1 Lab+Hwk 4: Introduction to the Miniature E-puck Robot

This laboratory requires the following equipment:

- C30 programming tools for the e-puck robot (installed in BC 07/08)
- The "development tree" which is a set of files required by the C30 compiler
- Webots simulation software
- Webots User Guide and Reference Manual
- One e-puck robot, one Bluetooth USB dongle, and a spare battery
- The document "Introduction to the e-puck robot"
- The document "Tutorial for programming the e-puck robot in Linux"

The laboratory duration is about 3 hours. The report for this laboratory and homework will be due on the first Monday after **your lab session, at 12 noon**. Please format your solution as a **PDF file** with the name **[name]_lab[#].pdf**, where **[name]** is your account user name and **[#]** is the number of the lab+homework assignment, and upload it as the Report submission onto Moodle. If you have additional material (movies, code, and any other relevant files), upload a **[name]_lab[#].tar.gz** archive instead with all files, including the report (hint: `tar cvzf [name]_lab[#].tar.gz [directory or files]`).

Keep in mind that no late solutions will be accepted unless supported by health reasons (and a note from the doctor). If there are extreme circumstances which will prevent you from attending a particular lab session, it may be possible to make arrangements with the TAs ***before*** the lab in question takes place. The more advanced notice you can give in these situations, the more likely it is that we will be able to work something out.

1.1 Grading

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; **you are not required to write or produce anything for these questions**.
- The notation $\mathbf{Q_x}$ means that the question can be answered theoretically, without any simulation; **your answers to these questions must be submitted in your Report**. The length of answers should be approximately **two sentences** unless otherwise noted.
- The notation $\mathbf{I_x}$ means that the problem has to be solved by implementing a piece of code and performing a simulation; **the code written for these questions must be submitted in your Additional Material**.

Please use this notation in your answer file!

The number of points for each exercise is given between parentheses. The combined total number of points for the laboratory and homework exercises is 100.

1.2 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 following two documents:

- Introduction to the e-puck robot
- Tutorial for programming the e-puck robot in Linux

We strongly recommend you to **read these documents before coming to the lab session**, so that you can right away start working with the e-puck.

1.3 Controlling the e-puck robot

The e-puck robot can be controlled in two different ways:

- **Remote control:** Your computer runs a program and sends commands via Bluetooth (serial link) to your e-puck. The *sercom* program on the e-puck interprets and executes these commands.

- **Uploading a program:** You compile a program for the e-puck and upload it on its FLASH memory. The program runs directly on the e-puck. Hence, the e-puck only needs to be connected while you upload the program.

In the first part of the lab, you will remotely control the e-puck using Webots. In the second part, you'll write and upload your own programs.

2 Lab: Getting familiar with the e-puck

2.1 Preparing software and hardware

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

2.1.1 Downloading the software

First at all, get all the files needed for the lab in your home directory by checking out the following SVN (subversion.tigris.org) repository:

```
cd <your swarm intelligence directory>
svn checkout http://lanospc47.epfl.ch/svn/students/Lab4
svn checkout http://lanospc47.epfl.ch/svn/students/Epuck
```

Then, we need to add some paths to your PATH variable. Edit the file "Epuck/Tools/setupenvironment" and execute it by typing

```
Epuck/Tools/setupenvironment
```

This opens a new bash shell with the required PATH variables set correctly.

2.1.2 Preparing the hardware

Prepare your hardware setup now:

1. Plug in the Bluetooth USB dongle to your desktop computer.
2. Place the battery in the robot.
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 you robot ID is 999. Make sure to replace 999 with the number of your robot.

2.1.3 Uploading the *sercom* program on the e-puck

Questions 1-18 require the *sercom* program to be running on the e-puck. The e-pucks we provide you should contain this program already. If you programmed your e-puck with another program (questions 19-25), however, you can always revert to the *sercom* program by typing:

```
cd Epuck/EpuckDevelopmentTree/program/advance_sercom/
epuckupload -f sercom.hex 999
```

Press the blue reset button on the e-puck as soon as the dots appear. More information can be found in the "Tutorial for programming the e-puck robot in linux".

2.2 Warm-up

We'll start with some simple warm-up questions which should allow you to get a feeling of working with real hardware in general, and the e-puck in particular. You should spend about 30 minutes on this part. Recall that we expect **short intuitive answers without graphics** (roughly 2 sentences), unless otherwise noted.

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. Launch *minicom* by entering

```
minicom epuck999
```

Press the (blue) 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` key and type `Z`) you get to the main menu of *minicom*. Configure your terminal to have a local echo (type `E`).

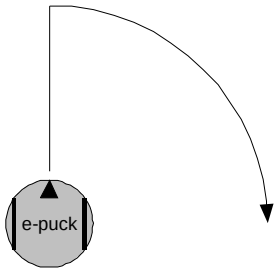
S: Enter `H <Enter>` in your minicom terminal. Your robot will answer with all the commands it understands through the serial line.

Q_1 (1): 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?

Q_2 (1): Put different objects (sheet of paper, pencil, hand, …) at the same distance in front of the sensor and take a measurement each. Explain with **one** sentence why the values are different, even though the distance is the same?

S: Experiment with the motor speed command (D). Try values between -1000 and 1000 for both wheels. Note that the real (physical) speed of the wheel is linear with respect to this value, and 1000 is the maximum speed.

Q_3 (2): Give a sequence of commands such that the robot would move along the following line! (Don't worry about exact angles and distances and timings – only concepts matter.)



Q_4 (2): Let your e-puck cross a sheet of A4 paper (21 x 29.7 cm) at speed 200 and count the seconds. Calculate the real speed (approximatively)! What speed would you set to both wheels to have the e-puck go backward at 5 cm/s?

S: 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.

Q_5 (1): Can you guess what P does? What do the values returned by Q mean? By how much does Q increase per second?

Apple's iPhone can detect it's orientation by means of an accelerometer. The e-puck has such an accelerometer as well, allowing to measure acceleration (and therefore forces) in all three dimensions. For each dimension, you get a value between 0 and 4096. If no force is present along an axis, the corresponding value is roughly 2000 – 2300 (the exact center value depends on the e-puck).

Q_6 (2): Put the e-puck on your desk and press `A <Enter>`. Then turn your e-puck on one side and measure the acceleration again. Why is one value out of the 2000 – 2300 range? Explain with **one** sentence how the iPhone is able to detect its orientation?

Q_7 (1): What would you measure if the e-puck was in free fall?

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

2.3 Remote Control via Webots

In the lab last week, you implemented a Braitenberg obstacle avoidance algorithm in Webots. This week, we will test that algorithm on a real e-puck. We already implemented this algorithm for you and assume that you know how it works.

In this section, we will use Webots to remotely control the e-puck robot. The Braitenberg algorithm runs in Webots (on your PC) and communicates in real-time with your e-puck over Bluetooth. It receives the proximity sensor values (via the "N" command that you tested before) and sets the motor speed (using the "D" command).

S: Open Webots and load the world *obstacleavoidance.wbt* from the folder *Lab4/webots/worlds*. Run the simulation and observe the e-puck. In

particular, have a look at how the e-puck moves straight in open space, i.e. if there is no obstacle in proximity.

S: Run the same controller in remote control mode now:
1. Stop and revert your simulation.
2. Double-click on the e-puck. A window with a model of the e-puck appears.
3. Select the rfcomm device corresponding to your e-puck.
4. Make sure your e-puck is switched on and select *remote control* instead of *simulation*. Webots now connects to the e-puck, which may take a couple of seconds.
5. Put the e-puck in your white arena and press the *Fast Forward* button (not the *Start* button) in Webots. Your e-puck will start moving.

Observe how your real e-puck moves if there are no obstacles around. Make this experiment several times with different initial directions and different starting points.

Q_8(10): How did the real e-puck move as compared to the simulated e-puck? What are two main issues that could cause this behavior?

2.4 Programming the E-Puck Robot

In the previous section, you controlled your e-puck robots remotely via Webots. In this section, you will run the Braitenberg obstacle avoidance algorithm on the e-puck itself. Again, we implemented the algorithm for you.

S: Type the following commands to program your robot with the Braitenberg obstacle avoidance program:

```
cd Lab4/obstacleavoidance/
epuckupload -f obstacleavoidance.hex 999
```

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. Once the upload is finished, put the e-puck in the white arena and observe it doing obstacle avoidance.

S: 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 linearizes the Braitenberg inputs.
- 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.

Q_9(2): What differences in obstacle avoidance behavior do you observe between the two algorithms?

Q_{10}(5): 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` in that folder and upload it. Does this improve the robot's behavior? Explain what happens.

S: Upload the *speed_oscillation* program on your e-puck now. The program makes the robot go straight at full speed (1000 steps/s) during half of the time, and stop during the other half. As a result, the average forward speed is about half the full speed.
In the file *main.c*, you will find a command *wait(200000);* in the main loop. This introduces a delay at this point and defines how long the e-puck keeps the same speed. A wait value of 200'000 corresponds to roughly 140 ms.

Q_{11}(3): Try wait values of 20'000, 2'000 and 200. Does the robot continue to move at half the full speed on average? (Describe your observations here. There is a more detailed question about this in the homework.)

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.

I_{12}(5): A simple object following algorithm can be obtained by inverting the Braitenberg weights of an obstacle avoidance algorithm. We therefore prepared the folder *objectfollowing* for you with a copy of the *obstacleavoidance* program.
Make the necessary modifications in *objectfollowing*, compile (`make`), upload (`epuckupload -f objectfollowing.hex 999`) and test your algorithm.

I_{13}(10): You probably noticed that the previous algorithm – despite following the object pretty well – hits (and even pushes) the object to follow. Implement a Braitenberg algorithm in *objectfollowing_nopush*, which still follows an object, but stops a few millimeters in front of the object (and stays there until the object is moved).

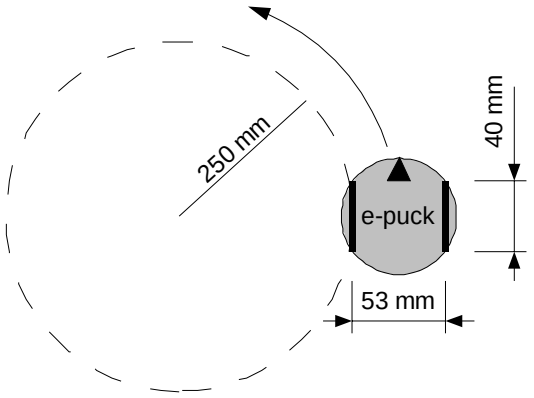
Q_{14}(5): Explain (in a few words) how you implemented *objectfollowing_nopush*.

3 Homework

Q_{15}(3): The basic obstacle avoidance algorithm used in Q_{10} has an update rate of roughly 12 Hz and a mean speed of 2.6 cm/s. Assume you want to run this same algorithm to remotely control an e-puck in Singapore. The delay introduced by the network is 180 ms (one way). What mean speed would you have to choose such that the obstacle avoidance quality remains the same?

Q_{16}(6): Would you use a remote control solution for the following applications? Motivate!
(a) Chess-playing miniature robot with a camera (to detect the current situation on the chess board) and a gripper (to move the men)
(b) Safety stop mechanism using bumper sensors on a mobile robot

Q17(8): Assume for this question that the wheel radius of the e-puck is 20 mm, the distance between the wheels (axis length) is 53 mm and turning one wheel by 360° corresponds to 1000 motor increments. What speed (motor commands) do you need to specify so that to robot moves once around a circle of radius 25 cm (radius of the trajectory of the inner wheel) within 20 seconds?



Q18(2): If you initialized the motor positions to 0, 0 (`P,0,0`), what would they be after one round along the circle?

Q19(8): For the same speed, calculate the trajectory, $\xi_I(t)$, using the forward kinematic model, but imagine the radius of the outer wheel was 0.07 mm (diameter of a human hair) bigger! If the robot starts at *x(0)=0*, *y(0)=0* and *θ(0)=0*, calculate its position (x, y) and heading (θ) after 20 seconds!

Q20(8): sin(), cos() and floating point operations are very time-consuming on microcontrollers. How could you speed up the following code:

```
float amplitude_at_freq6(int sample[]) {
     int i;
     float x=0., y=0.;

     for (i=0; i<512; i++) {
          x+=cos((float)i/6*2*PI) * (float)(sample[i]);
          y+=sin((float)i/6*2*PI) * (float)(sample[i]);
     }

     return sqrt(x**2+y**2);
}
```

(Note that the microcontroller on the e-puck would allow us to further optimize such code using DSP operations. This, however, requires assembly code and is not what we want you to do here.)

Q21(5): The following code implements the *speed_oscillation* program in Webots. What happens if you modify the wait value here? (Or: what mistake did the programmer make when porting the program from the e-puck to Weboots?)

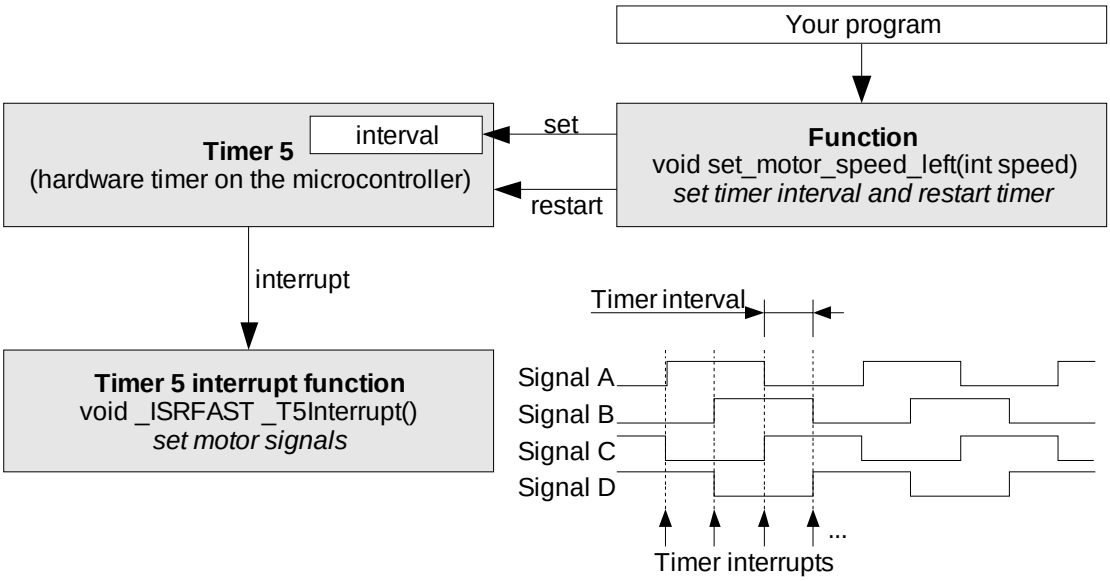
```
int speedZero=0;

void wait(unsigned long num) {
```

```
        while (num > 0) {num--;}
}

static int run(int ms) {
        if (speedZero) {
                differential_wheels_set_speed(0, 0);
                speedZero=0;
        } else {
                differential_wheels_set_speed(1000, 1000);
                speedZero=1;
        }
        wait(200000);
        return TIME_STEP;
}
```

Q22(5): The following sketch describes how the e-puck library generates the motor signals for the left motor. (The right motor works exactly the same way.)



Each time the motor signals change, the (stepper) motor moves one step. What happens if you call set_motor_speed_left extremely often, such as in **Q11** with a low wait value?

Q23(5): What would happen (physically) if you set a too high speed (e.g. 1500)?

4 Submission

In addition to your report, you need to submit your object following algorithms *objectfollowing* and *objectfollowing_nopush*. Assuming that you have your report stored as *[username]_lab4.pdf* in the *Lab4* folder, you can create a tar.gz file as follows:

```
cd Lab4
tar cfz [username]_lab4.tar.gz [username]_lab4.pdf
            objectfollow objectfollow_nopush (on the same line)
```

Don't forget to fill out the feedback form!

5 The forward kinematic model: Example

Exercise:
Assume your robot is moving forward at speed 2 cm/s, i.e., each wheel moves at this speed. Furthermore, let's assume your robot starts at position *x(0) = 10 cm*, *y(0) = 5 cm* and $\theta(0) = 30^o$. Calculate the trajectory, $\xi_I(t)$, of the robot.

Solution:
Using the equation for a differential-drive robot (see lecture notes), we can write

$$\dot{\xi}_I = R^{-1}(\theta)\dot{\xi}_R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{r\dot{\varphi}_1}{2} + \frac{r\dot{\varphi}_2}{2} \\ 0 \\ \frac{r\dot{\varphi}_1}{2l} + \frac{-r\dot{\varphi}_2}{2l} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2cm/s \\ 0 \\ 0 \end{bmatrix}$$

In this simple example, the rotation speed is zero, since the robot is moving on a straight line. Hence, the third value of $\dot{\xi}_R$ (robot movement in the coordinate system of the robot) is equal to zero. Carrying out this multiplication yields

$$\dot{\xi}_I(t) = \begin{bmatrix} \dot{x}(t) \\ \dot{y}(t) \\ \dot{\theta}(t) \end{bmatrix} = \begin{bmatrix} \cos(\theta(t)) \cdot 2cm/s \\ \sin(\theta(t)) \cdot 2cm/s \\ 0 \end{bmatrix}$$

which describes the robot's movement in world coordinates. To obtain the trajectory, ξ_I, we only need to integrate $\dot{\xi}_I$ with respect to time. Since we need *θ(t)* for *x* and *y*, we first have to integrate the third line:

$$\theta(t) = \int \dot{\theta}(t)dt = \int 0dt = 0 + c_\theta = \theta(0)$$

Now, we have got everything to integrate the first two lines as well:

$$x(t) = \int \dot{x}(t)dt = \int 2cm/s \cdot \cos(\theta(0))dt = 2cm/s \cdot \cos(\theta(0)) \cdot t + c_x$$
$$y(t) = \int \dot{y}(t)dt = \int 2cm/s \cdot \sin(\theta(0))dt = -2cm/s \cdot \sin(\theta(0)) \cdot t + c_y$$

After calculating the integration constants c_x and c_y and plugging in all given values, the robot trajectory can be written as

$$\xi_I(t) = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix} = \begin{bmatrix} 2cm/s \cdot \cos(30^o) \cdot t + 10cm \\ 2cm/s \cdot \sin(30^o) \cdot t + 5cm \\ 30^o \end{bmatrix}$$