RYERSON UNIVERSITY

Department of Electrical and Computer Engineering
Faculty of Engineering and Architectural Science

Real-Time Digital PID Controller

Authors:
Steve Singh | 500389934
Rishabh Kumar | 500398457

Table of Contents

1 Introduction and Objective

A real-time digital PID controller was implemented in the lab to control the motor position of a DC motor. This was done primarily using concurrent programming methods with POSIX threads in the C language. The project had three development phases:

- **Part A:** Design the PID controller with anti-windup functionality by obtaining the necessary parameters using the Ultimate Sensitivity Method, then develop a C program which uses the DLaB library to interface with the motor and provide control signals in simulation mode.
- **Part B:** Modify the control program in part A to execute on the hardware and physically control the motor with and without a CPU load present.
- **Part C:** Extend the controller further by allowing controller parameters to be changed during execution.

Part A: Program Development and Simulation

2.1 Proportional Controller

The proportional controller provides a foundational component to control systems, scaling a calculated error value by a proportional gain (K_p). The error represents the difference between a desired reference signal and the actual output signal from the plant. The general behaviour of the proportional controller on its own is unsatisfactory. Depending on how critical the system is, a proportional controller alone may or may not be sufficient. Figure 1 below shows how the proportional controller responds to a step input of 50^o and a proportional gain of 1.

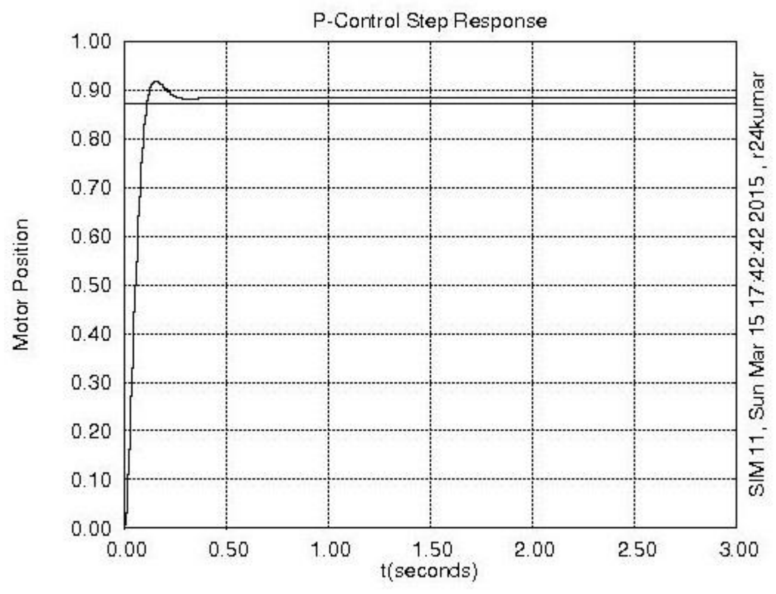

Fig. 1: Proportional Controller step response using $K_p = 1$.

It can be immediately observed that the proportional response has an inability to provide zero steady-state error.

2.2 PID Controller Parameters

Since a proportional controller on its own may be insufficient, adding two more components (integral and derivative, thereby making it a PID controller) may significantly improve tracking abilities. The PID control system provides additional benefits with some minor loss to other response attributes. A benefit that one may observe from the use of a PID controller is that the steady state error is practically zero. The cost for this, however, may be higher overshoots or longer settling times.

First, a transfer function model of the plant or motor system was obtained in order to determine the critical gain. In general, the motor transfer function G_m can be represented as follows:

$$G_m(s) = \frac{K_m}{s\tau_m + 1} \cdot \frac{1}{n} \cdot \frac{1}{s}$$

where K_m represents the motor gain constant, τ_m as the motor time constant and n is the gear ratio. The motor gain constant and time constant are represented more specifically as follows:

$$K_m = \frac{K_t}{RB + K_t K_e} \qquad \tau_m = \frac{RJ}{RB + K_t K_e}$$

where $K_t$is the motor torque constant, K_e the motors counter-electromotive (CEMF) force constant, J is the equivalent moment of inertia due to the load and motor, B is the viscous friction coefficient due to the load and motor and R is the armature resistance.

Using the values ascertained from the motor specifications [1], the following transfer function was obtained for the motor process:

$$G_m(s) = \frac{1}{s^2 + 0.6165s}$$

And the resulting closed loop function as:

$$G_{cl}(s) = \frac{Kp}{s^2 + 0.6165s + Kp}$$

In order to design the PID system correctly, the *ultimate sensitivity method* or Ziegler-Nichols method was used. This approach requires the gain, K_u, at which the system reaches marginal stability or shows oscillatory behaviour as well as the period of the oscillations (P_u). These values are then scaled appropriately by predefined values in the Ziegler-Nichols table (table 1).

TABLE 1
ZIEGLER-NICHOLS TABLE

	K_p	T_i	T_d
P	$0.5K_u$		
PI	$0.45K_u$	$P_u/1.2$	
PID	$0.6K_u$	$P_u/2$	$P_u/8$

One could obtain the closed loop function mathematically, or by using trial and error. The mathematical approach requires taking the characteristic equation from the closed loop function and obtain the poles of the closed loop system for some gain K_u. Placing the resulting poles and zeros on a root locus graph provides a description of the system's behaviour. When there are poles or zeros on the imaginary access, the gain at that point of the system will result in marginal stability. Using this concept along with trial and error, the critical gain where the system reaches marginal stability was determined to be 21.75 with an oscillation period of approximately 0.05 seconds. A simulation of the proportional system under marginally stable conditions is depicted in figure 2.

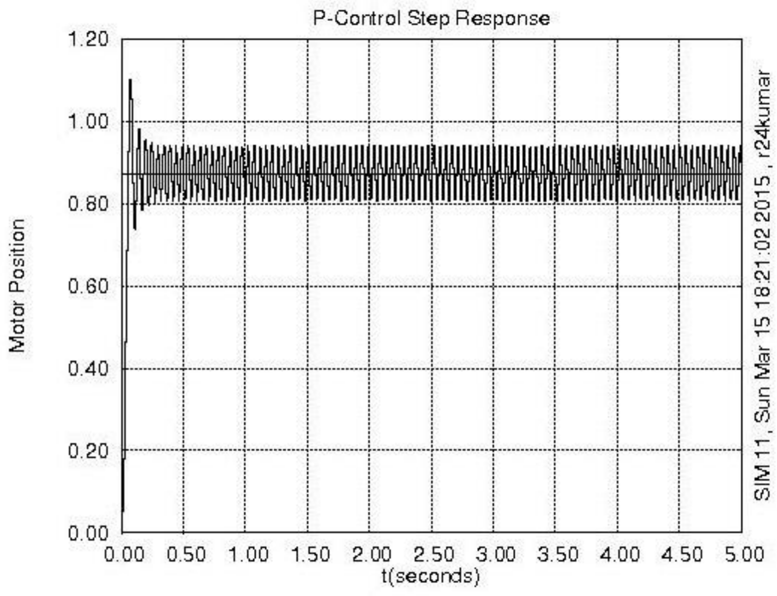

Fig. 2: Marginal stability obtained at the critical gain of 21.75.

Applying these results to table 1 results in the following set of PID parameters:

TABLE 2
PARAMETERS FOR PID SYSTEM ACCORDING TO ZIEGLER-NICHOLS TABLE

	K_p	T_i	T_d
P	10.875		
PI	9.7875	0.04167	
PID	13.05	0.025	0.00625

The response of the motor under the PID parameters from table 2 is depicted in figure 3. There is an observable amount of instability in the transient response before the system reaches steady state.

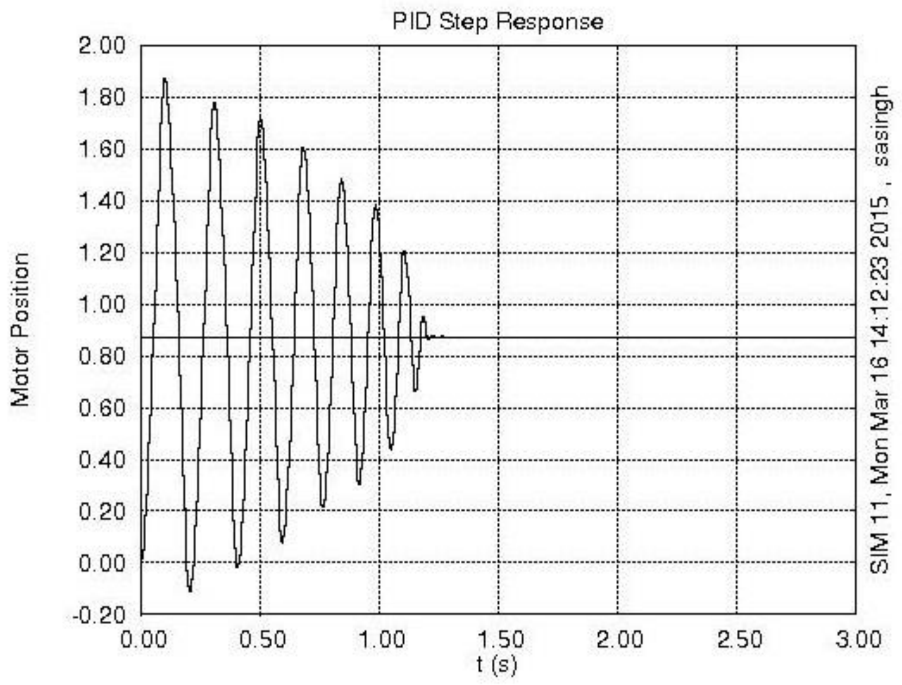

Fig. 3: PID response using parameters from table 2.

2.3 PID Difference Equation

The difference equation for the PID controller above can be derived by obtaining the discretized equation for each component and then combining. The s-domain equation is used as a reference:

$$G_c(s) = K_p\left(1 + \frac{1}{T_i \cdot s} + \frac{T_d \cdot s}{1 + \frac{T_d \cdot s}{N}}\right)$$

The discretized equation for the proportional component, P(kT), is simply:

$$P(kT) = K_p \cdot e(kT)$$

Since we are using the *forward rectangular rule* for numerical integration, I(kT) without wind-up can be defined as:

$$I(kT) = I\big((k-1)T\big) + \frac{K_p}{T_i} \cdot e((k-1)T) \cdot T$$

The *backward difference rule* is used for numerical differentiation. Therefore, D(kT) can be defined as:

$$D(kT) = \frac{T_d}{N \cdot T + T_d} \cdot D\big((k-1)T\big) + \frac{K_p \cdot T_d \cdot N}{N \cdot T + T_d} \cdot (e(kT) - e((k-1)T))$$

Combining all three components, we get:

$$u(kT) = P(kT) + I(kT) + D(kT)$$

$$u(kT) = K_p \cdot e(kT) + I\big((k-1)T\big) + \frac{K_p}{T_i} \cdot e\big((k-1)T\big) \cdot T + \frac{T_d}{N \cdot T + T_d} \cdot D\big((k-1)T\big) + \frac{K_p \cdot T_d \cdot N}{N \cdot T + T_d} \cdot (e(kT) - e((k-1)T))$$

Which is the final difference equation for the PID controller.

2.4 PID Controller with Anti-Windup

When the integrator component becomes saturated, the output of the plant gets clipped and results in an inaccurate error value being calculated. The inaccuracy then gets propagated to the controller, where undesirable control signals are generated. To combat this issue, an anti-windup scheme is added to the controller (see figure 4).

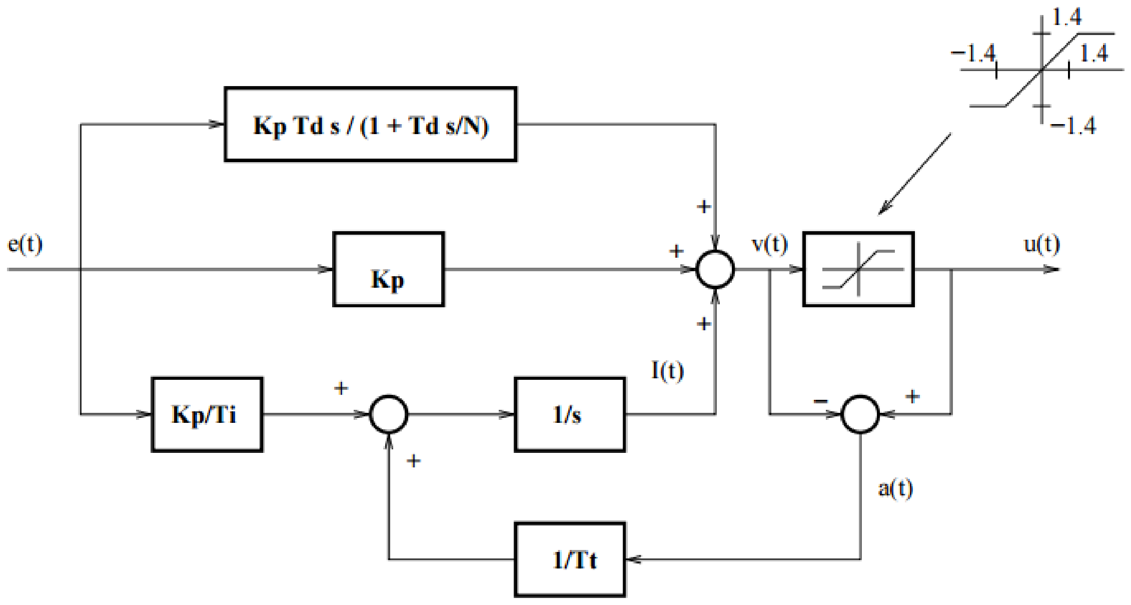

Fig. 4: PID Controller with anti-windup added [2]

According to this control scheme, the integrator component, I(kT), with anti-windup then becomes:

$$I(kT) = I\big((k-1)T\big) + \tfrac{K_p}{T_i} \cdot e\big((k-1)T\big) \cdot T + \tfrac{T}{T_t} \cdot a(kT)\,, \quad \text{where } a(kT) = u_k - v_k$$

Re-combining with the previously derived equation, the difference equation for the PID controller with anti-windup is obtained:

$$u(kT) = K_p \cdot e(kT) + I\big((k-1)T\big) + \frac{K_p}{T_i} \cdot e\big((k-1)T\big) \cdot T + \frac{T}{T_t} \cdot a(kT)$$

$$+ \frac{T_d}{N \cdot T + T_d} \cdot D\big((k-1)T\big) + \frac{K_p \cdot T_d \cdot N}{N \cdot T + T_d} \cdot \big(e(kT) - e((k-1)T)\big)$$

The responses of this controller to a step and square-wave input are shown in figures 5 and 6, respectively. Both inputs have a magnitude of 50^o. The square-wave has a frequency of 1 Hz and duty cycle of 50%.

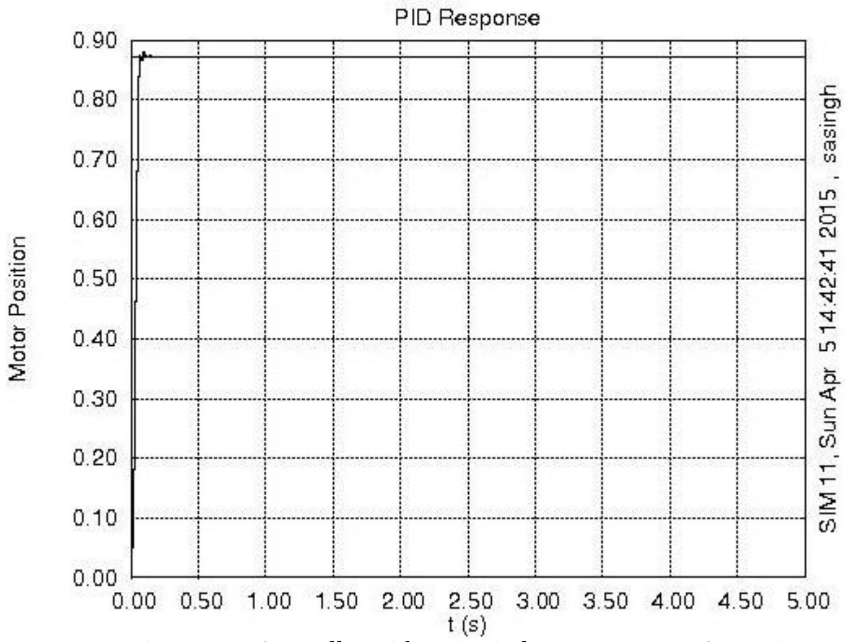

Fig. 5: PID Controller with anti-windup response to 50^o step input

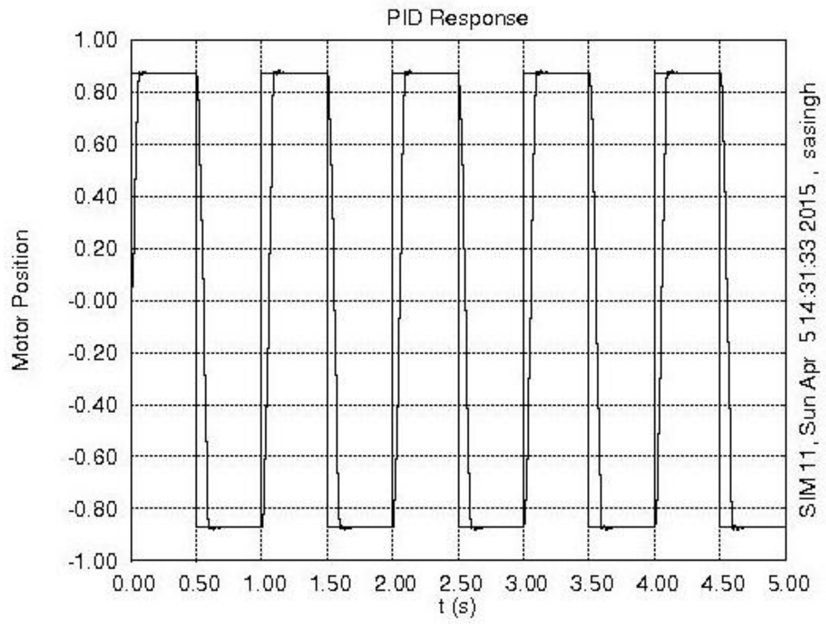

Fig. 6: PID Controller with anti-windup response to 50^o square-wave at 1 Hz with 50% duty cycle

Part B: Real-Time Implementation

In this phase of the project, the control program was modified to enable access to the hardware module of the motor.

3.1 Soft Real-Time Modifications

The control program was modified to run in **soft real-time** under the Real-Time Applications Interface (RTAI). The testing procedure involved obtaining two responses: one with a separate program running concurrently to create a load on the CPU, and the other without such a load. The resultant responses with and without the CPU load can be seen in figures 7 and 8, respectively.

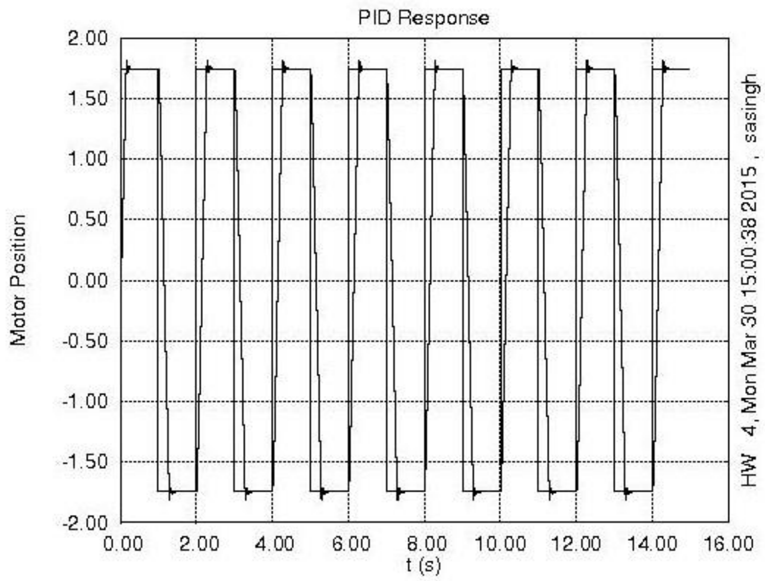

Fig. 7: Soft real-time PID Controller response running under RTAI with CPU load

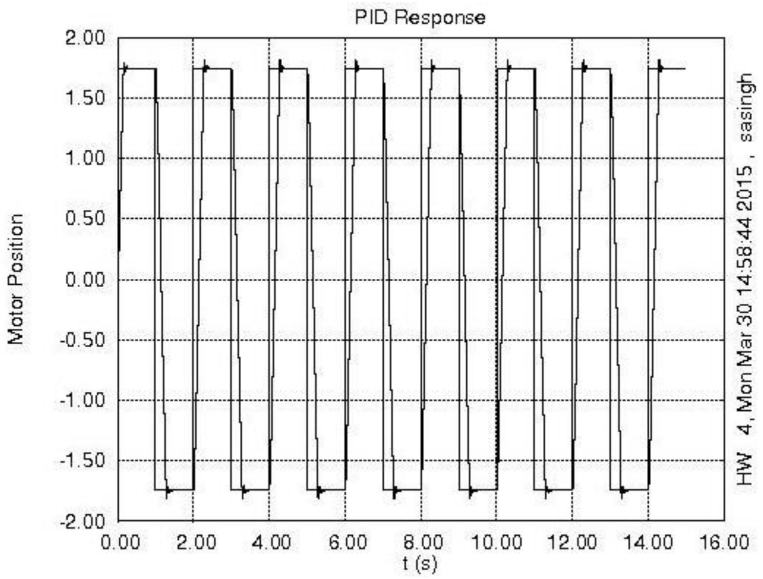

Fig. 8: Soft real-time PID Controller response running under RTAI without CPU load

3.2 Hard Real-Time Modifications

Similarly, the controller was modified to run in hard real-time mode under RTAI. As with the soft real-time variant, the load program was used in one of the tests. Figures 9 and 10 show the response with and without the CPU load, respectively.

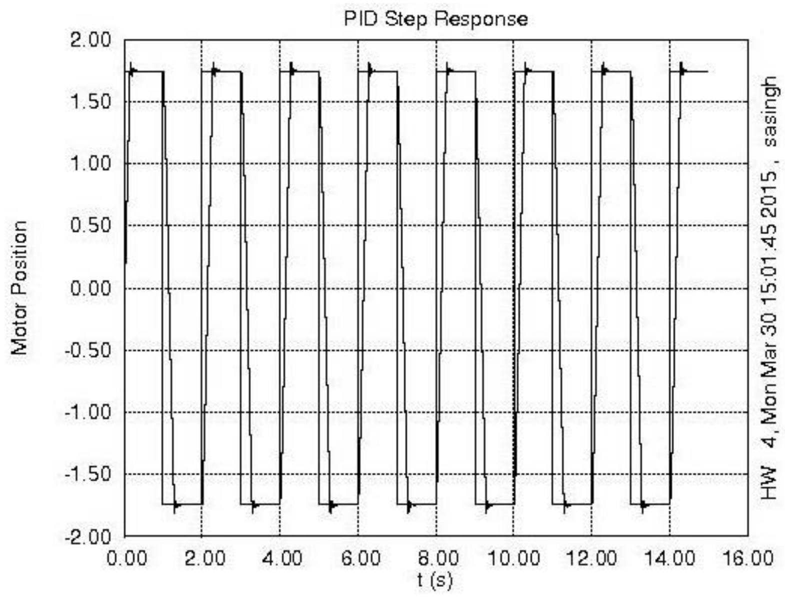

Fig. 9: Hard real-time PID Controller response running under RTAI with CPU load

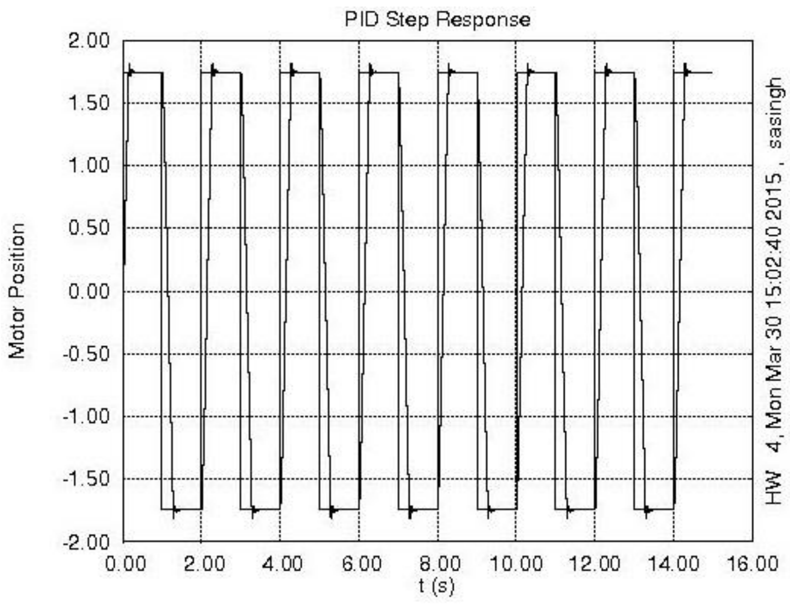

Fig. 10: Hard real-time PID Controller response running under RTAI without CPU load

Part C: Extended Functionality

The final modification to the controller involved implementing a way of changing the parameters during execution, as opposed to letting the system run for the specified run time with the configured parameters unchanged. A user could change one or more parameter by entering an option from the menu, and the resulting change in the motor position would be reflected in the plot. Figure 11 shows an example where the initial K_p value is set to 21.75 and then changed to 2 after three seconds.

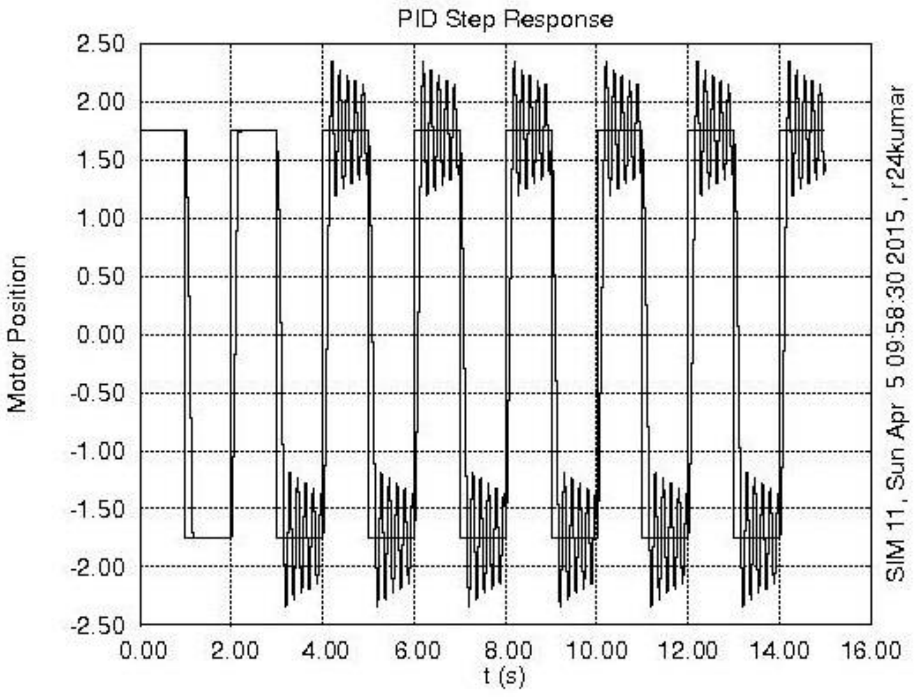

Fig. 11: Response transition after changing K_p from 21.75 to 2.0

A drastic change in the response can be clearly observed once the proportional gain is altered.

References

[1] Chen, Y.C. (2015). *W2015 Project — Real-Time Digital PID Controller*. Laboratory Manual. Retrieved March 27, 2015, from https://courses.ryerson.ca/bbcswebdav/pid-2880762-dt-content-rid-6547381_2/courses/ele709_w15_01/projw15(2).pdf

Appendix – Source code

task2.1.c – P Controller

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <mqueue.h>
#include <errno.h>
#include "dlab.h"

#define MAXS 5000   // Maximum no of samples
                    // Increase value of MAXS for more samples

sem_t data_avail;   // Do not change the name of this semaphore

float theta[MAXS];  // Array for storing motor position
float ref[MAXS];    // Array for storing reference input
int N = 20;

//Info to be passed to the Control thread
struct thread_info {
  float Fs, run_time, Kp, Ti, Td, N;
};

//Make declaring thread info types easier
typedef struct thread_info thread_info_t;

void *Control(void *arg)
{
  thread_info_t *info;

  float Fs, run_time, Kp, T;
  int samples_taken;
  int k = 0;
  float ik_prev = 0, ek_prev = 0, dk_prev = 0;//initialize e(k-1), i(k-1) and d(k-1) to 0


  info = (thread_info_t *)arg;
  Fs = info-> Fs;
  run_time = info-> run_time;
  Kp = info-> Kp;
  T = 1/Fs;     //Period

  samples_taken = (int)(run_time * Fs);



  while(k < samples_taken)
  {
    //Acquire semaphore
    sem_wait(&data_avail);

    //Retrieve motor position (radians)
    float motor_position = EtoR(ReadEncoder());
    //Determine the error rate
    float ek = ref[k] - motor_position;

    //Determine the proportional term
    float uk = Kp*ek;

    DtoA(VtoD(uk)); // It is converted by VtoD and sent to the D/A.
    theta[k] = motor_position;
    k++;
  }
  pthread_exit(NULL);
}


int main(void *arg)
{
  pthread_t control_t;

  //-- Defaults
  float Kp = 1;              // Initialize Kp to 1.
  float N;
  float run_time = 3.0;      // Set the initial run time to 3 seconds.
  int motor_number = 11;     // Motor number for specific workstation
  int Fs = 200;              // Sampling frequency (default = 200 Hz)

  //-- Plotting variables
  int no_of_points = 50;     // # of points when plotting
  int input_t = 0;           // 0 = step function, 1 = square wave

  int i = 0;                 // Iterating integer
  int samples_taken;         // # of samples taken
  char choice;               // User choice from menu
  float step_deg, step_rad;
  float mag, freq, dc;


  while(1)
  {
    //Print out menu
    printf("\n\t -==== Menu ====- \n");
    printf("\tr: Run the control algorithm \n");
    printf("\tp: Change value of Kp \n");
    printf("\tf: Change the sampling frequency \n");
    printf("\tt: Change the run time \n");
    printf("\tu: Change the input type (Step or Square) \n");
    printf("\tg: Plot results on screen \n");
    printf("\th: Save the Plot results in Postscript \n");
    printf("\tq: exit \n");
    scanf("%s", &choice);

     switch(choice){

      //-- Run the control algorithm
      case 'r':
        sem_init(&data_avail, 0, 0);
        if (Initialize(DLAB_SIMULATE, Fs, motor_number) != 0)
        {
          printf("Error initializing..\n");
          exit(-1);
        }

        samples_taken = (int)(run_time * Fs);

        //Prepare info for the control thread
        thread_info_t thread_info;
        thread_info.Fs = Fs;
        thread_info.run_time = run_time;
        thread_info.Kp = Kp;
        thread_info.Ti = Ti;
        thread_info.Td = Td;
        thread_info.N = N;

        //Dispatch control thread
        if (pthread_create(&control_t, NULL, &Control, &thread_info) != 0)
        {
          printf("Error creating Sender thread.\n");
          exit(-1);
        }
        //Wait for control thread to finish
        pthread_join(control_t, NULL);
        //Terminate motor connection
        Terminate();
        //Destroy semaphore
        sem_destroy(&data_avail);
        break;

      //-- Change value of Kp
      case 'p':
        printf("Enter new Kp: \n");
        scanf("%f", &Kp);
        printf("New value of Kp is %f \n", Kp);
      break;

      //-- Change value of sampling_freq
      case 'f':
        printf("Enter new sampling_freq: \n");
        scanf("%f", &Fs);
        printf("New sampling frequency is %f \n", Fs);
      break;

      //-- Change value of run_time
      case 't':
        printf("Enter new run_time: \n");
        scanf("%f", &run_time);
        printf("New run-time is %f \n", run_time);
      break;

      //-- Change the type of inputs
      case 'u':
        printf("Select input type (0 = step | 1 = square): \n");
        scanf("%d", &input_t);
        if (input_t == 0)
        {   //Step Input
          printf("Enter degrees: \n");
          scanf("%f", &step_deg);
          step_rad = step_deg * (PI/180);
          printf("Step value (radians) = %f\n", step_rad);

          //Populate reference array with step input
          for (i = 0; i < MAXS; i++)
          {
            ref[i] = step_rad;
          }

        }
        else if (input_t == 1)
        {
          //Square Input
          printf("Enter magnitude: \n");
          scanf("%f", &mag);
          printf("Enter Frequency: \n");
          scanf("%f", &freq);
          printf("Enter duty cycle: \n");
          scanf("%f", &dc);

          //Populate reference array with square wave input
          Square(ref, MAXS, Fs, mag*(PI/180.0), freq, dc);
        }

      break;

      //-- Plot results on screen
      case 'g':
        // Plot the graph of reference and output vs time on the screen
              no_of_points = run_time * Fs;
        plot(ref, theta, Fs, no_of_points, SCREEN, "Graph Title", "x-axis", "y-axis");

      break;
```

```
      //-- Save the Plot results in Postscript
      case 'h':
        // Save the graph of reference and output vs time in Postscript
        no_of_points = run_time * Fs;
        plot(ref, theta, Fs, no_of_points, PS, "Graph Title", "x-axis", "y-axis");
      break;

      //-- Exit
      case 'q':
        printf("Finished.");
        pthread_exit(NULL);
      break;

      default:
        printf("Invalid choice.\n");
      break;
     }

  }
}
```

task2.3.c – PID Controller

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <mqueue.h>
#include <errno.h>
#include "dlab.h"

#define MAXS 5000   // Maximum no of samples
                    // Increase value of MAXS for more samples

sem_t data_avail;   // Do not change the name of this semaphore

float theta[MAXS];  // Array for storing motor position
float ref[MAXS];    // Array for storing reference input
int N = 20;

//Info to be passed to the Control thread
struct thread_info {
  float Fs, run_time, Kp, Ti, Td, N;
};

//Make declaring thread info types easier
typedef struct thread_info thread_info_t;

void *Control(void *arg)
{
  thread_info_t *info;

  float Fs, run_time, Kp, T, Ti, Td, N;
  int samples_taken;
  int k = 0;
  float ik_prev = 0, ek_prev = 0, dk_prev = 0;//initialize e(k-1), i(k-1) and d(k-1) to 0


  info = (thread_info_t *)arg;
  Fs = info-> Fs;
        run_time = info-> run_time;
        Kp = info-> Kp;
  Ti = info-> Ti;
  Td = info-> Td;
  N  = info-> N;
```

```
      T = 1/Fs;       //Period

  samples_taken = (int)(run_time * Fs);


  while(k < samples_taken)
  {
    //Acquire semaphore
    sem_wait(&data_avail);

    //Retrieve motor position (radians)
    float motor_position = EtoR(ReadEncoder());
    //Determine the error rate
    float ek = ref[k] - motor_position;

    //Determine the proportional term pk
    float pk = Kp*ek;
    //Determine the integral term ik using forward rectangular rule
    float ik = ik_prev + (Kp/Ti)*(ek_prev)*T;  //use i(k-1) + Kp/Ti * e(k-1)*T

    //Determine the derivative term dk using backward rectangular rule
    float dk = (Td/(N*T + Td))*dk_prev + (((Kp*Td*N)/(N*T+Td))*(ek - ek_prev));

    float uk = pk + ik + dk; //calculate control value at=uk-v;
    ik_prev = ik; //Store current ik into prev variable for future use as i(k-1)
    ek_prev = ek; //store current ek into prev variable for future use as e(k-1)
    dk_prev = dk; //store current dk into prev variable for future use as d(k-1)

    DtoA(VtoD(uk)); // It is converted by VtoD and sent to the D/A.
    theta[k] = motor_position;
    k++;
  }
  pthread_exit(NULL);
}



int main(void *arg)
{
  pthread_t control_t;

  //-- Defaults
  float Kp = 1;             // Initialize Kp to 1.
  float Ti, Td;
  float N;
  float run_time = 3.0;     // Set the initial run time to 3 seconds.
  int motor_number = 11;    // Motor number for specific workstation
  int Fs = 200;             // Sampling frequency (default = 200 Hz)

  //-- Plotting variables
  int no_of_points = 50;    // # of points when plotting
  int input_t = 0;          // 0 = step function, 1 = square wave

  int i = 0;                // Iterating integer
  int samples_taken;        // # of samples taken
  char choice;              // User choice from menu
  float step_deg, step_rad;
  float mag, freq, dc;


  while(1)
  {
    //Print out menu
    printf("\n -==== Menu ====- \n");
    printf("\tr: Run the control algorithm \n");
    printf("\tp: Change value of Kp \n");
    printf("\ti: Change Value of Ti \n");
    printf("\td: Change Value of Td \n");
    printf("\tf: Change the sampling frequency \n");
    printf("\tt: Change the run time \n");
```

```c
printf("\tu: Change the input type (Step or Square) \n");
printf("\tg: Plot results on screen \n");
printf("\th: Save the Plot results in Postscript \n");
printf("\tn: Change Value of N \n");
printf("\tq: exit \n");
scanf("%s", &choice);

 switch(choice){

  //-- Run the control algorithm
  case 'r':
    sem_init(&data_avail, 0, 0);
    if (Initialize(DLAB_SIMULATE, Fs, motor_number) != 0)
    {
      printf("Error initializing..\n");
      exit(-1);
    }

    samples_taken = (int)(run_time * Fs);

    //Prepare info for the control thread
    thread_info_t thread_info;
    thread_info.Fs = Fs;
    thread_info.run_time = run_time;
    thread_info.Kp = Kp;
    thread_info.Ti = Ti;
    thread_info.Td = Td;
    thread_info.N = N;

    //Dispatch control thread
    if (pthread_create(&control_t, NULL, &Control, &thread_info) != 0)
    {
      printf("Error creating Sender thread.\n");
      exit(-1);
    }

    //Wait for control thread to finish
    pthread_join(control_t, NULL);
    //Terminate motor connection
    Terminate();
    //Destroy semaphore
    sem_destroy(&data_avail);
    break;

  //-- Change value of Kp
  case 'p':
    printf("Enter new Kp: \n");
    scanf("%f", &Kp);
    printf("New value of Kp is %f \n", Kp);
  break;

  //Change the value of Ti
  case 'i':
    printf("Enter new Ti: \n");
    scanf("%f", &Ti);
    printf("New value of Ti is %f \n", Ti);
  break;

  //Change the value of Td
  case 'd':
    printf("Enter new Td: \n");
    scanf("%f", &Td);
    printf("New value of Td is %f \n", Td);
  break;

  //Change the value of N
  case 'n':
    printf("Enter new N: \n");
    scanf("%f", &N);
    printf("New value of N is %f \n", N);
  break;
```

```
      //-- Change value of sampling_freq
      case 'f':
        printf("Enter new sampling_freq: \n");
        scanf("%f", &Fs);
        printf("New sampling frequency is %f \n", Fs);
      break;

      //-- Change value of run_time
      case 't':
        printf("Enter new run_time: \n");
        scanf("%f", &run_time);
        printf("New run-time is %f \n", run_time);
      break;

      //-- Change the type of inputs
      case 'u':
        printf("Select input type (0 = step | 1 = square): \n");
        scanf("%d", &input_t);
        if (input_t == 0)
        {   //Step Input
          printf("Enter degrees: \n");
          scanf("%f", &step_deg);
          step_rad = step_deg * (PI/180);
          printf("Step value (radians) = %f\n", step_rad);

          //Populate reference array with step input
          for (i = 0; i < MAXS; i++)
          {
            ref[i] = step_rad;
          }

        }
        else if (input_t == 1)
        {
          //Square Input
          printf("Enter magnitude: \n");
          scanf("%f", &mag);
          printf("Enter Frequency: \n");
          scanf("%f", &freq);
          printf("Enter duty cycle: \n");
          scanf("%f", &dc);

          //Populate reference array with square wave input
          Square(ref, MAXS, Fs, mag*(PI/180.0), freq, dc);
        }

      break;

      //-- Plot results on screen
      case 'g':
        // Plot the graph of reference and output vs time on the screen
              no_of_points = run_time * Fs;
        plot(ref, theta, Fs, no_of_points, SCREEN, "PID Step Response", "t (s)", "Motor Posi-
tion");

      break;

      //-- Save the Plot results in Postscript
      case 'h':
        // Save the graph of reference and output vs time in Postscript
        no_of_points = run_time * Fs;
        plot(ref, theta, Fs, no_of_points, PS, "PID Step Response", "t (s)", "Motor Position");
      break;

      //-- Exit
      case 'q':
        printf("Finished.");
        pthread_exit(NULL);
      break;
      default:
        printf("Invalid choice.\n");
      break;
     }

  }
}
```

task2.4.c – PID + Anti-windup

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <mqueue.h>
#include <errno.h>
#include "dlab.h"

#define MAXS 5000   // Maximum no of samples
                    // Increase value of MAXS for more samples

sem_t data_avail;   // Do not change the name of this semaphore

float theta[MAXS];  // Array for storing motor position
float ref[MAXS];    // Array for storing reference input
int N = 20;

//Info to be passed to the Control thread
struct thread_info {
  float Fs, run_time, Kp, Ti, Td, Tt, N;
};

//Make declaring thread info types easier
typedef struct thread_info thread_info_t;

float satblk(float v){
  if(v>=-1.4 && v<=1.4)
    v=v;

  if (v>=1.4)
    v=1.4;

  if (v<=-1.4)
    v=-1.4;

  return (v);
}

void *Control(void *arg)
{
  thread_info_t *info;

  float Fs, run_time, Kp, T, Ti, Td, N;
  int samples_taken;
  int k = 0;
  float pk, ik, dk, ek, motor_position;
  float ik_prev = 0, ek_prev = 0, dk_prev = 0;//initialize e(k-1), i(k-1) and d(k-1) to 0

  //Anti-windup variables
  float Tt = 0.01, at = 0.0, antiwind, v;

  info = (thread_info_t *)arg;
  Fs = info-> Fs;
  run_time = info-> run_time;
  Kp = info-> Kp;
  Ti = info-> Ti;
  Td = info-> Td;
  Tt = info-> Tt;
  N  = info-> N;


      T = 1/Fs;       //Period

  samples_taken = (int)(run_time * Fs);



  while(k < samples_taken)
  {
    //Acquire semaphore
    sem_wait(&data_avail);

    //Retrieve motor position (radians)
    motor_position = EtoR(ReadEncoder());
    //Determine the error rate
    ek = ref[k] - motor_position;

        //Determine the proportional term pk
    pk = Kp*ek;

        if (k == 0){
          ik = 0;
          dk = (Kp * Td*N/(N * T + Td))*(ek);
        }
        else {
         //Determine anti-wind
         antiwind = (at*T)/Tt;
        //Determine the integral term ik using forward rectangular rule
        //and combine with anti-wind
    ik = ik_prev + (Kp/Ti)*(ek_prev)*T + antiwind;
        //Determine the derivative term dk using backward rectangular rule
        dk = (Td/(N*T + Td))*dk_prev + (((Kp*Td*N)/(N*T+Td))*(ek - ek_prev));
        }

    v = pk + ik + dk; //calculate control value
        float uk = satblk(v);

        at = uk - v;

    ik_prev = ik;  //Store current ik into prev variable for future use as i(k-1)
    ek_prev = ek;  //store current ek into prev variable for future use as e(k-1)
    dk_prev = dk;  //store current dk into prev variable for future use as d(k-1)

    DtoA(VtoD(uk)); // It is converted by VtoD and sent to the D/A.
    theta[k] = motor_position;
    k++;
  }
  pthread_exit(NULL);
}



int main(void *arg)
{
  pthread_t control_t;

  //-- Defaults
  float Kp = 21.75;            // Initialize Kp to 1.
  float Ti = 0.025, Td = 0.00625;
  float N = 20;
  float run_time = 20.0;      // Set the initial run time to 3 seconds.
  int motor_number = 11;     // Motor number for specific workstation
  int Fs = 200;              // Sampling frequency (default = 200 Hz)

  //-- Plotting variables
  int no_of_points = 50;     // # of points when plotting
  int input_t = 0;           // 0 = step function, 1 = square wave
  int i = 0;                  // Iterating integer
  int samples_taken;          // # of samples taken
  char choice;                // User choice from menu
  float step_deg, step_rad;
  float mag, freq, dc;


  while(1)
  {
    //Print out menu
    printf("\n -==== Menu ====- \n");
    printf("\tr: Run the control algorithm \n");
    printf("\tp: Change value of Kp \n");
    printf("\ti: Change Value of Ti \n");
    printf("\td: Change Value of Td \n");
    printf("\tf: Change the sampling frequency \n");
    printf("\tt: Change the run time \n");
    printf("\tu: Change the input type (Step or Square) \n");
    printf("\tg: Plot results on screen \n");
    printf("\th: Save the Plot results in Postscript \n");
    printf("\tn: Change Value of N \n");
    printf("\tq: exit \n");
    scanf("%s", &choice);

     switch(choice){

      //-- Run the control algorithm
      case 'r':
        sem_init(&data_avail, 0, 0);
        if (Initialize(DLAB_SIMULATE, Fs, motor_number) != 0)
        {
          printf("Error initializing..\n");
          exit(-1);
        }

        samples_taken = (int)(run_time * Fs);

        //Prepare info for the control thread
        thread_info_t thread_info;
        thread_info.Fs = Fs;
        thread_info.run_time = run_time;
        thread_info.Kp = Kp;
        thread_info.Ti = Ti;
        thread_info.Td = Td;
        thread_info.N = N;
               thread_info.Tt = 0.01;

        //Dispatch control thread
        if (pthread_create(&control_t, NULL, &Control, &thread_info) != 0)
        {
          printf("Error creating Sender thread.\n");
          exit(-1);
        }

        //Wait for control thread to finish
        pthread_join(control_t, NULL);
        //Terminate motor connection
        Terminate();
        //Destroy semaphore
        sem_destroy(&data_avail);
        break;

      //-- Change value of Kp
      case 'p':
        printf("Enter new Kp: \n");
        scanf("%f", &Kp);
        printf("New value of Kp is %f \n", Kp);
      break;

      //Change the value of Ti
```

```
case 'i':
  printf("Enter new Ti: \n");
  scanf("%f", &Ti);
  printf("New value of Ti is %f \n", Ti);
break;

//Change the value of Td
case 'd':
  printf("Enter new Td: \n");
  scanf("%f", &Td);
  printf("New value of Td is %f \n", Td);
break;

//Change the value of N
case 'n':
  printf("Enter new N: \n");
  scanf("%f", &N);
  printf("New value of N is %f \n", N);
break;

//-- Change value of sampling_freq
case 'f':
  printf("Enter new sampling_freq: \n");
  scanf("%f", &Fs);
  printf("New sampling frequency is %f \n", Fs);
break;

//-- Change value of run_time
case 't':
  printf("Enter new run_time: \n");
  scanf("%f", &run_time);
  printf("New run-time is %f \n", run_time);
break;

//-- Change the type of inputs
case 'u':
  printf("Select input type (0 = step | 1 = square): \n");
  scanf("%d", &input_t);
  if (input_t == 0)
  {   //Step Input
    printf("Enter degrees: \n");
    scanf("%f", &step_deg);
    step_rad = step_deg * (PI/180);
    printf("Step value (radians) = %f\n", step_rad);

    //Populate reference array with step input
    for (i = 0; i < MAXS; i++)
    {
      ref[i] = step_rad;
    }

  }
  else if (input_t == 1)
  {
    //Square Input
    printf("Enter magnitude: \n");
    scanf("%f", &mag);
    printf("Enter Frequency: \n");
    scanf("%f", &freq);
    printf("Enter duty cycle: \n");
    scanf("%f", &dc);

    //Populate reference array with square wave input
    Square(ref, MAXS, Fs, mag*(PI/180.0), freq, dc);
  }

break;

//-- Plot results on screen
case 'g':
  // Plot the graph of reference and output vs time on the screen
```

```
              no_of_points = run_time * Fs;
          plot(ref, theta, Fs, no_of_points, SCREEN, "PID Response", "t (s)", "Motor Position");

        break;

        //-- Save the Plot results in Postscript
        case 'h':
          // Save the graph of reference and output vs time in Postscript
          no_of_points = run_time * Fs;
          plot(ref, theta, Fs, no_of_points, PS, "PID Response", "t (s)", "Motor Position");
        break;

        //-- Exit
        case 'q':
          printf("Finished.");
          pthread_exit(NULL);
        break;

        default:
          printf("Invalid choice.\n");
        break;
       }

   }
}
```

task4.c – Extended functionality

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <mqueue.h>
#include <errno.h>
#include "dlab.h"

#define MAXS 5000   // Maximum no of samples
                    // Increase value of MAXS for more samples
pthread_mutex_t mutex_stop = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_modification = PTHREAD_MUTEX_INITIALIZER;
int STOP = 0;
int changed = 0;
sem_t data_avail;   // Do not change the name of this semaphore

float theta[MAXS];  // Array for storing motor position
float ref[MAXS];    // Array for storing reference input
int N = 20;

//Info to be passed to the Control thread
struct thread_info {
  float Fs, run_time, Kp, Ti, Td, Tt, N;
};

//Make declaring thread info types easier
typedef struct thread_info thread_info_t;

float satblk(float v){
  if(v>=-1.4 && v<=1.4)
    v=v;

  if (v>=1.4)
    v=1.4;

  if (v<=-1.4)
    v=-1.4;
  return (v);
}

void *Control(void *arg)
{
  thread_info_t *info;

  float Fs, run_time, Kp, T, Ti, Td, N;
  int samples_taken;
  int k = 0, starting_sample=0;
  float pk, ik, dk, ek, motor_position;
  float ik_prev = 0, ek_prev = 0, dk_prev = 0;//initialize e(k-1), i(k-1) and d(k-1) to 0

  //Anti-windup variables
  float Tt = 0.01, at = 0.0, antiwind, v;
  int STOP_INDICATED = 1;
  info = (thread_info_t *)arg;
  Fs = info-> Fs;
  run_time = info-> run_time;
  Kp = info-> Kp;
  Ti = info-> Ti;
  Td = info-> Td;
  Tt = info-> Tt;
  N  = info-> N;


  T = 1/Fs;  //Period

  samples_taken = (int)(run_time * Fs);


  while(STOP_INDICATED){
       k = starting_sample; //Reset K to iterate samples again from starting_samples
       //this makes sure that we only replace the thetas after 3seconds of sampling IF changed
== 1
      pthread_mutex_lock(&mutex_stop); //protect the following data
      if(STOP == 1){
        STOP_INDICATED = 0;
      }
      pthread_mutex_unlock(&mutex_stop); //protect the following data
      while(k < samples_taken)
      {
        //Acquire semaphore
        sem_wait(&data_avail);

        //Retrieve motor position (radians)
        motor_position = EtoR(ReadEncoder());
        //Determine the error rate
        ek = ref[k] - motor_position;

      //Determine the proportional term pk
        pk = Kp*ek;

      if (k == 0){
        ik = 0;
        dk = (Kp * Td*N/(N * T + Td))*(ek);
      }
      else {
       //Determine anti-wind
       antiwind = (at*T)/Tt;
      //Determine the integral term ik using forward rectangular rule
      //and combine with anti-wind
        ik = ik_prev + (Kp/Ti)*(ek_prev)*T + antiwind;
      //Determine the derivative term dk using backward rectangular rule
      dk = (Td/(N*T + Td))*dk_prev + (((Kp*Td*N)/(N*T+Td))*(ek - ek_prev));
      }

        v = pk + ik + dk;                            //calculate control value
      float uk = satblk(v);

      at = uk - v;
```

```
        ik_prev = ik; //Store current ik into prev variable for future use as i(k-1)
        ek_prev = ek; //store current ek into prev variable for future use as e(k-1)
        dk_prev = dk; //store current dk into prev variable for future use as d(k-1)

        DtoA(VtoD(uk)); // It is converted by VtoD and sent to the D/A.
        theta[k] = motor_position;
        k++;

          pthread_mutex_lock(&mutex_modification); //protect the following data
        if(changed == 1){
          starting_sample = 3.0*Fs;//new starting location before updating values
          k = starting_sample;//start at next samples
          Fs = info->Fs;
         // samples_taken = (int)(run_time * Fs);
          Kp = info->Kp;
          Ti = info->Ti;
          Td = info->Td;
          N = info->N;
          printf("System updated: \n");
          printf("FS %f, Kp %f, Ti %f, Td %f, N %f\n", Fs, Kp, Ti, Td, N);
          changed = 0;
        }
        pthread_mutex_unlock(&mutex_modification); //protect the following data
      }
  }
  pthread_exit(NULL);
}



int main(void *arg)
{
   //Prepare info for the control thread
  thread_info_t thread_info;
  pthread_t control_t;

  //-- Defaults
  float Kp = 21.75;             // Initialize Kp to 1.
  float Ti = 0.025, Td = 0.00625;
  float N = 20;
  float run_time = 3.0; // Set the initial run time to 3 seconds.
  int motor_number = 11;    // Motor number for specific workstation
  int Fs = 200;             // Sampling frequency (default = 200 Hz)

  //-- Plotting variables
  int no_of_points = 50;    // # of points when plotting
  int input_t = 0;          // 0 = step function, 1 = square wave

  int i = 0;                // Iterating integer
  int samples_taken;        // # of samples taken
  char choice;              // User choice from menu
  float step_deg = 50, step_rad;//50 mag for step degrees
  float mag = 50, freq = 0.5, dc = 50; //Default magnitude, frequency and duty cycle


  while(1)
  {
    //Print out menu
    printf("\n -==== Menu ====- \n");
    printf("\tr: Run the control algorithm continuously.\n");
    printf("\ts: Stop the control algorithm \n");
    printf("\tp: Change value of Kp \n");
    printf("\ti: Change Value of Ti \n");
    printf("\td: Change Value of Td \n");
    printf("\tf: Change the sampling frequency \n");
    printf("\tt: Change the run time \n");
    printf("\tu: Change the input type (Step or Square) \n");
    printf("\tg: Plot results on screen \n");
    printf("\th: Save the Plot results in Postscript \n");
    printf("\tn: Change Value of N \n");
    printf("\tq: exit \n");
    scanf("%s", &choice);

     switch(choice){

      //-- Run the control algorithm
      case 'r':
        sem_init(&data_avail, 0, 0);
        if (Initialize(DLAB_SIMULATE, Fs, motor_number) != 0)
        {
          printf("Error initializing..\n");
          exit(-1);
        }

        samples_taken = (int)(run_time * Fs);


        thread_info.Fs = Fs;
        thread_info.run_time = run_time;
        thread_info.Kp = Kp;
        thread_info.Ti = Ti;
        thread_info.Td = Td;
        thread_info.N = N;
         thread_info.Tt = 0.01;
         pthread_mutex_lock(&mutex_stop); //protect the following data
          STOP = 0;
         pthread_mutex_unlock(&mutex_stop); //protect the following data
        //Dispatch control thread
        if (pthread_create(&control_t, NULL, &Control, &thread_info) != 0)
        {
          printf("Error creating Sender thread.\n");
          exit(-1);
        }
        break;

    //Stop the continuous run
      case 's':
        pthread_mutex_lock(&mutex_stop); //protect the following data
          STOP = 1;
         pthread_mutex_unlock(&mutex_stop); //protect the following data

       //Wait for control thread to finish
        pthread_join(control_t, NULL);
        Terminate();
        sem_destroy(&data_avail);
        printf("Simulation stopped!\n");
        break;
      //-- Change value of Kp
      case 'p':
        printf("Enter new Kp: \n");
        scanf("%f", &Kp);
         pthread_mutex_lock(&mutex_modification); //protect the following data
             thread_info.Kp = Kp;
             changed = 1;
        pthread_mutex_unlock(&mutex_modification); //protect the following data
        printf("New value of Kp is %f \n", Kp);
      break;

      //Change the value of Ti
      case 'i':
        printf("Enter new Ti: \n");
        scanf("%f", &Ti);
         pthread_mutex_lock(&mutex_modification); //protect the following data
             thread_info.Ti = Ti;
             changed = 1;
        pthread_mutex_unlock(&mutex_modification); //protect the following data
        printf("New value of Ti is %f \n", Ti);
      break;

      //Change the value of Td
      case 'd':
```

```
  printf("Enter new Td: \n");
  scanf("%f", &Td);
   pthread_mutex_lock(&mutex_modification); //protect the following data
       thread_info.Td = Td;
       changed = 1;
 pthread_mutex_unlock(&mutex_modification); //protect the following data
  printf("New value of Td is %f \n", Td);
break;

//Change the value of N
case 'n':
  printf("Enter new N: \n");
  scanf("%f", &N);
   pthread_mutex_lock(&mutex_modification); //protect the following data
       thread_info.N = N;
       changed = 1;
  pthread_mutex_unlock(&mutex_modification); //protect the following data
  printf("New value of N is %f \n", N);
break;

//-- Change value of sampling_freq
case 'f':
  printf("Enter new sampling_freq: \n");
  scanf("%f", &Fs);
  pthread_mutex_lock(&mutex_modification); //protect the following data
       thread_info.Fs = Fs;
       changed = 1;
  pthread_mutex_unlock(&mutex_modification); //protect the following data
  printf("New sampling frequency is %f \n", Fs);
break;

//-- Change value of run_time
case 't':
  printf("Enter new run_time: \n");
  scanf("%f", &run_time);
  printf("New run-time is %f \n", run_time);
break;

//-- Change the type of inputs
case 'u':
  printf("Select input type (0 = step | 1 = square): \n");
  scanf("%d", &input_t);
  if (input_t == 0)
  {   //Step Input
    printf("Enter degrees: \n");
    scanf("%f", &step_deg);
    step_rad = step_deg * (PI/180);
    printf("Step value (radians) = %f\n", step_rad);

    //Populate reference array with step input
    for (i = 0; i < MAXS; i++)
    {
      ref[i] = step_rad;
    }

  }
  else if (input_t == 1)
  {
    //Square Input
    printf("Enter magnitude: \n");
    scanf("%f", &mag);
    printf("Enter Frequency: \n");
    scanf("%f", &freq);
    printf("Enter duty cycle: \n");
    scanf("%f", &dc);

    //Populate reference array with square wave input
    Square(ref, MAXS, Fs, mag*(PI/180.0), freq, dc);
  }

break;
```

```
      //-- Plot results on screen
      case 'g':
        // Plot the graph of reference and output vs time on the screen
        no_of_points = run_time * Fs;
        plot(ref, theta, Fs, no_of_points, SCREEN, "PID Response", "t (s)", "Motor Position");

      break;

      //-- Save the Plot results in Postscript
      case 'h':
        // Save the graph of reference and output vs time in Postscript
        no_of_points = run_time * Fs;
        plot(ref, theta, Fs, no_of_points, PS, "PID Response", "t (s)", "Motor Position");
      break;

      //-- Exit
      case 'q':
        printf("Finished.");
        pthread_exit(NULL);
      break;

      default:
        printf("Invalid choice.\n");
      break;
     }

  }
}
```