Physical Human-Robot Interaction Laboratory (2022-2023)

Prof. Andrea Calanca, Matteo Meneghetti

Installation

Requirements

- PC with any Linux distribution (preferred: Ubuntu from version 20.04 onwards)
- Visual Studio Code

PlatformIO extension

PlatformIO is a VSCode extension which offers a development environment for embedded applications. Since it provides a well-structured toolchain to compile, debug and test our codebase with multiple frameworks, it is currently one of the best free options to develop on embedded devices.

- Open the Extension Page and look for the **PlatformIO** extension, then press *Install*.
- Wait for the installation process to end (it may take a while). A possible error during the installation concerns the lack of the *venv* module. On Debian-based distributions, e.g. Ubuntu, it is possible to install it using the command:

  ```
  sudo apt install python3-venv
  ```

- When prompted, restart Visual Studio Code or just reload the window

User Interface

For our convenience, the GUI is hosted as a Python package on the Python Package Index (*PyPI*), as such we just need to download the relative package by using *pip*. Assuming we are using the default bash:

```
# Default version (web-browser)
pip3 install forecastui

# Windowed version
pip3 install forecastui[qt]
```

To start the application, just open a shell and enter the command

```
forecastui
```

Accessing the serial port

By default, a non-super user is unable to access the serial port. In order to proceed without having to start our application with the *sudo* command (terrible practice), we need to add our user to the *dialout* group (because that's the group which the serial devices under /dev are part of).

```
sudo adduser $USER dialout
```

Then don't forget to **reboot** your PC for the changes to have effect.

Firmware

The last step we need is to flash the firmware on the Nucleo-L432KC board. A default project is hosted on the PHRI Testbed Gitlab repository. Find a suitable location on your filesystem and then clone the project

```
# I am assuming you have already Git installed by now.
# If that's not the case, installing it on Ubuntu is pretty straightforward
sudo apt install git
git clone https://gitlab.com/altairLab/elasticteam/forecast/phri-testbed.git
```

That's just the default project with a barebone main.cpp file. Notice the *include* and *src* directories, where you are supposed to store **.hpp* headers and **.cpp* source files for the controllers you will develop during the laboratory lessons. The main library is hosted on another repository, Forecast PHRI. Enter the *lib* directory and clone the actual control framework:

```
cd phri-testbed/lib
git clone https://gitlab.com/altairLab/elasticteam/forecast/forecast-phri.git
```

Flash the firmware

Identify your setup

We have prepared three different setups (A,B and C). Each setup consists in a DC motor equipped with a load cell which measures the interaction force between the motor and the environment. Each motor has different torque constant, nominal current, gear ratio and encoders as well as other mechanical properties which are outside the focus of these lectures. All these parameters are already identified and stored in the control library Forecast PHRI.

Edit the file under *lib/forecast-phri/scripts/defaults/rpc.ini* by uncommenting the lines regarding your setup. If you are uncertain on what to do, ask for help to the teaching assistant.

Inspect the main file

We are almost done and ready to compile. The main.cpp file already includes the code for logging, instancing the reference signal and starting the control loop. A placeholder controller is already implemented, we are free to use it as a foundation to develop more complex control laws, in order to focus more on control and less on programming.

The Controller class

In order to use a custom controller, such controller must include and then inherit from the *Controller* class.

To get started, create a new file called *MyController.hpp* in the *include* directory of your project and include the main Controller class. Only then, we are free to define our controller class. For my convenience, I will also operate inside the *forecast* namespace; if you find yourself unsure about C++ namespaces, I advise you to do the same.

```
#ifndef MYCONTROLLER_H // In order not to define this class multiple times
#define MYCONTROLLER_H

#include <forecast/Controller.hpp>

namespace forecast {

    class MyController : public Controller {
        // TODO: our methods will go here!
    }
}

#endif
```

The only thing left in the header is to define these methods inside the class: constructors, init, getParamNames and process. Let's assume that our controller needs just one parameter: KP (float), as in the proportional parameter of a common PID implementation.

```
class MyController : public Controller {
    public: // notice the public keyword, don't miss it
        MyController();
        MyController(float kp);
        virtual bool init(const std::vector<float>& params) override;
        virtual std::vector<std::string> getParamNames() const override;
        virtual float process(const IHardware* hw,
                              float ref,
                              float dref = 0,
                              float ddref = 0) override;
   protected:

    float kp = 0.0;
    float theta = 0.0f;
    float err = 0.0;

}
```

Let's move on to the implementation, create a MyController.cpp file in the *src* directory. All we need is to include the header file we just created and implement the constructors and all the remaining virtual methods.

```
#include <MyController.hpp>

using namespace forecast; // or whatever you are using

MyController::MyController()
    : Controller(1) { // notice the 1, i.e. the number of parameters
    // nothing to do here
}

MyController::MyController(float kp)
    : Controller(1), // same as above
      kp(kp) // initialize the 'kp' member, copy by value
{
    Controller::initialized = true; // set the initialized member to true
}

bool MyController::init(const std::vector<float> &params)
{
    if (params.size() != numberOfParams)
        return false;

    kp = params[0];

    return initialized = true; // set the initialized member to true
}

float MyController::process(const IHardware *hw,
                            float ref,
                            float dref,
                            float ddref)
{
    theta = hw->getThetaM();
    err = ref - theta;
    float output = kp*err;
    return output;
}
```

```
std::vector<std::string> MyController::getParamNames() const
{
    return {"KP"}; // return a string vector, in this case
                   // of just one element
}
```