Digital Control TP

Philippe Schuchert - July 18, 2021

Figure 1: Quanser QUBE–Servo 2

Introduction

The objective of this TP is to control the QUBE – Servo 2 shown in Fig. 1. You will be in charge of implementing the controller: first code the controller using the `C` programming language, and then find appropriate values for the controller's coefficients. To tune the controller, MATLAB will be used, and the model of the DC motor is given. The control loop is implemented in LabVIEW, and will call your `C` controller implementation. A mac mini will serve as the "brain" of the operation: sending the correct control values to the DC motor, at the correct time. The bloc-diagram of the control loop can be found in Fig. 2.

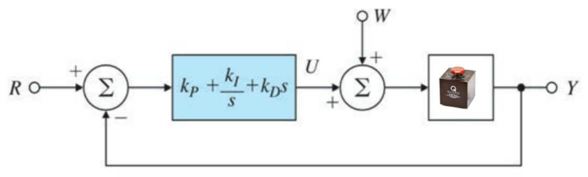


Figure 2: Control architecture. Change plot

Implementing a continuous time controller $K(s)$ on a computer system is difficult: it requires measuring the output y and computing the input to the system u infinitely fast. This problem is usually solved by using a digital controller $K(z)$, sampling the output y and computing the input u every T_s seconds. If the system is sampled sufficiently fast, the continuous time controller $K(s)$ can be converted to a discrete controller using the Zero-order-Hold (ZOH) discretization.

In the following sections, the z-transform variable is denoted z. The sampled version of a signal $x(t = kT_s)$ is denoted $x[k]$. The $z-$transforms of a signal is denoted using a capital letter, eg $\mathcal{Z}(x[k]) = X(z)$. The output is denoted y, the input u, the error $e = r - y$.

1 Digital PID controller

The PID controller is the most common controller in industry, as it is simple to tune and results in appropriate performance for relatively simple systems. The discrete formulation for a PID controller is given by:

$$K(z) = \underbrace{K_p}_{P} + \underbrace{K_i \frac{T_s}{z-1}}_{I} + \underbrace{K_d \frac{z-1}{(z-1)T_f + T_s}}_{D}, \quad (1)$$

where K_p the proportional gain, K_i the integral gain, K_d the derivative gain, T_f a filtering constant for the derivative of the error, and T_s the sampling time. In the lecture slides (Chapter 4, slide 60), the PID is described using $T_f = T_s$. This choice makes the control input very sensitive to measurement noise and abrupt changes in the reference. To reduce this effect, a filtering constant $T_f \approx 2 \cdot T_s - 10 \cdot T_s$ can be chosen, or in as seen in a later TP, a 2DoF controller with different paths for the measurements and reference. In the z-domain, the control value $U(z)$ is given by:

$$U(z) = K(z)E(z) = U_p(z) + U_i(z) + U_d(z), \qquad (2)$$

where U_p, U_i, U_d the proportional, integral and derivative contribution respectively. To implement the PID controller, U_p, U_i, U_d will be handled separately.

1.1 Proportional part

The proportional part affects the control input proportionally to the error, and acts like a spring: the larger the error, the more it will *pull* the system in the right direction. It is given by:

$$U_p(z) = K_p E(z)$$

and with corresponding difference equation:

$$u_p[k] = K_p e[k],$$

where $u_p[k]$ the proportional contribution at sample k.

1.2 Integral part

The integral part affects the control input proportionally to the integral of the error, and prevents constants tracking errors. It is given by:

$$U_i(z) = K_i \frac{T_s}{z-1} E(z).$$

Multiplying both sides of the equation by $z - 1$, and taking the inverse z-transform results in the following difference equation:

$$u_i[k] - u_i[k-1] = K_i T_s e[k-1],$$

where $u_i[k]$ the integral contribution at sample k. This can be rewritten as:

$$u_i[k] = u_i[k-1] + K_i T_s e[k-1].$$

1.3 Derivative part

The derivative part affects the control input proportionally to the derivative of the error. It acts as a damper, and will tend to slow-down the system. It is given by:

$$U_d(z) = K_d \frac{z-1}{(z-1)T_f + T_s} E(z).$$

Multiplying both sides of the equation by $(z-1)T_f + T_s$, and taking the inverse z-transform results in the following difference equation:

$$T_f u_d[k] - (T_s - T_f)u_d[k-1] = e[k] - e[k-1],$$

where $u_d[k]$ the derivative contribution at sample k. This can be rewritten as:

$$u_d[k] = \frac{(T_f - T_s)u_d[k-1] + e[k] - e[k-1]}{T_f}.$$

1.4 Control input and saturation

The control input at sample k is the sum of the three different contributions:

$$u[k] = u_p[k] + u_i[k] + u_d[k].$$

To prevent the saturation of the control input, use either of the conditional integration scheme proposed in Chapter 4, slide 62, to guarantee $|u| \leq u_{\max}$. If $u_{\max} = 0$, you should ignore the anti-windup.

1.5 TP 1

1.5.1 Implementation

Implement this PID controller with anti-windup as described using the `C` programming language. Complete the function `calc` in the file `./C/src/PID.c`. This file has two functions:

- `initialize`: This function is used to set the different coefficients (K_p, K_i, K_d, T_f, T_s, $u_{\max}$). The different values are read from a binary file generated in MATLAB. This function is already completed, and should not be modified.
- `calc`: This function is used to calculate the new output $u[k]$, given $y[k]$ and $r[k]$ as input to the function. This function must be completed.

You can use the Xcode project to troubleshoot your implementation. Your implementation should return the same values as listed at the end of `main.c`. Once the controller is properly implemented, you can compile it to a shared object using the following command[1] in the terminal:

```
$ clang -shared -undefined dynamic_lookup -o
pid.so PID.c
```

You should obtain a file called `pid.so`, which will be used to control the system. Double check that the controller is properly implemented by running the script `./MATLAB/TestPID.m` in MATALB.

[1]Remember to `cd` to the location of the file first!

1.5.2 Tuning

Tune a PID controller using the Ziegler-Nichols method by completing the `./MATLAB/TunePID.m` file. Save the PID controller data, and simulate the closed-loop using in the Simulink file `./MATLAB/Qube.slx`. Change the amplitude and the period of the square signal to see the effect the the anti-windup.
Note that this file assumes the controller is located at `./C/src/pid.so`, and the binary file at `./MATLAB/dataPID.bin`.

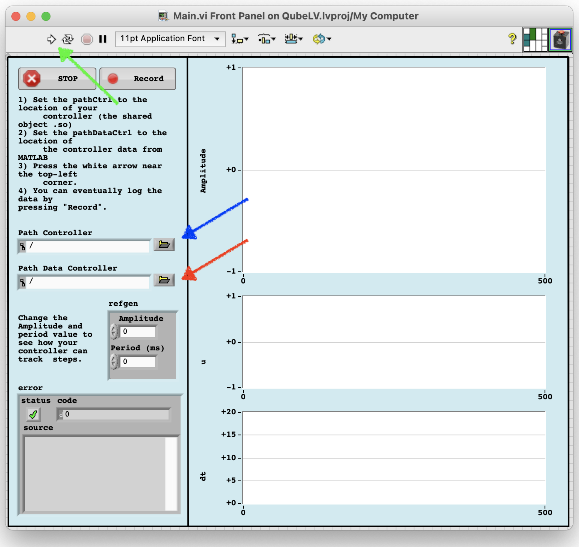
1.5.3 Quanser Qube

If the implementation is correct and controller tuned, you should test your controller on the Qube. To test your PID controller on the physical system, open the LabVIEW project `QubeLV.lvproj`, and then in the project explorer open `MAIN.vi`.

1. Click on the folder icon near the blue arrow and select your `C` controller compiled as shared object (`pid.so`).

2. Click on the folder icon near red arrow and select your binary file `dataPID.bin` from MATLAB.

3. You can test your control strategy by pressing on the small white arrow, new the green arrow. You can use square waveform as reference for the DC motor. Change the amplitude and the frequency. Perform the same experiment with anti-windup disabled.



1) Set the pathCtrl to the location of your
controller (the shared object .so)
2) Set the pathDataCtrl to the location of
the controller data from MATLAB
3) Press the white arrow near the top-left
corner.
4) You can eventually log the data by
pressing "Record".

Change the Amplitude and period value to see how your controller can track steps.

Amplitude: 0
Period (ms): 0
code: 0

Figure 3: Main VI, front panel