

Design of a 1-DOF Force-Reflecting Master-Slave Teleoperator

Gregor Altvater, Stephen Lee, and Barun Singh
EECS 567 Project Report

Abstract: This paper will explore the benefits and drawbacks of using position-based control versus a hybrid position-force control based upon bias estimation. Both a transfer function model and a state space model of a master-slave teleoperator system are developed and compared to find the most suitable control scheme. Furthermore, tests on a physical system are conducted to compare the theoretical results obtained via simulation for the two control schemes with empirical results. Finally, dynamic properties, effect of noise, and modeling error effects are analyzed for both control methodologies.

I. Motivation

In the increasingly complex world of surgery, where operations on a smaller scale have gained increasing importance, the utilization of robotic assistance has become essential. The skills of a human surgeon have a physical threshold, and today's more advanced surgical procedures, such as tumor removal from the brain, require precision that is beyond human capabilities. Force reflection is necessary for such tasks because it gives the operator a more realistic feel of the surrounding environment. Our initial aim was to develop a 3-DOF mechanism, which could be controlled in the x-y-z directions, to allow an operator to make planar incisions. However with inadequate time, complexity of the mechanical design, and limited resources, the scope of the problem was focused onto a 1-DOF system.

Previous work in the field by Goldenberg et al. [1] has produced a 1-DOF force-reflecting master-slave teleoperators. However, their system design included force sensors to measure the applied forces. Our goal, therefore, was to build a 1-DOF force-reflecting master-slave teleoperator without using a force sensor to measure the forces being applied on the systems. Instead of using force sensors, the controllers will use secondary sensors to gain information about the forces being applied. Eliminating the need for extra sensors is an important goal in industry applications where cost is an important issue. In addition, two different methods will be used to control the system: position control and force control.

II. Code Design

Simulation and design analysis were performed using Matlab and Simulink. Once optimal designs were obtained, the controllers were implemented in the hardware setup. The controller C++ code was developed and compiled using the Watcom development environment. An 8-axis Servo-To-Go, Inc. Model 2 I/O card was used to establish the interface between the computer and hardware. This card has eight encoder inputs, eight 13-bit DAC outputs, eight 13-bit ADC inputs, and thirty-two digital I/O lines.

The controller program itself was divided into two sections—interrupt code and task code. The task code set up the system parameters and then looped until 'quit' was selected. The loop within the task code implemented the user interface and displayed data updates to the screen. Two lines were drawn to illustrate the current angle of both the Master and the Slave, and two

text boxes displayed the angle and proportional constant of each system numerically. The control algorithm was implemented in the interrupt code, which was set to interrupt every 10 milliseconds. During each interrupt, the encoder values were read and a new control signal was computed using the control law developed during the simulation phase of the project. Finally, the control signal was outputted as a DAC command through the PC card to an amplifier that drove the physical system.

III. Physical System Design

The physical system consisted of two separate identical subsystems for the master and slave. Each of the subsystems consisted of a motor (Maxon, model #118777) and an encoder (Agilent Technologies, model #HEDS-5700-A02), which were mounted onto a 10"x6"x2" uniformly solid piece of wood, and a steering wheel. Extra triangular supports hold up the system. The steering wheel was formed from a solid, uniform disc of wood 8" in diameter and 1" thick. It was mounted on the 6mm diameter shaft of the wheel through a hole drilled in the wheel's center. The encoder shaft was connected to the motor shaft using a metal collar and electrical tape. A slot was carved into the wooden piece to allow room for the steering wheel. The encoders were connected to the computer via the Model 2 I/O card described previously. Figure 3.1 shows a picture of the final physical system.



Figure 3.1 Physical system (note that master and slave are identical)

IV. System Modeling

System Model

The modeling of the physical components of this system was undertaken per the approach described in [2]. The model of a DC servomotor is shown below (Figure 4.1). There are four main blocks. Armature losses, G_1 , includes motor inductance, L_m , and resistance, R_m . The torque constant, K_t , relates input current to output torque. The inertia and friction block, G_2 , includes J , a combination of rotor, shaft, encoder, and wheel inertia, as well as B , brush and bearing friction. Lastly, $H(s)$ represents the back-EMF term. The input to the system is a voltage, E_a , and the output is angular velocity, ω . I_a represents the current flowing through the motor coils, and T represents the torque exerted by the motor.

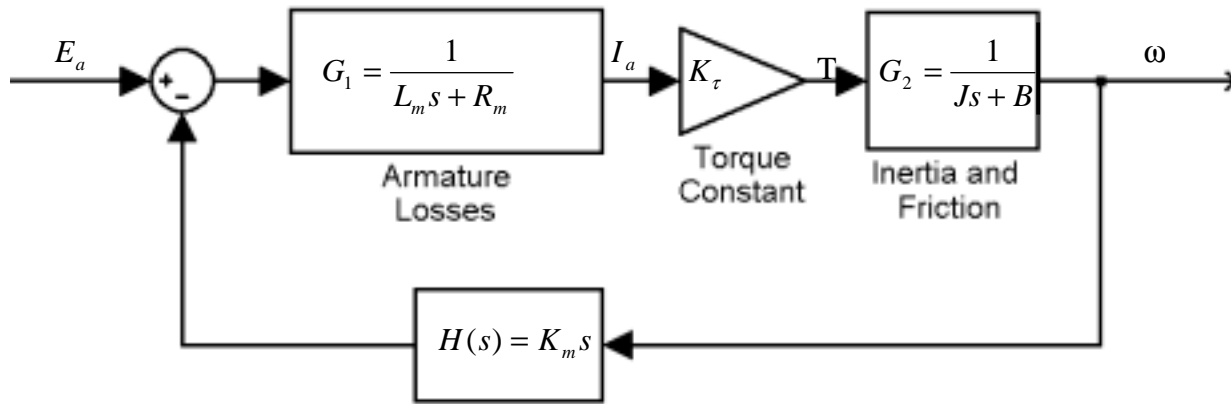


Figure 4.1. Servomotor model proposed by [2]

To simplify the modeling, the back-EMF term, $H(s)$, was initially ignored. This was justified because a transconductance amplifier was to be used. This type of amplifier has adaptive logic that increases the applied voltage to maintain a constant current flow. Unfortunately, a constant current amplifier was not available and corrective circuitry in the model had to be included to take the back-EMF term into account. We chose to ignore the motor inductance, L_m , because it is negligible. The servomotor's resistance, $R_m = 1.23 \Omega$, is included in the aforementioned back-EMF circuitry. The inertial terms and brush friction terms were calculated as follows. The wheel mass is represented by M , and the wheel radius by r . Hence the wheel inertia is:

$$\begin{aligned}
 J_{wheel} &= \frac{1}{2} \cdot M \cdot r^2 \frac{kg}{m^2} \\
 &= 6.75 \times 10^{-4} \frac{kg}{m^2}
 \end{aligned}$$

The rotor inertia and torque constant are given in the specification sheet ($J_{rotor} = 6.55 \times 10^{-6} \text{ kg/m}^2$ and $K_\tau = 0.0389 \text{ Nm/A}$). The bearing and brush friction term was found experimentally to be $B = 0.01 \text{ kg/s/m}^2$. A description of the method used to find this term is included at the end of this section.

A simple model of an integrator was used for the encoder. Combining the motor and encoder, the model for the motor is seen in Figure 4.2:

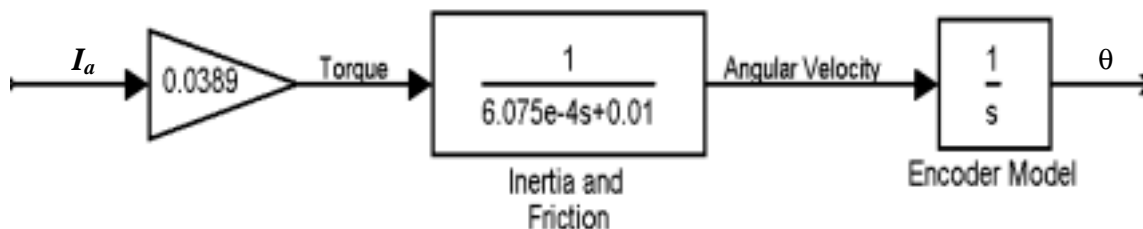


Figure 4.2. Motor and encoder model

The combined transfer function for the motor and encoder is given below. The back-EMF block can be modeled separately and will be given in the position control section.

$$L_{motor} = \frac{K_{\tau}}{J \cdot s^2 + B \cdot s} \frac{rad}{Amp}$$

$$= \frac{0.0389}{6.14 \times 10^{-4} \cdot s^2 + 0.01 \cdot s} \frac{rad}{Amp}$$

Calculation of brush and bearing friction term

In order to obtain a value for the internal friction of the motor, a constant input was applied to the motor. Once the motor reached a constant speed, power to the motor was removed, and the motor was allowed to spin down. The main force acting upon the motor was its internal friction, due mostly to the brushes. A calculation of the time constant of the motor speed's decay gives the frictional coefficient. Figure 4.3 shows the data collected during this test and the calculation of the frictional coefficient.

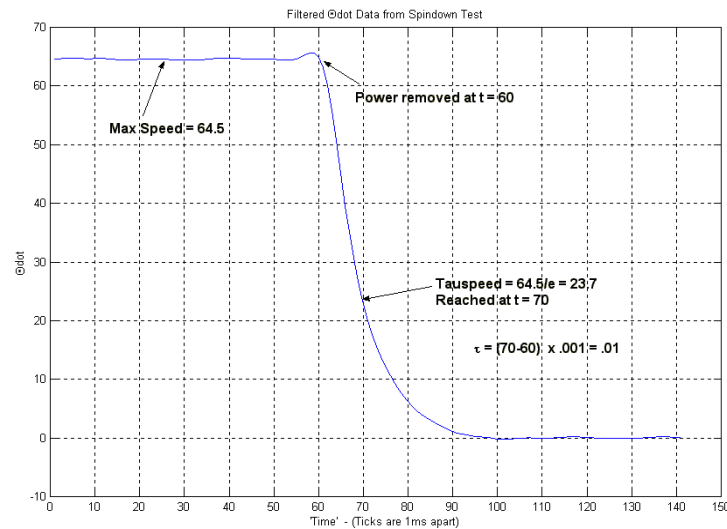


Figure 4.3 Spindown test data

The τ obtained, .01, was inserted into the model of the motor as the B term in the transfer function.

V. Position Control

Position Control Methodology: Software Simulation

In this control paradigm, both the master and slave control loops are controlling position. This means that θ_m is driven to θ_s and θ_s is driven to θ_m (where θ_m and θ_s are the angular positions of master and slave respectively). Since this is a 1-DOF system, forces applied to a wheel can be translated directly into a change in the wheel's angular position or a change in the wheel's angular velocity. If both wheel angles are driven toward each other, the slave will follow the master, while providing feedback about disturbances it encounters. For example, if an obstacle struck by the slave wheel prevents θ_s from reaching θ_m , this discrepancy is fed back to the master and manifests itself as a force that is felt by the hand grasping the wheel.

The software simulation was implemented in Matlab and Simulink. Appendix A illustrates the full model schematic as well as detailed circuitry of the position-based force-feedback controller. The calculations for the open loop transfer functions for each portion of the master loop are given below.

The transconductance amplification circuit modification (back-EMF) transfer function, where ω is the angular velocity of the motor, I_a is the input current into the back-EMF circuit, $R_m = 1.23 \Omega$ is the armature resistance, and V_c is the circuit output voltage, is given by:

$$V_c = [0 \quad R_m] \cdot \begin{bmatrix} \omega \\ I_a \end{bmatrix}$$

A PID controller was used to compensate the plant, since an integrator is required for zero steady state error and a derivative term is needed to control overshoot. The transfer function for the controller, where $K_d = 0.5$ is the differential gain, $K_p = 20$ is the proportional gain, and $K_i = 0.5$ is the integral gain is given by:

$$L_{comp} = \frac{K_d \cdot s^2 + K_p \cdot s + K_i}{s} \frac{Amp}{Volt}$$

The combined transfer function for the motor and encoder were given previously in section III. The layout of the slave circuitry parallels the master and all the values are kept identical. The final Simulink diagram used to simulate the system is given in Appendix A, Figure A.1. A simulation is depicted in Figure 5.1, illustrating the response of the system to a sinusoidal input, with amplitude = 0.04 N \approx 1A and frequency = 1 rad/sec, to the master and a step disturbance, with amplitude = -0.04N applied at time t = 50sec, applied to the slave.

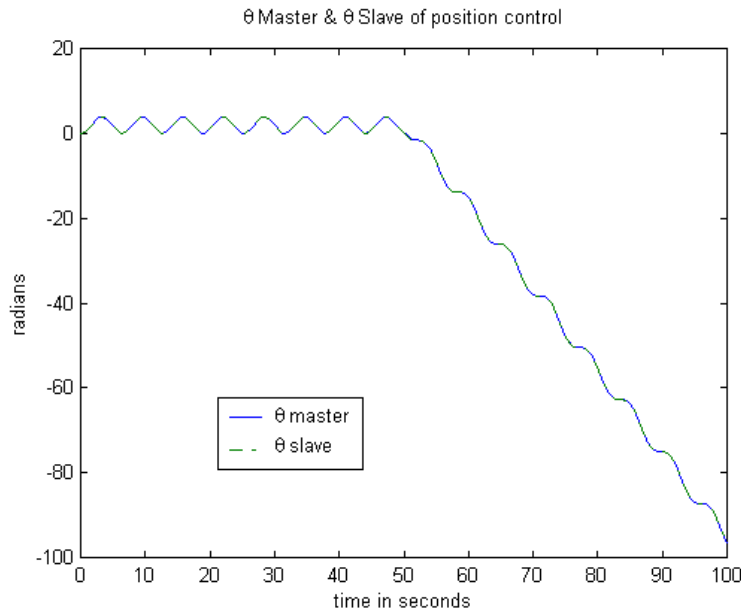


Figure 5.1. Angular response to sinusoidal applied force and step disturbance

As the graph indicates the two angular positions track each other closely. Illustrated in Figure 5.2 is a closer look at the gap between the signals, which might be attributable to time delay in the loops. When looking at the angular separations in various parts of Figure 5.1, it can be seen that the steady state error approaches zero.

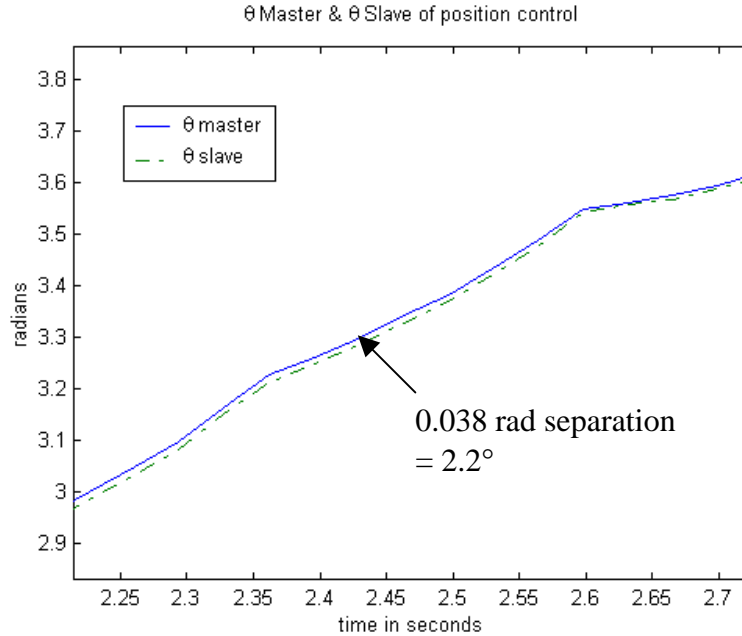


Figure 5.2. Angular response to sinusoidal applied force and step disturbance, closer look

To observe the effects of the time lag between the master and slave angular positions, we view Figure 5.3. Two sinusoids with the same amplitude, 180° out of phase, were fed into the system. In theory, θ_m and θ_s would remain 0. However, as can be seen, there are minute oscillations in the angles, which is attributable to time delays.

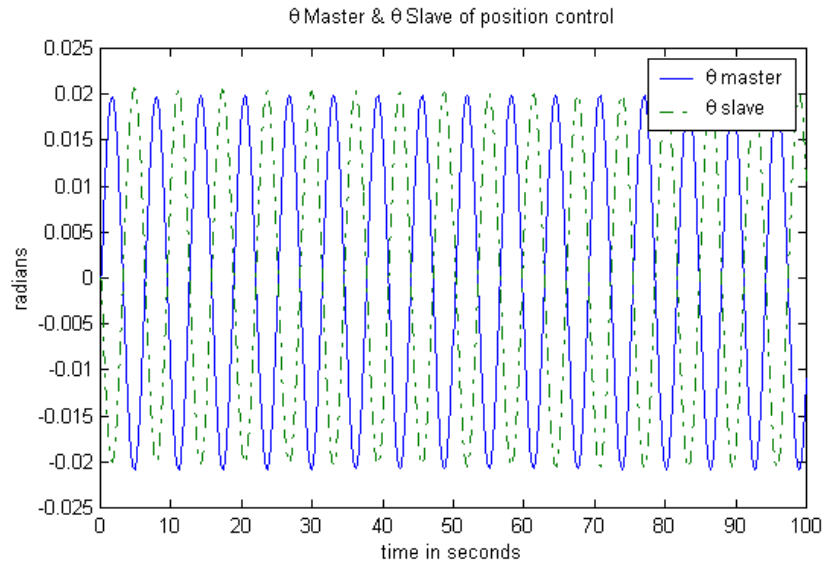


Figure 5.3. Minute oscillations in θ_m and θ_s instead of cancellation

Finally, by changing the controller parameters, one can obtain various types of responses for the system. If the derivative gain is turned up too far, the controller will try to overcompensate for changes in θ_m and θ_s , which in turn will have a dramatic effect on the control signal, u_{master} and u_{slave} . Figure 5.4 illustrates the high frequency oscillations in the control signals with $K_d = 10$.

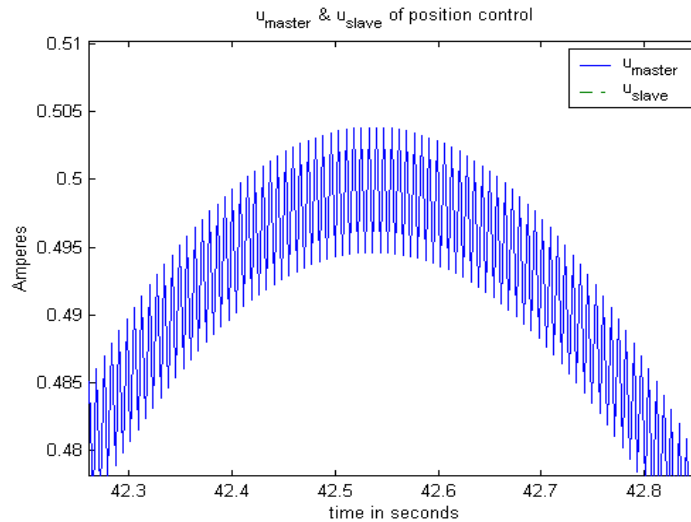


Figure 5.4. Control signal oscillations due to high derivative gain

The other parameters of the PID controller have various effects on the difference $\theta_m - \theta_s = \varepsilon$, where ε is the error as well as the control signals u_{master} and u_{slave} . For instance, by increasing the integrator gain, K_i , ε is decreased. This technique is effective between $K_i \approx 10$ to $K_i \approx 100$ after which any subsequent increase in K_i has no effect on ε . Increasing K_p , decreases ε as well. Again, there is a design trade-off. With large enough K_p , the oscillatory control signal effect, as with large K_d can be observed. This is to be expected, since K_p has similar effects as K_d only to a lesser extent.

Position Control Methodology: Noise Response and modeling error

By looking at certain stability criteria, we were able to investigate the properties of how the position controlled force feedback system reacted to measurement and process noise. Seen in Appendix A, Figure A.1, noise can be injected into the system at various locations, depicted by input nodes 3, 4, 5, and 6. In Figure 5.5, we graphed the bode plots from various noise locations to the outputs, θ_m and θ_s .

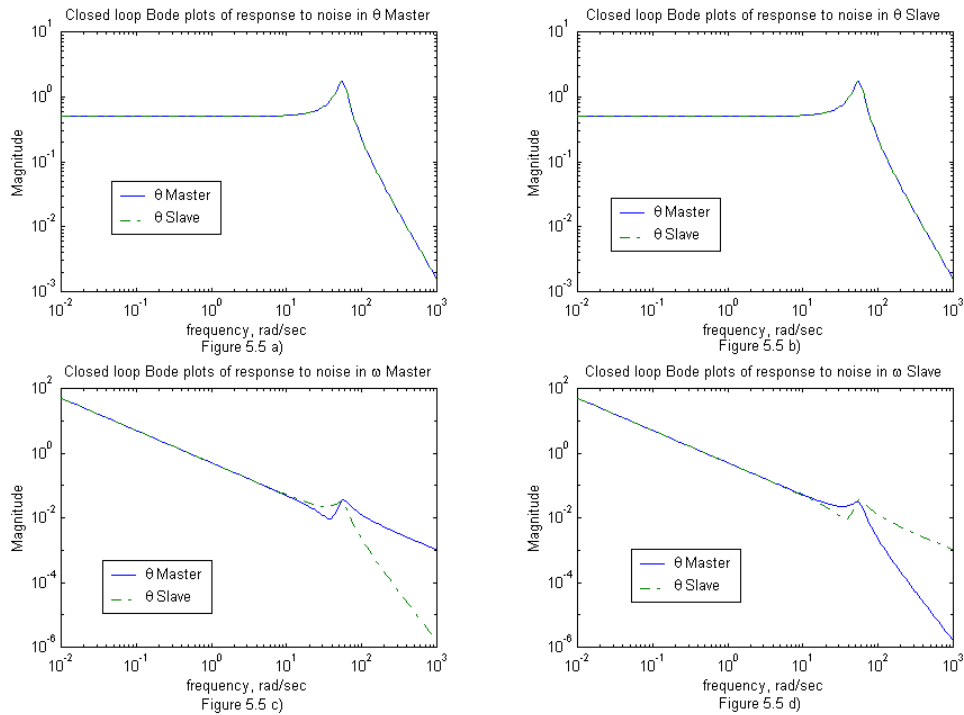


Figure 5.5. Closed loop bode plots of noise response from a) θ_m feedback to θ_m and θ_s b) θ_s feedback to θ_m and θ_s c) ω_m to θ_m and θ_s d) ω_s to θ_m and θ_s

As the plots illustrate, θ_m and θ_s parallel each other in response to noise. Figures 5.5 a & b show that θ_m and θ_s are extremely sensitive to noise between 30 – 80 rad/sec in the θ_m and θ_s feedback loops. The attenuation at the tail of these loops is partly attributable to the bandwidth constraints of the encoder (pole at $s = 0$). As for the noise response in ω_m and ω_s (Figures 5.5 c & d), the bode plot follows almost directly from the effects of the encoder on the signal.

Further analysis was conducted to determine the effects that improper modeling of the motor, wheel and encoder had on the system. We modified the inertial, friction and encoder terms by certain percentages and passed a sinusoid with amplitude 0.0389 N in the Force Applied node and a step at $t = 200\text{sec}$ also with amplitude 0.0389 N. The mean squared error of the angular outputs was compared and is tabulated below in Table 5.1 and Figure 5.6

J (Inertia)	B(Friction)	Encoder	Error Master
+2%	+0%	+0%	6.46E-06
+5%	+0%	+0%	4.03E-05
+10%	+0%	+0%	1.61E-04
+20%	+0%	+0%	6.44E-04
+50%	+0%	+0%	0.004
+0%	+2%	+0%	0.1153
+0%	+5%	+0%	0.6804
+0%	+10%	+0%	2.481
+0%	+20%	+0%	8.345
+0%	+50%	+0%	33.44
+0%	+0%	0.01	1.3
+0%	+0%	0.02	4.72
+0%	+0%	0.05	22.461
+0%	+0%	0.1	60.03
+0%	+0%	0.2	124.2

Table 5.1. Modeling error table

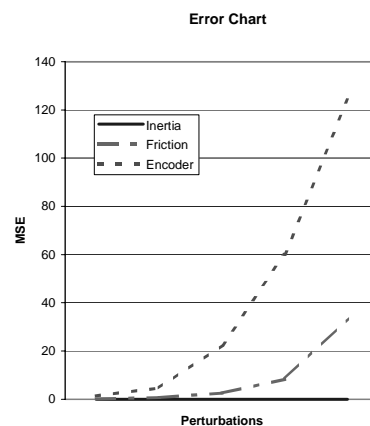


Figure 5.6. Modeling Error chart

It becomes evident that modeling errors in inertia will have minimal effect on system output, whereas encoder errors influence angular position significantly.

Position control in hardware

Position control was implemented in the C++ file *position.cpp*, given in Appendix B. In this program, the interrupt handler reads in the encoder outputs for the Master and the Slave, then uses a PID control law to generate two control inputs. One of these drives θ_m to θ_s , and the other drives θ_s to θ_m . Table 5.2 gives an explanation for symbols used in the program.

	Proportional Constant	Integral Constant	Derivative Constant
Master	K _{pm}	K _{im}	K _{dm}
Slave	K _{ps}	K _{is}	K _{ds}

Table 5.2 Definition of symbols

When position control was implemented in the physical setup, a problem was discovered with our initial hardware setup. This problem was with the original interrupt period of one millisecond. Figure 5.7 illustrates the problem. With a low K_{ds} term, the overshoot on θ_s is rather large. Even with fairly slow oscillations applied to the Master wheel, θ_s overshoots by up to 30° regularly. Also, we can already see the control signal for the Slave, u_s oscillating at high frequency.

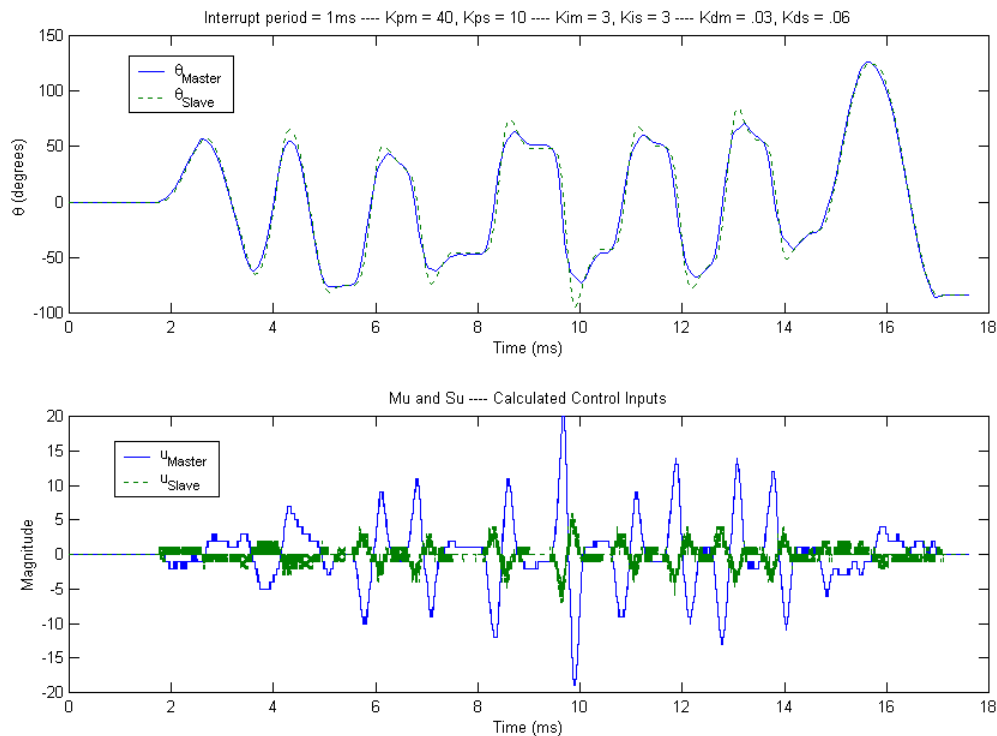


Figure 5.7 Simulation results with short interrupt period and low K_{ds}

In an attempt to reduce the overshoot, K_{ds} was increased to 0.7. The figure below shows the results. Clearly, overshoot in θ_s has been reduced. The maximum value seen below is 10° . However, the slave's control signal's magnitude has increased by a factor of between 5-10. This wildly oscillating control signal is not healthy for the motor and produces an audible humming noise as the motor is driven forward then backwards then forwards again at a rate of one kilohertz, since the period of the interrupt was initially set to one millisecond.

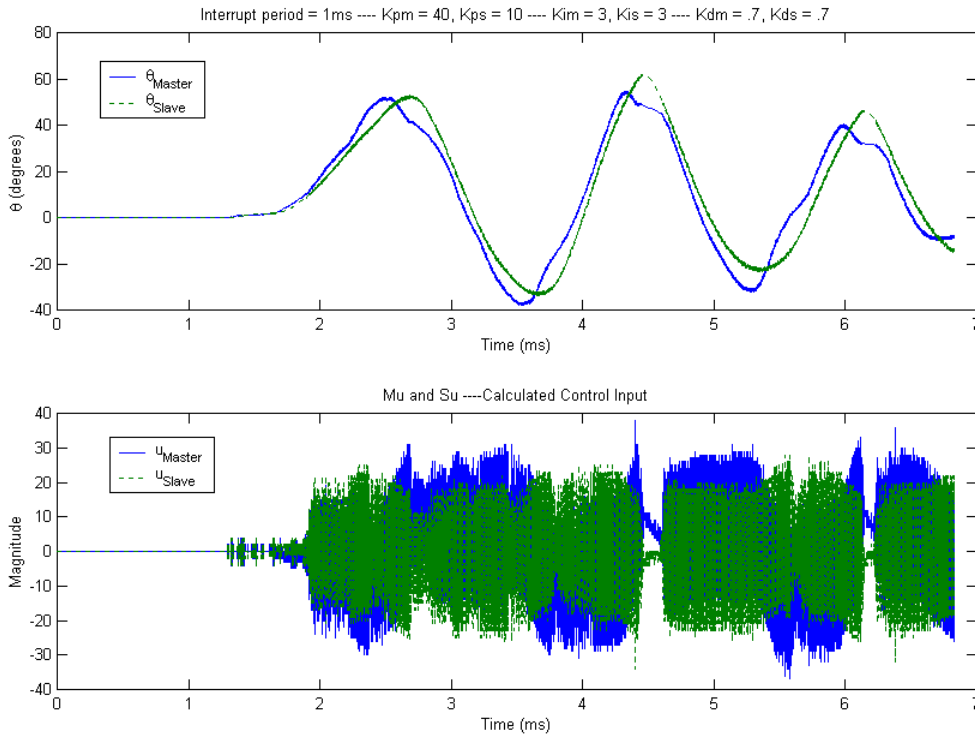


Figure 5.8 Simulation results with short interrupt period and high K_{ds}

To resolve this issue, the interrupt period was lengthened to ten milliseconds. While this did not eliminate the oscillations completely, their magnitude was greatly decreased. The remainder of this section will discuss results from algorithms in which the interrupt period was ten milliseconds.

Another problem encountered in the experimental setup was that the optimal constants obtained through simulation using Matlab were far from optimal. Simulation failed to show large overshoots in θ_s when a step was applied to θ_m . However, overshoot clearly occurred during experimentation. Generally, overshoot is a product of the wheel's inertia, which was accounted for in the system model when the controller was designed. This points to other possible errors in the model for the motor. When the effect of model errors was studied, it was seen that errors in inertia had little effect on controller design, while small errors in the models for brush friction and the encoder had a large impact on the system. Therefore, new control constants had to be generated experimentally using the hardware setup.

Figure 5.9 shows θ_m , θ_s , and their corresponding control signals using the constants generated through simulation. From $t = 0$ to 11 ms, step inputs was applied to the Master wheel. From $t =$

11 to 20 ms, a constant force disturbance was applied to the Slave wheel while the Master wheel was slowly rotated. Finally, from $t = 20$ to 25 ms, the Master wheel was oscillated at a high frequency. The top graph shows unacceptable transient behavior in θ_s as a result of a step in θ_m . Overshoot in θ_s occurs as a result of a high K_{ps} value, and the resulting oscillations die slowly due to underdamping that results from a low K_{ds} value. Also, when the Master wheel is oscillated quickly with a small amplitude, the resulting amplitude of the Slave wheel's oscillations is at least four times as large.

Figure 5.10 shows θ_m , θ_s and their corresponding control signals using a lower K_{ps} and a higher K_{ds} value. From $t = 0$ to 10, there are gradual steps applied to the Master wheel. Around $t=15$, quick oscillations are applied. Overshoot has been decreased and there are no oscillations. At $t = 20$, a constant disturbance is applied to the Slave wheel. The response in all cases is much improved from those seen previously.

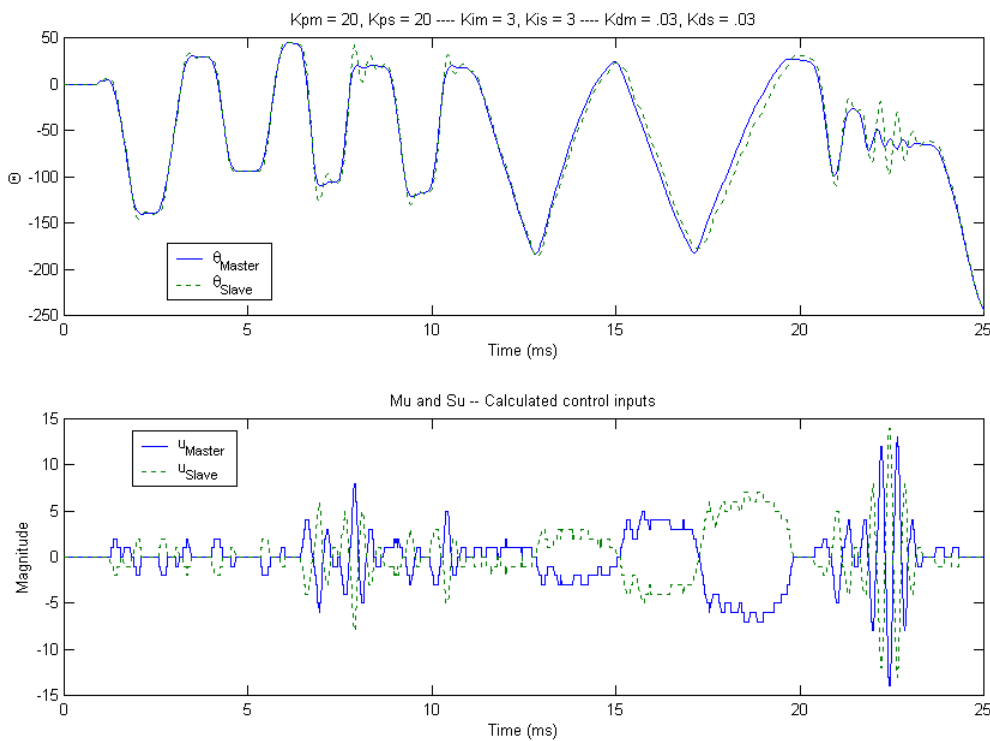


Figure 5.9 Simulation results with high K_{ps} and low K_{ds}

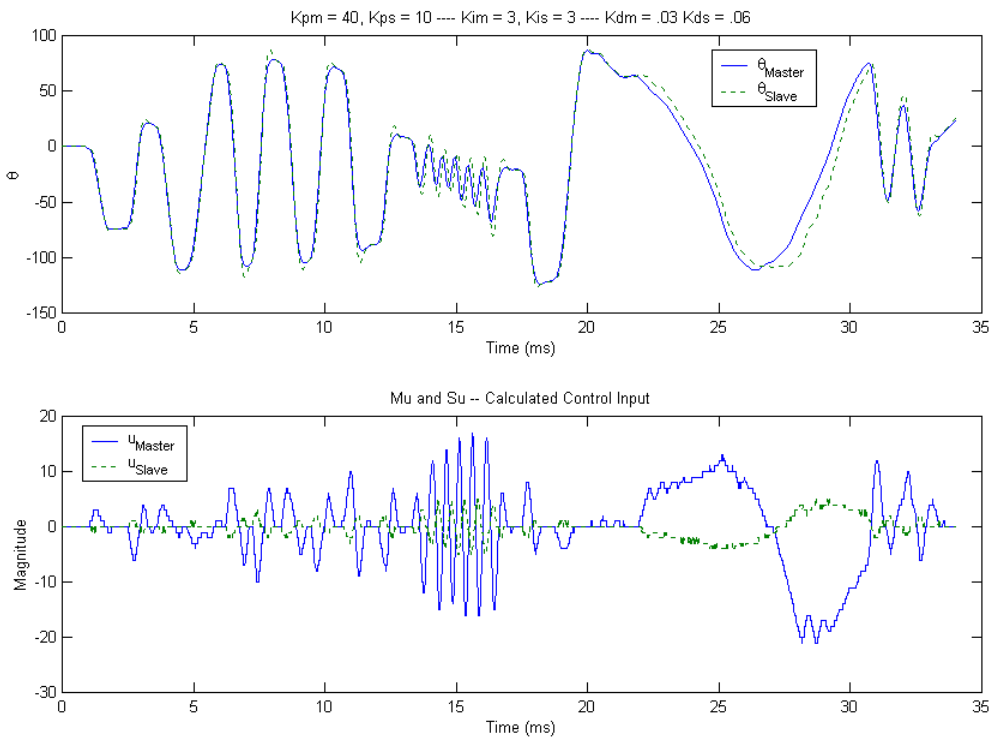


Figure 5.10 Simulation results with lower Kps and higher Kds

VI. Hybrid Control

Another approach to bilateral teleoperation is to use force control rather than position control. In this control paradigm, the master determines the disturbance on the slave, and the same amount of disturbance torque is applied to the master wheel. The human operator then would feel the same disturbance or a proportional disturbance, if there is a scaling difference between the master and slave. Note that in a position control scheme, no information about the disturbance on the slave is relayed back to the human user. In many instances, such knowledge may be useful.

In most conventional force control schemes, a force sensor is used to determine the disturbance to the slave. However, such force sensors can be eliminated through the use of observers. From an observer, an estimate of the disturbance force can be produced. This estimate is then fed back to the master subsystem. A position control scheme as described in previous section is used to control the slave. The resulting scheme is a hybrid system, using position control to control the slave wheel, while using force control in the feedback from the slave to the master.

The first step in designing the observer-based force control scheme is to write state equations describing the system. Ignoring back-EMF and motor resistance, which will be taken account later, the motor dynamics are given by:

$$\theta = \frac{K_\tau}{J s^2 + B s} \cdot I_a \quad \text{where } \theta = \text{angular position of motor shaft}$$

$$\theta (J s^2 + B s) = K_\tau \cdot I_a \quad I_a = \text{input current to motor}$$

$$J \ddot{\theta} + B \dot{\theta} = K_\tau \cdot I_a \quad J, B, K_\tau \text{ are motor parameters described in Section V}$$

Setting the state variables as: $x_1 = \theta$, $x_2 = \dot{\theta}$ yields:

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = -\frac{B}{J} \cdot x_2 + \frac{K_\tau}{J} \cdot E_a$$

Writing the above in matrix form and substituting the actual values for B, J, and K_τ yields:

$$\dot{x} = Ax + Bu \Rightarrow \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -82.3045 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 64.0329 \end{bmatrix} u$$

$$y = Cx + Du \Rightarrow y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

However, this model is not complete, since the torque disturbance has not been taken into account. From Figure 5.2, it can be seen that the torque on the motor shaft is proportional to the input current, I_a , by the constant K_τ . Therefore, if the effect on the states of the system by a current input is governed by the matrix B, the effect on the states by a torque input is governed by the matrix $E = B / K_\tau$. This yields the following state equations:

$$\dot{x} = Ax + Bu + Ed \Rightarrow \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -82.3045 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 64.0329 \end{bmatrix} u + \begin{bmatrix} 0 \\ 1646.09 \end{bmatrix} d$$

$$\dot{y} = Cx + Du \Rightarrow y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

After modeling the system, an observer must be designed to estimate the states x_1 and x_2 , as well as the disturbance, d . Thus, three new states, \hat{x}_1 , \hat{x}_2 , and \hat{d} are inserted into the model above. The general diagram for an observer is shown in Figure 6.1. The only inputs available to the observer are the control input, which is computed internally, and the final input, which is

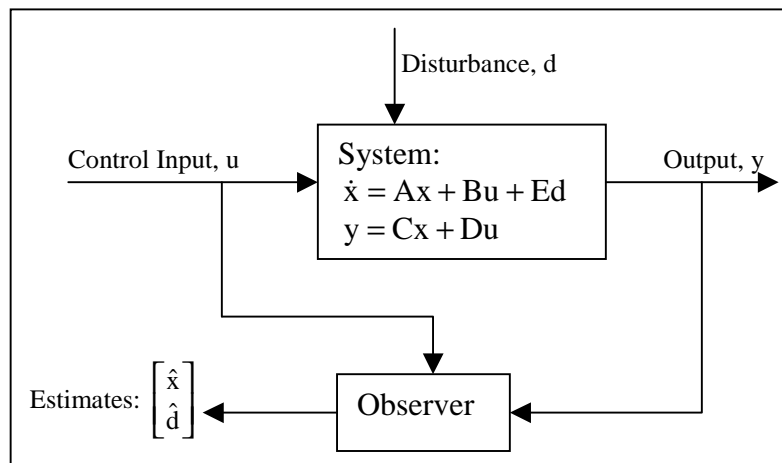


Figure 6.1 Block Diagram of Observer

measured through the encoder. From these inputs, \hat{x}_1 , \hat{x}_2 , and \hat{d} must be calculated.

As explained in [3] and [4], in order for the state and disturbance estimates to be accurate, they must obey the same dynamics as the actual states. The dynamics for the estimates must also contain a forcing term that drives the error, $x - \hat{x}$ to 0. Using the known state equations for the motor, and the assumption that we have a constant disturbance, the dynamics of the estimates are given by:

$$\begin{aligned}\dot{\hat{x}} &= A \hat{x} + B u + E \hat{d} + L1 (y - \hat{y}) \\ \hat{d} &= L2 (y - \hat{y}) \\ \hat{y} &= C \hat{x}\end{aligned}$$

The matrix L1 and constant L2 are chosen to appropriately drive the estimation error to zero. From modern control theory, we know that the observer for this system can be designed independently of any control schemes we may choose to implement (see [4]). Therefore, writing the observer equations in terms of matrices yields:

$$\begin{bmatrix} \dot{\hat{x}} \\ \dot{\hat{d}} \end{bmatrix} = \begin{bmatrix} A - L1 \cdot C & E \\ -L2 \cdot C & 0 \end{bmatrix} \begin{bmatrix} \hat{x} \\ \hat{d} \end{bmatrix} + \begin{bmatrix} B \\ 0 \end{bmatrix} u + \begin{bmatrix} L1 \\ L2 \end{bmatrix} y$$

In order to achieve zero steady state error in the estimation, the “A” matrix in the above equations must have stable eigenvalues. Therefore we choose L1 and L2 such that

$$\begin{bmatrix} A - L1 \cdot C & E \\ -L2 \cdot C & 0 \end{bmatrix} = \begin{bmatrix} A & E \\ 0 & 0 \end{bmatrix} - \begin{bmatrix} L1 \\ L2 \end{bmatrix} \begin{bmatrix} C & 0 \end{bmatrix}$$

has eigenvalues in the open left-half plane (eigenvalues with negative real parts). The eigenvalues of this matrix were placed at -70 , -80 , and -90 , yielding observer gains of $L1 = [157.7 \quad 6121]^T$, and $L2 = 306.18$.

The motor models for the master and slave in the system are identical, allowing the same observer design to be used for both. The disturbance estimate obtained from the slave observer is fed back as a control input to the master. Thus, the human user at the master interface feels the disturbance and inputs some force to the master wheel to compensate. This input, added to the signal generated by the controller, is used as the control signal for the slave. In the resulting loop, the controller must be designed so that the output angles of the master and the slave are equal. A PID controller is used to drive the difference between the outputs of the master and slave motors to zero. The output of the controller is given by:

$$\begin{aligned}\text{Controller output} &= K_p \cdot (\theta_m - \theta_s) + K_i \cdot \left(\int \theta_m - \int \theta_s \right) + K_d \cdot (\dot{\theta}_m - \dot{\theta}_s) \\ &= K_p \cdot (x_{1m} - x_{1s}) + K_i \cdot \left(\int x_{1m} - \int x_{1s} \right) + K_p \cdot (x_{2m} - x_{2s})\end{aligned}$$

Therefore, if an observer is used on the master, as well as the slave, to estimate the states of the plant, the control signal can be generated through the observed states. The final Simulink drawing for the system using force control is given in Appendix C.

The simulation results of the observer-based force-feedback system were very similar to the results achieved through position control. The angular outputs of the master and slave were

almost identical when using controller gains of $K_p = K_i = 100$ and $K_d = 10$. Simulation results are shown in Fig. 6.1, and an error plot is given in Fig. 6.2.

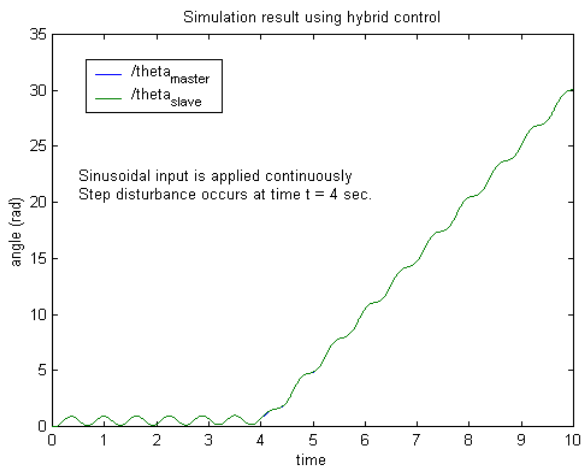


Figure 6.1 Simulation Results

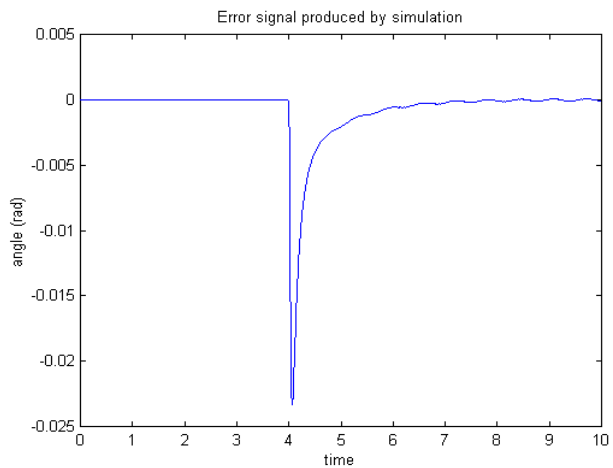


Figure 6.2 Error between master and slave angles

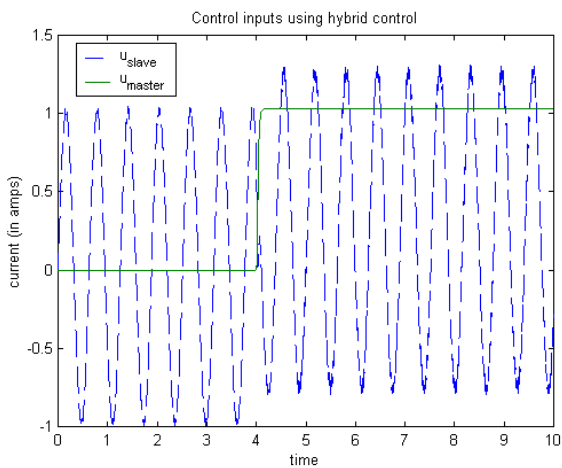


Figure 6.3a Control inputs using hybrid control

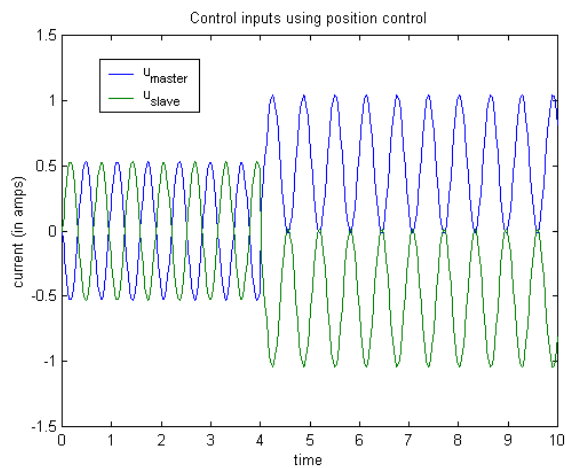


Figure 6.3b Control inputs using position control

The greatest difference in outputs was approximately 0.02 rad (1.15 deg). The control signals required to achieve these results are plotted in Fig. 6.3a. Note that the control signals, u_m and u_s , vary between ± 1.3 amps, which is below the saturation level of our motors. Figure 6.3a also verifies that our system is modeled correctly. As expected, the control signal to the master is simply equal to the disturbance present at the slave, and the control signal to the slave is the input supplied by the human (in this case a 1 amp sinusoid) plus the signal created by the position controller. Notice that when the control input to the master increases, the control input to the slave also increases. The control signals generated when applying the same input and disturbance forces to a pure position control scheme are shown in Figure 6.3b. It can be seen that when using position control, unlike hybrid control, the control signal for the master and slave are opposite in sign. This difference is to be expected, and will always exist between these two types of control schemes. The discrepancy is due to the fact that in the hybrid control

scheme, only the slave uses position control, and the input to the slave motor is the input to the master motor plus the control input, whereas in the position control scheme, both the master and slave use position control, and the only input to the slave motor is the control signal.

VII. Discussion

In the final analysis, neither one of the two control schemes discussed can be said to be absolutely better than the other. Both methods produce comparable results in terms of the final result. That is, the angular position of the master and slave wheels is almost equal (steady state error is zero) for both control methodologies. Hybrid control has the advantage that it supplies the user with a model of the disturbance occurring at the slave. Using hybrid control, the system could be designed so that the user feels twice (or some proportion of) the disturbance on the wheel as well. In order to do this without the observer-based control, one or more force sensors must be used, which could be expensive, or undesirable due to the physical design of the system. In the position-based scheme, one can similarly increase the force felt by the user, however it would not be implemented through the same, straightforward method. To increase feel on the master wheel, one would increase the proportional constant on the master controller. This method, as previously stated, is not directly proportional to the amount of the disturbance. However, such advantages in hardware come at a cost in software, for the observer based method. The hybrid control scheme is much more difficult to implement in software than the position control scheme. This is due to the fact that numerical integration is necessary in order to solve the differential equation describing the observer.

VIII. Conclusion

In the final analysis, it was found that position control works well in both simulation and experimentation. The hybrid control methodology appears to work just as well as position control in simulation. The coding necessary to implement hybrid control is much more complex than for position control, so no time was available in the course of this project to implement the hybrid control in hardware. Also, the physical design had many limitations. Shoddy construction added unmodeled errors. The physical system also had room for improvement. One specific area that could be improved is the driving motors. We found that the torque output of the motors used was not adequate to supply a full range of force feedback. For example, if the slave hits an immovable obstacle, the user is still able to turn the master wheel. A next-generation design would include precise machining and better quality components.

References

- [1] Goldenberg, A.A., Bastas, D., and Strassberg, Y. "On the bilateral control of master-slave teleoperators" *Robotersysteme.*, vol. 7, no. 2, May 1991 p91-99
- [2] Phillips, C. L.: Harbor, R. D. : *Feedback Control Systems*, 3rd ed., Englewood Cliffs, New Jersey: Prentice Hall, 1996
- [3] Freudenberg, J. S.: Hollot, C. V.: Looze, D. P.: *A First Graduate Course in Feedback Control*, Ann Arbor, Michigan, 2000
- [4] Skogestad, S. : Postlethwaite, I.: *Multivariable Feedback Control: Analysis and Design*: New York, John Wiley and Sons, 1996
- [5] Strassberg, Y., Goldenberg, A.A., Mills, J.K. "Stability Analysis of a Bilateral Teleoperating System" *Journal of Dynamic Systems, Measurement, and Control*, vol 115. Sept 1993. p 419-426

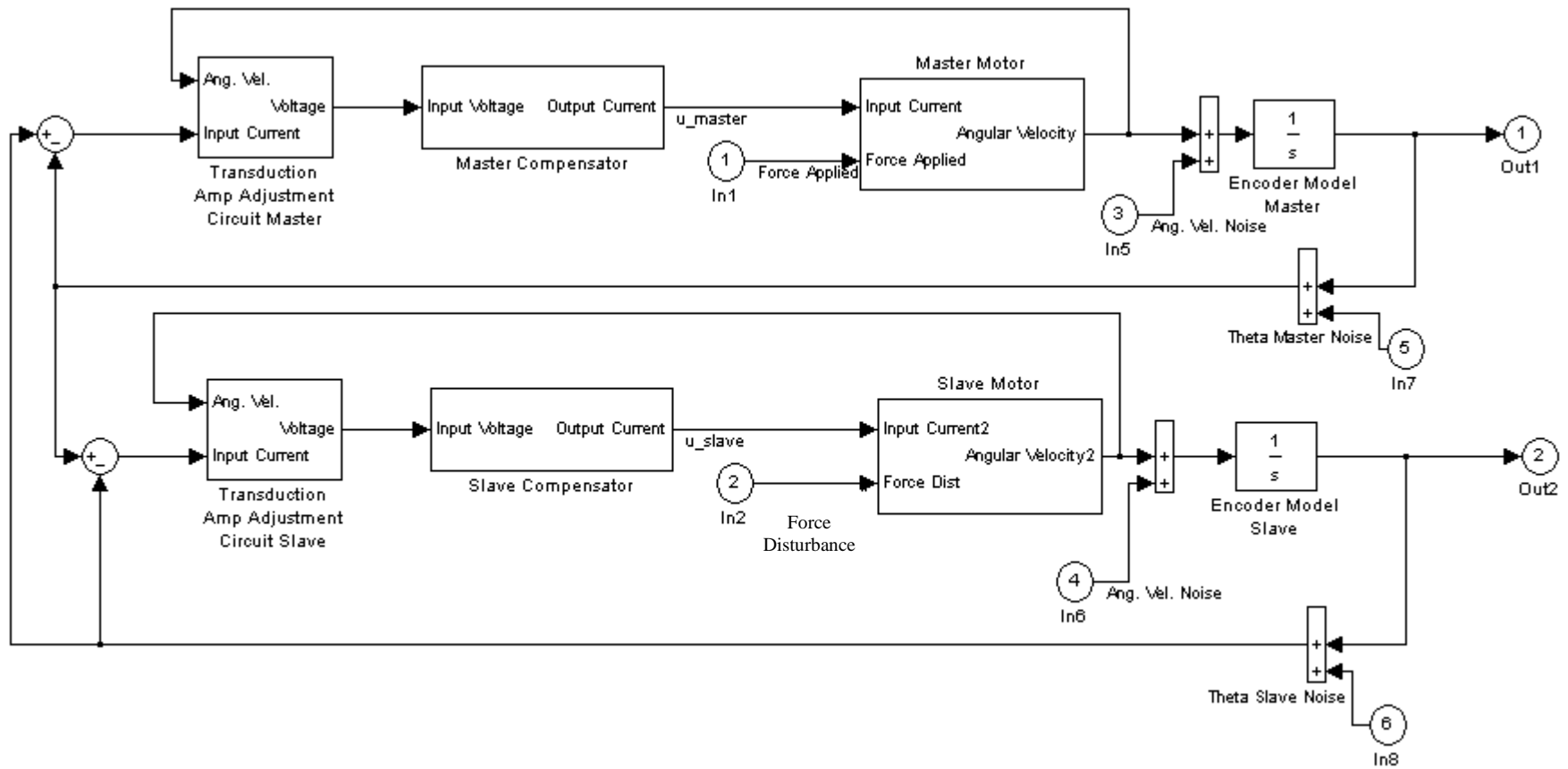


Figure A.1. Full Schematic of the Position Control

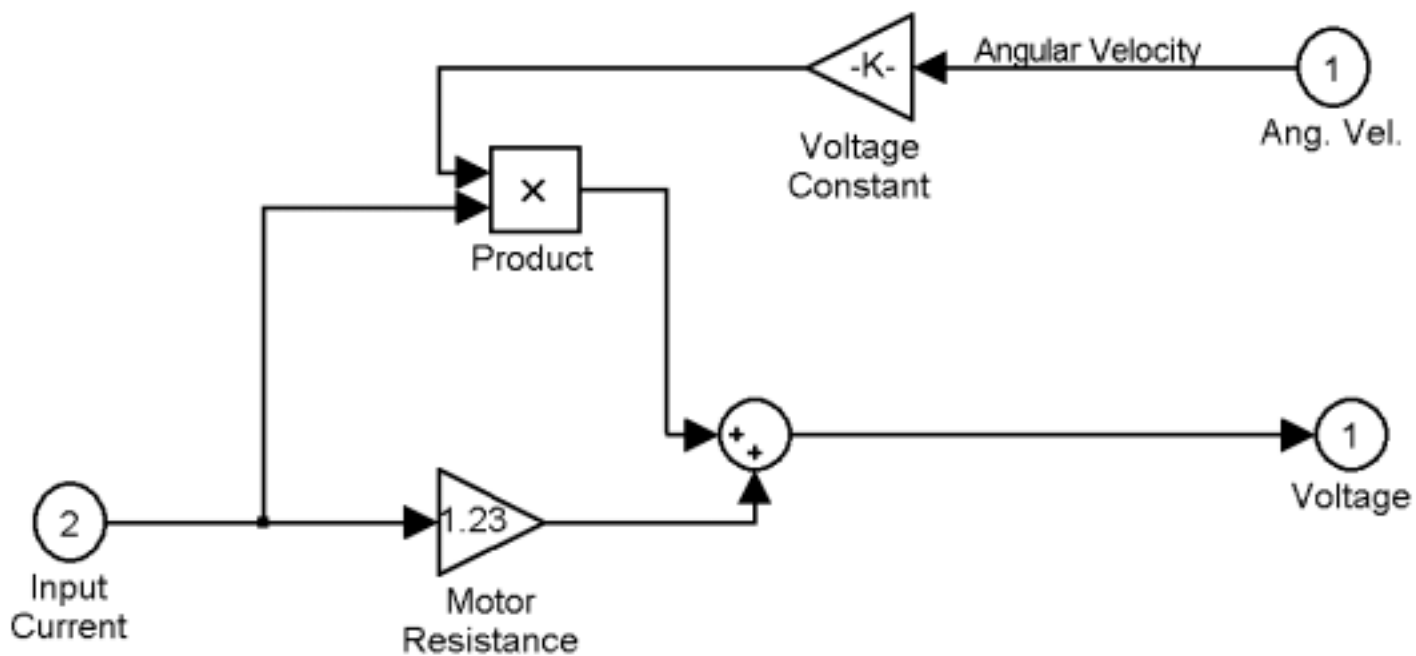


Figure A.2. Transduction amplifier adjustment circuitry

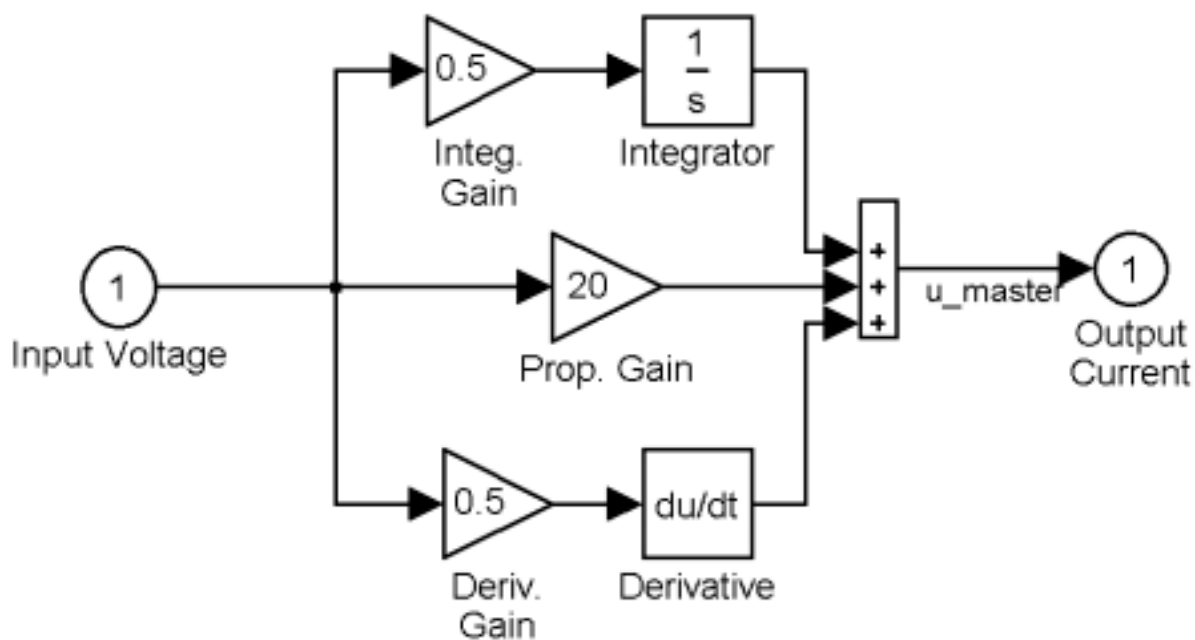


Figure A.2. Compensator circuitry

Appendix B

```
/* position.cpp - This file performs Master-Slave Force-feedback teleoperator
 * control using position control.
 */

#include <stdio.h>
#include <dos.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <graph.h>
#include <time.h>
#include <i86.h> //needed for delay function
#include "stgdefs.h"
#include "controlr.h"
#include "irq_defs.h"
#include "examples.h"
#include "..\..\Common\h\viewport.hpp" //graphics routines

#define DATA_ACQ_SAMPLES 100000L //Max num of data points to store and spool to disk.
#define SAMPLE_PERIOD .01 //10ms, Used in control law. Won't affect interrupt rate.

#define MASTER_ENC_CHAN 0 //Define master/slave motor/enc I/O channels
#define MASTER_MOTOR_CHAN 0
#define SLAVE_ENC_CHAN 2
#define SLAVE_MOTOR_CHAN 1

//function prototypes
double get_thetaM(void);
double get_thetaS(void);
double get_KpM(void);
double get_KpS(void);
void __interrupt __far MyInterruptHandler(__CPPARGS);

//global variable declarations
extern Controller StgController;
extern IrqDataStruct IrqData;
long encoderdata[4][DATA_ACQ_SAMPLES]; //4 X DATA_ACQ_SAMPLES data array
//Will store thetas and control inputs
long lastsampleindex=0;

//The volatile keyword tells the compiler that this variable's contents
//may be altered at any instant by an interrupt service routine, and therefore
//should not be cached in a register.
volatile double thetaM;
volatile double KpM=40.0; //Master Proportional Constant
volatile double thetaS;
volatile double KpS=10.0; //Slave Proportional Constant

/*****
/*
/* main
/*
/*
/*****/

void main(void)
{
    unsigned long int dataIndex;
    struct videoconfig vc;
    FILE *fil;
```

```

double tempthetaM=0.;
double tempthetaS=0.;
char paramstring1[100],paramstring2[100],paramstring3[100],paramstring4[100];
int done=0;

//Set up the video.
_setvideomode(CG1024X768X256);

//Creat a viewport.
VIEWPORT vp2(vc,50,50,924,668);//coords are x0, y0, xf-x0, yf-y0

//Set characteristics of the viewport.
vp2.SetBGColor(0);
vp2.SetTextColor(15);
vp2.SetFGColor(3);
vp2.SetOrigin(0,0);
vp2.SetBorder(3);

vp2.GetReady();

//Wait until the video mode switches over before we start doing control.
while(!kbhit()); // Controller starts upon keypress.
getchar();

//Zero data storage array.
for(dataIndex=0;dataIndex<DATA_ACQ_SAMPLES;dataIndex++) {
    encoderdata[0][dataIndex]=0;
    encoderdata[1][dataIndex]=0;
    encoderdata[2][dataIndex]=0;
    encoderdata[3][dataIndex]=0;
}

//Zero the encoders.
StgController.SetEncoderCounts(MASTER_ENC_CHAN,0);
StgController.SetEncoderCounts(SLAVE_ENC_CHAN,0);

//Set up interrupt stuff.
StgController.SelectInterruptPeriod(_10_MILLISECONDS); //10ms interrupt period
StgController.int_stal(MyInterruptHandler);
StgController.StartInterrupts();

//Main screen update loop.
while(!done){// Do non-control loop stuff here.
    //Draw lines to screen (thetaM and thetaS).
    vp2.SetFGColor(0);
    vp2.Line(694,50,694+300*sin(tempthetaS),50+300*cos(tempthetaS));
    vp2.Line(230,50,230+300*sin(tempthetaM),50+300*cos(tempthetaM));
    tempthetaS=get_thetaS();
    tempthetaM=get_thetaM();

    //Store parameters in strings.
    sprintf(paramstring1,"thetaM=%lf",(float)get_thetaM());
    sprintf(paramstring2,"KpM=%lf",(float)get_KpM());
    sprintf(paramstring3,"thetaS=%lf",(float)get_thetaS());
    sprintf(paramstring4,"KpS=%lf",(float)get_KpS());
    //Clear parameter area on screen and draw parameters to screen.
    vp2.Rectangle(0,390,430,550,470);
    vp2.Rectangle(0,90,430,250,470);
    vp2.MoveTo(400,450);
    vp2.PutsXY(paramstring3);
    vp2.MoveTo(400,440);
    vp2.PutsXY(paramstring4);
}

```

```

vp2.MoveTo(100,450);
vp2.PutsXY(paramstring1);
vp2.MoveTo(100,440);
vp2.PutsXY(paramstring2);
vp2.SetFGColor(3);
vp2.Line(694,50,694+300*sin(tempthetaS),50+300*cos(tempthetaS));
vp2.Line(230,50,230+300*sin(tempthetaM),50+300*cos(tempthetaM));
delay(30); //Delay to reduce screen flicker.
//Handle keyboard hits.
if(kbhit()){
    switch(getch()){
        case '+'://Increment master proportional gain.
        case '=':
            _disable();
            KpM+=0.1;
            _enable();
            break;
        case '-'://Decrement Master proportional gain.
            _disable();
            KpM-=0.1;
            _enable();
            break;

        case 's'://Increment slave proportional gain
            _disable();
            KpS+=0.1;
            _enable();
            break;
        case 'x'://Decrement slave proportional gain
            _disable();
            KpS-=0.1;
            _enable();
            break;

        case 'q'://Exit program.
            done=1;
            break;
        default:
            break;
    }//end switch

} //end if
} //end while

// Turn off interrupts and clean up.
StgController.StopInterrupts();
StgController.un_int_stal();
StgController.RawDAC(MASTER_MOTOR_CHAN,0);
StgController.RawDAC(SLAVE_MOTOR_CHAN,0);

_setvideomode(_DEFAULTMODE);

//Spool stored data to disk.
fil=fopen("encdat.ext","w");
if(fil!=NULL){
    for(dataIndex=0;dataIndex<lastsampleindex;dataIndex++)

fprintf(fil,"%li\t%li\t%li\t%li\n",encoderdata[0][dataIndex],encoderdata[1][dataIndex],
encoderdata[2][dataIndex],encoderdata[3][dataIndex]);
    fclose(fil);
} //end if
else

```

```

        printf("Couldn't open output file for writing!\n");
} // end main

double get_thetaM(void)
{
    //This function gets the correct value of the variable thetaM. It deals with the
    //shared data problem between task code and interrupt code.
    double temp=thetaM;
    while(temp!=thetaM)
        temp=thetaM;
    return temp;
} //end get_thetaM

double get_thetaS(void)
{
    //This function gets the correct value of the variable thetaS. It deals with the
    //shared data problem between task code and interrupt code.
    double temp=thetaS;
    while(temp!=thetaS)
        temp=thetaS;
    return temp;
} //end get_thetaS

double get_KpM(void)
{
    double temp=KpM;
    while(temp!=KpM)
        temp=KpM;
    return temp;
} //end get_KpM

double get_KpS(void)
{
    double temp=KpS;
    while(temp!=KpS)
        temp=KpS;
    return temp;
} //end get_KpS

void __interrupt __far MyInterruptHandler(__CPPARGS) //Interrupt Handler
{
    static unsigned long milliseconds=0;
    static long dataindex=0;
    static LONGBYTE encoderVals[8];
    static long MDACcommand=0;
    static long SDACcommand=0;
    static double Mu=0;
    static double Su=0;
    double encoderValsDouble[8];
    double Mr,My,Me,Sr,Sy,Se;
    static double Me_old,Se_old;
    static int first_interrupt=TRUE;
    static double Mi=0.0,Si=0.0;
    double Md,Sd;
    double KiM=3, KdM=.03; //DEFINE I and D CONTROL CONSTANTS
    double KiS=3, KdS=.06; //for Master and Slave

    //Disable other interrupts so this routine won't be interrupted.
    _disable( );
    fix_es();
    fOutP( IrqData.Mb_ControlReg, IrqData.SpecificEOI ); // allow other interrupts.

    //Increment our clock.

```

```

milliseconds++;

//Here's our control law.
//First, read in encoders.
StgController.EncoderLatch();
StgController.EncReadAll(encoderVals);

//Store encoder values and control inputs for writing to disk later.
encoderValsDouble[0]=(double)encoderVals[MASTER_ENC_CHAN].Long;
encoderValsDouble[1]=(double)encoderVals[SLAVE_ENC_CHAN].Long;
if(dataindex<DATA_ACQ_SAMPLES){
    encoderdata[0][dataindex]=encoderVals[MASTER_ENC_CHAN].Long;
    encoderdata[1][dataindex]=encoderVals[SLAVE_ENC_CHAN].Long;
    encoderdata[2][dataindex]=Mu;
    encoderdata[3][dataindex]=Su;
    lastsampleindex=dataindex;
    dataindex++;
}

//end if

//Now calculate position and error signals.
thetaM=My=encoderValsDouble[0]*2.*3.1416/2000.; //2000 counts per revolution
thetaS=Sy=encoderValsDouble[1]*2.*3.1416/2000.;
Sr=thetaM; //Drive Slave to thetaM
Mr=thetaS; //Drive Master to thetaS
Me=Mr-My; //Define error signals
Se=Sr-Sy;

//If this is the first time that this function is being called,
//we must initialize the old errors.
if(first_interrupt){
    Me_old=Me;
    Se_old=Se;
    first_interrupt=FALSE;
}

//end if

//Now calculate error integrals and derivatives
Mi+=Me*SAMPLE_PERIOD;
Md=(Me-Me_old)/SAMPLE_PERIOD;
Si+=Se*SAMPLE_PERIOD;
Sd=(Se-Se_old)/SAMPLE_PERIOD;

//Next, use a PID control law to calculate motor control signal.
Mu=KpM*Me+KiM*Mi+KdM*Md;
Su=KpS*Se+KiS*Si+KdS*Sd;

MDACcommand=(long)(-Mu/10.*(double)0x0FFF);
SDACcommand=(long)(-Su/10.*(double)0x0FFF);

//Finally, output control signal to motor.
StgController.RawDAC(MASTER_MOTOR_CHAN, MDACcommand);
StgController.RawDAC(SLAVE_MOTOR_CHAN, SDACcommand);

//Store current values of the errors for use in next interrupt.
Me_old=Me;
Se_old=Se;

//Re-enable interrupts.
_enable();

}

} //end MyInterruptHandler

```

Appendix C

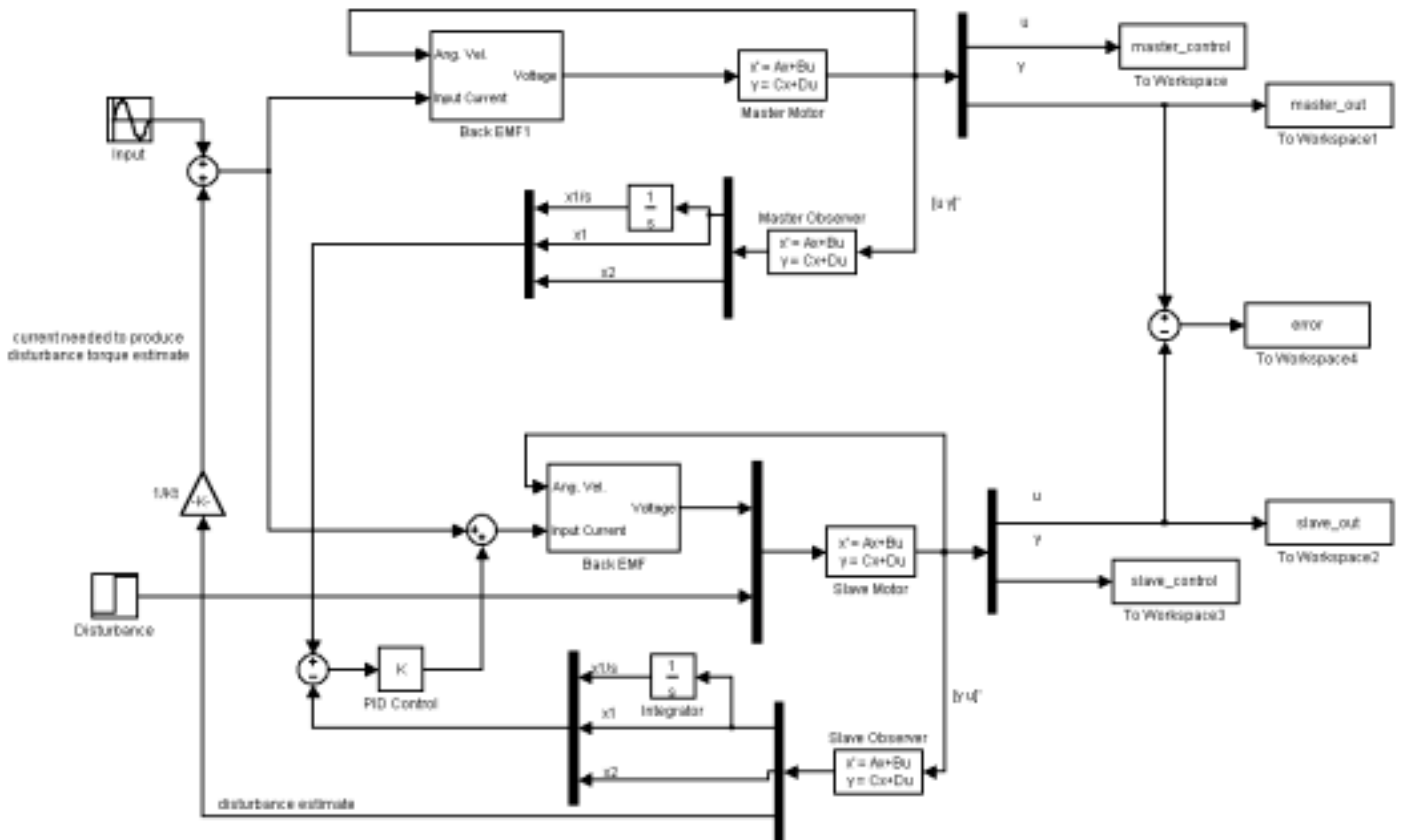


Figure C.1 Hybrid Control Schematic