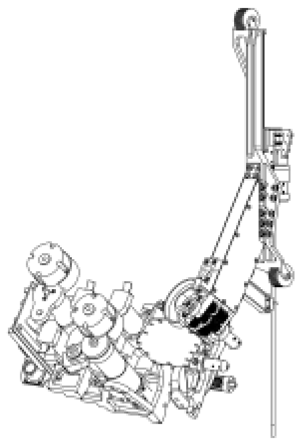
EE449 Project Milestone Report V

Cable Winding Controller for Surgical Robot Arm Capstans



Prepared for
BioRobotics Laboratory
Department of Electrical Engineering
University of Washington
Seattle, WA

Prepared by
Imam Tjung and Kiran Thomas

June 11, 2010

Executive Summary

The BioRobotics Lab at the University of Washington, Seattle is building new RAVEN surgical robots for testing and development purposes. These robots consist of a complex cable mechanism that requires following a tedious cabling procedure during construction and maintenance. It was determined that a controller could be used to improve the cabling procedure by involving the motors themselves during cabling.

The EE 449 project's goal is to design a controller that will assist in the cabling procedure. This was accomplished by using the various hardware of the RAVEN system and software. The project duration was 3 months and followed a specific program. The first step was to model the system and then simulate the model. Next the state space model was developed and used to design a controller based on the specifications obtained from the customer. This controller design was then implemented on a test rig which is similar to the actual robot.

The entire project was implemented successfully with the exception of a foot pedal. The foot pedal helps the user by not having to use the keyboard to send various commands to the computer. Even though the cabling procedure can be completed without the foot pedal mechanism, the ease of use of the controller is limited by the fact that the user will be dependent on access to the computer keyboard.

The foot pedal is scheduled to be implemented within one week of submission of this report. Once this is complete the code and instructions will be handed over to the Bio-Robotic Lab, so that it can be send to a student in UC Santa Cruz who will be constructing seven new RAVEN robots.

Table of Contents

List of Illustrations

1. Project Description

The RAVEN surgical robot developed by the BioRobotics lab has two robot arms each of which use a complex pulley system to move the arms in different directions. The pulley system is powered by separate motors that have one steel cable attached onto its shaft. The reels have threads on them into which the steel cable wraps onto. These reels are located at hard to reach places, which makes the initial cabling a tedious process. Also anytime a person has to remove and replace the cable during maintenance.

Our project originated from this issue during cabling. Using the motors to assist in the cabling process would benefit any person involved with cabling. So the idea was to create a controller for the motors to wind up the cable onto the motor shafts using a controller and hold that position while the other end was wrapped around. The goal in this project is to create a closed loop controller that will perform an automatic cable winding for the RAVEN surgical robot. The system should be able to wind the cable on the capstan (reel on which the motor shaft is attached to) at a specified velocity. The user should be able to stop the motor once it has wound using a footpedal and wind it in the opposite direction whenever required. In this report, we include the discussion about our customer and his expectations of the project. We will also explain about our plant, actuator, sensor, and control resources. In addition, we will cover our results and conclusions.

In designing a control system the general flow of tasks are system modeling, simulation, control design, controller performance and robustness testing. This final milestone report covers the entire progress of the Automatic Cable Winding for Surgical Robot Arms project. The purpose of this report is to show the progress of the project leading to the final controller.

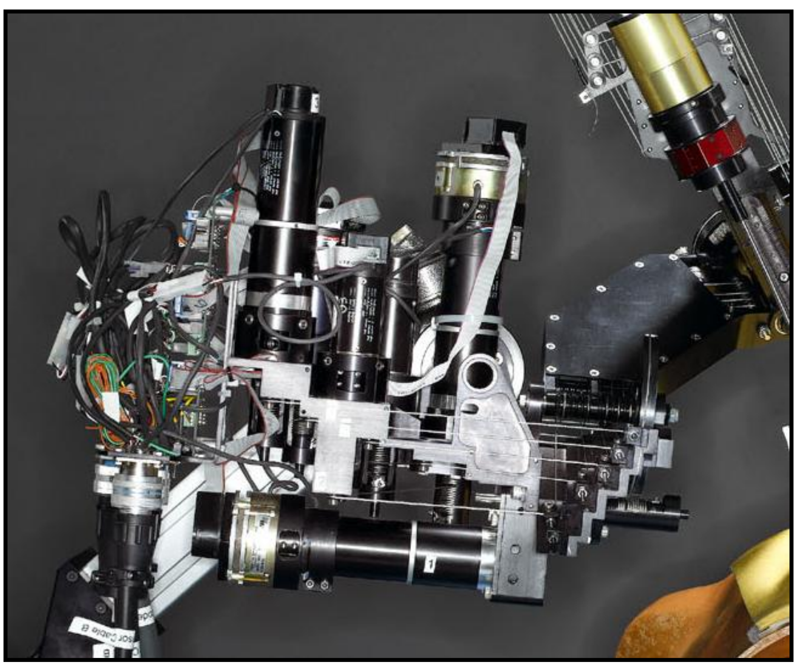


Figure 1. Close up of the motor cluster on the RAVEN

2. Literature Review

The BioRobotics laboratory, Department of Electrical Engineering, of the University of Washington, Seattle mainly focuses on haptic devices and remote surgery. One of the main projects of the Lab is the development of remote surgical technology. Haptic Devices are high performance mechantronic (computer and mechanical) devices that allow the physical interaction between humans and computer models. Surgical technology is the idea to create robotic devices that makes the surgeon be able to perform a surgery safely and effectively. One of these initiatives is the RAVEN remote surgical robot partly funded by the military. The idea behind the robot is to deliver immediate surgical assistance to battlefields, disaster areas, rural areas etc. The project has accelerated in recent years and is now under rigorous testing. To enable more testing and improvements, seven new robots are being constructed by students in UC Santa Cruz. The customer for the project is the BioRobotics Lab. They will hand over the software and instructions to the students in UC Santa Cruz. The customer also wants the system to turn at a certain direction with a specified number of turns. The system also should be able to rotate at a specified speed.

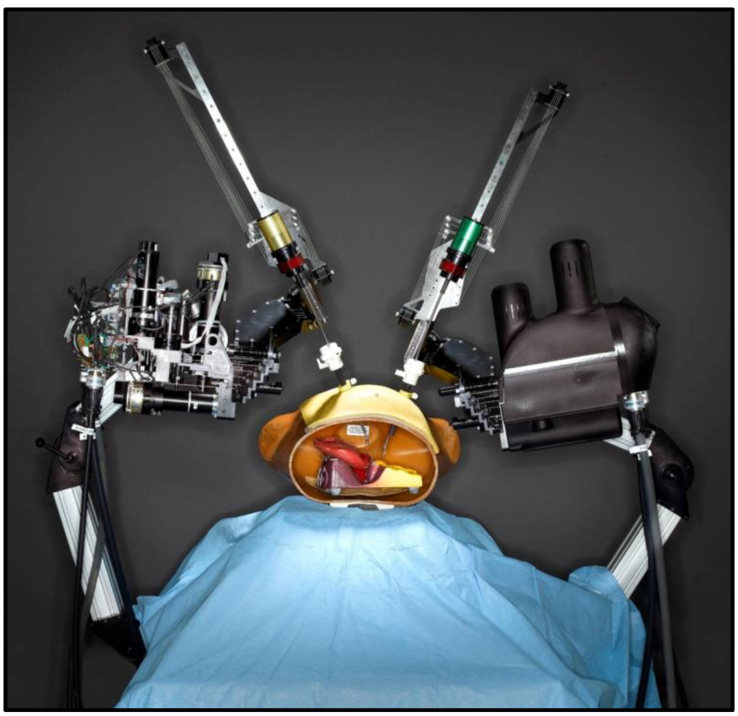


Figure 2. RAVEN remote surgical robot

3. Symbols and Units

The symbols and their corresponding units used in this report are:
Input voltage $V_a(t)$ (V)
Current i(Amps)
Load torque $T_L(t)$ (N-m)
Torque constant K_T (N-m/A)
Speed constant K_v (V/(rad/sec))
Back emf voltage e(t) (volts)
Viscous friction B_m (N-m)
Motor terminal resistance R_m (Ω)
Motor terminal inductance L_a (H)
Motor torque T(t) (N-m)
Motor angle θ (rad)
Angular velocity ω (rad/sec)
Amplifier Gain K_A
Rotor + capstan inertia J_m (kg-m2)

4. Modeling

The first step in the project was the development of the model. This was accomplished by first examining the system, deriving equations, developing the state space model, collecting parameters and then simulating the model.

4.1 Equations and State Space Model

The state space model was derived from the physical system by dividing the motor into two subsystems resulting in the equations:

Electrical Equation:
$V_a(t) = L_a\, di/dt + R_m\, i(t) + K_v\, \omega(t)$

Mechanical Equation:
$T_L(t) = K_t\, i(t) - b_m\, \omega(t) - J_m\, d\omega/dt$

Friction torque inside the motor is modeled by the term $b_m(\omega)(t)$ which is a non-linear function of ω. It would be a simple linear function of $\omega(t)$ when we consider only the viscous friction model (i.e., $b_m(\omega)(t) = b_{viscous}\, \omega(t)$ [2]. The non linear damping term was obtained from the motor datasheet. The figure below shows the state space representation of the model [3]. The matrix X is the state consisting of the current I, angle theta and angular velocity omega. The matrix u is the input of the system, the voltage v and the load torque.

$$X = \begin{bmatrix} X1 \\ X2 \\ X3 \end{bmatrix} = \begin{bmatrix} I \\ \Theta \\ w \end{bmatrix} \begin{matrix} \text{amps} \\ \text{rad} \\ \text{rad/sec} \end{matrix} \qquad u = \begin{bmatrix} u1 \\ u2 \end{bmatrix} = \begin{bmatrix} V \\ Tl \end{bmatrix} \begin{matrix} \text{volts} \\ \text{N-m} \end{matrix}$$

$$\begin{bmatrix} dX1/dt \\ dX2/dt \\ dX3/dt \end{bmatrix} \begin{matrix} = \\ = \\ = \end{matrix} \begin{bmatrix} -Rm/La & 0 & -K/La & X1 \\ 0 & 0 & 1 & X2 \\ K/Jm & 0 & 0 & X3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -bm(X3)/Jm \end{bmatrix} + \begin{bmatrix} KA/La & \Theta \\ 0 & 0 \\ 0 & -1/Jm \end{bmatrix} \begin{bmatrix} u1 \\ u2 \end{bmatrix}$$

Figure 3. System state space model

4.2 System Diagram

The figure 4 below shows a detailed description of the mechanical components of the system. The system can be divided into three parts, motor, capstan and a variable tension from a person holding the cable. Each part of the hardware is characterized by certain parameters which were included in the modeling in milestone two.

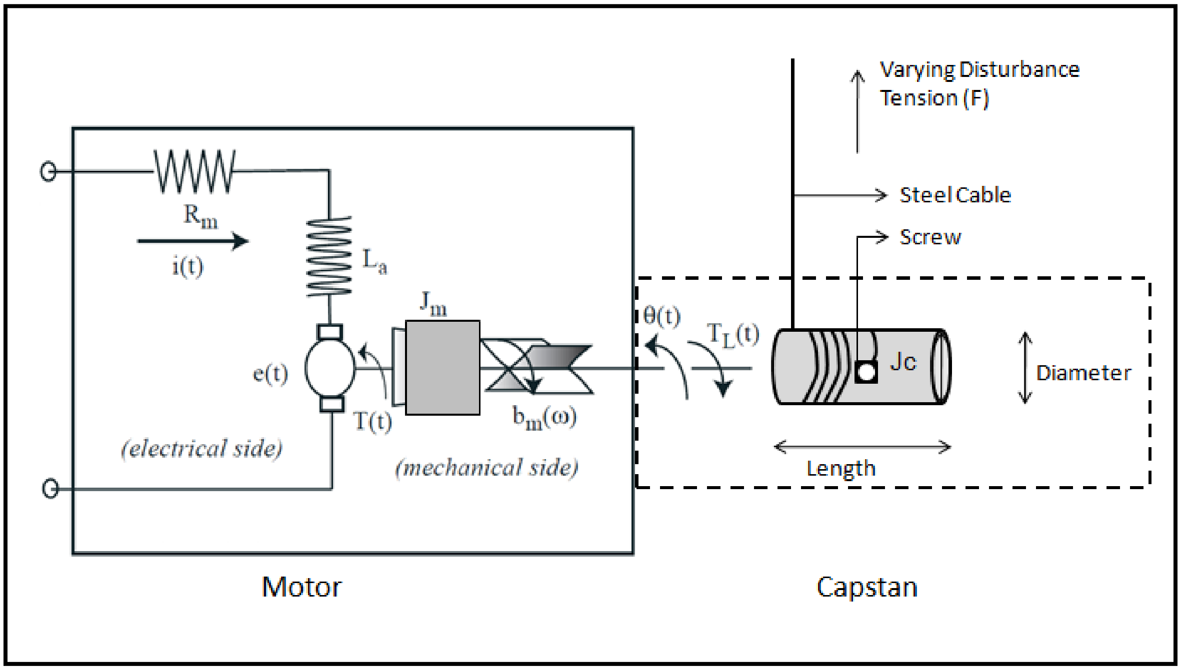


Figure 4. System mechanical diagram

4.3 Parameters

The parameters for the established model are for the motor on the pulley board, which is where all the initial testing will be conducted. The values shown below were obtained from the data sheet and also calculated based on measured data. The final implementation will use the information from the actual robot's parameters which is shown in the Appendix A.

Electrical Parameters:
Self inductance (La) = 1280 mH
Terminal Resistance (Rm) = 4.94 ohms

Mechanical Parameters:
Torque to Speed Ratio (Bm) = 1.1507e-3 Nm/(rad/sec)
Plant inertia (Jm) = 85 gcm^2 + 21.932 gcm^2
Torque constant (Kt) = 0.09167 Nm/Amp
The plant inertia consists of the inertia contribute by the motor rotor and the capstan. Since the capstan could not be isolated the total inertia was calculated by using the rotor inertia from the data sheet and calculating the capstan inertia using the equation:

$$I_z = \frac{1}{2}\pi\rho h\left({r_2}^4 - {r_1}^4\right)$$

Where, h is the height of the capstan, rho(Þ) is the density of steel, r2 and r1 are the inner and our radii of the capstan.

4.4 Model Simulation

A simulation was done to assess the model's response to a constant input. Simulink was used to create a block diagram [2] for the simulation as shown in Figure 5. The plant block consists of the electrical and mechanical systems. The input signal used for the simulation was a 1 volt step and a 0.4 N-m disturbance torque. The results of the simulation are shown below in figure 7.

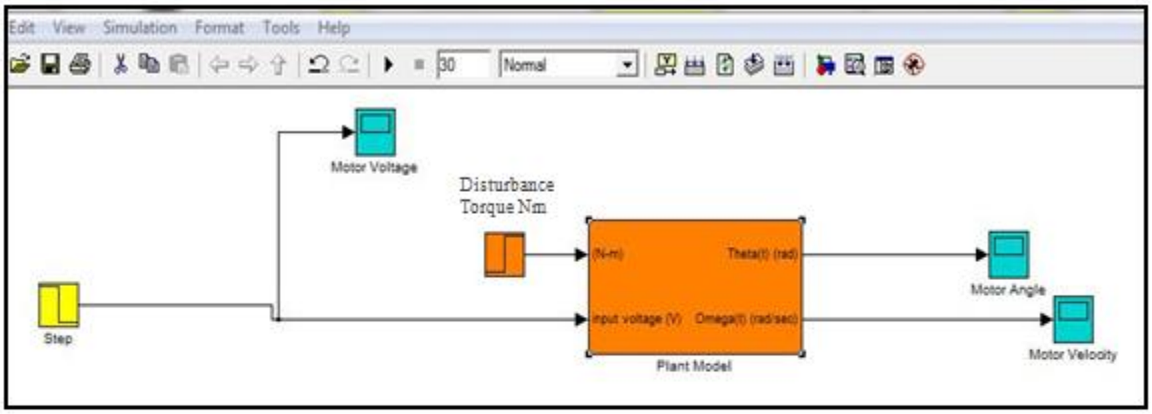


Figure 5. Simulink top level block diagram

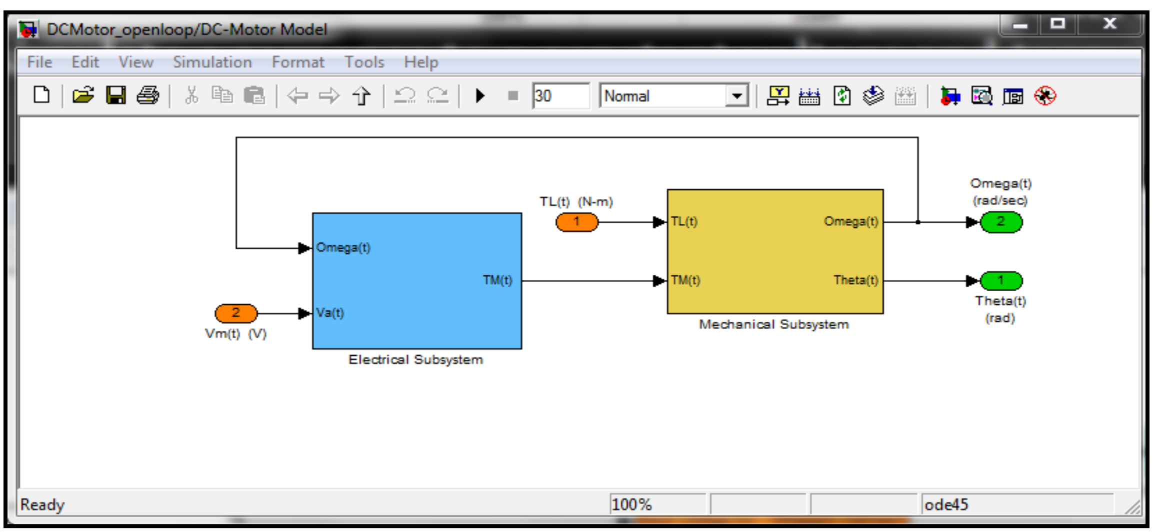


Figure 6. Simulink inner plant model

The graph shown below in Figure 7 shows the position of the motor as seen from the output of the system. The straight line shows that the angle is changing at a constant rate as expected.

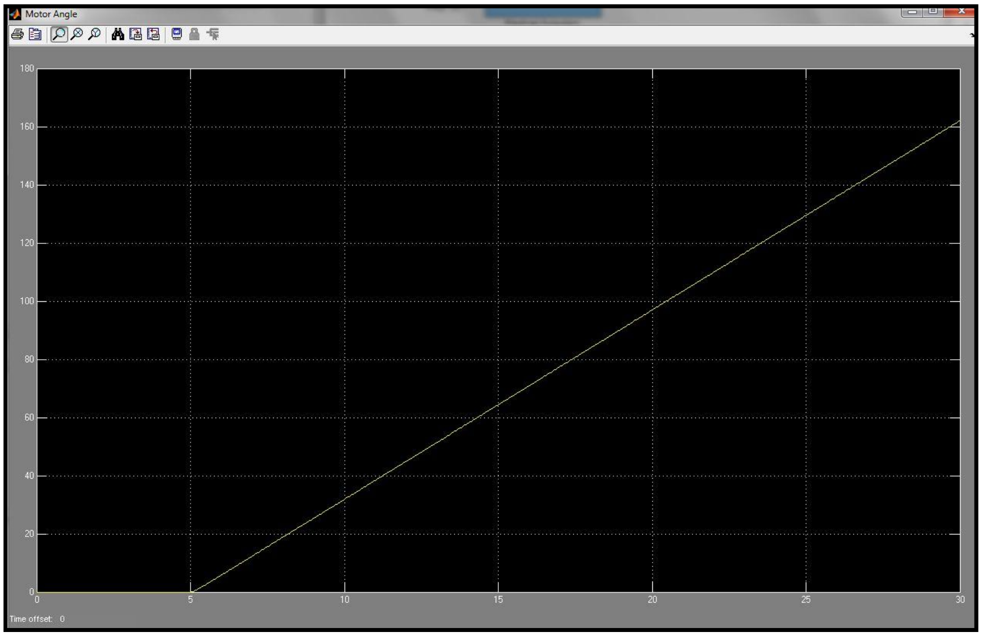


Figure 7. Motor angle result from simulation

The graph below shows the motor velocity when there is a step input.

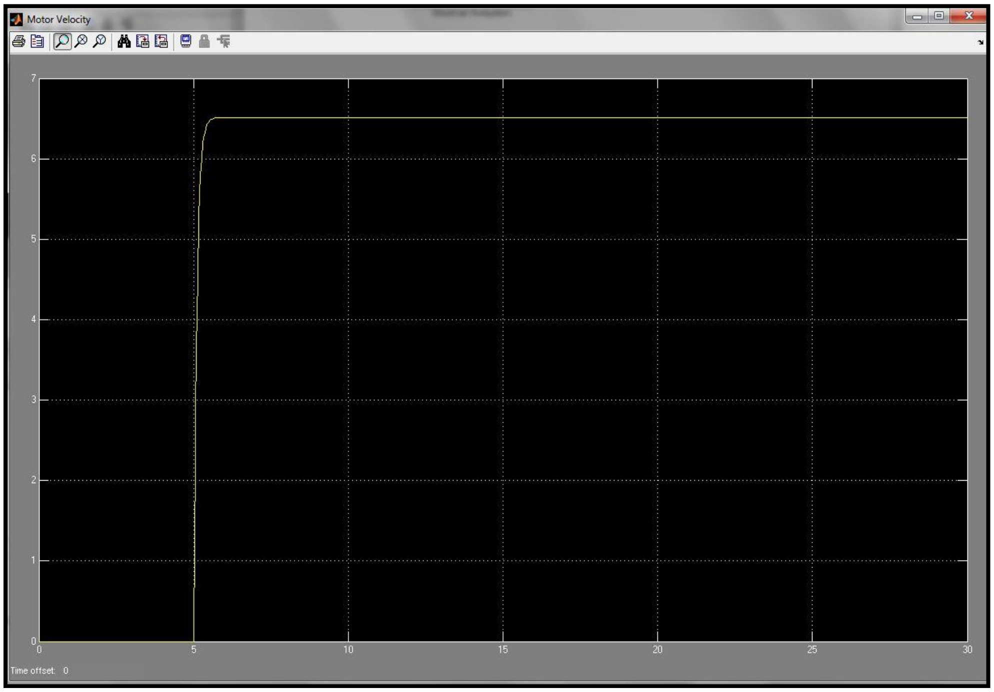


Figure 8. motor velocity result from simulation

4.5 Controllability and Observability

The controllability and observability of the system was assessed using a method shown in the book "Control Systems Engineering" [1]. The controllability was assessed by obtaining the controllability matrix using the A and B matrix from the state equation. A MatLab command directly calculates the matrix and also the rank. If the rank of the controllability matrix is the same as the order of the plant, then the system is controllable. The figure below shows a MAtLab screen shot of the calculation.

```
>> Cm = ctrb(A,B)

Cm =

  1.0e+009 *

    0.0000         0   -0.0000   -0.0000    0.0000    0.0007
         0         0         0   -0.0001    0.0000    0.0101
         0   -0.0001    0.0000    0.0101   -0.0007   -1.1382

>> rank(Cm)

ans =

     3
```

Figure 9. Controllability calculation

Similarly the observability can be calculated from the A and C matrices. Matlab was used to calculate the observability matrix also and the rank was 3 which is the same as the order of the plant. Therefore the plant is observable.

```
>> Om=obsv(A,C)

Om =

  1.0e+003 *

         0    0.0010         0
         0         0    0.0010
    8.5672         0   -0.1075

>> rank(Om)

ans =

     3
```

Figure 10. Observability calculation

5. Design

The design of the controller involved evaluating the specifications of the project and designing a controller that is capable of meet those requirements.

5.1 Performance Specifications

The performance specifications that were established after multiple revisions were:

1. All the hardware used for the project must be from the BRL lab since that will be accessible to students building or re-cabling the robots.
2. The user should be able to control the number of turns wound by the capstan and also the velocity of rotation.
3. The system should be able to wind the cable in two different directions.
4. The system should hold the position of the capstan for a specified amount of time until the cable can be wound around the actuator joint are brought back to attach on the other end of the shaft.
5. The controller should be able to track the specified velocity with 95%+ accuracy.
6. The controller should detect and stop the motor in less than 1 sec if the cable gets dislodged from the capstan.

5.2 Controller Design

We started designing our controller design by writing a MATLAB script to find the motor position transfer function and motor velocity transfer function. After that we use root locus method in Sisotool (MATLAB toolbox) to find the best proportional gain (Kv_p) and integral gain (Kv_I) for velocity control and proportional gain (Kp) for position control. We use these values as starting points for our Simulink simulation. In the simulation, we used trial and error method to improve the results beyond what was obtained in the sisotool design to get the best possible gains. The best controller should have fast rising time, minimum overshoot, and no steady state error. Below shows the best result that we achieved using Sisotool. (See figure 11 and 12). The final values that worked for the system was different since the designed values did not work effectively.

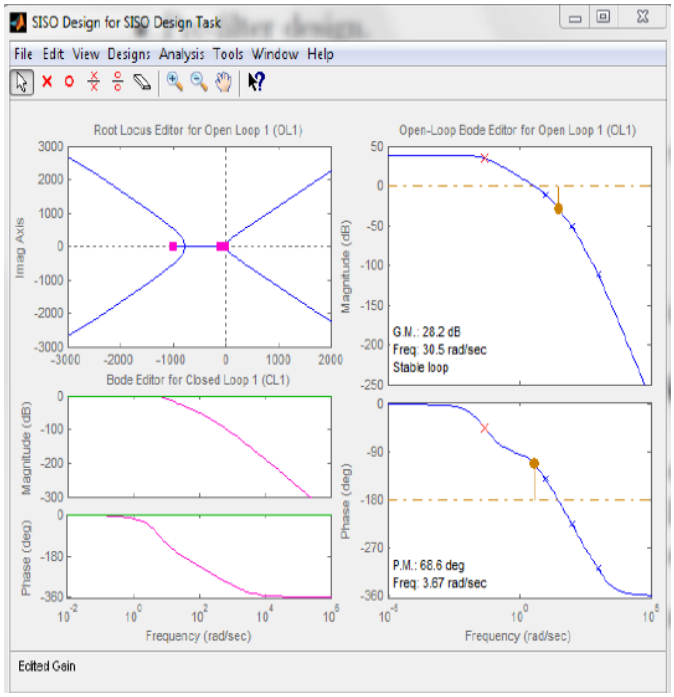


Figure 11. Sisotool screen shot showing the design method for obtaining gains

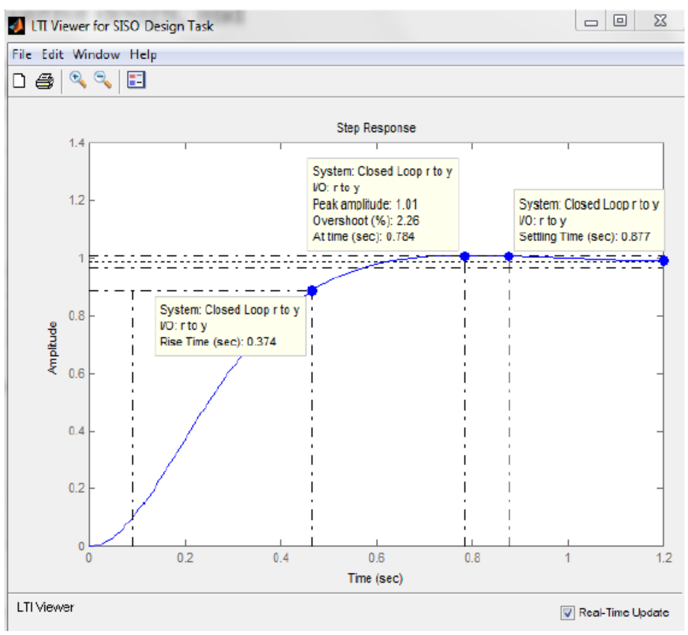


Figure 12. Step response results showing the performance of the controller

5.3 Simulations

We used simulink (program inside MATLAB) to simulate our controller. For the velocity control, we choose PI controller because the motor need to rotate at a constant velocity with a little steady state error. We did not want to add D controller because our system has a lot of noise and it would not help with the performance of the controller. Adding D controller will cause our system to have even higher noise. The input of the velocity control need to go to pre-filter to get rid of overshoot. The motor controller has a saturation limit which is 24 volts. However, for safety issues, we choose the motor saturation limit to be 15 volts. The I controller need to have integral anti-windup to avoid the integral goes unstable. The blue box convert the controller output and disturbance load to the angle theta. Then, the result go to pseudo derivative to get rid of noises and convert angle theta (position) to omega (velocity). The diagram of velocity controller is shown in figure 13.

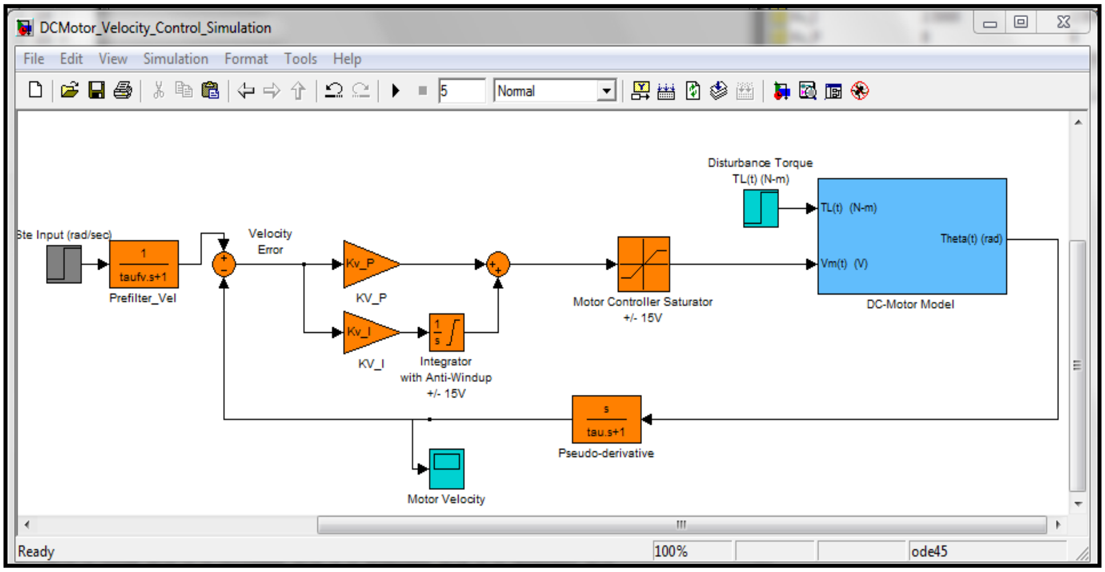


Figure 13. Simulink block diagram showing velocity controller

For the position control, we choose to create a position control (outer loop) outside the velocity control (inner loop). This design will help while transitioning between controllers since the velocity controller block is always part of the control loop. The diagram of position controller is shown in figure 14.

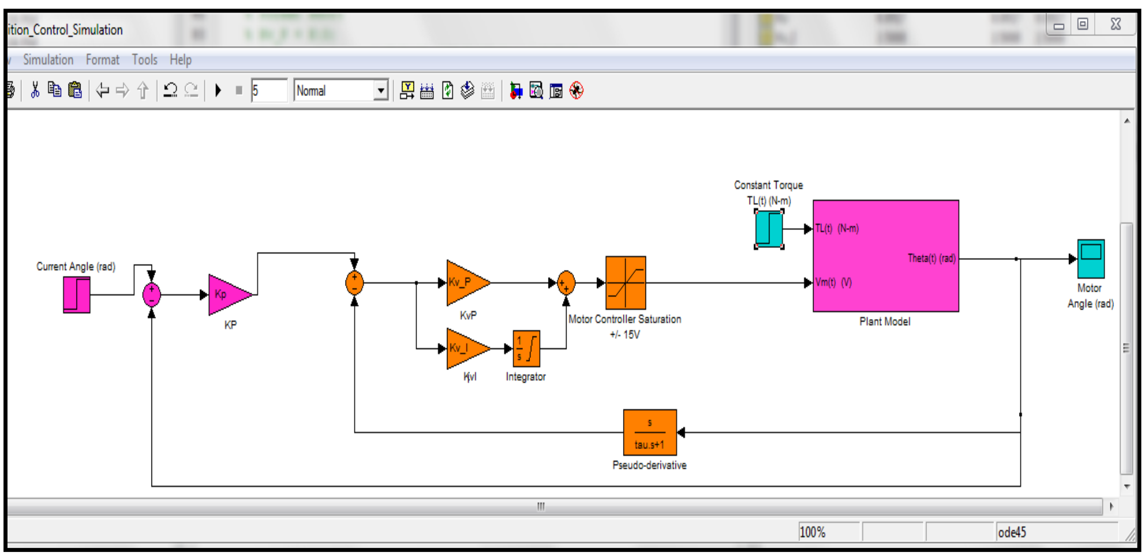


Figure 14. Simulink block diagram showing position controller

Implementation
The project was implemented using the following hardware and software:

5.4 Hardware

The hardware used for the project was limited to those available in the BioRobotics Lab as requested by the customer. The hardware on which the design and testing was done was based on the pulley board. The descriptions of each are given below:

1. Motor:

The motor is a MAXON brushless DC motor. It works with a motor controller which can also acts as an amplifier. The motor has a power rating of 120W with a max stall torque of 0.7 Nm. The remaining motor data is available in the data sheet attached in the appendix.

2. Encoder:

The encoder is built onto the motor and is also powered by the motor controller.

3. Capstan and Cable:

The capstan is a steel cylinder that is welded onto the motor shaft. The cable is a 4mm thick steel cable. The inertia of the capstan and cable combination was calculated based on the dimensions and included in the system model.

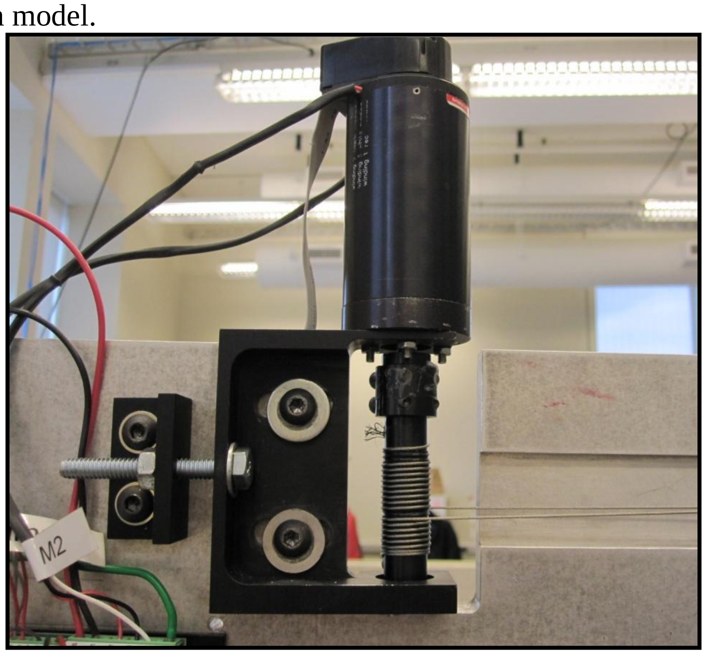


Figure 15. Photo showing various hardware

4. Foot pedal:

Since under the new design the user will be cabling the capstan by hand, typing commands or using the keyboard during cabling is impractical. Since haptic devices used in the BRL lab use a USB foot pedal, we are including that into our system. The pedal is an extension of the keyboard and has three pedals. One pedal will be used to start rotating in one direction, second for the opposite direction and third for emergency stop. Figure 16 below shows a photograph of the foot pedal.

Figure 16. Foot pedal

5.5 Software

The system uses RTAI (Real Time Application Interface) that allows the computer to communicate to the pulley-board in the real time. First, the system needs to initialize the USB-board and set up the communication protocol. Then, it will compute the correct values to the USB-board to operate the correct DC motor motion. These tasks are happen periodically every 1 millisecond. Inside the function where it computes the correct motor motion, we program our controller which is proportional control for position and proportional and integral control for velocity.

The control for the system is done in C and built onto the RTAI module on the USB I/O board. The velocity controller was implemented using the encoder data and making necessary manipulations. The position controller is switched on currently after a certain time. Once the foot pedal is implemented the corresponding pedals will control the switching between states.

The gains of the final design were Kp = 0.02 for position control.
For the velocity control Kp = 0.4 and Ki = 0.01.

5.5.1 State Machine

Based on the need to isolate the functions of the system, a finite state machine was developed. The purpose of the state machine is to clearly define and separate the functions of the system into parts at a top level stage.

The figure 17 below shows the state diagram. There three states named A, B and C. Each of these states output a certain value which either shuts off the system or activates either the velocity control or position control. The possibilities that are not shown in the diagram are considered as “dont cares”. The start command is initiated when the user presses the left pedal.

The Emergency stop command is initiated when the user presses the right pedal.

The opposite direction command is initiated by the capstan reaching the angular position specified by the user in the first rotational cycle.

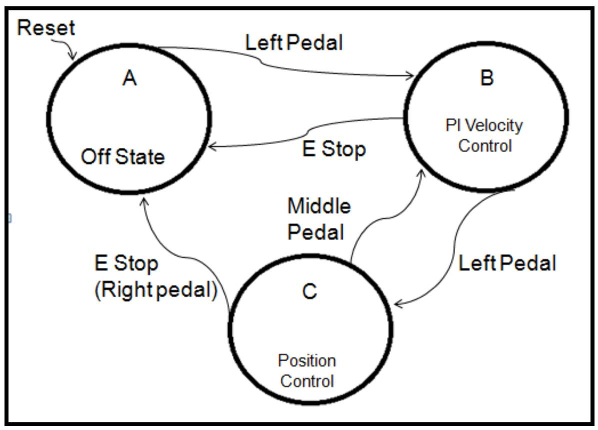


Figure 17. State Machine diagram

6. Experimental Data

The data shown here are the results on the day of the report compilation. The velocity control and the state transitions work effectively. The foot pedal has not been implemented and the program required to run the USB interface is being developed. The data below shows the various other parts of the project.

6.1 Velocity Controller

The velocity control of the controller was limited to 500 deg/sec so that the user can stop the motor when required. This might be necessary when the cable falls into the wrong thread or if there is a delay before the foot pedal is pressed. The graph below in Figure 18 shows the motor velocity of the motor when there is no resistance. The max tension provided by the motor while winding is **0.5N.**

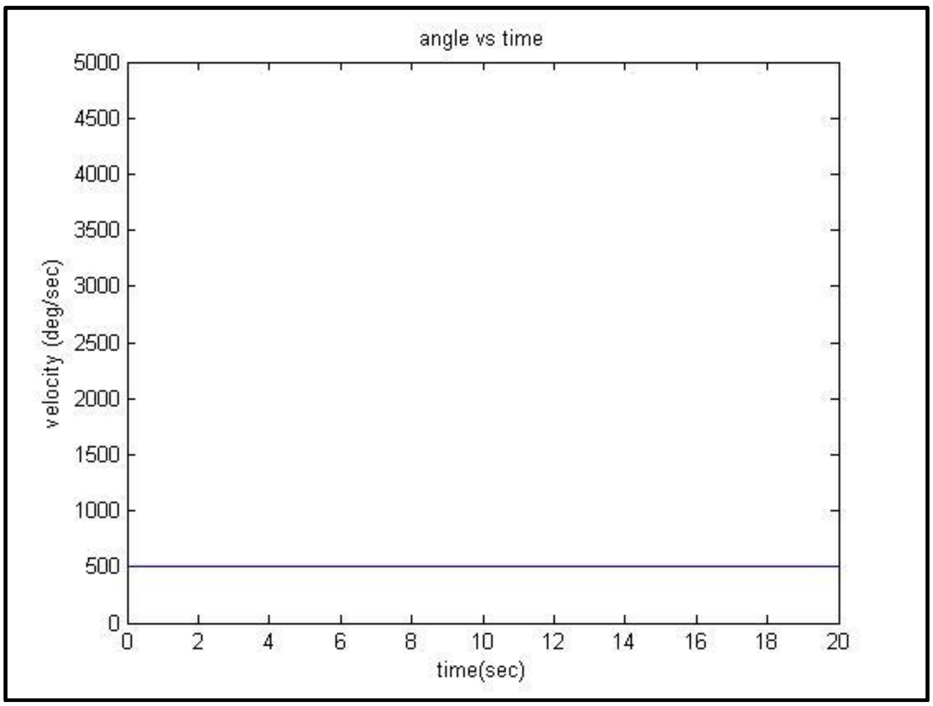


Figure 18. graph showing constant velocity for the velocity controller

6.2 Position Controller

The position controller worked effectively even though it was only a proportional control. The motor managed to hold the position up to a measured tension of **55N** (measured using a force sensor). The graph below in Figure 19 shows the motor holding its position after 20secs of rotation.

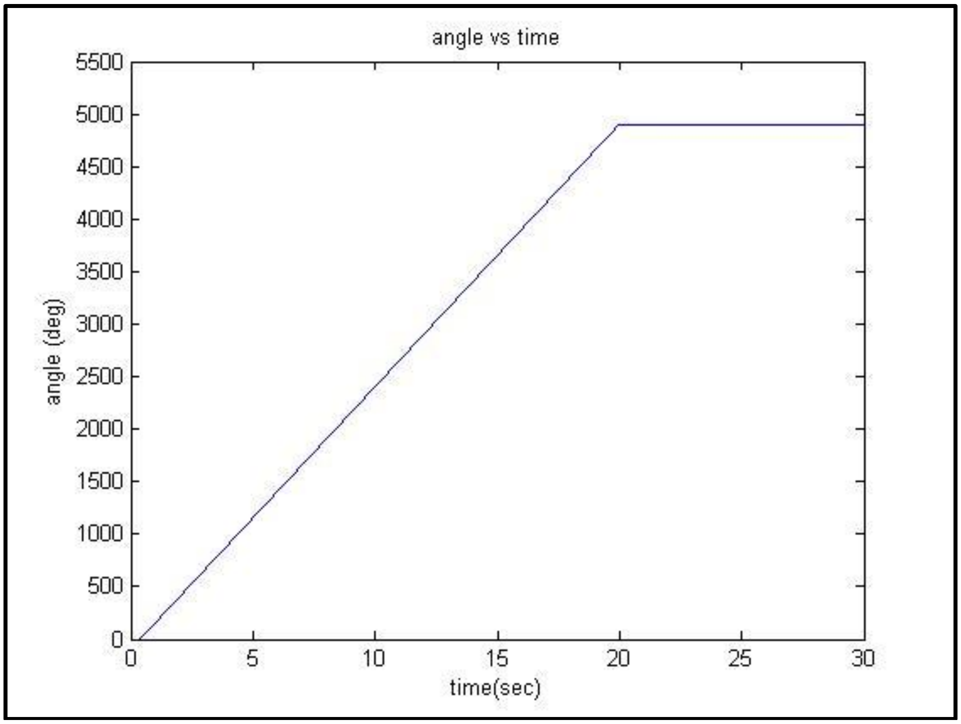


Figure 19. Motor position for position control

7. Conclusions

The purpose of the EE 449 class is to learn about real world implementation of control principles for an actual customer. This project was created out of necessity and provided the necessary level of complexity. Over the course of the quarter we successfully modeled the system. Then we designed a controller and implemented the controller. We learned the different ways to implement a controller. Also implementing a theoretical design in the real world has numerous challenges. Planning for obstacles and unexpected behavior should be part of the project plan. We also learned about effective ways to conduct technical presentations and write reports.

In terms of the project, most of the parts have been implemented with the exception of the foot pedal. Even though the controller can be used without the foot pedal, since making cabling easier is the main goal of the project, the project will be complete only once the foot pedal can be properly used. The data from the other motors on the actual robots being built are included in the appendix. The next step is to complete the foot pedal information and then hand over this info to the BioRobotics Lab.

8. Bibliography

[1] Norman S. Nise, "Control Systems Engineering", Wiley 2008
- Used to learn about control principle and to calculate controllability and observability matrix.

[2] Uy-Loi Ly, "DC Motor Control", 2010, https://courses.washington.edu /aa448/DCMotorControl_LabDescription.pdf
- Used to help model and simulate system.

[3] "dcmodel", 2010. https://courses.washington.edu/aa448/dcmodel.pdf
- Used to determine the State Space equation.

[4] "Maxon EC Motor Data Sheet", 2010. http://shop.maxonmotor.com/ishop/article/article/118898.xml
- Used to get the model parameters

[5] Atef Saleh Othman Al Mashakbeh, "Proportional Integral and Derivative Control of Brushless DC Motor" European Journal of Scientific Research. ISSN 1450-216X Vol.35 No.2 (2009), pp.198-203
-Used the author's method to model the BLDC motor.

[6] J. Rosen, B. Hannaford, 'Doc at a Distance,' IEEE Spectrum, pp. 34-39, October 2006.
- The paper contained detailed information about the RAVEN projects and photographs

[7] M. Lum, D. Friedman, J. Rosen, G. Sankaranarayanan, H. King, K. Fodero, R. Leuschke, M. Sinanan, B. Hannaford, 'The RAVEN - Design and Validation of a Telesurgery System,' International Journal of Robotics Research, vol. 28, pp. 1183-1197, September 2009.
-Used to understand the design of the RAVEN.

[8] RTAI documentation, https://www.rtai.org/documentation/magma/html/api/
-Used to learn about the USB I/O board's existing code.

[9] Borsjie, P.; Chan, T.F.; Wong, Y.K.; Ho, S.L.; , "A Comparative Study of Kalman Filtering for Sensorless Control of a Permanent-Magnet Synchronous Motor Drive," *Electric Machines and Drives, 2005 IEEE International Conference on* , vol., no., pp.815-822, 15-15 May 2005
-To understand implementation of the unscented kalman filter

9. Appendix A

Motor information for the motors used on the RAVEN

1. **Maxon Motor – 310009 motor RE30 + 166155gear GP32C**

Combination data		
Nominal voltage	V	48
No load speed	min^{-1}	2295
Max. continuous torque	mNm	0.75
Stall torque	mNm	1.1
Gear data		
Article No.	166155	
Program	Planetary Gearhead GP 32 A Ø32 mm, 0.75 - 4.5 Nm, Metal Version	
Reduction		3.7:1
No. of stages		1
Max. continuous torque	Nm	0.75
Intermittently permissible torque at gear output	Nm	1.1
Sense of rotation, drive to output		=
Max. efficiency	%	80
Average backlash no load	°	0.7
Mass inertia	gcm^2	1.5
Gearhead length L1	mm	26.4
Weight	g	118
Max. motor shaft diameter	mm	6
Motor data		
Article No.	310009	
Program	RE 30 Ø30 mm, Graphite Brushes, 60 Watt	
Assigned power rating	W	60
Nominal voltage	V	48
No load speed	min^{-1}	8490
Stall torque	mNm	1020
Max. continuous torque	mNm	88.2
Speed / torque gradient	min^{-1} / mNm^{-1}	8.33
No load current	mA	78.5
Starting current	A	19
Terminal resistance	Ohm	2.52
Max. permissible speed	min^{-1}	12000
Nominal current (max. continuous current)	A	1.72
Max. efficiency	%	88
Torque constant	mNm / A^{-1}	53.8
Speed constant	min^{-1} / V^{-1}	178
Mechanical time constant	ms	3.01
Rotor inertia	gcm^2	34.5
Terminal inductance	mH	0.513
Thermal resistance housing-ambient	KW^{-1}	6.0
Thermal resistance winding-housing	KW^{-1}	1.7
Thermal time constant winding	s	16.9
Motor lenght	mm	68
Weight	g	260

2. Maxon Motor – 148877 motor RE40 + 203115 gear GP42C

Combination data		
Nominal voltage	V	48
No load speed	min^{-1}	632
Max. continuous torque	mNm	7.5
Stall torque	mNm	11
Gear data		
Article No.	203115	
Program	Planetary Gearhead GP 42 C Ø42 mm, 3 - 15 Nm, Ceramic Version	
Reduction		12:1
No. of stages		2
Max. continuous torque	Nm	7.5
Intermittently permissible torque at gear output	Nm	11
Sense of rotation, drive to output		=
Max. efficiency	%	81
Average backlash no load	°	0.8
Mass inertia	gcm^2	15
Gearhead length L1	mm	55.4
Weight	g	360
Max. motor shaft diameter	mm	10
Motor data		
Article No.	148877	
Program	RE 40 Ø40 mm, Graphite Brushes, 150 Watt	
Assigned power rating	W	150
Nominal voltage	V	48
No load speed	min^{-1}	7580
Stall torque	mNm	2500
Max. continuous torque	mNm	184
Speed / torque gradient	min^{-1} / mNm^{-1}	3.04
No load current	mA	68.6
Starting current	A	41.4
Terminal resistance	Ohm	1.16
Max. permissible speed	min^{-1}	12000
Nominal current (max. continuous current)	A	3.12
Max. efficiency	%	92
Torque constant	mNm / A^{-1}	60.3
Speed constant	min^{-1} / V^{-1}	158
Mechanical time constant	ms	4.39
Rotor inertia	gcm^2	138
Terminal inductance	mH	0.329
Thermal resistance housing-ambient	KW^{-1}	4.65
Thermal resistance winding-housing	KW^{-1}	1.93
Thermal time constant winding	s	41.9
Motor lenght	mm	71
Weight	g	480

3. SERIES Z12A PWM SERVO AMPLIFIERS

SPECIFICATIONS:

	MODELS	
POWER STAGE SPECIFICATIONS	**Z6A6**	**Z12A8**
DC SUPPLY VOLTAGE	16 – 60 VDC	16 – 80 VDC
PEAK CURRENT (2 sec. max., internally limited)	± 6 A	± 12 A
MAX. CONTINUOUS CURRENT (internally limited)	± 3 A	± 6 A
MINIMUM LOAD INDUCTANCE *	100 μH	100 μH
SWITCHING FREQUENCY	50 kHz ± 15%	33 kHz ± 15%
HEATSINK (BASE) TEMPERATURE RANGE **	0^{o} to $+75^{o}$ C, disables if > 75^{o} C	
POWER DISSIPATION AT CONTINUOUS CURRENT	10 W	24 W
OVER-VOLTAGE SHUT-DOWN (self reset)	67 V	88 V
BANDWIDTH (load dependent)	5 kHz	

10. Appendix B

Code Comments

pulley_board_control.c

```
/*

File: pulley_board_control.c
Author: Hawkeye
Created October 2008

   This is the main control loop content for the pulley board.

Inputs:  Encoder values
Outputs: DAC values

*/


#include "pulley_board_control.h"
#include <rtai_fifos.h>
#include "velocity.h"

#include <stdio.h>

unsigned char kb;
//------------------Foot Pedal-------------
//extern int input;
//#include <string.h>
//#include <iostream>
//#include <sys/time.h>
//#include <termios.h>
//#include <stdlib.h>

//-------------------------------------------------------------Foot
Pedal------------------------------------------
//static struct termios g_old_kbd_mode;
/*
// did somebody press somthing???
static int kbhit(void){
    struct timeval timeout;
    fd_set read_handles;
    int status;

    // check stdin (fd 0) for activity
    FD_ZERO(&read_handles);
    FD_SET(0, &read_handles);
    timeout.tv_sec = timeout.tv_usec = 0;
    //status = select(0 + 1, &read_handles, NULL, NULL, &timeout);
```

```
    return status;
}
*/
/*
// put the things as they were befor leave..!!!
static void old_attr(void){
    tcsetattr(0, TCSANOW, &g_old_kbd_mode);
}
*/

//----------------------------------------------------------------------
--------------------------------------------------
//-----------------------------------------

// Hack to get rid of annoying compiler warnings w/ math.h
#ifdef __attribute_used__
#undef __attribute_used__
#endif
#ifdef __attribute_pure__
#undef __attribute_pure__
#endif
#ifdef __always_inline
#undef __always_inline
#endif
#include <math.h>

#include "../core_code/RTAI_modules/motor.h"
#include "../core_code/RTAI_modules/t_to_DAC_val.h"
#include "../core_code/RTAI_modules/utils.h"
#include "defines.h"
#include "UnscentedKalmanFilter.h"

#ifndef INT_COUNT
    int count = 1;
#define INT_COUNT
#endif
int positionhold=1;
extern unsigned long int globalTime;
int globalNumMech = 1;

struct tagDOF_type globalDOF_types[1];


#define I_MAX_DES7010 31.97 // Includes fudge-factor from empirical
measurement.  Nominally 30.0 Amps
#define I_MAX_EC40_PULLEYB 9.725
#define I_CONT_EC40_PULLEYB 1.77
#define GEAR_BOX_TR_PULLEYB 1
#define T_PER_AMP_EC40_MEASURED 0.09167


//header file with experiment settings in it
```

```
#include "settings.h"

//place to cache the UKF's estimate of the parameter value.  The
estimates for the state
//variables are placed in tagDOF, but because there is no place to put
this number there,
//it can't be done that way.
static double damping_estimate;
static double stiffness_estimate;

// Initialize some stuff.
void init_pulley_board(struct tagDOF *in_j1)
{

  //Load IMAX
  globalDOF_types[0].currentPeakMax = I_MAX_DES7010;
  globalDOF_types[0].currentContMax = (double)(I_CONT_EC40_PULLEYB);

  //Load the torquePerAmp
  globalDOF_types[0].tauPerAmp = (double)(T_PER_AMP_EC40_MEASURED *
GEAR_BOX_TR_PULLEYB);

  //Initialize the old positions and velocities
  int k=0;
  for (k = 0; k < STATE_HISTORY_LEN; k++)
    {
      globalDOF_types[0].stackMotorPos[k] = 0;
      globalDOF_types[0].stackMotorPosDes[k] = 0;
      globalDOF_types[0].stackMotorVel[k] = 0;
      globalDOF_types[0].stackMotorVelDes[k] = 0;
    }

  //- Initialize torque to zero
  in_j1 -> tau_d = 0;

  // Initialize the Kalman filter
  initUKF();
}



//- sinusiod parameters
//the actual values are defined in settings.h.  See this file
//for more information.
//These are used in the function pulley_board_control() and
output_parameters().
#ifdef SUM_OF_SINUSOIDS
static const int numSin          =     6;
static const double scaleMag          =     MAGNITUDE_SCALAR;
static const double const Mag[]       =
      SUM_OF_SINUSOIDS_MAGNITUDES;
```

```
static const double const w[]          =
      SUM_OF_SINUSOIDS_FREQUENCIES;
#endif


#ifdef SIMPLE_SINUSOID
//the actual values are defined in settings.h.  See this file
//for more information.
static double const SingleMag          =     SIMPLE_SINUSOID_MAGNITUDE;
static double const Singlew            =     SIMPLE_SINUSOID_FREQUENCY;
#endif

#ifdef ROTATE
//the actual values are defined in settings.h.  See this file
//for more information.
static double const piToAngle          =     PI_TO_ANGLE;
static double const rotAngle         =   ROTATION_ANGLE;
static double const numbRotate         =     NUMBER_OF_ROTATION;
static double const medSpeed         =   MEDIUM_SPEED;
#endif

//- This is the control code for the pulley board.
void pulley_board_control(struct tagDOF *const j1){



  //- frequency sweep variables
  #ifdef SWEEP
  double freqHz       =     SWEEP_FREQUENCY;
  const double sweepMag     =     SWEEP_MAGNITUDE;
  #endif

  //these variables are only required if low pass position
  //filtering is being used.  See settings.h for more information.
  #ifdef LPF_POS
  //arrays used to store previous filtered and unfiltered encoder
  //positions
  static float enc_in[3] = {0,0,0};
  static float enc_out[3] = {0,0,0};
  #endif

  //- Coefficients for position LPF
  //coefficient values are defined in settings.h
  #ifdef LPF_POS
  static const float const B[] = LPF_IN_COEFFS;
  static const float const A[] = LPF_OUT_COEFFS;
  #endif

  double xhk[N]; // State estimate
      //Changed to N so that it is compatible with the new
      //Kalman filter       --Cooper
```

```
//- Get time in seconds
const double secs = (double)(globalTime) / 1000;

//-  LPF for position
float enc_cur = (float)j1->enc_val;

#ifdef LPF_POS
float filt_enc = (B[0]*enc_cur +
         B[1]*enc_in[0] +
         B[2]*enc_in[1] +
         B[3]*enc_in[2] +
         A[1]*enc_out[0] +
         A[2]*enc_out[1] +
         A[3]*enc_out[2] );
#endif


//LPF_POS can be defined to enable position filtering in
//settings.h.  See this file for more information.
#ifdef LPF_POS
  //- calculate motor angle from filtered position
  j1->mpos = (float)filt_enc * (2 * PI) /ENC_PER_REV;
#else
  //- calculate motor angle from un-filtered position
  j1->mpos = (float)enc_cur * (2 * PI) /ENC_PER_REV;
  printk("unfilt\n");
#endif


//- update old values for position LPF filter
#ifdef LPF_POS
enc_in[2]  = enc_in[1];
enc_in[1]  = enc_in[0];
enc_in[0]  = enc_cur;
enc_out[2] = enc_out[1];
enc_out[1] = enc_out[0];
enc_out[0] = filt_enc;
#endif

//- Apply unscented kalman filter
if ( -1 == doUKF(xhk,
           (float)enc_cur * (2 * PI) / ENC_PER_REV,
           ( -1 * j1->tau_d ) ) ){
  printk("UKF estimation failed. (-1)\n");
}

//- save kalman estimates to data structure
j1->jpos   = xhk[0];   // estimated motor angle
j1->jpos_d = xhk[1];   // estimated link angle;
j1->jvel   = xhk[2];   // estimated motor velocity;
j1->jvel_d = xhk[3];   // link velocity;
```

```
  stiffness_estimate = xhk[4]; //cache it so that it can be output as
data
  damping_estimate = xhk[5]; //cache it so that it can be output as
data

  //The following compilation is contingent on a #define in
  //settings.h.  See this file for more information.
  #ifdef SWEEP
    //- Generate frequency sweep
    freqHz = floor(secs / 10) + 1;
    if (freqHz > 50) freqHz = 1;
    j1->mpos_d = sweepMag * sin( freqHz * 2 * PI * secs);
  #else
  #ifdef SUM_OF_SINUSOIDS
    //- Generate sin wave
    //- desired position = a Sin (w*t)

    j1->mpos_d = 0;
    int i=0;
    for (i=0; i<numSin; i++){
      j1->mpos_d += Mag[i] * scaleMag * sin( w[i] *2*PI * secs );
    }
  #else
  #ifdef SIMPLE_SINUSOID
    //- Generate sin wave
    //- desired position = a Sin (w*t)

    j1->mpos_d = SingleMag * sin( Singlew *2*PI * secs );
  #else
  #ifdef STEP
    //- Generate step input
    //-   Stay at zero for 5 sec.  Then step to 90 degrees
    if (secs < 5.0){
      j1->mpos_d = 0;
    } else {
      j1->mpos_d = PI / 2;
    }
  #else
  #ifdef SQUARE_WAVE
    //- Generate square wave input
    if (secs < 2.0){
      j1->mpos_d = 0;
    }
    else if ( ((int)floor(secs) % 4) < 2 ) {
      j1->mpos_d = PI / 4;
    }
    else if ( ((int)floor(secs) % 4) < 4 ) {
      j1->mpos_d = -1 * PI / 4;
    }
    else if (secs < 8) {
      j1->mpos_d = 0;
    }
```

```
    else {
      j1->mpos_d = 0;
    }
  #else
  //#ifdef ZERO
  //  //- desired position = 0;
  //  j1->mpos_d = 0;
  //#else
  #ifdef ROTATE
    // rotate from 0 to desire position
    //j1->mpos = 0;
    //double lastSec = 0;
/*   int YES = 0;

if (YES == 0) {
    char ch;
    //static char init;
    struct termios new_kbd_mode;

    //if(init)
    //    return;
    // put keyboard (stdin, actually) in raw, unbuffered mode
    tcgetattr(1, &g_old_kbd_mode);
    memcpy(&new_kbd_mode, &g_old_kbd_mode, sizeof(struct termios));

    new_kbd_mode.c_lflag &= ~(ICANON | ECHO);
    new_kbd_mode.c_cc[VTIME] = 0;
    new_kbd_mode.c_cc[VMIN] = 1;
    tcsetattr(1, TCSANOW, &new_kbd_mode);
    atexit(old_atr);
    YES = 1;
}
*/

//    while (!kbhit()){
        //char ch1;
//while (1) {
        // getting the pressed key...
        //int a = 'a' + 13;
        //int s = 's' + 13;
        //int d = 'd' + 13;
        //ch1 = getchar();
        //delay(100);
         //char inp[2];
         int input = 1;

/*
    while (strcmp (inp,"ex") != 0 ) {
         //puts ("Enter text: ab, cd, or ef");
         scanf("%s", inp);
         printf("Inp is: %s \n", inp);
         if ( strcmp(inp,"ab") ==0){
```

```
            input = 1;
         }
         else if ( strcmp(inp,"cd") ==0){
            input = 2;
         }
         else if ( strcmp(inp,"ef") ==0){
            input = 3;
         }
         else {
            input = 3;
         }
         printf("Input is: %d \n", input);
    }
*/
/*
    if ( kb == 'a' ){
        input = 1;
    }
    else if ( kb == 's' ){
        input = 2;
    }
    else if ( kb == 'd'){
        input = 3;
    }
    else {
        input = 2;
    }
*/
if (secs < numbRotate){
        // rotate right to left
        if( input == 1 ){
                j1->mpos = rotAngle * piToAngle * secs;
                j1->mpos_d = (rotAngle * piToAngle * secs) - medSpeed;
        }
        // stop
        else if ( input == 2) {
              j1->mpos_d = rotAngle * piToAngle * numbRotate;
        }
        // rotate left to right
        else if ( input == 3 ){
              j1->mpos = rotAngle * piToAngle * secs;
              j1->mpos_d = (rotAngle * piToAngle * secs) + medSpeed;
        }
        else{
             j1->mpos_d = 0;
        }
}

//    }

//    }
```

```
if (secs > numbRotate){
        // get rid of j1->mpos (how to do that)
        j1->mpos = ((float)enc_cur * (2 * PI) / ENC_PER_REV);

 if(positionhold==1){
      j1->mpos_d = j1->mpos;//((float)((int)(j1->mpos/(2*PI))))*2*PI;
      positionhold=0;
}

        //- update old values for position LPF filter
        enc_in[2]  = enc_in[1];
        enc_in[1]  = enc_in[0];
        enc_in[0]  = enc_cur;
        enc_out[2] = enc_out[1];
        enc_out[1] = enc_out[0];
        enc_out[0] = filt_enc;

  #else

    //no reference signal was chosen--compilation will fail with
message
    #error no reference signal defined in settings.h

  //Nesting implements "elif"-style structure; only one reference
signal is compiled.
  #endif
  #endif
  #endif
  #endif
  #endif
  #endif




  //- Calculate motor velocity
  //- This should be done after calculating desired trajectory to
determine mvel_d


  //The following four function calls invoke code in velocity.c
  GetJointVelocity( j1, VEL_DESIRED);
  GetJointVelocity( j1, VEL_ACTUAL);
  // NOTE: These two lines of code above are correct in that the same
function is called
  // twice with a different second parameter.  The first time the
desired velocity is
  // computed based on past desired motor positions, and cached in the
mvel_d field.
  // The second time the actual velocity is computed based on past
actual motor positions,
  // and cached in the mvel field.  --Cooper
```

```
  PushPos(j1);
  PushVel(j1);

    //- compute torque
    // mvel stores actuals
    // jvel stores UKF estimates

      //The following compilation is contingent on a #define in
      //settings.h.  KP and KD are also defined there.  See this
      //file for more information.
    #ifdef P_CONTROL
        j1->tau_d = KP * (j1->mpos - j1->mpos_d); //- P-control
        //#error P_CONTROL not implemented!!!
    #else

    #ifdef PI_CONTROL

     if (secs < numbRotate) {
            j1->tau_d = KPV * (j1->mpos - j1->mpos_d) + KI * (j1->mvel
- j1->mvel_d);  //- V-control
        }
     else{
            j1->tau_d = KP * (j1->mpos - j1->mpos_d);// - j1->jpos);
//- P-control
        }
        //if (secs > numbRotate){
            //j1->mpos = 0;
            //j1->mpos_d = 0;
        //    j1->tau_d = j1->tau_d - (j1->tau_d - 0.01);
        //}
    #else

    #ifdef PD_TRAJECTORY_FOLLOWING
      j1->tau_d = ((KP) * (j1->mpos - j1->mpos_d) + (KD) * (j1->mvel -
j1->mvel_d) );  //- PD control
    #else

    #ifdef PD_STEP_FOLLOWING
      j1->tau_d = ((KP) * (j1->mpos - j1->mpos_d) + (KD) * (j1->mvel -
0) );  //- PD control
    #else

    #ifdef D_CONTROL
      // j1->tau_d =  KD * (j1->mvel - j1->mvel_d);  //- D-control
      #error D_CONTROL not implemented!!!
    #else

    #ifdef ZERO_CONTROL
      //  j1->tau_d = 0;  // Zero-control
      #error ZERO_CONTROL not implemented!!!
    #else
```

```
    #ifdef PD_TRAJECTORY_MOTOR_ESTIMATE
      j1->tau_d = ((KP) * (j1->jpos - j1->mpos_d) + (KD) * (j1->jvel -
j1->mvel_d) );  //- PD control with motor estimates
    #else

    #ifdef PD_STEP_MOTOR_ESTIMATE
      j1->tau_d = ((KP) * (j1->jpos - j1->mpos_d) + (KD) * (j1->jvel -
0) );  //- PD control with motor estimates relative to static target
    #else

      #error no control type defined in settings.h

    //Nesting implements "elif"-style structure; only one control type
is compiled.
    #endif
    #endif
    #endif
    #endif
    #endif
    #endif
    #endif
    #endif

  //- Set constant torque
  //  j1->tau_d = 0.1623;

  //- Calculate a DAC value from the desired torque
  j1->current_cmd = DACFromTorque(j1);

  // print state
  if (globalTime % 99 == 0){
    printk("mpos: (%d),\tmpos_d: (%d),\tmvel: (%d), \tjpos: (%d),
\tjpos_d: (%d), \tdac:( %d / %d )\n\n",
         (int)( j1->mpos   * 180 / M_PI * 1e2),
         (int)( j1->mpos_d * 180 / M_PI * 1e2),
         (int)( j1->enc_val  * 180 / M_PI * 1e2),
       (int)( j1->jpos ),
       (int)( j1->jpos_d   * 180 / M_PI * 1e2),
         (int)(100*j1->tau_d) ,
        j1->current_cmd ); //+ ZERO_DAC);

  }
}

//void output_parameters(int out);

void pulley_board_output(struct tagDOF *j1, int out){

  char foo[400];

#ifdef PRINT_ASCII
```

```
//only if the option is selected to print the header,
//otherwise don't do this
#ifdef PRINT_OUTPUT_HEADER
  static int first_time = 1;

  if (first_time){
    //this header information wasn't really that useful...

    ///sprintf(foo,"%% new data collection\n\n");
    ///rtf_put(out, foo, strlen(foo));

    //sprintf(foo, "\n%% globalTime (ms), mpos (deg * 100), mpos_d
(deg * 100), j1->mvel (deg/sec * 100), tau_d (torque*100), DAC val,
enc_val (motor), enc_val(link)");
    //rtf_put(out, foo, strlen(foo));
    //sprintf(foo, "(9)mpos(est), linkAngle(est), mvel(est),
linkVel(est)\n");
    //rtf_put(out, foo, strlen(foo));

    //try this header instead
    //macro to send line to FIFO
    #define out_to_fifo(...)     \
      sprintf(foo, __VA_ARGS__);       \
      rtf_put(out, foo, strlen(foo))         \

      //basically, print the comment made below
      out_to_fifo("\tThe output format is as follows:\n");
      out_to_fifo("\n");
      out_to_fifo("\tcolumn  1 --Time elapsed in milliseconds\n");
      out_to_fifo("\tcolumn  2 --Motor position as determined from
motor encoder\n");
      out_to_fifo("\t       in degrees * 100 (centidegrees?).  This
may\n");
      out_to_fifo("\t       have been low-pass filtered first\n");
      out_to_fifo("\tcolumn  3 --Reference signal in angle units
in\n");
      out_to_fifo("\t       degrees * 100 (centidegrees?)\n");
      out_to_fifo("\tcolumn  4 --Motor velocity computed from previous
motor\n");
      out_to_fifo("\t       positions.  This is done in
velocity.c.\n");
      out_to_fifo("\t       Units are degrees/sec * 100
(centidegrees/sec?)\n");
      out_to_fifo("\tcolumn  5 --Torque values sent to motor
(units?)\n");
      out_to_fifo("\tcolumn  6 --Integer value sent to USB board to
effect desired\n");
      out_to_fifo("\t       torque\n");
      out_to_fifo("\tcolumn  7 --Some encoder value, don't know
which\n");
      out_to_fifo("\tcolumn  8 --Some encoder value, don't know
which\n");
```

```
     out_to_fifo("\tcolumn  9 --Estimated motor angle from Kalman
filter\n");
     out_to_fifo("\t       in degrees * 100 (centidegrees?)\n");
     out_to_fifo("\tcolumn 10 --Estimated link angle from Kaman
filter\n");
     out_to_fifo("\t       in degrees * 100 (centidegrees?)\n");
     out_to_fifo("\tcolumn 11 --Estimated motor velocity from Kalman
filter\n");
     out_to_fifo("\t       in degrees/sec * 100
(centidegrees/sec?)\n");
     out_to_fifo("\tcolumn 12 --Estimated link velocity from Kalman
filter\n");
     out_to_fifo("\t       in degrees/sec * 100
(centidegrees/sec?)\n");
     out_to_fifo("\tcolumn 13 --Experimental column used for parameter
estimation.\n");
     out_to_fifo("\t       Currently parameter being estimated is\n");
     out_to_fifo("\t       transmission damping in Ns/m * 100\n");
     out_to_fifo("\n");

    --first_time;
  }
#endif //PRINT_OUTPUT_HEADER


//////////////////////////////////////////////////////////////////////
/////////
//
//    The output format is as follows:
//
//    column  1 --    Time elapsed in milliseconds
//    column  2 --    Motor position as determined from motor encoder
//                in degrees * 100 (centidegrees?).  This may
//                have been low-pass filtered first
//    column  3 --    Reference signal in angle units in
//                degrees * 100 (centidegrees?)
//    column  4 --    Motor velocity computed from previous motor
//                positions.  This is done in velocity.c.
//                Units are degrees/sec * 100 (centidegrees/sec?)
//    column  5 --    Torque values sent to motor (units?)
//    column  6 --    Integer value sent to USB board to effect desired
//                torque
//    column  7 --    Some encoder value, don't know which
//    column  8 --    Some encoder value, don't know which
//    column  9 --    Estimated motor angle from Kalman filter
//                in degrees * 100 (centidegrees?)
//    column 10 --    Estimated link angle from Kaman filter
//                in degrees * 100 (centidegrees?)
//    column 11 --    Estimated motor velocity from Kalman filter
//                in degrees/sec * 100 (centidegrees/sec?)
//    column 12 --    Estimated link velocity from Kalman filter
//                in degrees/sec * 100 (centidegrees/sec?)
```

```
//    column 13 --     Experimental column used for parameter
estimation.
//               Currently parameter being estimated is
//               transmission damping in Ns/m * 100
//
//////////////////////////////////////////////////////////////////////
/////////


  sprintf(foo, "%d,\t%d,\t%d,\t%d,\t%d,\t%d,\t%d,\t%d,\t",
        globalTime,
        (int)( j1->mpos   * 180 / M_PI * 1e2), //
        (int)( j1->mpos_d * 180 / M_PI * 1e2),
        (int)( j1->mvel   * 180 / M_PI * 1e2),
        (int)(1e5 * j1->tau_d) ,
        j1->current_cmd + ZERO_DAC,
        j1->enc_val,
        j1->enc_offset
        );
  rtf_put(out, foo, strlen(foo));

//old, without parameter estimation

//  sprintf(foo, "%d,\t%d,\t%d,\t%d\n",
//      (int)( j1->jpos    * 180 / M_PI * 1e2),  // estimated motor
angle
//      (int)( j1->jpos_d  * 180 / M_PI * 1e2),  // estimated link
angle;
//      (int)( j1->jvel    * 180 / M_PI * 1e2),  // estimated motor
velocity;
//      (int)( j1->jvel_d  * 180 / M_PI * 1e2)   // link velocity;
//      );
 // rtf_put(out, foo, strlen(foo));


//new, with parameter estimation

  sprintf(foo, "%d,\t%d,\t%d,\t%d,\t%d,\t%d\n",
        (int)( j1->jpos    * 180 / M_PI * 1e2),  // estimated motor
angle
        (int)( j1->jpos_d  * 180 / M_PI * 1e2),  // estimated link
angle;
        (int)( j1->jvel    * 180 / M_PI * 1e2),  // estimated motor
velocity;
        (int)( j1->jvel_d  * 180 / M_PI * 1e2),   // link velocity;
        (int)( stiffness_estimate  * 1e2),   // parameter;
        (int)( damping_estimate  * 1e2)   // parameter;
        );
  rtf_put(out, foo, strlen(foo));

#else //PRINT_ASCII not defined, output binary
```

```
  rtf_put(out, &globalTime, sizeof(unsigned long int)); // 1
  rtf_put(out, &j1->mpos, sizeof(float)); // 2
  rtf_put(out, &j1->mpos_d, sizeof(float)); // 3
  rtf_put(out, &j1->mvel, sizeof(float)); // 4
  rtf_put(out, &j1->tau_d, sizeof(float)); // 5
  rtf_put(out, &j1->current_cmd, sizeof(s_16)); // 6
  rtf_put(out, &j1->enc_val, sizeof(s_24)); // 7
  rtf_put(out, &j1->enc_offset, sizeof(int));
  rtf_put(out, &j1->jpos, sizeof(float));
  rtf_put(out, &j1->jpos_d, sizeof(float));
  rtf_put(out, &j1->jvel, sizeof(float));
  rtf_put(out, &j1->jvel_d, sizeof(float));
  rtf_put(out, &stiffness_estimate, sizeof(double));
  rtf_put(out, &damping_estimate, sizeof(double));

#endif //PRINT_ASCII


}
```

Settings.h

```
/*********************************************************************
****************
 * settings.h is a header file designed to contain experiments
settings
 * used in pulley_board_control.c
 *
 * Cooper Clauson                February 25, 2010
 *

*********************************************************************
***************/


#ifndef _SETTINGS_H__
#define _SETTINGS_H__

/*********************************************************************
*****************
 *
 * REFERENCE SIGNAL TYPE
 *
 * One of the following reference signal types should be defined:
 *    -ZERO
 *    -STEP
 *    -SWEEP
 *    -SUM_OF_SINUSOIDS
 *    -SIMPLE_SINUSOID
 *    -SQUARE_WAVE
 *
 * If more than one is defined, the program behavior is indeterminate.
```

```
 *
*******************************************************************
****************/

/*One of these should be uncommented, the rest should be commented:*/
//#define ZERO
//#define STEP
//#define SWEEP
//#define SUM_OF_SINUSOIDS
//#define SIMPLE_SINUSOID
//#define SQUARE_WAVE
#define ROTATE
/******************************************************************
******************
 *
 * SUM OF SINUSOIDS PARAMETERS
 *
 * The following parameters control the SUM_OF_SINUSOIDS reference
signal.
 * Both SUM_OF_SINUSOIDS_MAGNITUDES and SUM_OF_SINUSOIDS_FREQUENCIES
are six element
 * float array literals of the form "{XXX, XXX, XXX, XXX, XXX, XXX}",
where each XXX
 * is a floating point number.  MAGNITUDE_SCALAR is a floating point
number.
 *
 * SUM_OF_SINUSOIDS_MAGNITUDES contains the magnitudes of the various
component sinusoids
 * (units?), while MAGNITUDE_SCALAR is a premultiplier that multiplies
each of them.
 * SUM_OF_SINUSOIDS_FREQUENCIES is the frequency of each sinusoid in
herz.
 *
*******************************************************************
*****************/

#ifdef SUM_OF_SINUSOIDS
  #define SUM_OF_SINUSOIDS_MAGNITUDES {1,  -0.7,    0.5,    0.8,  -
0.2,  0.1}
  #define MAGNITUDE_SCALAR 0.5
  #define SUM_OF_SINUSOIDS_FREQUENCIES {1./3,   1.5/3,    2./3,
2.5/3,    3./3,    3.5/3}
#endif

/******************************************************************
******************
 *
 * SIMPLE SINUSOID PARAMETERS
 *
```

```
 * The following parameters control the SIMPLE_SINUSOID reference
signal.
 * Both SIMPLE_SINUSOID_MAGNITUDE and SIMPLE_SINUSOID_FREQUENCY are
floating point numbers.
 * SIMPLE_SINUSOID_MAGNITUDE is the magnitude of the signal (units?),
and SIMPLE_SINUSOID_FREQUENCY
 * is the frequency in herz.
 *

**********************************************************************
*****************/

#ifdef SIMPLE_SINUSOID
  #define SIMPLE_SINUSOID_MAGNITUDE 1
  #define SIMPLE_SINUSOID_FREQUENCY 0.5
#endif

/*********************************************************************
********************
 *
 * SWEEP PARAMETERS
 *
 * These parameters are relevant to the SWEEP reference signal.
 *
 * ?????
 *

**********************************************************************
******************/

#ifdef SWEEP
 #define SWEEP_FREQUENCY 1.0
 #define SWEEP_MAGNITUDE PI / 3
#endif

/*********************************************************************
********************
 *
 * ROTATE PARAMETERS
 *
 * These parameters are relevant to the ROTATE reference signal.
 *
 * ?????
 *

**********************************************************************
******************/

#ifdef ROTATE
 #define PI_TO_ANGLE PI / 180
 #define ROTATION_ANGLE 5
 #define NUMBER_OF_ROTATION 0.28 * (360 / ROTATION_ANGLE)
```

```
 #define MEDIUM_SPEED 0.035
#endif

/*****************************************************************
************
 *
 * CONTROL TYPE
 *
 * One of the following control types should be defined:
 *    -P_CONTROL
 *    -PD_TRAJECTORY_FOLLOWING
 *    -PD_STEP_FOLLOWING
 *    -D_CONTROL
 *    -ZERO_CONTROL
 *    -PD_TRAJECTORY_MOTOR_ESTIMATE
 *    -PD_STEP_MOTOR_ESTIMATE
 *
 * If more than one is defined, the program behavior is indeterminate.
 *

******************************************************************
************/

/*One of these should be uncommented, the rest should be commented:*/
//#define P_CONTROL             //In P control, the torque is
proportional between the
                             //desired motor position and the current
motor position
//#define PD_TRAJECTORY_FOLLOWING     //NOTE: This one tends not to
result in vibrations.
//#define PD_STEP_FOLLOWING
//#define D_CONTROL                   //NOTE: Not currently
implemented
//#define ZERO_CONTROL                //NOTE: Not currently
implemented
//#define PD_TRAJECTORY_MOTOR_ESTIMATE
//#define PD_STEP_MOTOR_ESTIMATE           //NOTE: This is the one
that tends to cause vibrations.
#define PI_CONTROL
/*****************************************************************
***************
 *
 * CONTROL GAIN
 *
 * A number of the control types depend on the parameters KP and KD.
KP is the gain
 * related to the position error.  KD is the gain related to the
velocity error.
 *

******************************************************************
**************/
```

```
#define KPV 0.4             // Proportional for Velocity
#define KP 0.02             //       0.7 // Proportional for Position
#define KD 0.01        //  0.006
#define KI 0.01             // Integral for Velocity

// NOTE:
// 0.7 / 0.001 was unstable on 25-Sep-09 without UKF
// 0.4 / 0.005 was stable on 25-Sep-09 without UKF, unless excited
// 0.4 / 0.005 was totally stable on 25-Sep-09 with UKF
// 0.6 / 0.014 was totally stable on 28-Sep-09 with UKF

/*********************************************************************
**********
 *
 * POSITION LOW PASS FILTERING
 *
 * To enable low pass filtering of position, define LPF_POS.
Otherwise, leave
 * it undefined.
 *

**********************************************************************
**********/

/*uncomment to enable, comment to disable*/
#define LPF_POS

/*********************************************************************
***********
 *
 * POSITION LOW PASS FILTERING PARAMETERS
 *
 * The following parameters are coefficients used to "weigh" previous
values of
 * the filtered and unfiltered position.
 *
 * Each should be a four element float array literal of the form
"{XXX, XXX, XXX, XXX}",
 * where each XXX represents a floating point number.
 *
 * The first value in each array multiplies the most recent position
value, and the
 * last value multiplies the oldest.  LPF_IN_COEFFS[0] multiplies the
current encoder
 * value.  LPF_OUT_COEFFS[0] doesn't multiply anything--its value
doesn't matter.
 *

**********************************************************************
************/
```

```
#ifdef LPF_POS
  #define LPF_IN_COEFFS {0.0029,  0.0087,  0.0087,  0.0029}
  #define LPF_OUT_COEFFS {1.0000,  2.3741, -1.9294,  0.5321}
#endif

/**********************************************************************
************
 *
 * VELOCITY FILTERING
 *
 * If FILTER_VELOCITY is defined, then velocities are computed as a
linear combination
 * of previous velocities and previous positions.  This applies both
to the computation
 * of desired velocities from desired positions, and actual velocities
from actual
 * positions.
 *
 * This impacts code in velocity.c
 *

**********************************************************************
************/

//comment this line out to disable filtering
#define FILTER_VELOCITY

/**********************************************************************
***********
 *
 * VELOCITY FILTERING PARAMETERS
 *
 * The following parameters are coefficients used to "weigh" previous
values of
 * the velocity and position.
 *
 * Each should be a four element float array literal of the form
"{XXX, XXX, XXX, XXX}",
 * where each XXX represents a floating point number.
 *
 * The first value in the position coefficient array multiplies the
current position,
 * the second value the last position, the third the position before
that, and so on.
 * The second value in the velocity coefficient array multiplies the
last velocity,
 * the third the velocity before that, etc.  The first value in the
velocity coefficient
 * array does nothing, so it can be anything.
 *
 * Note that these parameters do nothing if velocity filtering is
disabled.
```

```
 *
**********************************************************************
************/

#ifdef FILTER_VELOCITY

  // When importing coefficients from Matlab:
  //
  //  *** CHANGE THE SIGN OF A TERMS ***
  //

  // 100Hz
  #define VF_POSITION_COEFFS {0,66.1659,-30.8672,-35.2987}
  #define VF_VELOCITY_COEFFS {1.0, 1.6005, -0.8538, 0.1518}


  // 50Hz
//  #define VF_POSITION_COEFFS {0, 11.3235, -3.0528, -8.2707}
//  #define VF_VELOCITY_COEFFS {1, 2.1912, -1.6005, 0.3897}

  // 10Hz
  //Poles Zeros configured for triple pole at w = 10Hz
//  #define K_10HZ 0.056517
//  #define VF_POSITION_COEFFS {K_10HZ, K_10HZ, - K_10HZ, - K_10HZ}
//  #define VF_VELOCITY_COEFFS {1, 2.8172, -2.6456, 0.8282}

#endif

#define PRINT_ASCII
#define PRINT_OUTPUT_HEADER

#endif //_SETTINGS_H__
```