Sensors and Sensing
Lab 1: Arduino, Raspberry and Motor Control

Todor Stoyanov

September 18, 2018

1 Objectives and Lab Materials

The objective of this lab is to introduce you to programming of embedded controllers for motors and encoders, using the Arduino boards. In addition, you will learn how to connect to your robot and how to implement simple serial communication in C++.

For this lab you are given:

- MakeBlock kit, including a MegaPi board and two DC motors.
- Raspberry Pi with Ubuntu and the Arduino IDE pre-installed.

The following tasks will guide you through setting up an Arduino controller and a corresponding C++ application to read values from the microcontroller and set PID parameters. In your report, describe the steps you have taken to solve each task, as well as the results you obtained and the methods you used to verify that the system functions properly. Explain what experiments you performed, how and why. Provide your Arduino sketch and C++ source codes as attachments and demonstrate the final system to the lab assistant. If you think it is necessary, you may take pictures of the system in action, and/or include screenshots.

1.1 Preliminaries

Before starting with this lab please verify you have performed these preliminary steps:

- For this lab, unmount the tracks of the robot and rest the robot onto a stand, so that the wheels do not touch the ground.
- Mount the Raspbery Pi to your robot, then insert the SD card and connect the charger cable.
- Connect the MegaPi USB cable to any free USB slot on the Raspberry.

- Connect an Ethernet cable between your workstation (desktop) PC and the Raspberry Pi.
- Connect the battery pack.
- Using the network manager on your workstation, set up a static IP for the wired connection onto the subnet 192.168.100.255.
- Use ssh to connect remotely into the Raspberry Pi:

  ```
  ssh pi@192.168.100.2 –X
  ```

- From here on we assume you are working on the Raspberry Pi reomtely over ssh.
- Clone the github code for this lab.
  `git clone https://github.com/tstoyanov/sensors_lab1_2018.git`
- Copy the file `sensors_lab1_2018/src/sensors_lab1_2018_stub.ino` into your arduino sketch directory on the Raspberry.
- Verify that the C++ code compiles by the following steps:

  ```
  cd sensors_lab1_2018/
  mkdir build
  cd build
  cmake ..
  make
  ```

  If this step does not work, you can continue with Tasks 1 and 2, and call the lab instructor.

2 Task 1: Basic Setup (5 points)

For this task, you will verify your setup and test some basic Arduino code.

- Start up the arduino IDE:

  ```
  cd ~/arduino-1.8.5
  ./arduino
  ```

- Open the following example sketch:
  `File>Examples>MakeBlockDrive>Me_On_Board_encoder>Me_Megapi_encoder_pwm`
- Save the sketch to a local directory (so that you can modify it)
- Compile and upload the sketch

- Open the serial monitor and set the correct baud rate
- Experiment by setting different commands to the microcontroller
- Modify the sketch to print the speed and position of each encoder as space separated values. Do not print any additional characters.
- Use the serial plotter tool to generate plots of the velocity and position of the encoders over time for different PWM set values.

3 Task 2: Communication Protocol (5 points)

In our next steps we will need to command the motor through a high-level controller. Unfortunately, this often neccessitates writing serialization code. In this task, we will set up and test the communication protocol between the Raspberry and the Arduino board. The commands you should implement are shown in Table 1. Follow these steps to implement the communication protocol.

Table 1: Simple communication protocol

Command	Payload	Purpose
VEL	ID[short]VAL[short]	Set a target velocity in rad/s to a motor
SET	ID[short]Kp[short]Ki[short]Kd[short]	Set the pid parameters of a motor

- On the Arduino side (`sensors_lab1_2018_stub.ino`):
  - Edit the function *sendStatus()* to communicate the internal state of the board. You should print out at least the value of each encoder, the value of the current velocity and the controller set point. Easiest option is to print as comma separated integer values followed by a carriage return (`\r\n`), but you may also pack the bytes directly in a character buffer and define your own termination sequence.
  - Edit the processMessage function which should check for incoming messages on the serial port.
  - Since you will be using the serial port with your C++ program, you cannot use the serial monitor at the same time. For debugging you can try switching on and off an LED.
- On the C++ side (arduino_interface.cc):

- Implement the methods *getStatus()*, *setPIDParameters()*, and *updateStatus()*. Follow the instructions in the comments.
- Implement the *main()* method in the file `src/arduino_interface.cc`. Print out the status variables and try sending different commands.

- Upload your arduino code and test the communication protocol. Try setting PID parameters and setpoints and verify that the status reported back by the arduino changes accordingly.

4 Task 3: PID Control (10 points)

PID control is a classical control method for achieving a desired setpoint. PID control uses a feedback loop in which the error between a desired control setpoint $r(t)$ and a measured signal $y(t)$ is used to derive an appropriate control action $u(t)$. The schematic of a PID controller is shown in Figure 1.

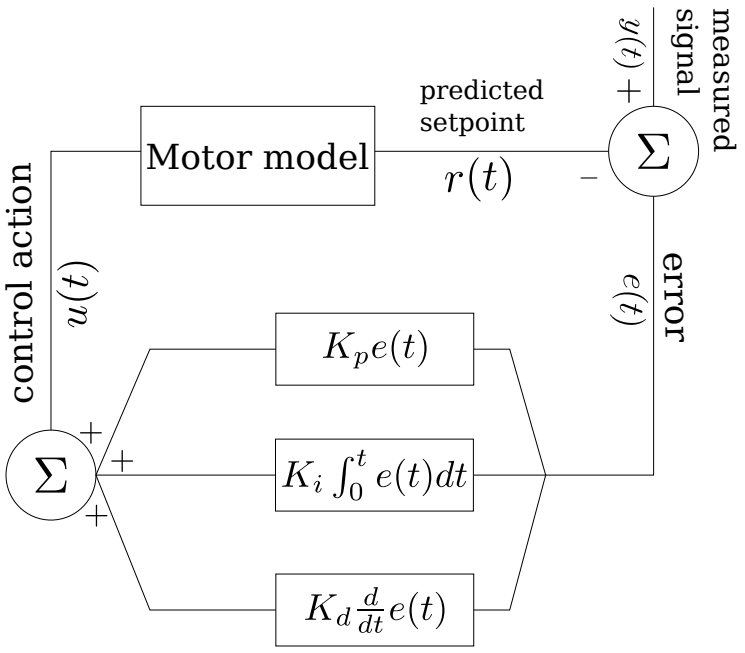

Figure 1: Block scheme of a PID controller.

The basic control equation in this case is:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{d}{dt} e(t) \tag{1}$$

where $e(t) = y(t) - r(t)$ is the error between the desired set point and the current measurement. The constants K_p, K_i and K_d regulate the behavior of the controller and will be tuned in the next exercise. In this task we will implement a PID controller for our motor

- In your Arduino sketch, instantiate the struct which holds the PID parameters K_p, K_i and K_d.
- Instantiate the control state parameter struct: the current set point r, the error e, the error derivative de and the control output u.
- Implement the function *pid* which computes the control output, given the error, error derivative and PID parameters. Keep track of the integrator term inside the PID parameter struct.
- Implement the missing parts of the *loop* function as to compute the error and error derivatives, and use them to obtain the PID controls.
- Note: to compute the error derivative de you should use the fixed sampling period dT.
- Initialize your controller with a low K_p and keep K_i and K_d as zeros. Plot the control outputs in your status message and display them using the serial plotter. Test your controller for several hardcoded setpoints and generate plots. Do not worry if the motor does not move at this point as the control outputs may be too low to generate motion. We will tune them in the next task.

5 Task 4: Controller Tuning (5 points)

The final task in this lab is to tune the PID controller to achieve smooth motion. You have already most of the interfaces needed, but still need to find good values for K_p, K_i and K_d.

- As a first step we will tune the controller for a pre-defined setpoint of $4rad/sec$. Hardcode this as the setpoint when you initialize the controller struct.
- Use the serial plotter tool to display the current velocity, setpoint, and error of the controller.
- There are two ways you can now tune the controller: either by manually setting values and re-uploading the sketch, or by using the C++ interface. If you choose the first option, you will have to re-upload the code multiple times and this might be slow. If you choose the second option, you will have to devise a way to plot the relevant varriables (error, setpoint, current value) in C++, or alternatively log the data and feed it into a plotting tool such as gnuplot or octave. Take whichever option is more convenient for you.
- Vary K_p and set target values for your controller and observe the graphs.

- Once you are satisfied with the rise time and/or you see an overshoot, you can try adding some damping by increasing K_d to achieve smoother tracking performance.
- Once you reach a stable behavior, you can start adding a small amount of K_i to reduce the steady state error. Be careful not to overshoot as this can result in erratic oscillations.
- Generate plots of your controller behavior for at least three different set points. What is the maximum and minimum velocity you can track? Describe your results and stress-test your controller.