Lab Exercise 08 DC motor PID position control

Lab 08.1 Objectives

The objectives of this exercise are to:

1. implement a position control system for an inertia dominated load,
2. explore appropriate path planning, and
3. integrate the use of a standard Matlab design tool into the application development system.

Lab 08.2 Introduction

In this exercise, a closed-loop position control system for the DC motor will be developed. The physical system is identical to that of Lab Exercise 07: as shown in Figure 08.1, the optical encoder (through the FPGA), the D/A converter (connected to the motor amplifier), and the periodic timer interrupt will be combined to control the DC motor.

A Matlab tool will be used to design an appropriate proportional-integral-derivative (PID) controller, shown in Figure 08.2. Later, you will evaluate the controller performance for a time-varying position reference path $x_{\text{ref}}(t)$.

This project builds on your past work. The program is structurally similar to that of Lab Exercise 07, and many of its components are reused.

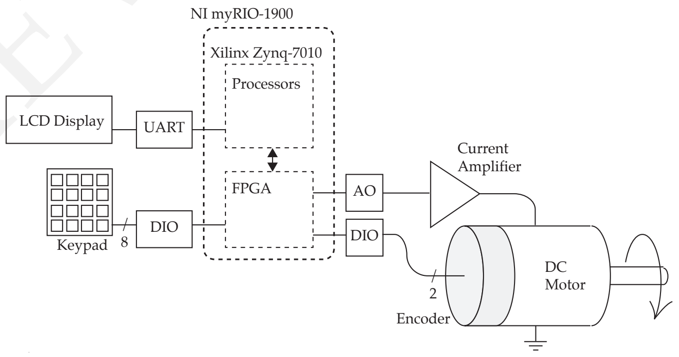

Figure 08.1: schematic of the test apparatus.

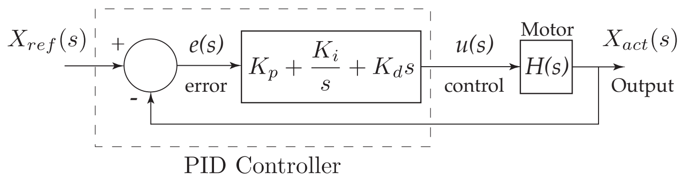

Figure 08.2: block diagram of the system with a PID controller in the loop.

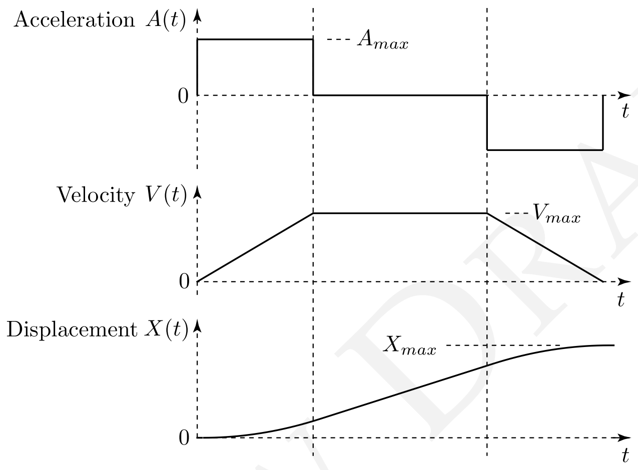

Figure 08.3: a method of path planning for position control is to integrate piecewise-constant acceleration (top), to obtain piecewise-linear velocity (middle), also to be integrated to obtain piecewise-quadratic (and continuously differentiable) position (bottom).

Lab 08.3 Path planning

A common task for a positioning system is to start from a stationary position, move to a new location, and then stop. Of course, one way to do this is to apply an appropriate size step to the reference input of the position control system. However, depending on the system bandwidth, a sufficiently large step may require torques (current) and/or velocities (voltages) that exceed the motor/driver capabilities. In addition, the dynamic characteristics (e.g. rise time and overshoot) may be inconsistent with the application requirements. One remedy is to use a form of truncated ramp instead of the step reference input.

Suppose that we wish to reposition a mass-dominated load by X_{max} as rapidly as possible, subject to limitations on the maximum acceleration A_{max} and velocity V_{max}, while avoiding discontinuities in the position slope. One such command is constructed as shown in Figure 08.3.

The motion has been divided into three sections: acceleration, constant velocity, and deceleration. Within this scheme, many variations are possible: High A_{max} would result in a long constant velocity section, with short accelerations. Alternately, for high V_{max}, the displacement would approach an s-shaped curve with no constant velocity section. Finally, by allowing both high A_{max} and V_{max}, the curve would approach a step.

In this lab exercise, you will use a C function `Sramps` that implements this time-varying displacement as the control system reference input. The function can link any number of ramp segments in succession, including specified dwell times at the end of each segment. It can also repeat the sequence of ramps indefinitely. See Lab 08.6 and Resource 16 for details.

Lab 08.4 PID control design and evaluation

For the lab exercise, you will write two Matlab scripts: one to design your PID controller and another to compare its performance to an analytical model. Specifically, the first script will design a PIDF controller using the MATLAB Control System toolbox function `pidtune`. This compensator should be designed to track the reference input, and to have control bandwidth of 8 Hz. A PID$\underline{F}$ controller improves noise immunity of a PID controller by limiting the high-frequency response of the derivative term. Check your controller design by plotting the closed-loop step response using the plant parameters from Lab Exercise 07.

The script should convert the continuous-time transfer function to discrete-time (`c2d`, `tf`, and `tfdata`, with sample time $T = 0.0005$ s), and then use `tf2sos` (transfer functions to second order sections) to break the transfer function into biquads. Finally, use the `sos2header` function (see Resource 17) to write the biquad filter to a C header file (`PIDF.h`) in your Lab Exercise 08 project folder. That header can be *`#include`*d in your myRIO C program (after the definition of the `biquad` **`struct`**.) In this way, when you run your C program in Eclipse, it will automatically incorporate the latest version of your compensator design.

As in Lab Exercise 07, your second script will load the actual response of position control system (`Lab8.mat`), and compare it to both the ideal reference displacement and the dynamic model prediction. See below for details.

Lab 08.5 Program description

The program is similar in structure to that of Lab Exercise 07, consisting of (1) a Main thread that initializes the task and calls `ctable2` to communicate with the user, and (2) a Timer thread that maintains timing using an interrupt, implements the position control, and saves the results. Your specific controller definition is derived from the header file written from your Matlab script.

Lab 08.5.1 Two threads

Main thread The main thread performs the following tasks.

1. Initialize the table editor variables.
2. Initialize the path profile variables as follows.

```
typedef struct {
  double xfa; double v; double a; double d;
} seg;
```

3. Set up the timer IRQ interrupt (as in Lab Exercise 06 and Lab Exercise 07).
4. As in Lab Exercise 07, register and create the Timer thread to catch the timer interrupt. The Timer thread will gain access to both the table data and the path profile through pointers in the Thread resource. For example,

```
typedef struct {
  NiFpga_IrqContext irqContext;   // context
  table             *a_table;     // table
  seg               *profile;     // profile
  int               nseg;         // no. of segs
  NiFpga_Bool       irqThreadRdy; // ready flag
} ThreadResource;
```

5. Call the table editor. The table should contain three "show" values, labeled as follows.

```
P_ref: revs
P_act: revs
VDAout: mV
```

6. When the table editor exits, signal the Timer thread to terminate. Wait for it to terminate.

Timer thread The Timer thread calls the interrupt service routine (ISR). At the beginning of the starting function, declare convenient names for the table entries from the table pointer, and for the ramp segment variables. For example,

```
double *pref  = &((threadResource->a_table+0)->value);
double *pact  = &((threadResource->a_table+1)->value);
double *VDAmV = &((threadResource->a_table+2)->value);
seg *mySegs = threadResource->profile;
int nseg     = threadResource->nseg;
```

The Timer thread includes a loop timed by the IRQ, and terminated only by its ready flag.

Before the control loop begins:

- initialize the analog I/0, and set the motor voltage to zero, using `Aio_Write` (as is Lab Exercise 07) and
- set up the encoder counter interface (as in Lab Exercise 04).

Each time through the loop, it should:

1. Get ready for the next interrupt by: waiting for IRQ to assert, then writing the Timer Write Register, and writing `TRUE` to the Timer Set Time Register.
2. Call `Sramps` to compute the value of the current reference position P_{ref}. See below.
3. Call `pos`, to obtain the position of the motor P_{act}. See below.
4. Compute the current error $e = P_{ref} - P_{act}$.
5. Call `cascade` to compute the control value from the current error using PIDF control filter. **Important**: limit the computed control value to the range $[+7.5, -7.5]$ V.
6. Send the control value to the D/A converter `C00` using `Aio_Write`.
7. Change the table to reflect the current conditions of the controller.
8. Save the results of this BTI for later analysis. See below.

Lab 08.6 Functions

`cascade` – The `cascade` function, called once, from the ISR, during each BTI, implements the general-purpose linear difference equation algorithm from Lab Exercise 06. For this lab use the *same C code* that you used in Lab Exercise 06. In this case, the number of biquad sections will be 1.

Note that, as in Lab Exercise 06, all calculations should be made in (**double**) floating-point arithmetic.

`pos` – Write a `pos` function to read the encoder counter and return the displacement as a `(`**double**`)` in units of BDI (encoder counts), relative to the first position read.

`Sramps` – The C function `Sramps`, given in Resource 16, returns the current input reference position $\mathrm{P}_{\mathrm{ref}}$. The function accepts an input array of structures, each describing a separate displacement ramp segment. Called once each cycle of the control loop, `Sramps` steps through the segments, then repeats the complete path indefinitely.

We will initialize the path array in `main`, then pass the array and the number of segments to the Timer thread through the Thread Resource (described above in the Main thread section).

First, define the new segment data type `seg`:

```
typedef struct {
  double xfa;  // position (revs)
  double v;    // velocity limit
  double a;    // acceleration limit
  double d;    // dwell time (s)
} seg;
```

Then, to test the position control system, initialize an array `mySegs` of type `seg` as follows:

```
vmax = 50.;         // rev/s
amax = 20.;         // rev/s^2
dwell = 1.0;        // s
seg mySegs[8] = { // rev
  {10.125, vmax, amax, dwell},
  {20.250, vmax, amax, dwell},
  {30.375, vmax, amax, dwell},
  {40.500, vmax, amax, dwell},
  {30.625, vmax, amax, dwell},
  {20.750, vmax, amax, dwell},
  {10.875, vmax, amax, dwell},
  { 0.000, vmax, amax, dwell}
};
nseg = 8;
```

Notice that `mySegs` consists of four increasing ramps of 10.125 revolutions each, followed by four similar decreasing ramps that will return the motor to the starting position. All of the segments are subject to the same velocity and acceleration limits, and all dwell for one second before proceeding to the next segment.

You should declare the prototype of `Sramps` as:

```
int Sramps(
  seg *segs,     // segments array
  int nseg,      // number of segments
  int *iseg,     // current segment index
  int *itime,    // current time index
  double T,      // sample period
  double *xa     // next reference positon
);
```

At the end of the last segment, `Sramps` returns the total number of time steps in all of the segments. It returns `0` otherwise.

A typical call of `Sramps` might be:

```
nsamp = Sramps(mySegs, &iseg, nseg, &itime, T, &Pref);
```

When `Sramps` is called for the first time, set `*itime = -1`, and `*iseg = -1`, to initialize its operation.

Lab 08.6.1 Saving the responses

The data can be conveniently saved by defining data arrays in the ISR for each of the reference position, the actual position, and the torque. Then an auto-incremented index variable is used to store the data in the arrays during each BTI. Increment the index as needed, stopping when it reaches the length of the arrays. A convenient length would be 4000 points each.

After the main loop terminates, but while still in the Timer thread, write the results, to the `Lab8.mat` file. The results should include:

1. your name (`string`),
2. the reference position array (rad), cast to **`double *`**,
3. the current position array (rad),
4. the torque array (N-m),
5. the PIDF array, cast to **`double *`**, and
6. the BTI length (s).

Use the same methods as Lab Exercises 04, 06 and 07 to bring the `Lab8.mat` file to Matlab.

Lab 08.7 Laboratory procedure

Test and debug your program.

Lab 08.7.1 Matlab analysis

In the second of your Matlab scripts:

1. Load the experimental results from the `Lab8.mat` file.
2. Define a discrete version of the motor/load plant transfer function from Lab Exercise 07. Consider using `c2d`.
3. Form the discrete controller from the values in the PIDF array in `Lab8.mat`.
4. Form the closed loop system models relating the reference position P_{ref} input to the position P_{act} and torque T outputs:

$$G_1(z) = \frac{P_{act}(z)}{P_{ref}(z)} \quad \text{and} \quad G_2(z) = \frac{T(z)}{P_{ref}(z)}. \tag{08.1}$$

5. Using `lsim`, simulate the system to find the theoretical responses for both the position $P_{act}(t)$ and the torque $T(t)$ to the reference position $P_{ref}(t)$ array that you stored in `Lab8.mat`.
6. In a single Matlab figure plot the results in three `subplot`s versus time, as follows:
   a) reference position, theoretical position, and experimental position;
   b) experimental error (reference − experimental position) and theoretical error (reference − theoretical position); and
   c) theoretical and experimental torque.

What do you conclude?