Lab 9: Navigation and digital filtering on the e-puck Robot

This laboratory requires the following equipment:

- 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 (ev. spare battery)
- The document "Introduction to the e-puck robot"

The laboratory duration is approximately 3 hours. Although this laboratory is not graded, we encourage you to take your own personal notes as the final 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 this assignment, you will find several exercises and questions.

- The notation $\mathbf{Q_x}$ means that the question can be answered theoretically or with simple commands in the Linux operating system, without implementing or running any code.
- The notation $\mathbf{S_x}$ means that the question can be solved only by compiling and running a piece of code or an additional simulation.
- The notation $\mathbf{I_x}$ means that the problem has to be solved by implementing, possibly compiling, and running a piece of code.
- The notation $\mathbf{B_x}$ means that the question is optional (bonus) and should be answered if you have enough time at your disposal.

1.2 Getting Started (Short reminder)

To start with this lab, you will need to download the material available on Moodle. Download *lab09.tar.gz* in your personal directory. Now, extract the lab archive (you can type: `tar xvfz lab09.tar.gz`.)

1.3 General remarks and documentation

In this lab, you will continue working with the e-puck robot which we introduced in Lab 8. Here you will understand the importance of sensor calibration to obtain an accurate interpretation of the data, and how noise on these data can be eliminated using the signal processing techniques that you learned in previous sessions. The first part demonstrates how the e-puck can navigate using dead reckoning. The second part shows how sensors can be used to control the LEDs. The last part demonstrates how digital filtering can be used to process sensor signals.

1.4 Compiling and uploading a program for 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, then creating a connection between your computer and the robot, and finally uploading the program

onto the robot's FLASH memory. 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 E-puck motion control and digital filtering

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. If you did not delete anything after the previous lab you can skip this step.

2.1.1 Preparing the software

In case you did not download the e-puck development environment the last time, you need to do this now as described in this paragraph. Otherwise you can proceed to the next step.

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 (on one line):

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

The download might take a few minutes.

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

```
> cd EpuckDevelopmentTree/library
> 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 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: ……

2.2 Navigation with odometry

Recall Lab 8 which introduced the e-puck motors and how to use them. This exercise will do similar trajectories as before, but using dead reckoning for navigation.

Q_1: A differential robot (e.g., an e-puck) starts at t_0 from position $\vec{x}_0 = [x_0 \quad y_0 \quad \theta_0]^T = [0 \quad 0 \quad 0]^T$ (i.e., robot is located at the origin, pointing towards x-axis) in a very narrow straight corridor. The data sheet of the encoder indicates that it has 1000 increments per revolution, the wheels have a diameter of d = 4 cm and the distance between the wheels is b = 5.3 cm. Assume no wheel slip. After some time going **straight, with forward motion and constant velocity** (so only positive pulses on the odometry encoders), the processor registered 9455 pulses from both wheel encoders. What is the pose (i.e. position and orientation) of the robot at this time? Note: the e-puck does not have a motor with an encoder, but instead has a stepper motor which turns a certain amount of steps per revolution. For the purpose of this question a stepper motor and a motor with encoder are equivalent.

S_2: For this simple demonstration the robot has to move 20 cm forward and then perform a rotation of 180 degrees on the spot. The code is in *drnavigation/main.c*. Open the file and try to understand how the code works. The position of the robot is updated, in the function *update_odometry*, from the number of steps performed by each wheel since the last update. This requires an intermediate step, which is converting the number of steps performed by motors to the actual distance made by the wheels, which is done in the function *step_to_distance*.

Before compiling the program, go to the directory and change the `EPUCKLIBROOT` variable in the Makefile to the `EpuckDevelopementTree/library` directory of your installation:

for example: `EPUCKLIBROOT=~/Desktop/MyFiles/epuck/EpuckDevelopmentTree/library`)

Now you can compile your code by typing *make*.

Then upload and execute the code. To do this, do

```
> ssh localhost
```

Then navigate to the directory which contains *drnavigation.hex* and upload it onto the robot (replace ### with your 3 digit robot number):

```
> epuckupload -f drnavigation.hex ###
```

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 (where ### is your 3 digit robot number):

```
[/dev/rfcomm###] Transmission of drnavigation.hex
complete!
```

S3: Now, open a connection to the e-puck by typing (where ### is your 3 digit robot number):

```
> epuckconnect ### && cat /dev/rfcomm###
```

in a terminal. Make sure the robot has enough space to travel 20 cm or more forward. Using a pencil, mark the starting position on the table. Reset the e-puck and note how it performs the movements. For each maneuver (forward and simple rotation), note the distances traveled by each wheel displayed on the terminal. Use the equations from *update_odometry* to find the respective (x,y,theta) displacements. Do they correspond to what the robot is supposed to do in each maneuver? Why does not the real robot making the same displacements?

S4: The previous problem occurs because of calibration issues, namely the *distance between the wheels* in centimeters and *steps traveled per wheel rotation* are not properly set. Try to set them to the correct values in the first lines of *main.c* (WHEEL_AXIS=0.053 and STEPS_PER_WHEEL=1000 respectively).

2.3 Sensor reading

In the following exercise you will use the e-puck's acceleration sensor to figure out whether it is going up or down hill. We will use the LEDs to indicate this. The e-puck has a 3-axis accelerometer, but only one of the three axes will be used to determine the e-puck's orientation. Unfortunately, the e-puck's wheels do not have a very good grip, so we cannot make it going up or down a steep slope. This is why we will tilt the robot in the air to simulate going up or down.

Now we want to read data from the accelerometer and figure out what values are depending on the position of the e-puck.

S5: Open the file *epuck_read_accelerometer.c* and try to understand what the different parts of the program are doing.

- Initialization of the hardware
- Computation of the offset (more on that in a later question)
- Reading the accelerometer
- Writing the value to the terminal (**Note**: the accelerometer reads all three axes, but only outputs the y-axis, as this is the only one we need)

Compile the program (do not forget that you need to modify the *Makefile* in the same directory as *epuck_read_accelerometer.c* first with the path to the

parent folder of the library tree), and upload the program to the robot, as written before.

You can check the values given from the accelerometer by typing (where ### is your 3 digit robot number):

```
> epuckconnect ### && cat /dev/rfcomm###
```

S_6: What values do we get when the e-puck sits on the table or when it points 45°, 90° up or 45°, 90° down?

To process the data it helps if the accelerometer reading is around 0 when the e-puck is on a flat surface.

S_7: Open the file *calibrate_acc.c* and try to understand how it computes the offset. Then go back to *epuck_read_accelerometer.c* uncomment the line (remove `//`) which removes the offset from the measured value. Compile, upload and execute the code. **Note:** before executing the code, make sure that the e-puck sits on a flat surface. Check the sensor values again.

S_8: What values do we get *now* when the e-puck sits on the table or when it points 45°, 90° up or 45°, 90° down?

I_9: Now we want to turn on the front or back LEDs depending on whether the e-puck is pointing roughly 45° (or more) up or 45° (or more) down. To turn on the front LEDs you can use the function `front_LED_on()` and to turn on the back LEDs you can use the function `back_LED_on()`. To turn all LEDs off use `all_LED_off()`. Add a piece of code which calls the appropriate function depending on the value read from the accelerometer. Compile, upload and execute the code and check if the LEDs light up appropriately.

2.4 Digital filtering

We will now simulate the robot going over a bumpy road and see how this affects our up/down-indicator from the previous section. Then we will design a digital filter to clean the accelerometer's signal to make sure that the LEDs really only indicate if the e-puck is going up or down and do not give a "false alarm" on a bumpy road.

I_{10}: First we will simulate the e-puck going over a bumpy road. Add the line

```
go_bumpy();
```

to the file *epuck_read_accelerometer.c* before the `while()` loop. Make sure you still have the code inside which turns the LEDs on and off, depending on the orientation. Compile, upload and execute the code. The e-puck should start moving (bumping) forward now.

Q_{11}: You should see the LEDs which indicate the inclination of the robot flashing occasionally. Why is that?

Now we will analyze the signal from the accelerometer using MATLAB.

S_{12}: Type (where ### is your 3 digit robot number):

```
> epuckconnect ### && cat /dev/rfcomm### | tee acc_data.txt
```

into your terminal. In case this does not work, close the terminal, open a new one and try again. Now lift up the e-puck, hold it horizontally and restart the program by pushing the reset button. The data from the accelerometer is now stored in the file *acc_data.txt*. Very slowly rotate the e-puck back and forth in the air a few times (the wheels should still be spinning). Then put the e-puck down and let it move (bump) forward for a bit. Stop the data collection by typing `CTRL+C`. Stop the e-puck by turning it off.

S_{13}: Open MATLAB and plot the values recorded in `acc_data.txt`. Can you see where the values changed because you rotated the e-puck and where the values changed because of the bumping robot?

Now we would like to filter the signal such that the LEDs *do not* come on when the robot is bumping, but still *do* come on when the robot is rotated in air.

Q_{14}: What kind of filter would we need? Low-pass, high-pass, or band-pass?

For filtering, we will use the *Direct FIR*, presented in Lab 6:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k]$$

where the filtered value `y[n]` is obtained by summing up `M` previous unfiltered values `x[n-k]` where each previous value has been multiplied with the corresponding coefficient b_k. The number of coefficients (`M`) is called the filter order. The higher the filter order the more the *real* filter approaches an *ideal* filter.

I_{15}: Open MATLAB's filter design tool. Now design the filter taking into consideration the following aspects (recall Lab 6):

- Select in the `"Response type"` the desired type of filter.
- In `"Design Method"` select `"FIR equiripple".`
- Next we will have to tell the design tool at which frequency our signal has been sampled. In our case it is 50 Hz, so set F_S to `50` and make sure that `Units` is set to `Hz`.
- `Fpass` indicates the frequency up to which the signal should be unfiltered.
- If we would have an ideal filter everything above `Fpass` would be completely filtered out. In the real world, however, we can never

completely filter out anything. `Fstop` indicates the frequency at which the higher frequencies have been filtered out to a certain degree.

- Set the filter order. Select `"Specify order"` and set it to `50`.

Chose the `"Response type"`, and design the values for `Fpass` and `Fstop` considering that we want to filter out the high-frequency bumps of the moving robot but still register slow changes such as a slow rotation of the robot. **Note:** inclination is approximately a constant signal, so Fpass should be between 0 and 1 Hz. Also you can notice that the bumps have a period less than 1 second so the noise frequency should be > 1 Hz, so Fstop should be inside that interval.

After the filter computation is done, you should now see the magnitude response of the filter in the upper right window.

Now we would like to apply the filter with order 50 to our signal.

I$_{16}$: Export the computed coefficients to the MATLAB workspace and filter the data using the MATLAB routine *filter_data.m*. Finally plot the result:

```
y=filter_data(acc_data,Num);
figure;
plot(y);
```

Compare the new plot of the filtered data with the previous plot of the raw data from **S$_{13}$**.

1.1 Implementation on the e-puck

Now that we have seen that the filtered data clearly shows the slow rotation of the robot, but mostly filters out the bumpy driving, we want to implement the filter on the e-puck. The filter will directly process the data after they are read from the sensor and we will use the filtered data to control the LEDs.

I$_{17}$: Implement the filter on the e-puck

- In order to filter data, it is not enough to have the newest value from the accelerometer, we also need to have older samples. Looking again at the formula for our digital filter

$$y[n] = \sum_{k=0}^{M} b_k x[n-k]$$

We can see that the number of old samples `x[n]` we need, depends on the order of the filter `M`. The function

```
store_buffer(x,sample_buffer,filter_length);
```

in the file *buffer.c* puts the new sample `x` into an array called `sample_buffer` and pushes all the older ones back and thereby deletes the oldest one. In file *epuck_read_accelerometer.c* add the line

```
store_buffer(y,sample_buffer,51);
```

after the offset has been removed from the sample. Make sure that you also add `sample_buffer` as a new variable `int sample_buffer[51];`

- Now that we have all the old samples we need, we can implement the filtering. Open the file *filter.c*. It contains the implementation of the filter formula. The only thing which is missing are the filter coefficients which we previously computed in MATLAB.
- Go back to MATLAB, and print out the coefficients which should still be stored in `Num`. Simply type `Num` in the command line and the coefficients will be printed out. Copy the 51 coefficients and paste into the file *filter.c* by replacing the 51 zeros.
- Now we can filter the last sampled value using our `sample_buffer and` the function `filter_last_sensor_value(sample_buffer);` Add the line `y=filter_last_sensor_value(sample_buffer);` below `store_buffer(y,sample_buffer,51);`
- `y` now contains the filtered version of the last sample.
- Compile, upload and execute your code. Your e-puck should bump-drive forward now, but the LEDs should not or only very rarely ever come on. If you lift up the e-puck however and rotate it, the LEDs should come on again.

Q_{18}: Do you notice any delay in the response of the LEDs? If yes, what do you think is the reason?