CS331 Lab Report #5

Andrew Janssen (apjansse@uiuc.edu)
Michael Tucknott (tucknott@uiuc.edu)
Terrence Janas (tjanas@uiuc.edu)

Thursday 5 pm
Workstation: the one across from emb6 ;-)
Username: group26

Program implementation:

In this lab we used the threads.cc code from lab4 to finish the marble balance program. We added an extra shared global variable, `unsigned char vf_vd[4]`, which stores the fill and drain feedback voltages each as 12-bit values. Also, we added PID control calculations to the display thread. Our calculations were simple, given the fact that our sampling period was constant at 100 ms. The angle derivative was derived by taking the current angle, the previous angle, and finding the slope. The angle integral was calculated by finding the area bounded by the current and previous angles, and finding the sum of that area and the previous integral value. The rest of our PID calculations were taken from the lab5 handout. In our io thread, we added code that reads from the new shared feedback buffer, and outputs the values to the drain and fill D/A channels, appropriately. Everything else is the same from lab4.

Problems encountered:

This lab went smoothly. The only part that took any significant amount of time was finding values that balanced the marbles quickly. Basically we only had to add another shared variable for feedback, resulting in bidirectional communication between the display and io threads. PID control calculations added to the display thread were trivial, since nearly all required information was provided on the lab5 handout.

Team Member Contributions:

Michael: Helped develop the code, debugged.
Andrew: Wrote final report, helped develop the code, debugged.
Terrence: Helped developed the code, coded, debugged.

Lab Questions:

1. Kp = 4.0, Kd = 2.0, Ki = 0.3
2. Honestly, in the beginning we just tried guessing (we wanted to see what it would do with all sorts of values). After that we started inching towards our final values by setting Ki=0, picking a Kp and inching up Kd until we were happy with it (or backing up and finding a new Kp). Then we started fine-tuning to find a Ki value. The process took about 20 minutes and now the balance settles most of the way in about 15 seconds.
3. The real life model reacts loosely like MATLAB model we worked with in lab 3. For example, having done the MATLAB model we knew what we should expect Kp, Kd, and Ki to do but it was not precise enough to use the numbers we found in the real balance. We think the MATLAB model does a decent job modeling the real one, but it also reinforces the idea that modeling physical things is much more complex than it first seems.

```
/************************ BEGIN threads.cc ************************/

#include <stdlib.h>
#include <iostream.h>
#include <stdio.h>
#include <time.h>
#include <signal.h>
#include <string.h>
#include <mqueue.h>
#include <pthread.h>
#include <math.h>

pthread_mutex_t wave_spec_mutex;    // protects buffer[]
unsigned char buffer[2];            // loByte, hiByte of current voltage
unsigned short digitalL, digitalR;  // digital left/right bounds of seesaw

sigset_t io_signals;         // signal set used by SIGWAIT for io
sigset_t display_signals;    // signal set used by SIGWAIT for display

float kp, kd, ki;            // LAB5: coefficients from command line
unsigned char vf_vd[4];      // vf_loByte, vf_hiByte, vd_loByte, vd_hiByte
pthread_mutex_t feedback_mutex;     // protects vf_vd[]



extern "C"
{
  // ASM function that outputs voltage to D/A
  void hw_out(unsigned int port, unsigned char val);
  unsigned char hw_in(unsigned int port);
}



// function executed by display thread
void* display(void* args)
{
  int status;
  unsigned char loByte, hiByte;  // data from A/D controller
  unsigned short digitalV;       // 12-bit raw voltage of seesaw
  //float voltage;                 // current seesaw voltage
  float old_angle = -15.0;       // previous seesaw angle
  float angle;                   // current seesaw angle

  /* variables added for LAB5 */
  float ai = 0.0;        // integral of the angle
  float ad;              // derivative of the angle
  float fp,fd,fi;        // feedback (proportional,derivative,integral)
  float feedback;        // total PID feedback
  float vf, vd;          // fill & drain motor voltages to feedback
  unsigned char vf_loByte, vf_hiByte;     // vf data to send to D/A
  unsigned char vd_loByte, vd_hiByte;     // vd data to send to D/A
  unsigned short digital_vf;              // 12-bit raw vf voltage
  unsigned short digital_vd;              // 12-bit raw vd voltage
```

```
while(1)
{
  sigwait(&display_signals,&status);  // wait until SIGUSR2 is received
  if (status == SIGINT) {
    // CTRL-C was pressed
    // Send zero volts to both motors before exiting
    hw_out( 0x304,0 );   hw_out( 0x305,0x80 );
    hw_out( 0x306,0 );   hw_out( 0x307,0x80 );
    exit(0);
  }

  status = pthread_mutex_lock(&wave_spec_mutex);
  /***  enter critical section, read global variable  ***/
   loByte = buffer[0];
   hiByte = buffer[1];
  /***  exit critical section  ***/
  status = pthread_mutex_unlock(&wave_spec_mutex);

  // convert loByte and hiByte into one 12-bit value
  digitalV = hiByte & 0x00FF;
  digitalV = (digitalV << 4) & (0x0FF0);
  digitalV = ((loByte >>4)&(0x000F)) | digitalV;

  // calculate current voltage of seesaw
  //voltage = (((float)digitalV/4095.0)*20.0)-10.0;

  // calculate current angle of seesaw
  angle=(((float)(digitalV-digitalL)/(float)(digitalR-digitalL))*30.0)-15.0;
  // calculate derivative of angle
  ad = (angle - old_angle)/0.1;
  // calculate integral of angle
  ai = ai + 0.5*(angle + old_angle)*0.1;


  // now we can update the value of old_angle...
  old_angle = angle;


  // feedback from proportional control
  fp = (-1.0) * kp * angle;
  // feedback from derivative control
  fd = (-1.0) * kd * ad;
  // feedback from integral control
  fi = (-1.0) * ki * ai;
  // F = -Kp*(angle) - Kd*<derivative(angle)> - Ki*<integral(angle)>
  feedback = fp + fd + fi;


  if (feedback < 0) {              // compute the fill motor voltage and
    vf = 1.0;                      // the drain motor voltage
    vd = 1.0 + fabsf(feedback);
    if (vd > 4.0)
      vd = 4.0;
  }
  else {
    vf = 1.0 + fabsf(feedback);    // 1.0 + |feedback|
    vd = 1.0;
    if (vf > 4.0)
      vf = 4.0;
  }
```

```
    // convert final vf to 12-bit raw values for the D/A controller
    digital_vf = (unsigned short) ( ((vf + 5.0)*4095.0)/10.0 );
    vf_loByte = (digital_vf <<4) & (0xF0);
    vf_hiByte = (digital_vf >>4) & (0xFF);

    // convert final vd to 12-bit raw values for the D/A controller
    digital_vd = (unsigned short) ( ((vd + 5.0)*4095.0)/10.0 );
    vd_loByte = (digital_vd <<4) & (0xF0);
    vd_hiByte = (digital_vd >>4) & (0xFF);

    status = pthread_mutex_lock(&feedback_mutex);
    /***  enter critical section, write global variable  ***/
    vf_vd[0] = vf_loByte;
    vf_vd[1] = vf_hiByte;
    vf_vd[2] = vd_loByte;
    vf_vd[3] = vd_hiByte;
    /***  exit critical section  ***/
    status = pthread_mutex_unlock(&feedback_mutex);

    printf("a=%.2f  ad=%.2f  ai=%.2f  fp=%.2f  fd=%.2f  fi=%.2f  f=%.2f  vd=%.2f
            vf=%.2f\n", angle,ad,ai,fp,fd,fi,feedback,vd,vf);
  }
}



// function executed by main thread after SIGUSR1 interrupt occurs
void io_timer_handler()
{
  int status;
  unsigned char loByte, hiByte;

  hw_out(0x300,0x00);   // start A/D conversion
  // Cheap hack for getting stable A/D input
  // input loByte from BASE+0, hiByte from BASE+1
  while(hw_in(0x308) & 0x80 != 0) { };
  loByte = hw_in(0x300);
  hiByte = hw_in(0x301);


  /*** send LAB5 feedback data to the controller ***/
  status = pthread_mutex_lock(&feedback_mutex);
  // enter critical section, read global variable
   hw_out(0x304, vf_vd[0]);   // fill, loByte
   hw_out(0x305, vf_vd[1]);   // fill, hiByte
   hw_out(0x306, vf_vd[2]);   // drain, loByte
   hw_out(0x307, vf_vd[3]);   // drain, hiByte
  // exit critical section
  status = pthread_mutex_unlock(&feedback_mutex);


  status = pthread_mutex_lock(&wave_spec_mutex);
  // enter critical section, write global variable
   buffer[0] = loByte;
   buffer[1] = hiByte;
  // exit critical section
  status = pthread_mutex_unlock(&wave_spec_mutex);
}
```

```
int main(int argc, char *argv[])
{
  int status;
  char prompt;
  unsigned char hiByteR, loByteR, hiByteL, loByteL;

  pthread_mutex_init(&wave_spec_mutex, NULL);  // initialize mutexes
  pthread_mutex_init(&feedback_mutex, NULL);
  pthread_t wave_thread_id;                    // thread ID

  // initialize the A/D, D/A controller
  hw_out( 0x309, 0 );   // disable int & DMA
  hw_out( 0x308, 0 );   // clear trigger flip flop for interrupt
  hw_out( 0x302, 0 );   // select analog input channel

  // send zero volts to both motors
  hw_out( 0x304,0 );   hw_out( 0x305,0x80 );
  hw_out( 0x306,0 );   hw_out( 0x307,0x80 );

  cout << "**********************************\n"
       << "**********************************\n"
       << "**                              **\n"
       << "**    CS 331 Lab 5: Group 26    **\n"
       << "**                              **\n"
       << "**********************************\n"
       << "**********************************\n\n";

  if (argc != 4)
  {
    fprintf(stderr,"Usage: lab5 <Kp> <Kd> <Ki>\n");
    exit(1);
  }

  kp = atof(argv[1]) / 50.0;     // LAB5: convert command line args to
  kd = atof(argv[2]) / 50.0;     // floating-point coefficients
  ki = atof(argv[3]) / 50.0;

  vf_vd[0] = 0;     vf_vd[2] = 0;     // initialize vd,vf to 0 volts
  vf_vd[1] = 0x80;  vf_vd[3] = 0x80;


  cout << "Tilt the seesaw all the way to the right, and hit ENTER...\n";
  cin.get(prompt);

  hw_out(0x300,0x00);                    // start A/D conversion
  // Cheap hack for getting stable A/D input
  // input loByte from BASE+0, hiByte from BASE+1
  while(hw_in(0x308) & 0x80 !=0) { };
  loByteR = hw_in(0x300);                // get calibration data,
  hiByteR = hw_in(0x301);                // right-side of seesaw

  cout << "Tilt the seesaw all the way to the left, and hit ENTER...\n";
  cin.get(prompt);

  hw_out(0x300,0x00);                    // start A/D conversion
  // Cheap hack for getting stable A/D input
  // input loByte from BASE+0, hiByte from BASE+1
  while(hw_in(0x308) & 0x80 !=0) { };
  loByteL = hw_in(0x300);                // get calibration data,
  hiByteL = hw_in(0x301);                // right-side of seesaw
```

```
// convert "left" calibration voltage data into 12 bit discrete value
digitalL = hiByteL & 0x00FF;
digitalL = (digitalL << 4) & (0x0FF0);
digitalL = ((loByteL >>4)&(0x000F)) | digitalL;

// convert "right" calibration voltage data into 12 bit discrete value
digitalR = hiByteR & 0x00FF;
digitalR = (digitalR << 4) & (0x0FF0);
digitalR = ((loByteR >>4)&(0x000F)) | digitalR;


struct itimerspec io_timer_spec, display_timer_spec,
                  io_old_timer_spec, display_old_timer_spec;

// create io timer
timer_t io_timer;
struct sigevent io_timer_evp;
memset(&io_timer_evp, 0, sizeof(io_timer_evp));
io_timer_evp.sigev_signo = SIGUSR1;        // io responds to SIGUSR1 signal
io_timer_evp.sigev_notify = SIGEV_SIGNAL;
timer_create(CLOCK_REALTIME, &io_timer_evp, &io_timer);

// set io timer settings (goes off 10 times per second)
io_timer_spec.it_value.tv_sec = 0;   // first SIGUSR1 will go off only 1 ns
io_timer_spec.it_value.tv_nsec = 1;   // after the timer is started
io_timer_spec.it_interval.tv_sec = 0;   // our timer interval is
io_timer_spec.it_interval.tv_nsec = 200000000;

// create display timer
timer_t display_timer;
struct sigevent display_timer_evp;
memset(&display_timer_evp, 0, sizeof(display_timer_evp));
display_timer_evp.sigev_signo = SIGUSR2;   // display responds to SIGUSR2
display_timer_evp.sigev_notify = SIGEV_SIGNAL;
timer_create(CLOCK_REALTIME, &display_timer_evp, &display_timer);

// set display timer settings (goes off 10 times per second)
display_timer_spec.it_value.tv_sec = 0;   // 1st SIGUSR2 will go off only
display_timer_spec.it_value.tv_nsec = 1;  // 1 ns after the timer is started
display_timer_spec.it_interval.tv_sec = 0;   // our timer interval is
display_timer_spec.it_interval.tv_nsec = 200000000;

sigemptyset(&io_signals);              // Create signal set for SIGWAIT,
sigaddset(&io_signals, SIGUSR1);       //  used by io.
sigaddset(&io_signals, SIGINT);
sigemptyset(&display_signals);         // Create signal set for SIGWAIT,
sigaddset(&display_signals, SIGUSR2);  //  used by display thread.
sigaddset(&display_signals, SIGINT);

// start io timer
timer_settime(io_timer,0,&io_timer_spec,&io_old_timer_spec);

// start display timer
timer_settime(display_timer,0,&display_timer_spec,&display_old_timer_spec);

io_timer_handler();  // store initial data in the buffer

// spawn the display thread...
status = pthread_create(&wave_thread_id, NULL, display, NULL);
```

```
  while(1) {                        // pass raw data to buffer
    sigwait(&io_signals,&status);   // 10 times per second, until
    if (status == SIGINT) {         // user presses CTRL-C
      // CTRL-C was pressed
      // Send zero volts to both motors before exiting
      hw_out( 0x304,0 );   hw_out( 0x305,0x80 );
      hw_out( 0x306,0 );   hw_out( 0x307,0x80 );
      exit(0);
    }
    io_timer_handler();
  }
}
```

/********************** END threads.cc ************************/**

/********************** BEGIN iodas.cc ************************/**

```
void hw_out(unsigned int port, unsigned char val)
{
   __asm__ __volatile__ ("outb %b0, %w1"
                         : /* no output */
                         : "a" (val), "d" (port) );
}

unsigned char hw_in(unsigned int port)
{
   unsigned char val;
   __asm__ __volatile__ ("inb %w1, %b0"
                         : "=a" (val)
                         : "d" (port) );
   return val;
}
```

/********************** END iodas.cc ************************/**