigation/main.c*. This code will make the e-puck go forward for 20 cm then turn 180 degrees go forward 20 cm again and turn another 180 degrees. After uploading the program, push the RESET button. This will make the program start. At the beginning of the program there is a 5 second pause before the robot starts. This will give you enough time to properly place the robot in the corridor.

After the robot starts it will execute the sequence (go 20 cm→ turn 180° → go 20 cm→ turn 180°) infinitely many times, but after a while, enough dead reckoning error will have accumulated such that the robot will start pushing the back wall. Let the robot go back and forth a few times and observe the drift.

I22(5): Now we want to use the back wall as a feature each time we are next to it. We will do this by developing a function that measures the distance to the wall (i.e., the feature) and adjust the robot's position such that it is the same every time the robot finishes going back and forth. Open the file *featurenavigation/main.c* and look for the already defined function `align`. First, remove the comments ("//") where this function is called in the `main` function. Then, fill the contents of function `align`, so that it does the described behavior. Recompile, re-upload, and set the robot in the corridor again, just as in the previous figure.

Hint: Note, that function `align` *accepts an argument* `(ref_point)` *which is the sensor value (or the distance) that the robot measures to the wall when it is initially placed in the corridor.*
Hint: The sensor measurement after the robot having executed the back and forth behavior is already implemented in the function.