Design and Analysis of a Ball-Balancing Plate

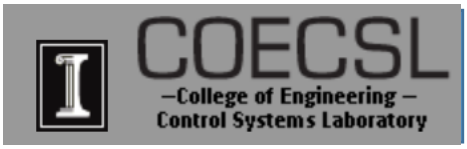
Maximilian Sieb

Bachelor Thesis

Spring 2016

TECHNISCHE UNIVERSITÄT DARMSTADT

TECHNISCHE
UNIVERSITÄT
DARMSTADT



Maximilian Sieb
Matrikelnummer/*Student ID in Germany*: 2714433
Studiengang/*Major*: Mechanical and Process Engineering

Bachelorarbeit/*Bachelor Thesis*

Thema/*Topic*: Design and Analysis of a Ball Balancing Plate

Eingereicht/*Handed in*: 05/21/2016

Betreuer/*Advisor*: Dan Block *(University of Illinois at Urbana-Champaign, Control Systems Laboratory)*

Prof. Tropea
Maschinenbau
Stroemungslehre und Aerodynamik
Technische Universität Darmstadt
Hochschulstraße 1
64289 Darmstadt

Erklärung zur vorliegenden Arbeit gemäß § 22/7 bzw. § 23/7 APB

Hiermit versichere ich, die vorliegende Bachelorthesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Sämtliche aus fremden Quellen indirekt oder direkt übernommenen Gedanken sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen und wurde noch nicht veröffentlicht.

Ort, Datum

Name, Vorname

Content

1 Introduction

For the final project of my undergraduate studies, I am conceiving and building a model of a ball balancing system. This system consists of a small plate that is mounted on its center of mass that possesses two rotational degrees of freedom provided by a double U-joint. The plate itself can be rotated by a set of direct-current motors that are connected to the plate by mechanical links.

The plate is to be moved in such a way that the ball rolls to the desired position that is given to the control system. The mechanics of how the plate is to be actuated and the mechanical components themselves have to be designed and built. Furthermore, the feedback control algorithms have to be implemented using a microcontroller. A sensor system consisting of a set of encoders attached to the DC motors and a resistive touchpad provides the necessary information to effectively balance the ball.

As for the control algorithm that takes the major part of this thesis, many approaches already exist, e.g. relatively straight-forward methods such as simple PID-feedback or elaborate methods such as ones utilizing fuzzy control [1] or sliding mode control [2]. My approach will utilize an inverse dynamic controller that will linearize the fully actuated part of the system by providing adequate feed-forward control combined with a full state feedback controller applied to the entire linearized system.

Another fact that should be mentioned is that some approaches like [2] assume the plate's angular displacement as the input to the system whereas others assume the torque provided by the DC motors as the input. Thus, it can be chosen whether to utilize a servo-motor that directly provides an angle as the control input or a DC-motor that provides a torque.

In this thesis, a DC-motor will be used because this provides much more flexibility in terms of choosing an appropriate control torque in order to design more elaborate control algorithms.

Firstly, I will briefly describe the system dynamics and the mathematical derivations along with it. Then I will introduce the mechanical design I came up with which is followed by an extensive simulation of both simple PID-control and the more elaborate non-linear state feedback controller.

In the end I will also cover the electronics and software implementations that came along with implementing the control algorithms.

2 Mechanical System Analysis

The system from a mechanical standpoint consists of the plate, a support beam on which the center of the plate is mounted and can freely rotate about two axes, two sets of one DC-motor, two connected links that connect the plate with the output shaft of the DC-motor and a casing that mounts the DC-motors on the ground. The geometry of the components can be seen in the table below.

Parameter	Description
a	Length of link 1
b	Length of link 2
c	Length from center of motor to connection point with plate
h_t	Height of center of plate
h_p	Height of connection point
h_m	Height of motor
d_m	Horizontal distance of motor from center of plate
d_p	Direct distance of connection point from center of plate
θ_m^x	Motor angle
α	Plate angle
δ	Angle between horizon and c
β	Angle between horizon and b

Table 2.1: Declaration of the main dimensions of the linkage system

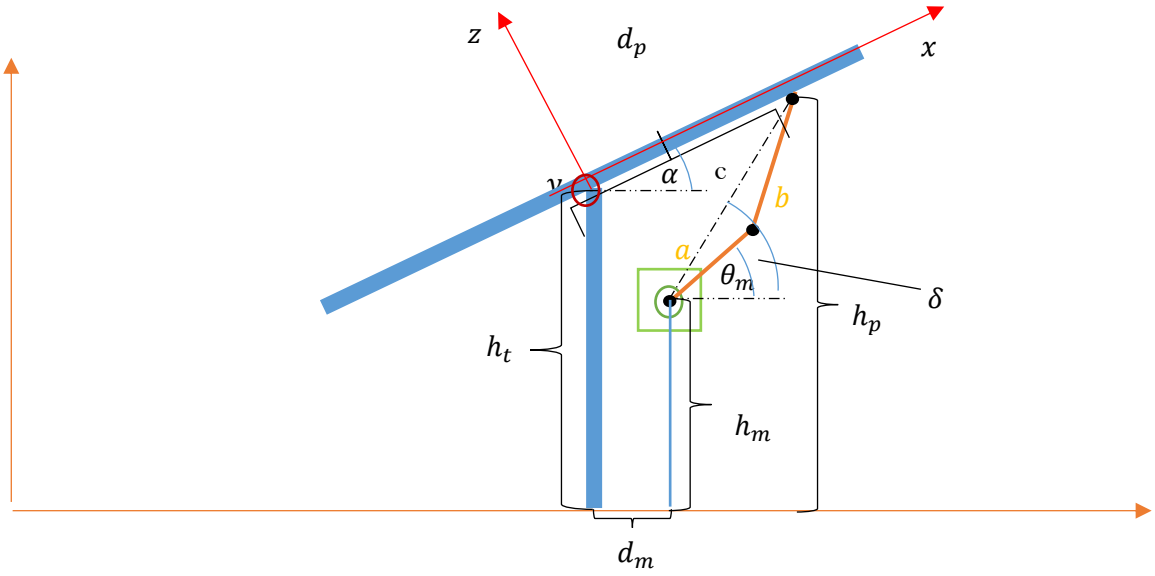


Figure 2.1: Schematic of the mechanical system

Above, a schematic of the system can be seen. Displayed is only one of the direct-current motors, namely the one responsible for the torque in the y-direction. As for the derivation of the mechanical system with respect to the links, it should be noted that the same holds true for the other DC-motor supplying the torque in x-direction.

In order to find out the optimal placement of the DC motors and the linkages while also finding the optimal length of the respective links, a MATLAB script was written. The following relations hold true for the above-described system.

$$h_p = h_t + \sin(\theta)\, d_p \tag{2.1}$$

$$c = \sqrt{\left(h_p - h_m\right)^2 + \left(d_p \cos(\theta) - d_m\right)^2} \tag{2.2}$$

$$\delta = \sin^{-1}\left(\frac{h_p - h_m}{c}\right) \tag{2.3}$$

Using the first and second the relation yields

$$\theta_m^x = \delta - \cos^{-1}\left(\frac{a^2 + c^2 - b^2}{2ac}\right) = \cos^{-1}\left(\frac{h_t + \sin(\alpha)\, d_p}{c}\right) - \cos^{-1}\left(\frac{a^2 + c^2 - b^2}{2ac}\right) := f(\alpha) \tag{2.4}$$

The motor angular displacement θ_m is a function of the plate angular displacementα. Prior to any analysis of the system for the setup of an adequate control algorithm, it has to be checked if the motor is capable of providing sufficient torque. In order to do that, D'Alambert's principle was applied to determine the necessary torque to keep the ball in a certain equilibrium position.

Applying D'Alambert's principle yields

$$-J_T\ddot{\alpha}\delta\alpha - mg * x * cos(\alpha)\delta\alpha + \tau\delta\theta_m^x = 0 \text{ with } \delta\theta_m^x = \frac{\partial f(\alpha)}{\partial\alpha}\delta\alpha \coloneqq q \tag{2.5}$$

$$\tau = \frac{J_T\ddot{\alpha} + mg * x * \cos(\alpha)}{q} \tag{2.6}$$

Where q is evaluated in MATLAB. It is apparent that a higher value of $q \coloneqq \frac{\partial f(\alpha)}{\partial\alpha}$ corresponds to a lower torque needed while a lower value of q corresponds to a larger value for α that can be achieved before the linkage setup reaches a deadlock positon, i.e. $c = b + a$. This equation provides also an estimation on what torque needs to be provided in the worst case when the ball is on the far side of the longer side of the plate.

The MATLAB script was run with different setups for the mechanical system as a whole in order to plot the derivative for different angles. Important considerations are that the linkage length should not exceed a certain threshold in order for the linkages to be rigid and that the function $\theta_m^x = f(\alpha)$ should be nearly constant since the transfer function from motor is much easier to express in this case.

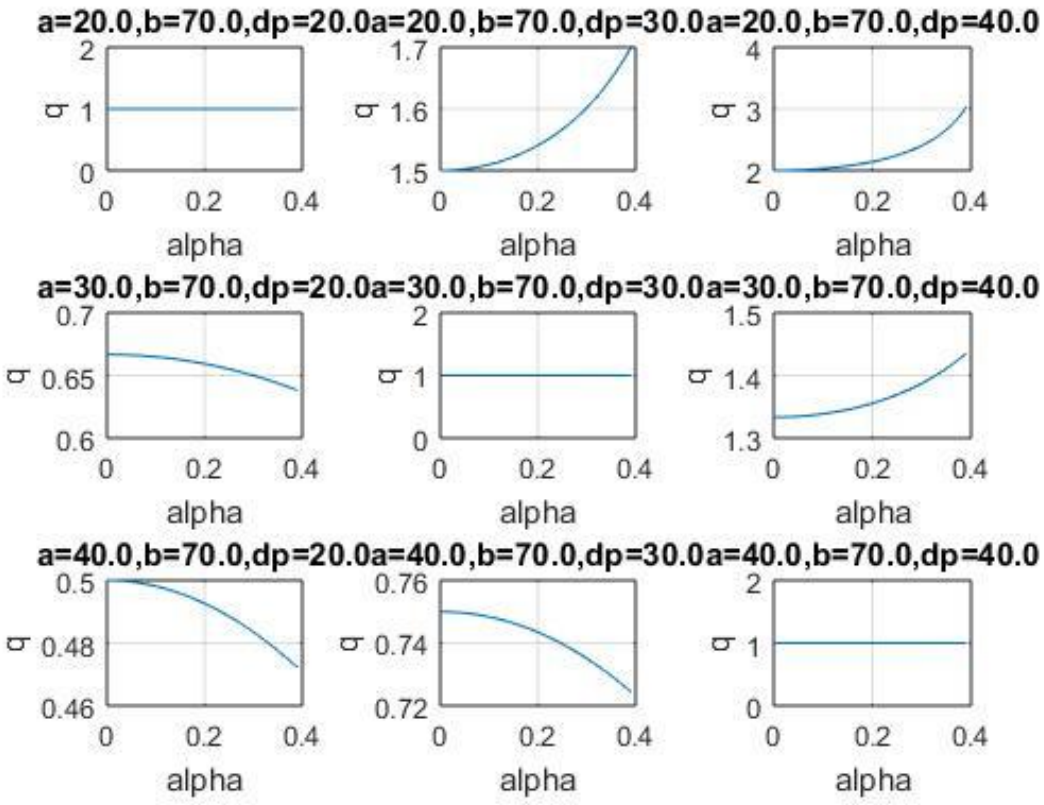


Figure 2.2: Plot of *q* vs. *alpha* for different link configurations

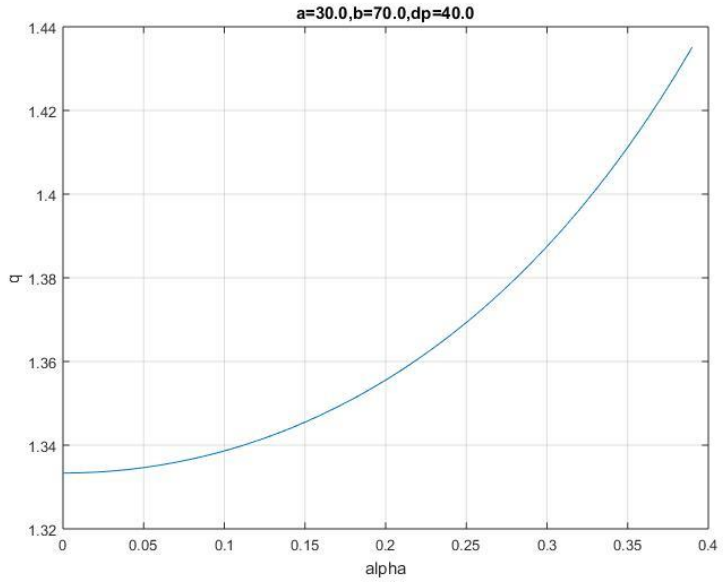


Figure 2.3: Plot of *q* vs. *alpha* for chosen link configuration

This procedure was conducted for both DC-motors and, hence, for both the rotation around the x-axis and the y-axis. After analyzing the graphs and results of the simulation that was run in MATLAB, the best configuration for the rotation about the y-axis was found to be $a = 30mm, b = 70mm \; and \; d_p = 40mm$. This configuration keeps the transmission coefficient q close to unity, which yields an approximated required equilibrium torque of 0.34 Nm for a 500g ball, which is still below the 0.4 Nm of the used DC-motor. However, a more detailed analysis on the dynamics and the motors themselves will be covered later on. Also it has to be taken into account that one half of the plate in y-direction is only 50mm long which poses a limit on the value of d_p. Furthermore, it can be observed that every configuration will ultimately encounter singular position where no more angular displacement of the plate can be achieved no matter how high the torque – it is undoubtedly important to avoid reaching this position.

The best configuration is characterized by allowing the table to revolve sufficiently far, keeping the necessary torque below a certain threshold specified by the motor specifications and also providing, at least to a certain extent, a constant transmission ratio $q \coloneqq \frac{\partial f(\alpha)}{\partial \alpha}$. Obviously, compromises have to be made, and the configuration in Figure 2.3 realizes the best compromise.

In the following sections, the dynamics of the entire sections are taken into account and the required torque can be more precisely specified – the calculations made above, however, provide valuable inside of the overall functionality of the linkage system and also give a first estimate on how the system should be dimensioned.

3 Mechanical System Design

After calculating the necessary dimension for the linkage system, the entire mechanical system could be modelled using SolidWorks. The modelling was characterized by an iteration of different designs. The first design as seen in Figure 3.1 and Figure 3.2 should provide for a nice and firm hold of the touchpad.

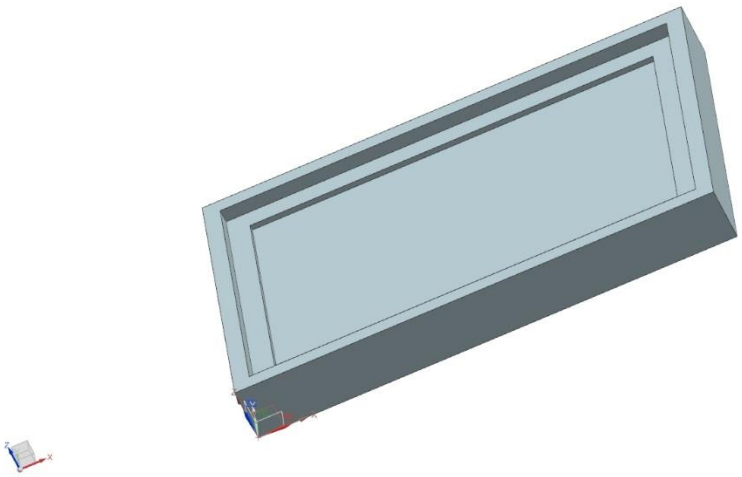


Figure 3.1: CAD model of the plate - top view

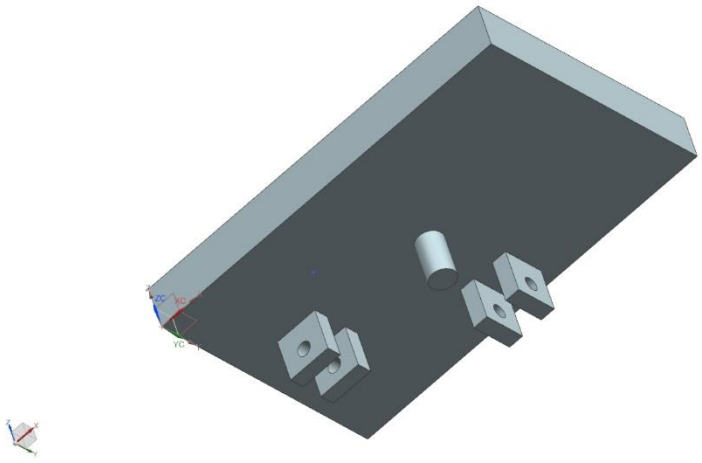


Figure 3.2: CAD model of the plate - bottom view

The plate on which the touchpad is placed is created using 3D printing. Because the surface itself is not perfectly smooth when using 3D printing, I designed the plate to have a cavity in the middle. This provides for the touchpad to only have contact with the plate on the outer rim, which minimizes the destabilizing effects of the previously described unevenness. The bottom part of the plate was modelled

in order to address the corresponding design choices for the linkage system, which will be discussed in detail now.

However, in order to print the model as proposed, the 3D-printer could only operate successfully with an extensive use of support structures. Since this would inevitably cause uneven surfaces, I decided to redesign the model and to be flat on the top, which would alleviate any concerns with rough surfaces in any way. This would cause the touchpad to not be fixed in place. This problem can be tackled with using tape, for instance, in order to fix the touchpad on the top. The revised model can be seen below. The cavities are supposed to reduce the overall weight of the plate.

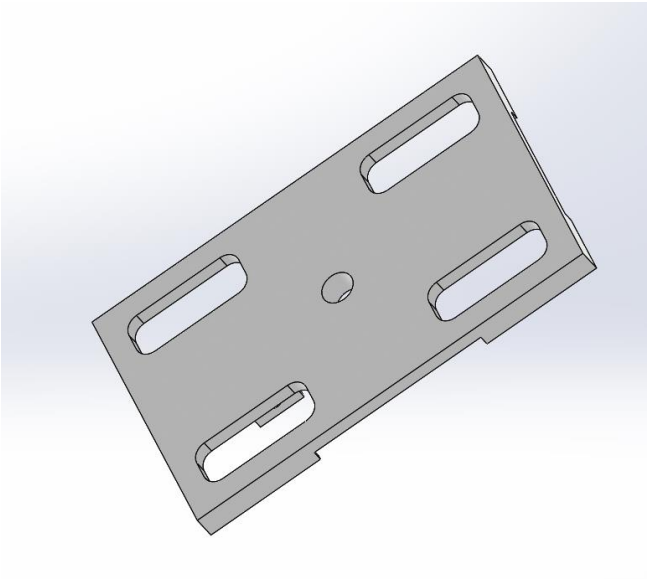


Figure 3.3: Revised CAD model of the plate - top view

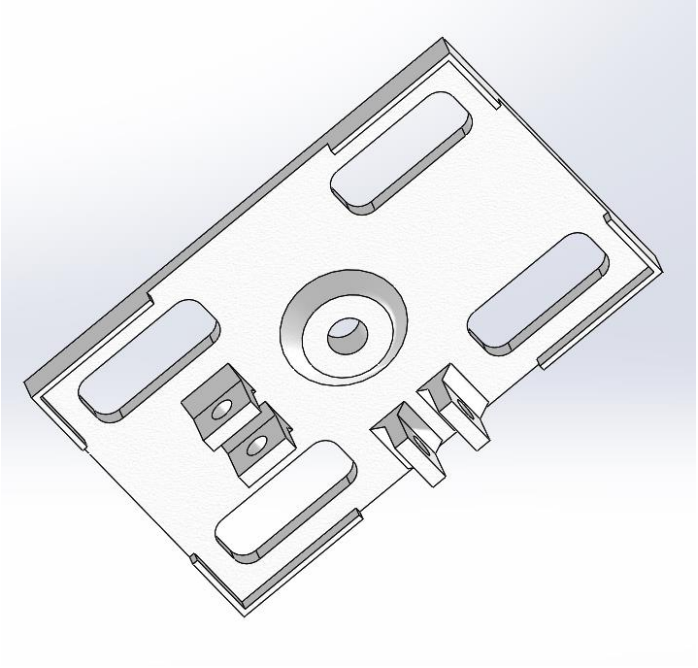


Figure 3.4: Revised CAD model of the plate – bottom view

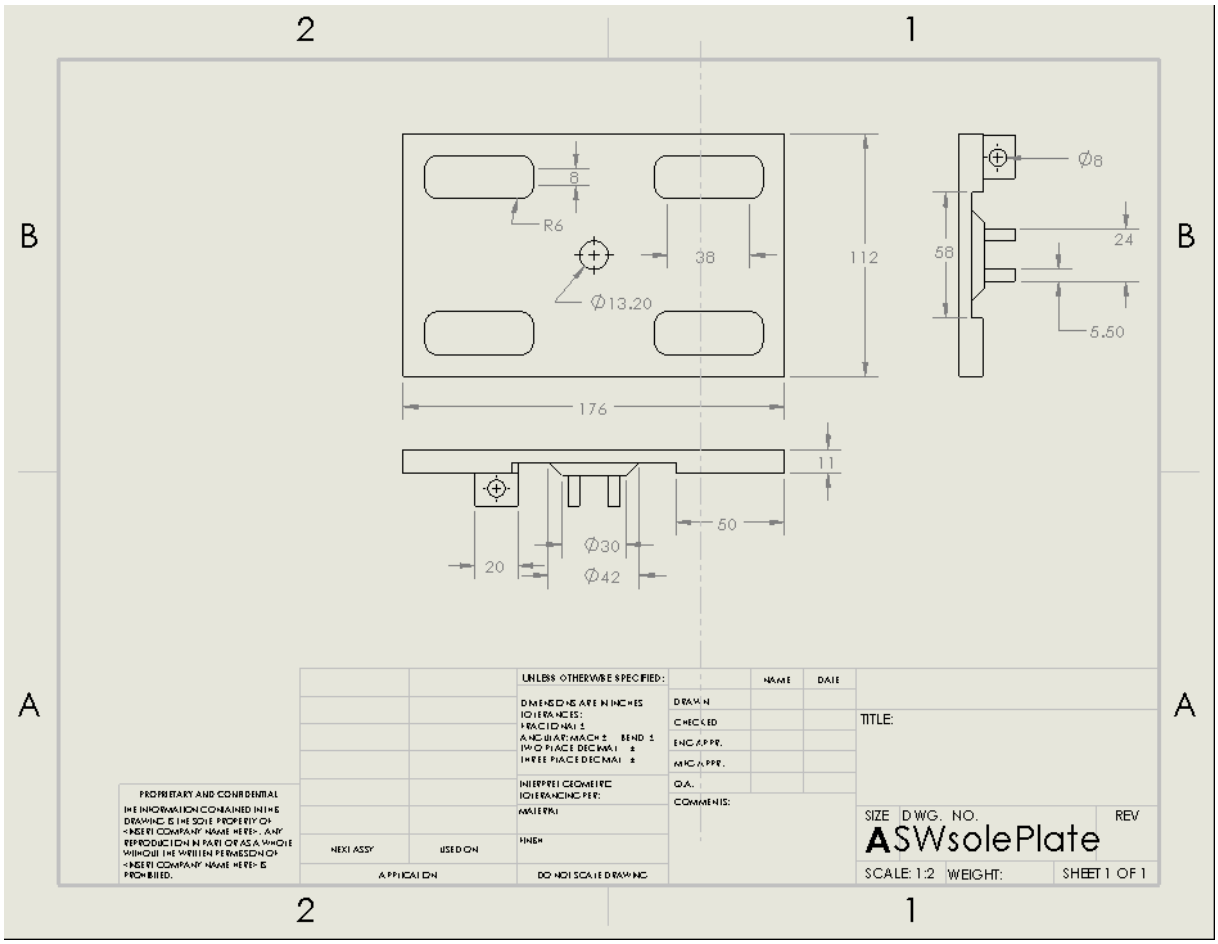


Figure 3.5: Technical drawing of the plate

Ideally, the ball should not roll off the plate if the control torques are adequately set. However, this cannot always be guaranteed and especially during the process of gain tuning, the gains can be off far enough so that the ball would roll over the plate. This can be prevented by attaching four pieces of a 3D-printed wall-like structure on the side of the plate that prevent the ball from falling off.

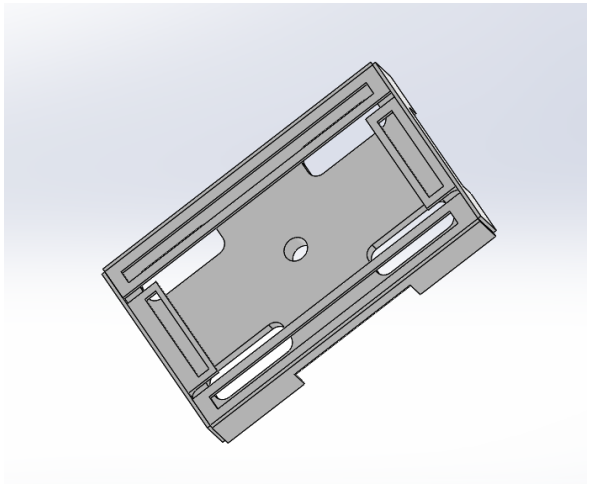


Figure 3.6: Revised CAD model of the plate with safety structures – top view

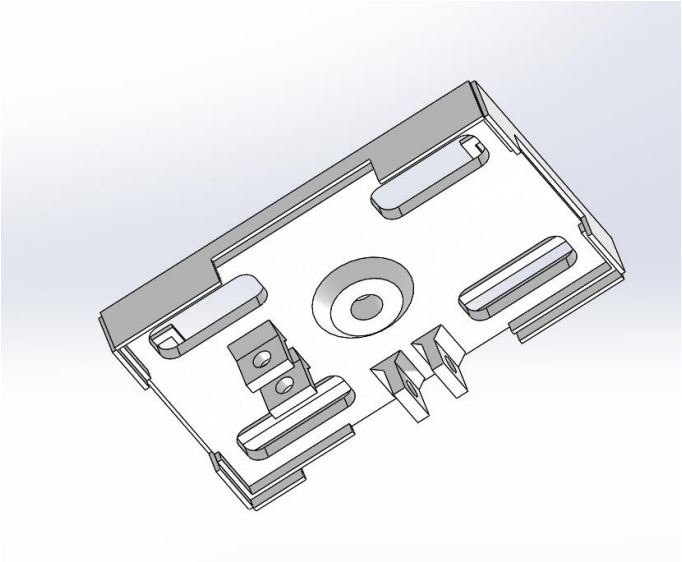


Figure 3.7: Revised CAD model of the plate with safety structures – bottom view

As for the linkage system, link *b* depicted in Figure 1 will consist of a metal rod with two ball joints attached to its ends. These ball joints provide two degrees of freedom and are necessary in order to provide unimpeded motion for the entire system. A screw will be put through the ball joints and the corresponding fittings that I designed on the bottom of the plate.

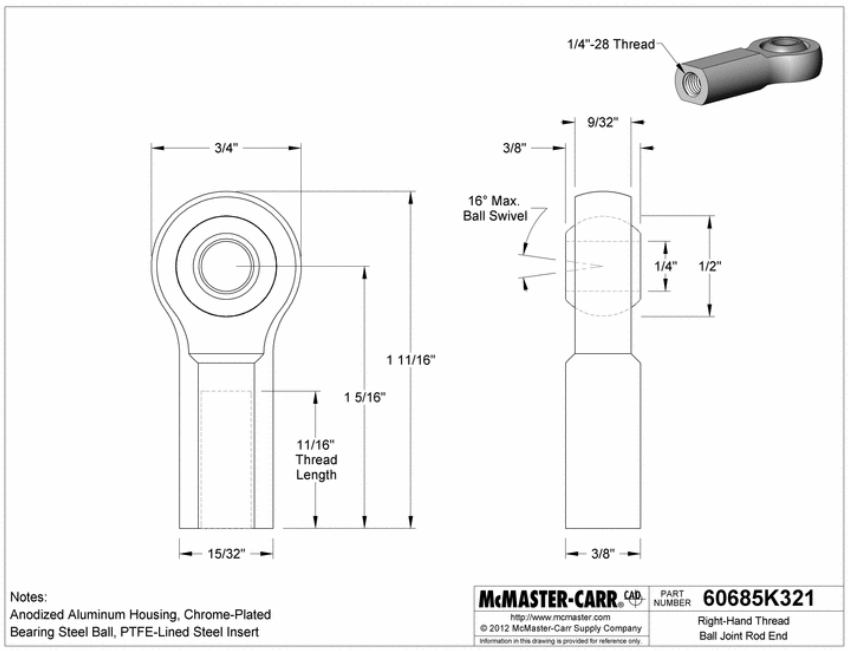


Figure 3.8: Technical drawing of the ball joint

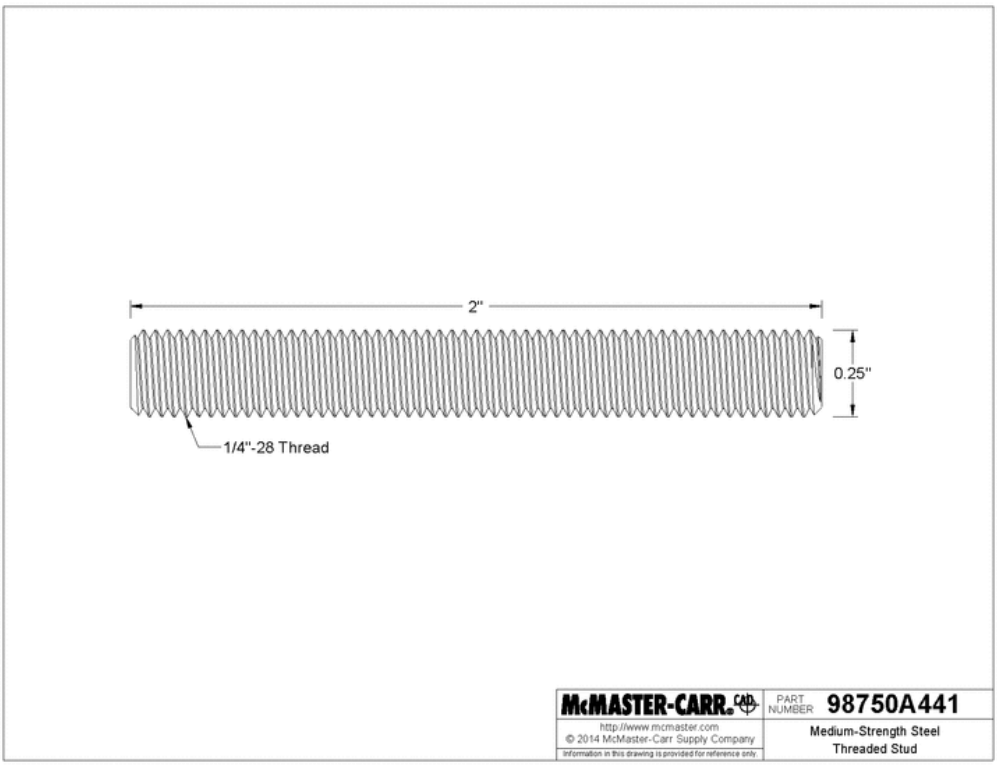


Figure 3.9: Technical drawing of the connection rod

The joint connecting link *a* and *b* is composed of two parts, namely the ball joint attached to link *b* and the designed fitting seen in Figure 3.10. This fitting provides the exact same functionality as the fittings on the bottom of the plate. This fitting, in turn, serves as link *a* also and will be attached to the shaft of the DC motor using a set of screws.

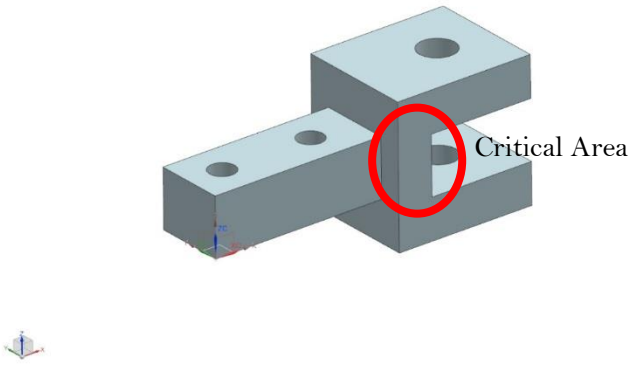


Figure 3.10: CAD model of link *a* – first version

In the end, the link design needed to be revised in order to for the ball joint not to hit the inner rim of the link.

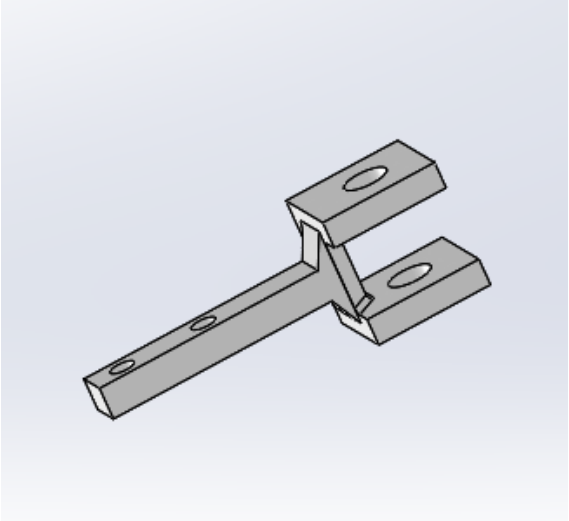


Figure 3.11: CAD model of link *a* – finalized version

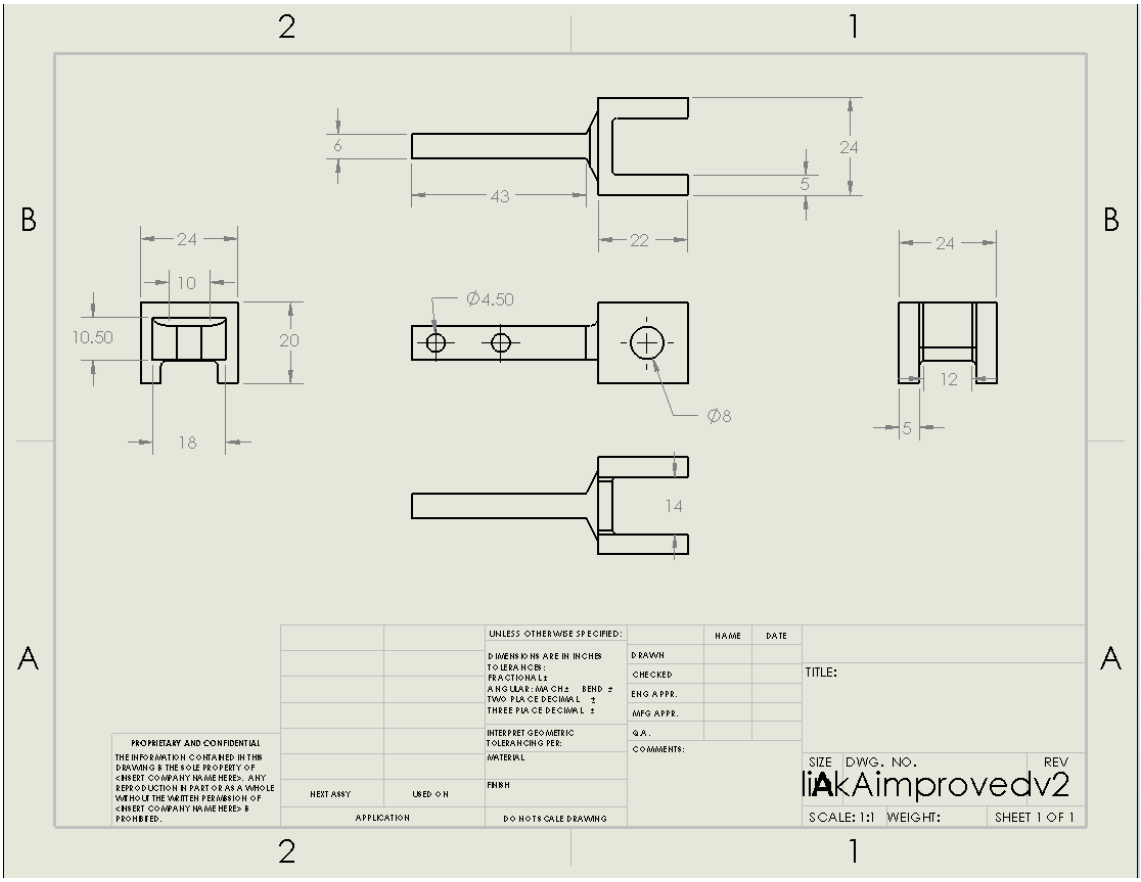


Figure 3.12: Technical drawing of finalized version of link *a*

The center of the plate itself is mounted using a double U-joint that provides two degrees of freedom, namely rotation around the two principle axis as discussed before. The main advantage of using this type of joint is that it only permits rotations about the specified degrees of freedom without allowing any other type of motion.

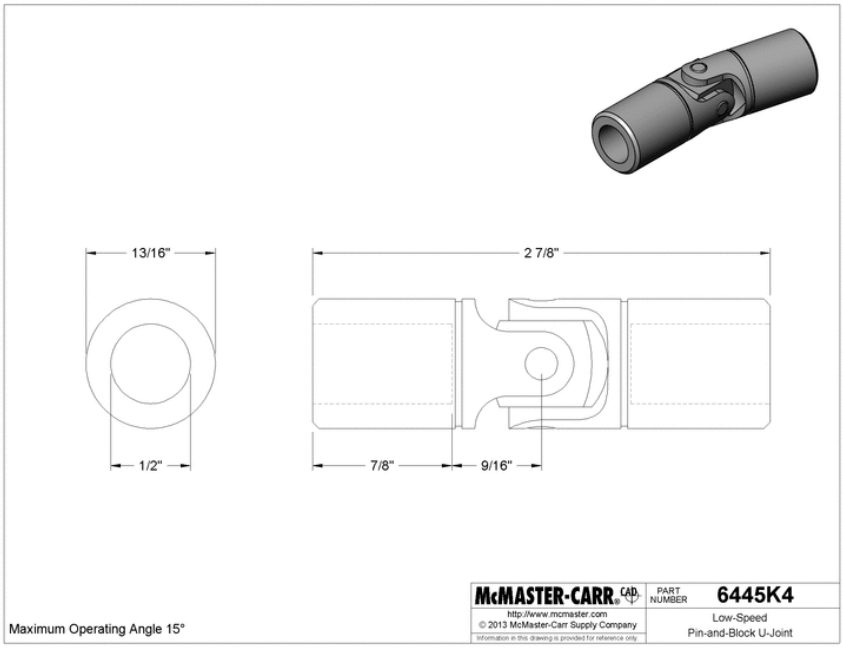


Figure 3.13: Technical drawing of the double U-joint

A picture of the assembled system can be seen below. As for the assembly, a short rod is inserted through the ball joint and two washers on each side are added. The ball joints are then assembled by screwing them in their respective place using nuts. The details can be inferred by the picture below.

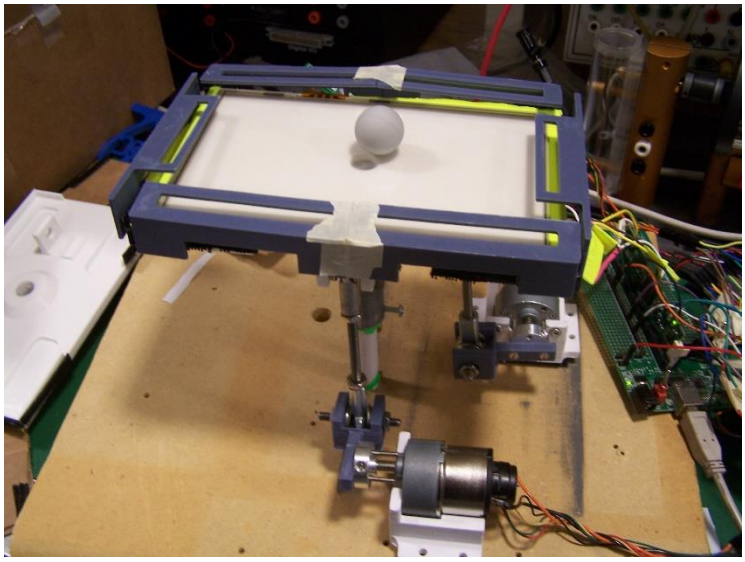


Figure 3.14: The assembled real-world system

4 System Modelling and Analysis

The main components of the mechanical system consist of the plate balancing the ball and the DC motors. The DC motor shaft is connected to the 3D-printed link. The two ball-joints together with the metal rod form the second link that connects to the plate. Overall, one pair of DC-motors is used to provide torque around both the x-axis and y-axis. The relationship between the torque provided by each DC-motor and the torque applied to the plate is described by the time-varying transmission coefficient q introduced in the previous chapter.

The coordinates relevant to the system that are used in the following and previous derivations all assume a plate-fixed coordinate frame σ compared to the world-coordinate frame σ'. Furthermore, the angles have been defined in such a way that a positive angle α corresponds to a negative acceleration in x-direction and a positive angle β corresponds to a negative acceleration in y-direction. A graphical depiction of the simplified system can be seen below.

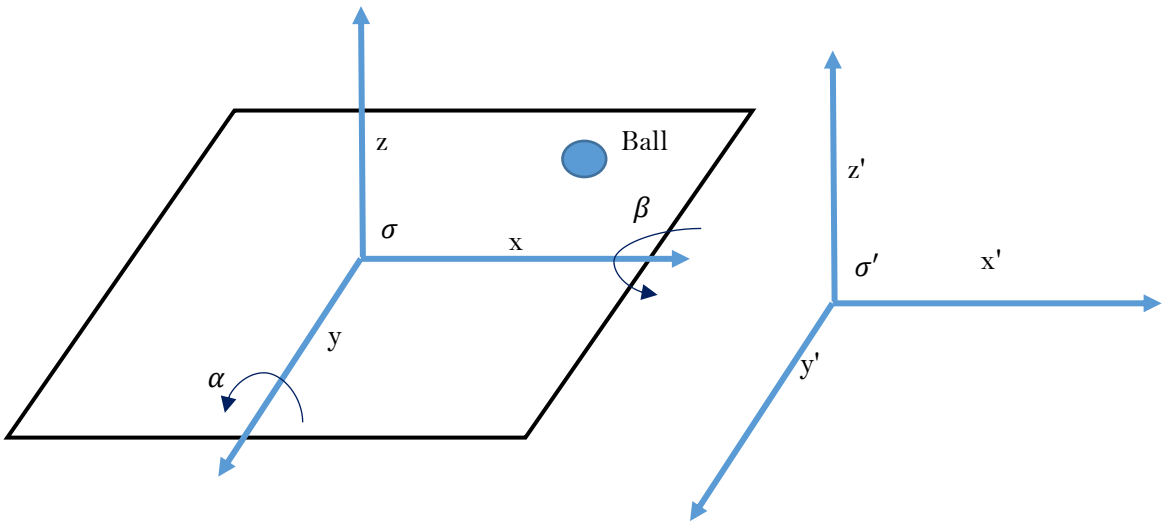


Figure 4.1: Visualization of model simplification of ball-plate system

The system was deliberately laid out in a way such that the transmission coefficient q stays constant throughout the range of operation. However, this proves to be difficult also because of viscous and kinematic friction between the links that are difficult to estimate. The links are theoretically fixed in such a way that the movement of one of the DC motor just provides rotation around one axis. Due to the usage of the double U-joint at the center of the plate, the rotational motions are not perfectly decoupled anymore.

However, this effect will be ignored for the sake of analysis. As for the motor, the well-known motor dynamics will be briefly covered and the final equations will be laid out. The main parameters relevant to the entire system are listed in table 4.1 below.

Parameter	Description	Value
J_p^x	Principle moment of inertia of plate around x-axis	$7.483 * 10^{-3}\ [\frac{Nms^2}{rad}]$
J_p^y	Principle moment of inertia of plate around y-axis	$6.482 * 10^{-3}\ [\frac{Nms^2}{rad}]$
J_b	Moment of inertia of ball	$12 * 10^{-5}\ [\frac{Nms^2}{rad}]$
m_p	Mass of plate	$0.1445\ [kg]$
m_b	Mass of ball	$0.003\ [kg]$
g	Gravitational constant	$9.81\ [\frac{m}{s^2}]$
r_b	Radius of ball	$0.01\ [m]$

Table 4.1: System parameters for the mechanical system

The moments of inertia of the plate have been obtained by an analysis in *SolidWorks* with the designed CAD-model.

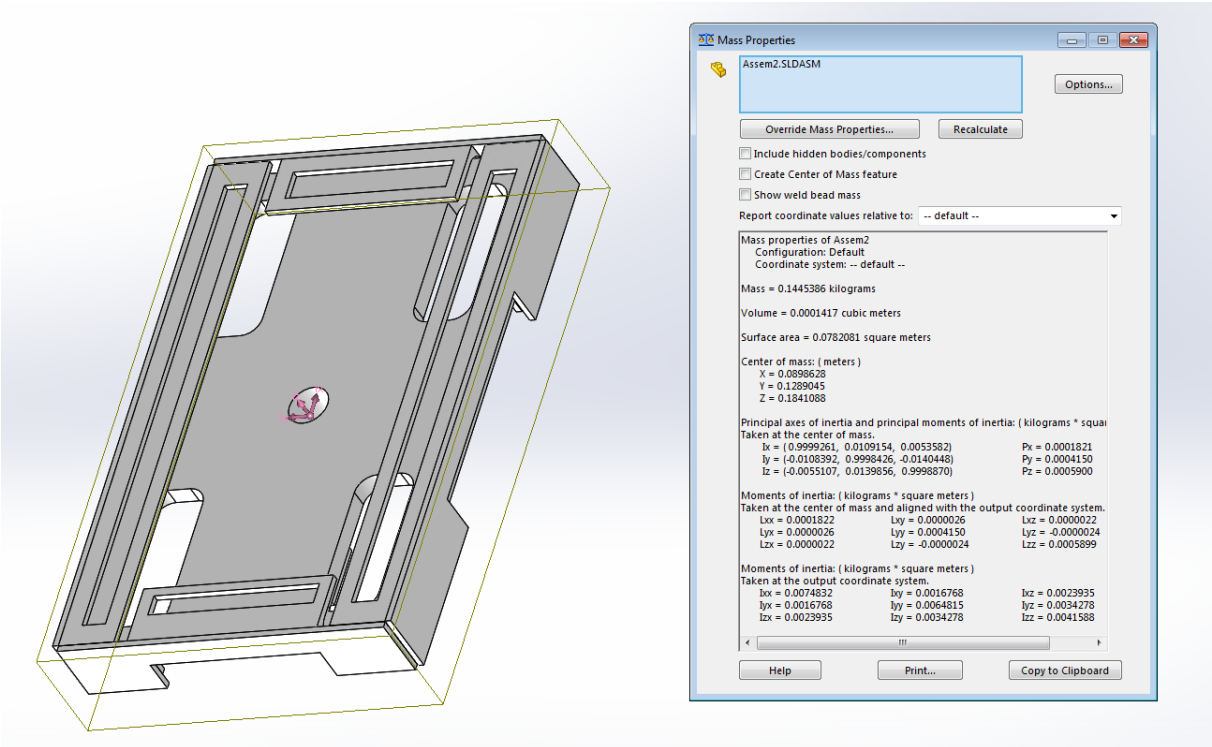


Figure 4.2: Screenshot of the parameter identification process in *SolidWorks*

First, the governing equations of the mechanical system and the DC-motor will be derived and explained. Thereafter, the control design will be discussed and suitable control algorithms will be analyzed.

4.1 Mechanical Model

The system equations of the mechanical system with the applied torque to the plate as in input and the balls position as an output will be derived using the Euler-Lagrange equation of the system.

The kinetic energy of the mechanical system is composed of both the plate's and the ball's kinetic energy:

$$T_b = \frac{1}{2}\left(m_b + \frac{I_b}{r_b^2}\right)\left(\dot{x_b}^2 + \dot{y_b}^2\right) \tag{4.1}$$

$$T_p = \frac{1}{2}I_p\left(\dot{\alpha}^2 + \dot{\beta}^2\right) + \frac{1}{2}m_b\left(x_b\dot{\alpha} + y_b\dot{\beta}\right)^2 \tag{4.2}$$

The overall kinetic energy is given by

$$T = T_b + T_p = \frac{1}{2}I_p\left(\dot{\alpha}^2 + \dot{\beta}^2\right) + \frac{1}{2}m_b\left(x_b\dot{\alpha} + y_b\dot{\beta}\right)^2 + \frac{1}{2}\left(m_b + \frac{I_b}{r_b^2}\right)\left(\dot{x_b}^2 + \dot{y_b}^2\right) \tag{4.3}$$

The potential energy of the system is described by

$$V = mg(xsin(\alpha) + ysin(\beta)) \tag{4.4}$$

Here the assumption is made that the plate is not contributing anything to the potential energy, although the center of mass of the plate does not lie in the center of rotation of the U-joint. This leads to the plate contributing a small amount of potential energy to this system which is neglected here.

Defining $L \coloneqq T - V$, the Lagrange equation is then given by

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = Q_i \tag{4.5}$$

where Q_i are the generalized forces on the system that are defined by

$$Q_i = \sum_{j=1}^{4} F_i \frac{\partial r_i}{\partial q_j} \tag{4.6}$$

Here, $\boldsymbol{F}$ are the external forces and torques, $\boldsymbol{r}$ is the position on where the external force or torque acts on a d q_j are the generalized coordinates $q_1 \coloneqq x, q_2 \coloneqq y, q_3 \coloneqq \alpha, q_4 \coloneqq \beta$. Then the describing equations of the system come out to be

$$\left(m + \frac{I_b}{r^2}\right)\ddot{x} - mx\dot{\alpha}^2 - my\dot{\alpha}\dot{\beta} + mgsin(\alpha) = 0 \quad (4.7)$$

$$\left(m + \frac{I_b}{r^2}\right)\ddot{y} - my\dot{\beta}^2 - mx\dot{\alpha}\dot{\beta} + mgsin(\beta) = 0 \quad (4.8)$$

$$\left(J_p^x + J_b + mx^2\right)\ddot{\alpha} + 2mx\dot{x}\dot{\alpha} + mxy\ddot{\beta} + m\dot{x}y\dot{\beta} + mx\dot{y}\dot{\beta} + mgxcos(\alpha) = \tau_\alpha \quad (4.9)$$

$$\left(J_p^y + J_b + my^2\right)\ddot{\beta} + 2my\dot{y}\dot{\beta} + mxy\ddot{\alpha} + m\dot{x}y\dot{\alpha} + mx\dot{y}\dot{\alpha} + mgycos(\beta) = \tau_\beta \quad (4.10)$$

A valid assumption is that the angular displacements and velocities of the plate are very small, which means $\sin(\alpha) \cong \alpha$ and $\dot{\alpha}\dot{\beta} \cong 0$. This is verified later by comparing the encoder readings with the acceptable angle ranges in order to apply the small angle approximation. Using this and $J_b = \left(m + \frac{I_b}{r^2}\right) = \frac{7}{5}m$, which can be easily shown to hold true for a sphere, equation (**Error! Reference source not found.**1) and (**Error! Reference source not found.**2) can be reduced to

$$\frac{7}{5}\ddot{x} + g\alpha = 0 \quad (4.11)$$

$$\frac{7}{5}\ddot{y} + g\beta = 0 \quad (4.12)$$

The equations derived here will be used later when designing the controller.

4.2 Motor Model

The motors used are Gear Head Motors from *Lynxmotion*. The motor provides an internal gear reduction of 30:1. However, for the analysis of the system the output shaft angles are used. Therefore, the inertia of the moment as well as all motor constants related to the motion of the motor such as K_τ, K_b and B are determined with respect to the output shaft. Considering this, the internal gear reduction plays no longer any role on the following analysis of the motors and the motor can be seen as a 1:1 geared substitute model. The DC-motors then have the parameters listed below.

Parameter	Description	Value
K_τ	Torque constant	$0.3602\ [\frac{Nm}{A}]$
K_b	Back-emf constant	$0.3602\ [\frac{Vs}{rad}]$
J_m	Moment of inertia of motor	$4.455 * 10^{-4}\ [\frac{Nms^2}{rad}]$
$\tilde{B} \coloneqq B_m + \frac{K_\tau K_b}{R}$	Overall damping	$0.01178\ [\frac{Nms}{rad}]$
R	Armature resistance	n/a
L	Armature inductance	n/a
τ_{stall}	Stall torque	$0.3962\ [Nm]$

Table 4.2: Parameters for the DC-motors

The motor constant K_τ and K_b can be calculated from the data sheet attached in the appendix. The moment of inertia and the overall damping have been experimentally determined by observing the motor dynamics. The resistance and inductance have not been determined since they are not needed – the inductance is not needed because the electrical dynamics are much faster than the mechanical ones, and the resistance is not needed because it only occurs in the calculation of the overall damping, which has been already experimentally determined.

Taking into account the motor's mechanical and electrical dynamics, the governing system equations can then be written as

$$J_m\ddot{\theta}_m + B_m\dot{\theta}_m = \tau_m - \frac{\tau_l}{q} \tag{4.13}$$

$$L\dot{I}_a + RI_a = V - V_b \tag{4.14}$$

$$\tau_m = K_\tau I_a \tag{4.15}$$

$$V_b = K_b \dot{\theta}_m \tag{4.16}$$

Assuming $\frac{L}{R} \ll \frac{J_m}{R_m}$, which is generally the case for small DC motors, the above equations can be reduced to

$$J_m \ddot{\theta}_m + (B_m + \frac{K_b K_\tau}{R})\dot{\theta}_m = u - \frac{\tau_l}{q} \text{ with } u \coloneqq \frac{K_\tau}{R} V \tag{4.17}$$

Applying the Laplace transformation, the motor equations can be rewritten as follows

$$\frac{\theta_m}{V} = \frac{\left(\frac{K_\tau}{R}\right)}{s(J_m s + B_m + \frac{K_\tau K_b}{R})} \tag{4.18}$$

$$\frac{\theta_m}{\tau_l} = \frac{1}{s(J_m s + B_m + \frac{K_\tau K_b}{R})} \tag{4.19}$$

The entire system depicted in a block diagram can be seen below

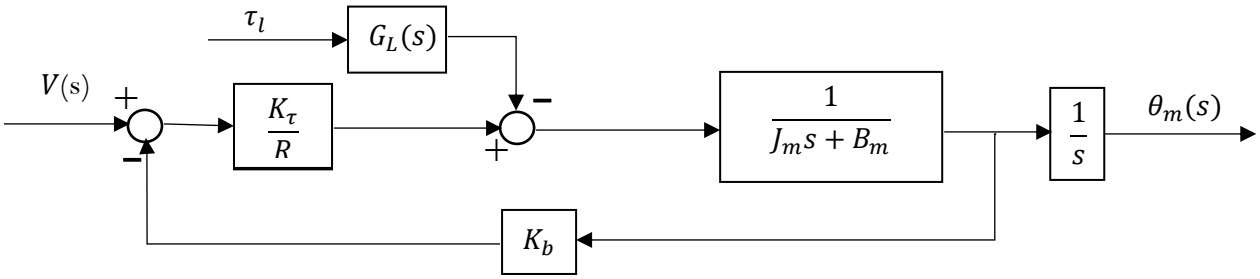


Figure 4.3: Block diagram of DC motor with load torque τ_l and input voltage V

Using $\frac{\theta_m}{q} = \alpha, \beta$ and $u_\alpha, u_\beta \coloneqq \frac{qK_\tau}{R} V$, as well as setting $\tilde{B} \coloneqq B_m + \frac{K_b K_\tau}{R}, \tilde{J_x} \coloneqq J_p^x + J_b + mx^2, \tilde{J_y} \coloneqq J_p^y + J_b + my^2$, the two governing equations for the entire DC motor model (**Error! Reference source not found.**) and (**Error! Reference source not found.**) can be combined to

$$J_m q\ddot{\alpha} + \tilde{B} q\dot{\alpha} = u_\alpha - \frac{1}{q}[\tilde{J_x}\ddot{\alpha} + 2mx\dot{x}\dot{\alpha} + mxy\ddot{\beta} + m\dot{x}y\dot{\beta} + mx\dot{y}\dot{\beta} + mgxcos(\alpha)] \tag{4.20}$$

which can be rearranged to

$$\ddot{\alpha} q\left(J_m + \frac{\tilde{J_y}}{q^2}\right) + \tilde{B} q\dot{\alpha} + \frac{1}{q}[\tilde{J_x}\ddot{\alpha} + 2mx\dot{x}\dot{\alpha} + mxy\ddot{\beta} + m\dot{x}y\dot{\beta} + mx\dot{y}\dot{\beta} + mgxcos(\alpha)] = u_\alpha \tag{4.21}$$

Doing the same steps with the second motor and the other set of variables, respectively, the second governing equation can be obtained:

$$\ddot{\beta}q\left(J_m + \frac{\tilde{J}_y}{q^2}\right) + \tilde{B}q\dot{\beta} + \frac{1}{q}[\tilde{J}_y\ddot{\alpha} + 2my\dot{y}\dot{\beta} + mxy\ddot{\alpha} + m\dot{x}y\dot{\alpha} + mx\dot{y}\dot{\alpha} + mgycos(\beta)] = u_\beta \qquad (4.22)$$

These two equations, which represent the system dynamics from motor to plate can be combined into the following set of equations, which is quite commonly seen for mechanical systems

$$\boldsymbol{M}(x, y, \alpha, \beta)\begin{bmatrix}\ddot{\alpha}\\ \ddot{\beta}\end{bmatrix} + \boldsymbol{B}(\dot{\alpha}, \dot{\beta})\begin{bmatrix}\dot{\alpha}\\ \dot{\beta}\end{bmatrix} + \boldsymbol{C}(x, \dot{x}, y, \dot{y}, \alpha, \dot{\alpha}, \beta, \dot{\beta})\begin{bmatrix}\dot{\alpha}\\ \dot{\beta}\end{bmatrix} + \boldsymbol{G}(x, y, \alpha, \beta) = \boldsymbol{u} \qquad (4.23)$$

Here, **M** represents the inertia matrix of the system, **B** the friction matrix, **C** the velocity-coupling matrix, and **G** the gravity vector. All values of these matrices can be obtained by the measurement of the encoders and the touchscreen.

5 Control Design

As for the control design, I decided to simulate and test multiple control approaches. Firstly, I will implement a simple PID-controller that utilizes splitting up the entire physical system into two individual subsystems and then controlling every subsystem but itself. Then I will analyze a more sophisticated approach utilizing feedback linearization and full state feedback with reference input scaling which will be finally extended by introducing additional integrator states.

The entire system consisting of the ball, plate and motors can be thought of as two individual subsystems. The first subsystem assumes the motor torque as input and the plate angle as input while the second subsystem assumes the plate angle as input and the ball position as output.

System equations (4.7) and (4,8) correspond to the first subsystem, while equations (4.9) and (4.10) correspond to the second subsystem.

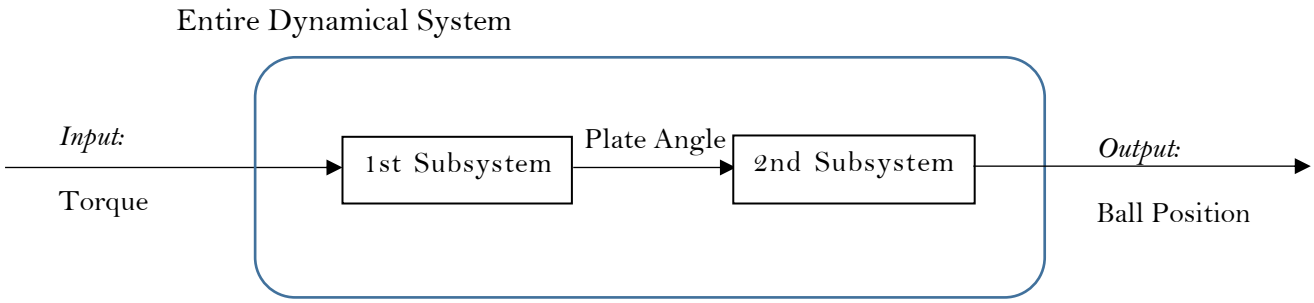


Figure 5.1: System schematic with individual subsystems

5.1 PID-Controller

The PID-controller will use all eight available physical states of the system as feedback. However, they are not all simultaneously used in one control equation. This state feedback approach will be analyzed later. Here, the system will be broken apart into the two subsystems laid out above and every subsystem itself will be individually controlled.

In order to implement the PID-controller, two separate control laws are designed. The first law governs the first subsystem and is set to equal

$$\begin{bmatrix} u_\alpha \\ u_\beta \end{bmatrix} = K_{p,2}\left(\begin{bmatrix} \alpha \\ \beta \end{bmatrix}^d - \begin{bmatrix} \alpha \\ \beta \end{bmatrix}\right) - K_{d,2}\begin{bmatrix} \dot{\alpha} \\ \dot{\beta} \end{bmatrix} \tag{5.1}$$

The second control law is designed for the first subsystem

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}^d = K_{p,1}\left(\begin{bmatrix} x \\ y \end{bmatrix}^d - \begin{bmatrix} x \\ y \end{bmatrix}\right) - K_{d,1}\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \tag{5.2}$$

It can be seen that the desired position of the ball yields a desired angle that, in turn, acts as the reference input for (5.24) in order to provide the final control torque supplied by the motors.

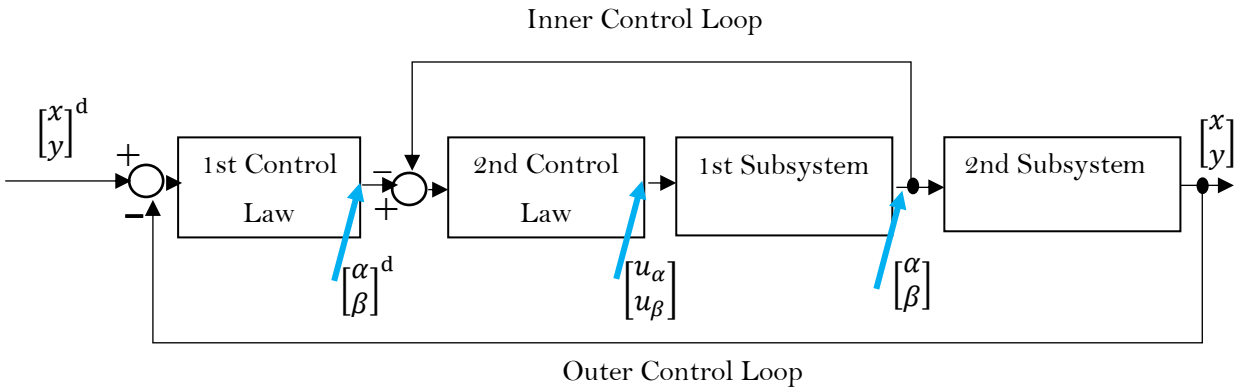


Figure 5.2: Block diagram incorporating the two control laws and subsystems

From the block diagram above it can be inferred that the subsystem consisting of motor and plate is somewhat "embedded" within the plate and ball subsystem. The goal now is to make the inner control system sufficiently fast in order to be able to provide the necessary angles to command the ball to the desired positions. As for the actual implementation, this approach also comes with the advantage of being able to saturate the reference angle not to exceed a certain threshold. This is important since the real-world system is not very stable and the motors would "lash out" multiple times. With saturating the reference angle, this problem could be adequately handled.

This approach, however, will always yield a steady-state error. Therefore, an integrator was implemented by modifying (5.24) to equal

$$\begin{bmatrix} u_\alpha \\ u_\beta \end{bmatrix} = K_{p,2}\left(\begin{bmatrix} \alpha \\ \beta \end{bmatrix}^d - \begin{bmatrix} \alpha \\ \beta \end{bmatrix}\right) - K_{d,2}\begin{bmatrix} \dot{\alpha} \\ \dot{\beta} \end{bmatrix} + K_i \int_0^{t_o} \left(\begin{bmatrix} x \\ y \end{bmatrix}^d - \begin{bmatrix} x \\ y \end{bmatrix}\right) dt \tag{5.3}$$

This integrator must be implemented in (5.24) and not in (5.25) because a non-zero torque must be provided to keep the system at zero steady-state error and torque is only affected by (5.24).

On top of that, gravity compensation as well as friction compensation can be added to further improve the controller design

- Gravity compensation: $\begin{bmatrix} u_\alpha \\ u_\beta \end{bmatrix}^{grav} = \begin{bmatrix} m_b g x cos(\alpha) \\ m_b g y cos(\beta) \end{bmatrix} \frac{1}{q}$
- Friction compensation: $\begin{bmatrix} u_\alpha \\ u_\beta \end{bmatrix}^{fric} = \begin{bmatrix} \tilde{B}\dot{\alpha} \\ \tilde{B}\dot{\beta} \end{bmatrix} q + \begin{bmatrix} C_{pos,neg} \\ C_{pos,neg} \end{bmatrix}$

The factor q within the friction compensation comes from the fact that $\tilde{B}$ was experimentally determined with respect to the output shaft whereas both angular velocities belong to the plate. Therefore, the term has to be scaled by q. The inner loop control law ultimately then comes out to be

$$\begin{bmatrix} u_\alpha \\ u_\beta \end{bmatrix} = K_{p,2}\left(\begin{bmatrix} \alpha \\ \beta \end{bmatrix}^d - \begin{bmatrix} \alpha \\ \beta \end{bmatrix}\right) - K_{d,2}\begin{bmatrix} \dot{\alpha} \\ \dot{\beta} \end{bmatrix} + K_i \int_0^{t_o} \left(\begin{bmatrix} x \\ y \end{bmatrix}^d - \begin{bmatrix} x \\ y \end{bmatrix}\right) dt + \begin{bmatrix} u_\alpha \\ u_\beta \end{bmatrix}^{grav} + \begin{bmatrix} u_\alpha \\ u_\beta \end{bmatrix}^{fric}$$

5.2 Computed Torque Control of Inner Loop

When dealing with control applications, often times it is required for the system output to follow a certain trajectory or set-point. For these kind of problems, the concept of feedforward control has proven to yield very good results, as seen in [6]. When dealing with this kind of control, there are two important cases to distinguish:

- Feedforward control with offline-computation due to prior knowledge of trajectory
- Feedforward control with online-computation for unknown trajectory

In this case, the trajectory will be chosen arbitrarily while operating the system. Therefore, the computation has to be done in real-time, which requires the microprocessor used to be sufficiently fast in terms of processing time.

To be specific, the approach chosen here is not a direct feedforward control because the reference input does not get fed directly into the system's input, but rather gets passed through the actual controller. However, the control law for the computed torque approach chosen here and other more common feedforward controllers looks quite similar and is, in fact, almost the same only that the control law used her utilizes an exact cancellation approach for the system itself.

As mentioned, a special kind of feedforward control will be implemented. As outlined in [5], the control input is set in such a way so that the system's nonlinear dynamics will be compensated by the control input. Since the control torque does not directly affect the ball position, the system is under-actuated. This can also be seen by inspecting (4.7) and (4.8) not to contain the torques of the motors. The entire under-actuated system can be viewed as two separate systems as depicted in Figure 5.1.

However, equations (4.9) and (4.10) suggest that the first subsystem is indeed fully actuated, and can therefore be feedback linearized. By decoupling the system like this and linearizing the second subsystem yielding (4.11) and (4.12), it is now only necessary to apply the computed torque approach to the fully actuated first subsystem. Then, the system dynamics of the first subsystem are described by simple decoupled second order dynamics. By setting the control input torque $\boldsymbol{u}$ equal to

$$\boldsymbol{u} = \boldsymbol{M}\boldsymbol{a} + \boldsymbol{B}\begin{bmatrix}\dot{\alpha}\\ \dot{\beta}\end{bmatrix} + \boldsymbol{C}\begin{bmatrix}\dot{\alpha}\\ \dot{\beta}\end{bmatrix} + \boldsymbol{G} \tag{5.4}$$

(4.23) reduces to

$$\begin{bmatrix}\ddot{\alpha}\\ \ddot{\beta}\end{bmatrix} = \boldsymbol{a}, \quad \boldsymbol{a} \in \mathbb{R}^{2x1} \tag{5.5}$$

This system is also known as a *double integrator system* [5]. The control law designed above is called inverse dynamics control because it 'inverses' the system's dynamics by using an inner-loop control architecture (5.24) to cancel out the non-linearity and an outer-loop control (5.25) to achieve the desired system dynamics.

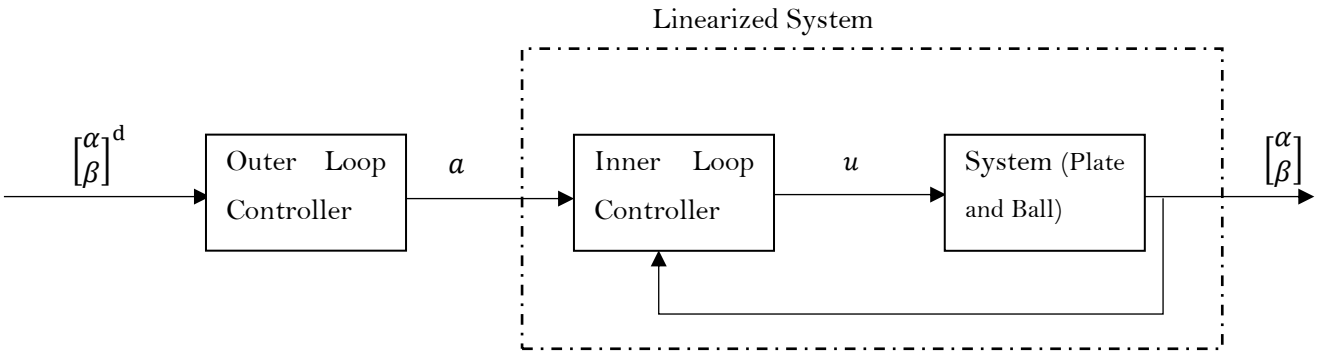


Figure 5.3: Control system schematic with inner and outer loop

The inner and outer loop architecture in this case must not be confused with the outer and inner loop introduced with the approach in chapter 5.1. There, the inner and outer loop are associated with a respective subsystem. Here, both the inner and outer loop are associated with only one physical subsystem where the inner loop cancels out the nonlinearity of the system and the outer loop provides the actual control law to provide the actual system dynamics.

This approach is rather sensitive to system disturbances and uncertainties - nonetheless it yields remarkable results when dealing with highly nonlinear systems where the dynamical equations are known precisely. With this approach, it is possible to design a decoupled controller for every output. Conveniently, **a** will be set to

$$\boldsymbol{a} = -\boldsymbol{K_p}\begin{bmatrix}\alpha\\ \beta\end{bmatrix} - \boldsymbol{K_d}\begin{bmatrix}\dot{\alpha}\\ \dot{\beta}\end{bmatrix} + \boldsymbol{r} \tag{5.6}$$

where **r** is a reference input

$$\boldsymbol{r} = \begin{bmatrix}\ddot{\alpha}\\ \ddot{\beta}\end{bmatrix}^{d} + \boldsymbol{K_p}\begin{bmatrix}\alpha\\ \beta\end{bmatrix}^{\mathrm{d}} + \boldsymbol{K_d}\begin{bmatrix}\dot{\alpha}\\ \dot{\beta}\end{bmatrix}^{\mathrm{d}} \tag{5.7}$$

It can be seen why this approach is rather similar to a feedforward control law: The reference input is used directly within the control input ***a***. Defining $\boldsymbol{e}(t) \coloneqq \begin{bmatrix}\alpha\\ \beta\end{bmatrix}^{\mathrm{d}} - \begin{bmatrix}\alpha\\ \beta\end{bmatrix}$, plugging (5.27) into (5.26), and plugging the resulting equation into (5.25), the error can be described by a second order system

$$\ddot{\boldsymbol{e}}(t) + \boldsymbol{K_d}\dot{\boldsymbol{e}}(t) + \boldsymbol{K_p}\boldsymbol{e}(t) = 0 \tag{5.8}$$

This control law governs the first subsystem and achieves better dynamical properties for the first subsystem in order to provide better results for the overall system.

5.2.1 Full State Feedback Control for Outer Loop

Having a completely linearized model now for all variables of the system, an LTI-state representation can be built. First, a set of useful state variables is declared by setting

$$\boldsymbol{x} = [x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]^T = \left[x_b, \dot{x}_b, \alpha, \dot{\alpha}, y_b, \dot{y}_b, \beta, \dot{\beta}\right]^T$$

Then, the state space equations can be written as

$$\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{a} \tag{5.9a}$$

$$\boldsymbol{y} = \boldsymbol{C}\boldsymbol{x} \tag{5.9b}$$

with $\boldsymbol{x} \in \mathbb{R}^{8x1}, \boldsymbol{u} \in \mathbb{R}^{2x1}, \boldsymbol{y} \in \mathbb{R}^{2x1}$ and $\boldsymbol{A} \in \mathbb{R}^{8x8}, \boldsymbol{B} \in \mathbb{R}^{8x2}, \boldsymbol{C} \in \mathbb{R}^{2x8}$.

In this case, it comes out to be

$$\dot{\boldsymbol{x}} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{5}{7}g & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{5}{7}g & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \boldsymbol{a} \tag{5.10a}$$

$$\boldsymbol{y} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \boldsymbol{x} \tag{5.10b}$$

with $\boldsymbol{a} = \begin{bmatrix} a_\alpha \\ a_\beta \end{bmatrix}, \boldsymbol{y} = \begin{bmatrix} x_b \\ y_b \end{bmatrix}$ and $\boldsymbol{x} = [x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]^T \coloneqq \left[x_b, \dot{x}_b, \alpha, \dot{\alpha}, y_b, \dot{y}_b, \beta, \dot{\beta}\right]^T$.

The eigenvalues of the **A** are all zero, which means the system is marginally stable in the case of open-loop control. Due to this reason, a closed-loop controller has to be designed. The poles of the system must lie in the left-hand side of the s-domain. In order to achieve this, the poles have to be placed by choosing the inputs appropriately.

Transforming (**Error! Reference source not found.**) into the Laplace domain yields

$$G(s) \coloneqq \frac{X(s)}{\mathrm{A}(s)} = \frac{-5g}{7}\frac{1}{s^2} \tag{5.11}$$

For a second order system, the poles are complex conjugates of each other. For each pole p, the following holds true

$$Re(p) = -\omega_n\zeta \text{ and } Im(p) = \omega_n\sqrt{\zeta^2 - 1} \tag{5.12}$$

As for the design specifications setting time T_s, the rise time T_r and the overshoot OS, the following relations can be used

$$T_s = \frac{4}{\omega_n \zeta} \text{ for settling within 2\% of final value} \tag{5.13}$$

$$T_r = \frac{1}{\sqrt{1-\zeta^2}}\left[\pi - \tan^{-1}\left(\frac{\sqrt{1-\zeta^2}}{\zeta}\right)\right], \text{which holds for overdamped systems} \tag{5.14}$$

$$OS = \exp\left(-\frac{\pi\zeta}{\sqrt{1-\zeta^2}}\right) \text{ or } \zeta = \sqrt{\frac{[\ln(OS)]^2}{\pi^2 + [\ln(OS)]^2}} \tag{5.15}$$

For achieving critical damping and a settling time of <1.2 seconds, $\zeta = 1$ and $\omega_n > 106$ are required. Overall, eight poles for all eight states will be calculated. In order to achieve a conventional second order system, all poles but two are placed far in the LHP with respect to these poles. Using (5.12) and (5.13), the real parts of those two poles are given by $Re(p) = -0.8929$. The other poles are somewhat set arbitrarily and overall the poles come out to be $p = [1, 18, 20, 21, 1, 18, 21, 30] * Re(p)$.

For the entire linearized system, a full state feedback will now be utilized. First, it has to be shown that the system is controllable. In order for this system to be controllable, the controllability matrix $\mathbf{Q} = [\boldsymbol{A}\ \boldsymbol{AB}\ \cdots \boldsymbol{A}^{n-1}\boldsymbol{B}]$ has to be of full rank. In this case, the matrix has full rank which can be easily checked using the MATLAB command *ctrb(A, B)* with the given state matrices.

The control input **u** is then set in dependency of the state variables $\boldsymbol{a} = -\boldsymbol{Kx}$, which creates the closed-loop system characterized by

$$\dot{\boldsymbol{x}} = (\boldsymbol{A} - \boldsymbol{BK})\boldsymbol{x} \tag{5.16a}$$

$$\boldsymbol{y} = \boldsymbol{Cx} \tag{5.17b}$$

Using Ackermann's formula, the entire gain matrix can be calculated depending on the desired pole locations. With the MATLAB command *place*, $\boldsymbol{K} \in \mathbb{R}^{2x8}$ is calculated to equal

$$\boldsymbol{K} = \begin{bmatrix} -219 & -282 & 606 & 33.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -219 & -282 & 606 & 33.5 \end{bmatrix} \tag{24}$$

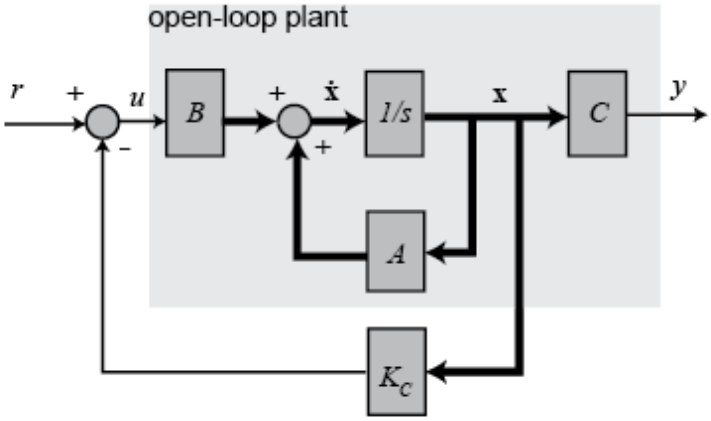


Figure 5.4: Block diagram of state feedback controller

It is necessary to mention that the pole placement method used here is only possible because a linearized system is assumed. This was achieved by using nonlinear control techniques for the motor-plate subsystem and the linearization of the plate-ball subsystem. Furthermore, the controller designed the way above only works for stabilization around the equilibrium point located in the origin. In order to introduce stabilization around arbitrary equilibrium points, a reference input $\boldsymbol{r}$ is introduced. The new outer loop control torque is calculated by

$$\boldsymbol{a} = \bar{\boldsymbol{N}}\boldsymbol{r} - \boldsymbol{K}\boldsymbol{x} \tag{5.19}$$

The scaling factor $\bar{\boldsymbol{N}} \in \mathbb{R}^{2x2}$ is necessary to scale the reference input appropriately since the gain matrix **K** is chosen without taking the reference input into account. Also it should also be noted that (5.13) is in fact equal to (5.26) the only difference being that (5.26) utilizes full state feedback while (5.13) only assumes feedback of the inner loop states,

The enhanced state space model then comes out to be

$$\dot{\boldsymbol{x}} = (\boldsymbol{A} - \boldsymbol{B}\boldsymbol{K})\boldsymbol{x} + \boldsymbol{B}\bar{\boldsymbol{N}}\boldsymbol{r} \tag{5.20a25}$$

$$\boldsymbol{y} = \boldsymbol{C}\boldsymbol{x} \tag{5.20b}$$

Applying the Laplace transform and solving for $\frac{Y(s)}{R(s)}$ yields

$$\frac{Y(s)}{R(s)} = \boldsymbol{C}\big(s\boldsymbol{I} - (\boldsymbol{A} - \boldsymbol{B}\boldsymbol{K})\big)^{-1}\boldsymbol{B}\bar{\boldsymbol{N}} = \boldsymbol{G}(s)\bar{\boldsymbol{N}} \tag{5.2126}$$

where $\boldsymbol{G}(s)$ denotes the closed loop gain matrix of the entire system when acting as a regulator system, i.e. the reference input set to zero. In order to make the entire transfer gain equal to one, which ultimately yields $error \to 0$ as $t \to \infty$. Applying the final value theorem, i.e. setting *s* to zero, and solving for $\bar{\boldsymbol{N}}$ gives the desired reference input gain.

Evaluating $\boldsymbol{G}(0)$ yields

$$\bar{N} = G(0)^{-1} = \begin{bmatrix} -219 & 0 \\ 0 & -219 \end{bmatrix}. \tag{5.22}$$

Assuming an exact model, this controller will always guarantee zero error as time goes to infinity. However, due to system uncertainties and the deviation from the nominal parameters to the actual ones, this controller will not yield zero steady state error since the calculation of $\bar{N}$ relies on a completely accurate model.

In order to guarantee zero error, also concerning time-varying set point tracking which has not yet been introduced, another approach utilizing a PI-controller will be analyzed in contrast to the approach utilizing the $\bar{N}$ gain. The necessary parameters for the entire system that are used for the simulation are listed below.

5.2.2 PI-Control for Set Point Tracking

The above designed controller works well for controlling around the equilibrium state at the origin that does not require a constant torque input to maintain that position. If one wants to stabilize the ball at a different position or even wants to make use of set-point tracking, the above designed controller does not suffice in terms of steady state and tracking error. In order for the steady state error to go to zero and the tracking error to go to zero as time goes to infinity, in theory, an integrator has to be built into the system. This is done by integrating the error of the output and feeding this additional state back into the system.

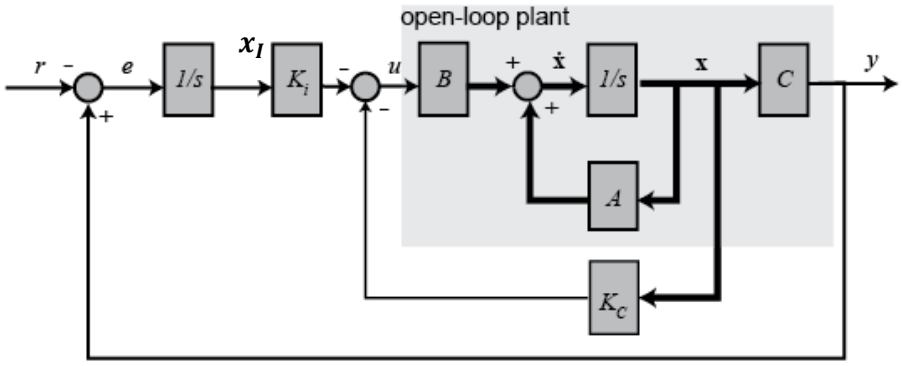


Figure 5.5: Block diagram of state feedback controller with integrator

The integrator increases the order of the system by one and, hence, introduces one more pole for each additional integrator. Again, the poles can be placed by using Ackermann's formula. Different sets of poles were tested utilizing Simulink in order to place the poles appropriately with respect to the DC-motors in order to avoid saturation when running the motors at too high torques,

In order to implement the integrator from a system's point of view, an additional state $\boldsymbol{x_I} \coloneqq \boldsymbol{y} - \boldsymbol{r}$ will be introduced along with an additional gain $\boldsymbol{K_I}$. Since the system at hand has two in- and outputs, there will be overall two more states and two more gains for the respective integrators.

The state space equations can then be reformulated as

$$\begin{bmatrix} \dot{\boldsymbol{x}} \\ \dot{\boldsymbol{x}}_{\boldsymbol{I}} \end{bmatrix} = \underbrace{\begin{bmatrix} \boldsymbol{A} & 0 \\ \boldsymbol{C} & 0 \end{bmatrix}}_{\boldsymbol{A_{ext}} \in \mathbb{R}^{10x10}} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{x_I} \end{bmatrix} + \underbrace{\begin{bmatrix} \boldsymbol{B} \\ 0 \end{bmatrix}}_{\boldsymbol{B_{ext}} \in \mathbb{R}^{10x2}} \boldsymbol{a} + \begin{bmatrix} 0 \\ -\boldsymbol{I} \end{bmatrix} \boldsymbol{r} \tag{5.23}$$

where $\boldsymbol{x_I} \in \mathbb{R}^{2x1}$ denotes the additional states for the integrators and $\boldsymbol{I} \in \mathbb{R}^{10x2}$ the identity matrix. Furthermore, $\boldsymbol{r} \in \mathbb{R}^{2x1}$ denotes the reference input, which can be either a constant or a time-varying trajectory. The control input is then chosen to be $\boldsymbol{a} = -[\boldsymbol{K_C}\ \boldsymbol{K_I}] \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{x_I} \end{bmatrix}$. By adding the two-integrator states, two additional poles are introduced. As before, they can be arbitrarily placed using Ackermann's formula.

Plugging in $\boldsymbol{a}$ and setting $\boldsymbol{K_{ext}} \coloneqq [\boldsymbol{K_C}\ \boldsymbol{K_I}]$ yields

$$\begin{bmatrix} \dot{\boldsymbol{x}} \\ \dot{\boldsymbol{x}}_{\boldsymbol{I}} \end{bmatrix} = (\boldsymbol{A_{ext}} - \boldsymbol{B_{ext}}\boldsymbol{K_{ext}}) \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{x_I} \end{bmatrix} \tag{5.24a}$$

$$\boldsymbol{y} = \boldsymbol{C}\boldsymbol{x} \tag{5.24b}$$

Choosing the poles to equal the poles specified in chapter 5.2.1 and introducing somewhat arbitrarily fast integrator poles yields the gain matrix

$$\boldsymbol{K_{ext}} = \underbrace{\begin{bmatrix} -4.00 & -0.874 & 1.05 & 0.0419 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -4.00 & -0.874 & 1.05 & 0.0419 \end{bmatrix}}_{\boldsymbol{K_C}} \underbrace{\begin{bmatrix} -2.92 & 0 \\ 0 & -2.92 \end{bmatrix}}_{\boldsymbol{K_I}} * 10^3 \tag{5.25}$$

Introducing integrators always brings the danger of making the system unstable. This holds especially true for real systems where saturation of actuators comes into play. If not handled properly, saturation can cause the integrator to integrate the error without providing the necessary torque, which can make the system response unstable. In order to fight this problem, the integration of the error could be halted as soon as saturation occurs. An even safer approach is to decrease the value of the integral once a certain torque threshold is exceeded. In this case, I decided to set $Integral_{error} = 0.98 * Integral_{error}$ once the torque provided by the motors get to high.

6 Microprocessor Implementation

As for the microprocessor, two different chips will be used. First, I implemented a simple PID-control with a MSP430 chip from Texas Instruments. After that, I used a F28335 motor control chip from Texas Instruments with higher capabilities in terms of computational power in order to implement the more thorough approach utilizing feedback linearization above while also providing much better debugging functionalities.

In order to implement the control algorithm and to get sensor readings from the resistive touchpad and the encoders, code has to be implemented and loaded onto the different microprocessors.

Since I using two different chips, I will cover the implementation process of both chips. In every section, I will talk about the MSP 430 first and after that I will cover the specifics associated with the F28335 chip.

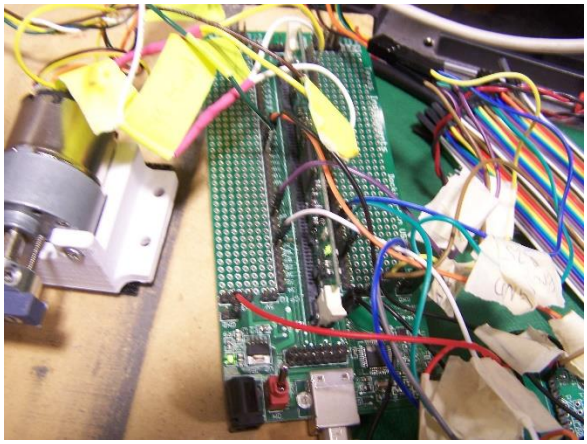


Figure 6.1: F28335 mounted on the control board

6.1 Touchpad

The touchpad is a so-called four-wire touchpad. This kind of touchpad is connected to four pins of the microprocessor with two resistors in each wire, respectively. When interfaced, the ADC connected to the touchpad provides the corresponding digital output depending on the position of the ball as indicated below. The dimensions of the touchpad are 103 mm in y-direction and 165 mm in x-direction. The touchpad, however, cannot register value up to the outer rims, which is why the maximum values are smaller than the actual touchpad dimensions. The 3D-printed walls also pose a certain limit on the range of movement of the ball. Below, the boundary values of the ranges can be seen.

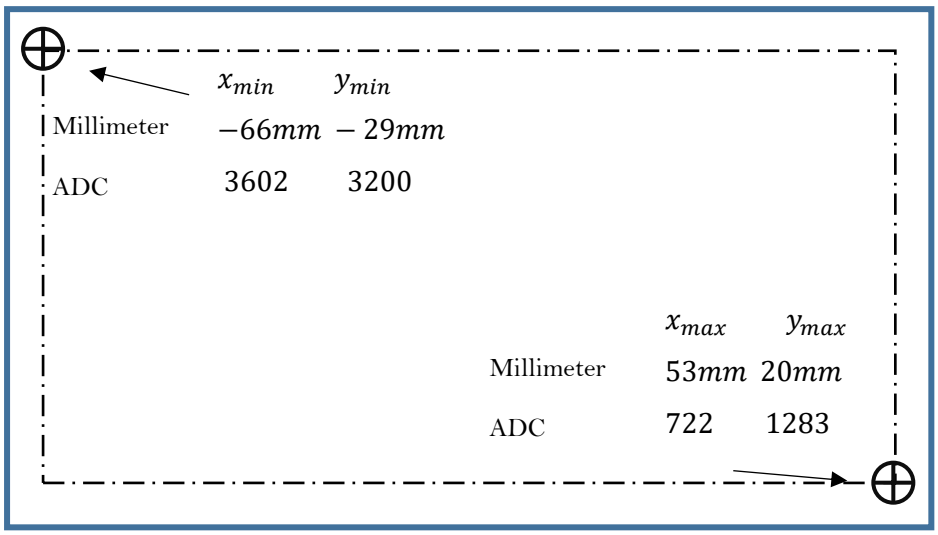


Figure 6.2: Layout of the resistive touchpad

If the x-value is to be read, a voltage is applied over the wire that lies in x-direction and the voltage is measured between the two resistors in this wire using Y1, thus acting is a kind of potentiometer that gives a proportional voltage depending on the x-position of the ball.

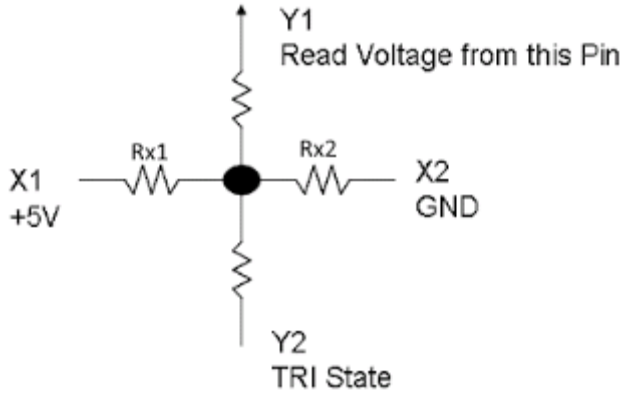


Figure 6.3: Touchpad circuit diagram for sampling x-position

The same procedure is used for reading the y-value, the only difference being that the voltage is applied over the wire that lies in the y-direction and the voltage is measured between the resistors in this wire using the pin X1.

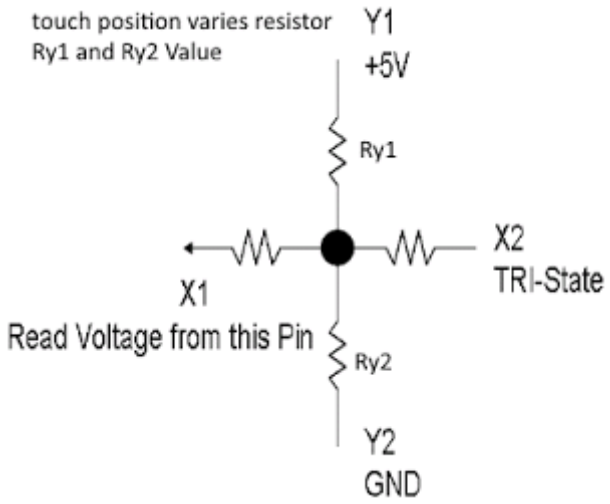


Figure 6.4: Touchpad circuit diagram for sampling y-position

The sampling procedure itself is different depending on which chip is used.

When using the MSP 430, the ADC had to be manually enabled within the code since the ADC do not have their own outputs on this chip. In addition, there are not enough GPIO pins available which is why one pin had to serve as both GPIO and ADC during the conversion process. The entire process can be described as follows:

1. Sampling of x-position
    a. Set up GPIO pins as described in figure x
    b. Enable the respective ADC channel (A0), start the conversion and read the value within the triggered ADC interrupt
2. Sampling of y-position
    a. Set up GPIO pins as described in figure xx
    b. Enable the respective ADC channel (A3), start the conversion and read the value within the triggered ADC interrupt

One entire sequence of measuring both directions takes 4ms. Hence, this is the shortest possible update time of the position and velocity values of the ball.

The output voltage is measured utilizing two channels of a 10-bit Analog-to-Digital converter (ADC) that is provided by the MSP430 microprocessor. Assuming the ADC is operating in an ideal way, this yields a signal-to-noise ratio of approximately $SQNR = 60dB$. Every 1ms the four respective pins are set up for the x-sampling, and once the ADC value has been read, the pins are set up for the y-sampling and the ADC is read once again. This does not only yield the position of the ball, but also the velocity by back differentiating and filtering the current and last values of the x- and y-position.

When using the F28335 chip, the procedure does not differ too much from the previously described one. However, the F28335 provides designated ADC pins that can be continuously enabled and sampled. This guarantees that there will be an ADC reading available every millisecond Furthermore, the ADC of the F28335 operates with 12 bits which lowers the signal-to-noise ratio to be effectively $SQNR = 74dB$. The process can be described as follows:

1. Sampling of x-position
    a. Set up GPIO pins as described in figure x
    b. Read the value of channel A0
2. Sampling of y-position
    a. Set up GPIO pins as described in figure xx
    b. Read the value of channel A1

6.2 DC-Motors

In order to interface the DC-motors with the microprocessor, an H-bridge for every one of the DC motors is soldered onto the circuit board. The H-bridge brings the advantage of controlling each one of them by generating one set of PWM signals. The PWM signal itself is modified according to the control law described in the previous chapter in order to generate the desired torque.

For generating the PWM signal with the MSP 430, the pins P2.2 and P2.4 are configured to be timer_A pins. Setting up the TTA1CCR0 register to equal 800, the carrier frequency of the PWM signal is set to $\frac{16Mhz}{800} = 20kHz$ assuming a clock frequency of 16 MHz. The TTA1CCR1 and TTA1CCR2 registers are continuously updated in order to provide a value that corresponds to the value calculated by the control law each time step. The voltage-to-torque ratio was experimentally determined to equal $\frac{0.3926Nm}{PWM-unit}$, where the PWM-unit range goes from -10 to 10. Scaling this appropriately to the PWM carrier frequency of 20 kHz, the required torque can be achieved.

The same scaling holds true for the F28335, except that predefined PWM functions write the required value to the control registers – therefore, only the PWM-value had to be defined in terms of a PWM-unit.

6.3 Encoders

The encoders are attached to the motor shaft of the DC-motors, which means that the rotational speed is the actual motor speed and the speed of the output shaft. The encoders used here are being operated in quadrature-count mode with two channels A and B.

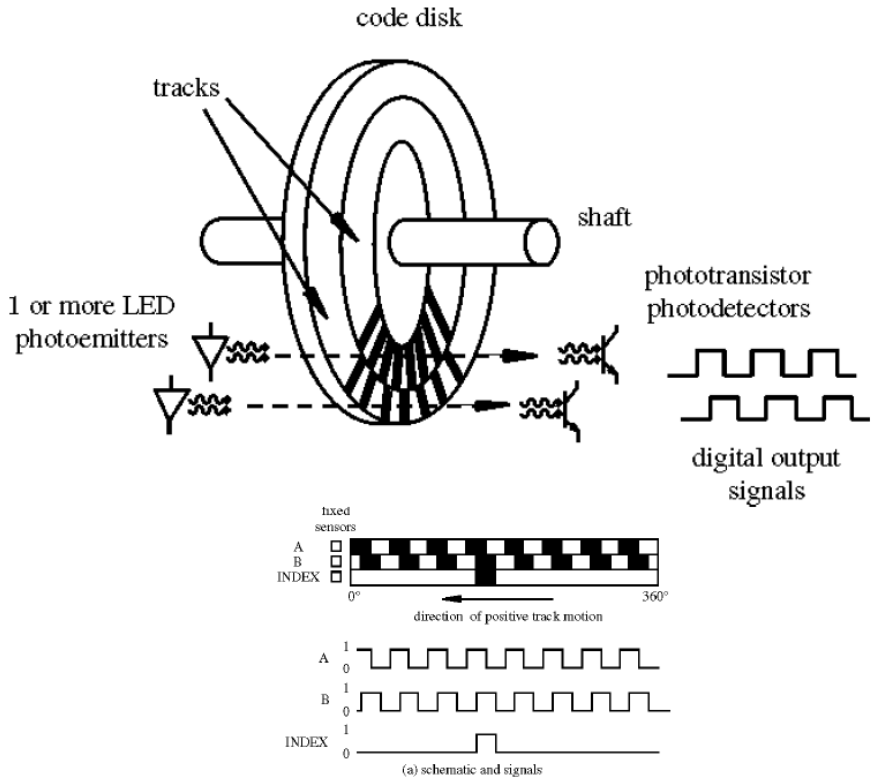


Figure 6.5: Quadrature encoder and method of function

The encoders yield 400 quadrature counts per revolution, which has to be multiplied by the effective gear ratio of 30:1 in order to yield the correct number of 12,000 counts in terms of angular displacement of the motor shaft.

As for the MSP 430, additional encoder chips had to be interfaced because the MSP 430 does not have its own encoders. Those chips are of the type LS7366 and a schematic of those is attached in the appendix. In order to sample the encoder values, the value is stored in the counter *(CTR)* register of the LS7366 chip and read by the MSP430. The two LS7366 chips are interfaced with the MSP430 by using SPI, one of the most standard data transfer protocols for embedded systems. SPI is a transfer protocol that enables the MSP430 chip to talk to different chips using just five pins.

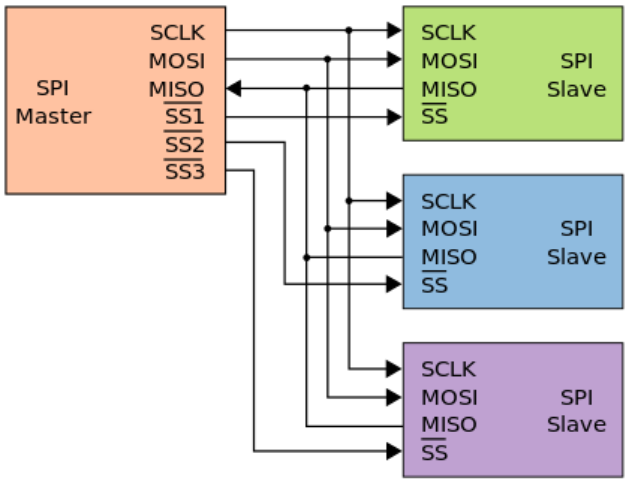


Figure 6.6: Serial Peripheral Interface overview

The main idea behind SPI is a master-slave architecture that allows one master to talk to multiple slaves. This is done via setting up a *SS* pin that is enabled or pulled down low, respectively, when talking to a specific slave is desired. The master, in this case the MSP430, provides the *clock rate* to synchronize the data transfer rate between master and slave. For shifting out the data, a *MOSI* pin is used while a *MISO* pin is used for receiving data from the slaves, in this case the LS7366 encoder chips.

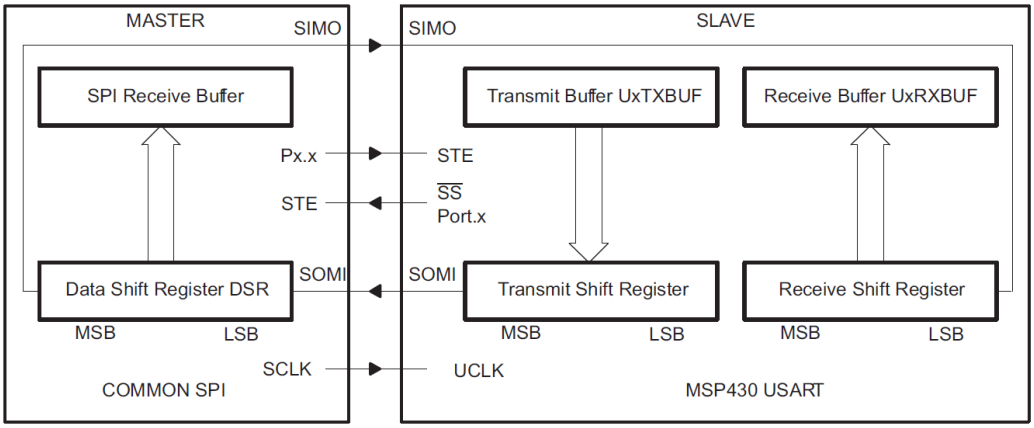


Figure 19-3. USART Slave and External Master

Figure 6.7: MSP430 master and slave registers

The LS7366 chips are configured at the beginning of the operation by writing to the two control registers MDR0 and MDR1, sending 8 addressing bits and 8 instructional bits, and after that, they are continuously read every millisecond. This is done by reading the 32-bit value that is stored within the encoder chips repeatedly. An inherent problem that comes with that is that the receive buffer of the MSP430 can only hold 8 bits. Therefore, the CNTR register is read four times transmitting 8bits each time in order to transmit the entire 32-bit value. Since the MSP430 does neither provide a receive nor a transmit FIFO, each transmit and receive has to be scheduled by utilizing the software interrupt

routine of the MSP430. The initialization of the LS7366 chips in the main-method has to be done using while-loops serving as a polling mechanism because the ISR is disabled in the main-method.

As for the F28335 chip, built-in encoders are provided that continuously sample the data.

6.4 Filter Design

A problem that comes with sensors is the introduction of noise. In order to tackle this problem, a digital filter is implemented. The velocities of the ball and the plate are indirectly measured by using the present and past encoder and touchpad value, respectively, to apply backwards differentiation. The time interval is taken to be 4ms because the setup of the code does not allow for a shorter time period.

Since differentiating sampling data can introduce large amounts of noise, this high-frequency noise has to be filtered out.

Since the data at hand is mainly used for time-domain analysis instead of frequency-domain analysis, a low pass IIR moving-average filter provides the best results and is designed in order to filter the acquired data adequately. The output of a general digital filter is given by

$$y[n] = \sum_{i=0}^{P} b_i x[n-i] - \sum_{j=1}^{Q} a_j y[n-i] \tag{6.1}$$

where b_i is the respective filter coefficient for the i^{th} data sample and a_j is the coefficient for the j^{th} output signal. For an IIR moving-average filter, each coefficient is given by $a_j = \frac{1}{N+1}$, where is the number of overall current and past samples being used and Q is the number of overall past output signal values being used.

In this case, I went for a third order moving-average filter with Q = 3. The overall filter came out to be the following:

$$y[n] = \frac{y[n-1] + y[n-2] + y[n-3]}{3} \tag{6.2}$$

This filter I applied when both calculating the velocity of the encoders and the touchpad samples.

When testing the entire system in real world, the control behavior seemed to be quite jerky. In order to overcome this, a digital low-pass filter was designed to filter the reference angle provided by the internal control law. The analog cut-off frequency was determined to be 50Hz in order to make the changes of the reference angle somewhat slower. This frequency, however, can be set differently in order to achieve a more adequate system response.The filter was then designed using MATLAB using the following steps:

1. Specify transfer function: *mytf = tf(50, [1 50])* Sampling period
2. Transform from s- to z-domain: *mydtf = c2d(mytf, 0.004, 'tustin')*

This yields the discrete-time transfer function

$$H(z) := \frac{Y(z)}{X(z)} = \frac{0.0909 + 0.0909z^{-1}}{1 - 0.818z^{-1}} \tag{6.3}$$

Here, Y stands for the output signal, in this case the filtered reference angle while X stands for the input signal, in this case the unfiltered reference angle. The source code implemented for the touchpad sampling is provided in the appendix.

7 Results

In order to simulate the different control designs, an extensive model of the system was built in MATLAB/Simulink using equations (4.7 – 4.10) along with the specific controller equations. The model is attached in the appendix. In this section, every controller will be simulated and compared with real-world results, if those have been obtained. For every simulation model, the following set of system parameters is assumed. These are comprised of the parameters introduced earlier in this thesis.

Parameter	Description	Value
K_τ	Torque constant	$0.3602\ [\frac{Nm}{A}]$
K_b	Back-emf constant	$0.3602\ [\frac{Vs}{rad}]$
J_m	Moment of inertia of motor	$4.455 * 10^{-4}\ [\frac{Nms^2}{rad}]$
$\tilde{B} \coloneqq B_m + \frac{K_\tau K_b}{R}$	Overall damping	$0.01178\ [\frac{Nms}{rad}]$
m_b	Armature resistance	$0.003\ [kg]$
r_b	Radius of ball	$0.01\ [m]$
g	Armature inductance	n/a
J_p^x	Principle moment of inertia of plate around x-axis	$7.483 * 10^{-3}\ [\frac{Nms^2}{rad}]$
J_p^y	Principle moment of inertia of plate around y-axis	$6.481 * 10^{-3}\ [\frac{Nms^2}{rad}]$
J_b	Moment of inertia of ball	$12 * 10^{-5}\ [\frac{Nms^2}{rad}]$
m_p	Mass of plate	$0.1445\ [kg]$
q	Effective transmission ratio between plate and motor angle	1.6

Table 7.1: Identified parameters for the entire system

7.1 PID-Controller

Since this control approach is not utilizing feedback linearization, the gains had to be found by simulating different gains and watching the system response. Simulating the system with different gains, a somewhat optimal set of gains was taken as a ground base for real-world testing. These gains came out to be $K_{p,1} = -0.76, K_{d.1} = -0.66, K_{p,2} = 5.5, K_{d,2} = 1.2 \; and \; K_i = -2.$

One should notice that the negative sign on the gains for the ball position come from the fact that the ball accelerates in the negative translational direction for positive angular displacement.

Using the specified gains and providing a step-input of $x_d = 0.04m \; and \; y_d = 0.02m$, the following results are obtained.

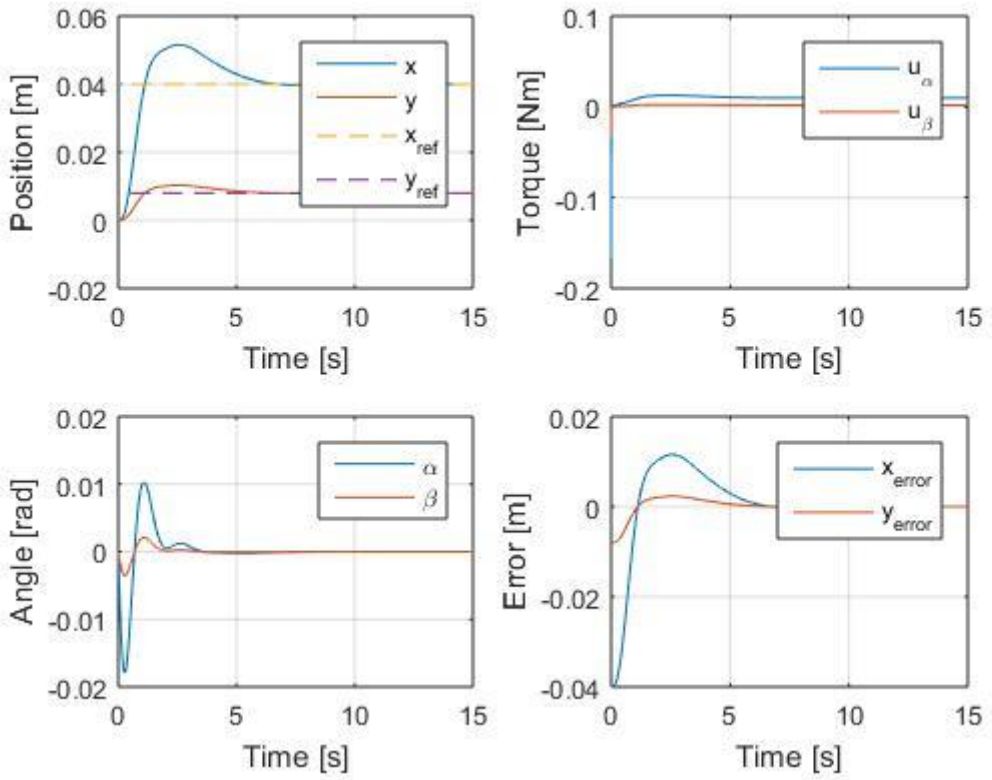


Figure 7.1: Simulation results using the PID-Controller for a given step input

It should be noted that the torque goes as high as -0.18 Nm. Considering that this is just a simulation and that the maximum torque of the motors is 0.3962 Nm, these results are deemed good and were used for testing in real-world.

7.2 Full State Feedback Controller

Using the specified gains given in (4), the following results are obtained for the same step input $x_d = 40m \; and \; y_d = 8m$. In order to compare the simulated results with the ones from the PID-controller, the poles were placed in such a way that they yield around the same torques. Then, the following results are obtained

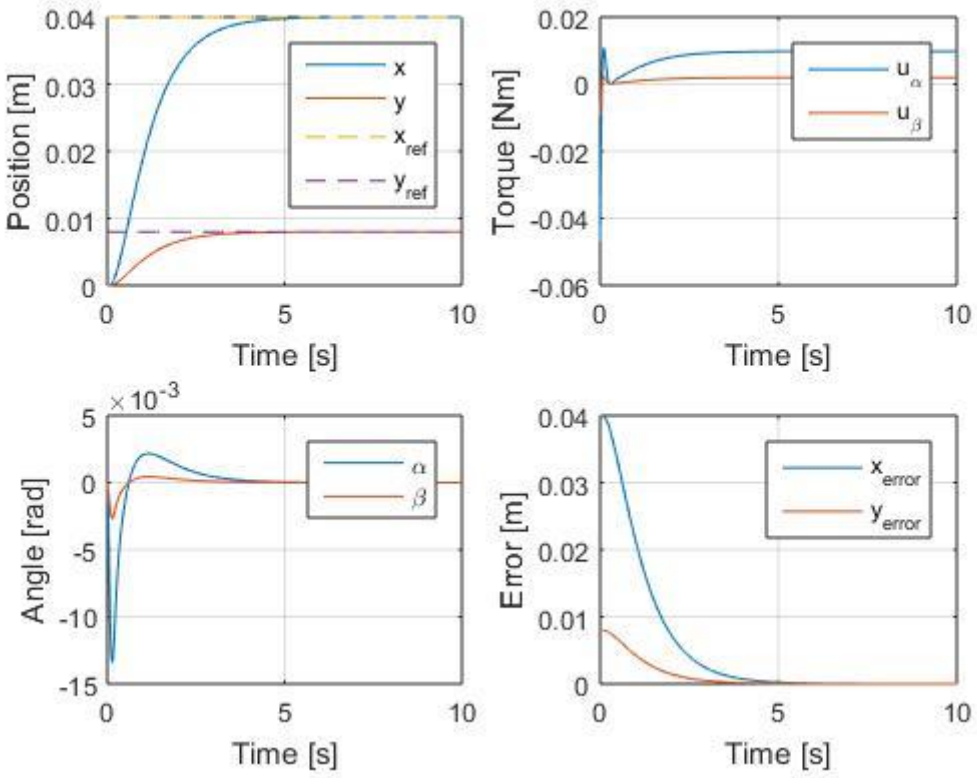


Figure 7.2: Simulation results using the full state feedback controller with partial feedback linearization for given step input

It can be seen that the step response is faster and there is no overshoot despite having similar torque outputs.

7.3 PI-Control for Set Point Tracking

Using the specified gains given in (xx), the following results are obtained for the same step input $x_d = 40mm \; and \; y_d = 8mm$. In order to compare the simulated results with the ones from the previous one, the same poles were chosen aside from the additional integrator poles. Then, the following results are obtained

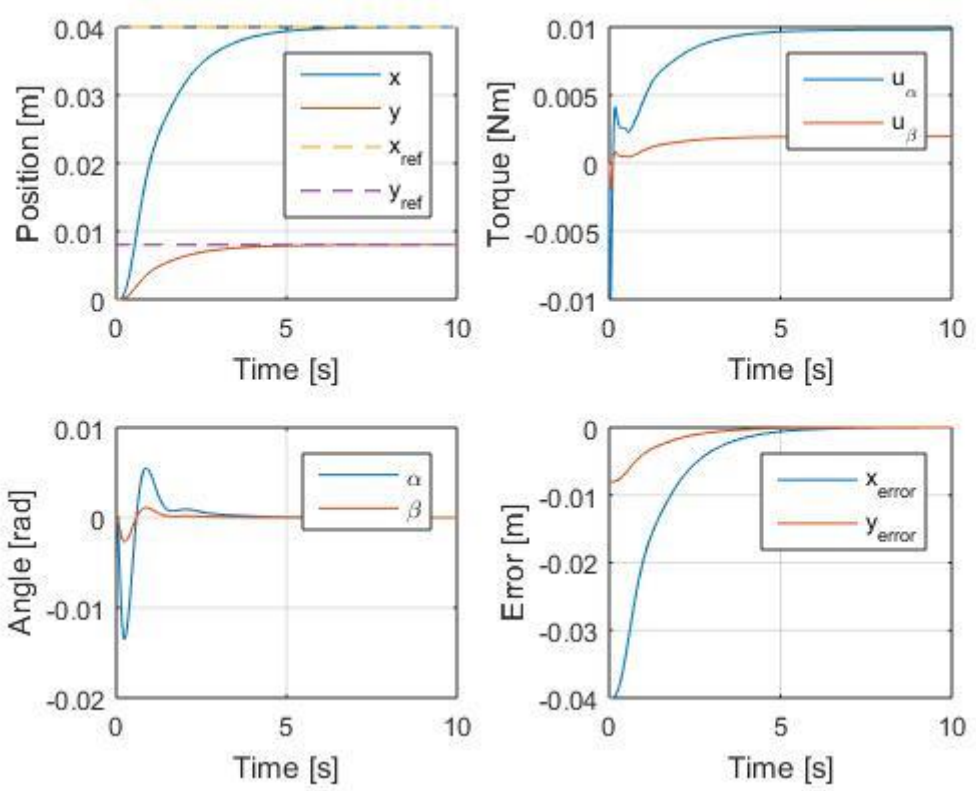


Figure 7.3: Simulation results using the state feedback controller with partial feedback linearization and added integral action for given step input

The time performance is about the same as the controller without integral action. However, the torque outputs are about ten times lower and therefore this controller is more energy efficient.

With this controller it is also possible to follow time-varying trajectories as seen below

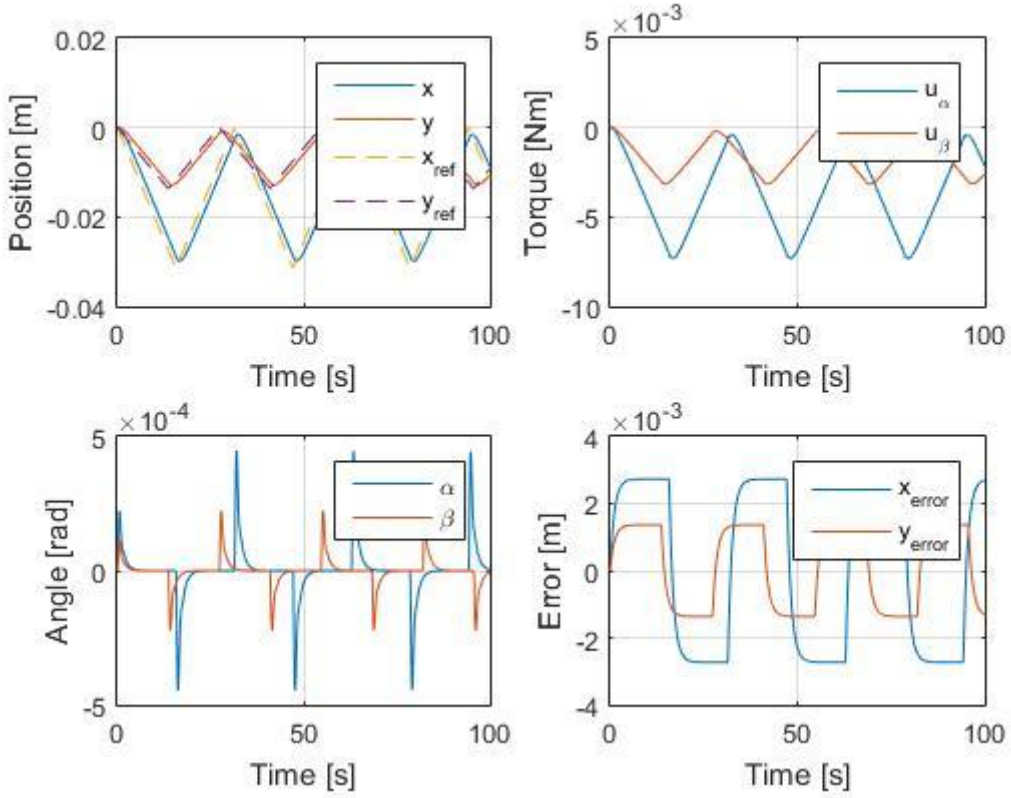


Figure 7.4: Simulation results using the state feedback controller with partial feedback linearization and added integral action for time-varying reference input

As can be seen, the controller is almost able to follow the trajectory perfectly with a very low amount of torque needed.

7.4 Testing of Real System

Overall, it is apparent that the real mechanical system behaves differently than the simulated system. This does not come unexpectedly since a real plant can rarely be perfectly identified by a theoretical model. Another main issue at hand is the backlash of the motor gears. The shaft being able to rotate forth and back by a small amount even when stalling is a direct consequence of this. This results in minor but strong enough movements of the plate, which make it difficult to keep the ball in a still position.

Since the MSP 430 did not allow for run-time change of gains, the testing and tuning was quite tedious. Obtaining actual data from the chip was quite difficult due to limited memory of the MSP430 chip.

A more critical problem was that the MSP 430 did not provide sufficient memory to introduce enough gains, and even ran into run-time issues due to limited memory. This was visible by motors going out-of-bounds repeatedly despite being saturated.

The F28335 provides run-time alteration of variables, which makes tuning of gains much easier, and it is possible to gather more data and send it to MATLAB for evaluation. Providing a step input of again $x_d = 40mm$ *and* $y_d = 8mm$, the following data is obtained

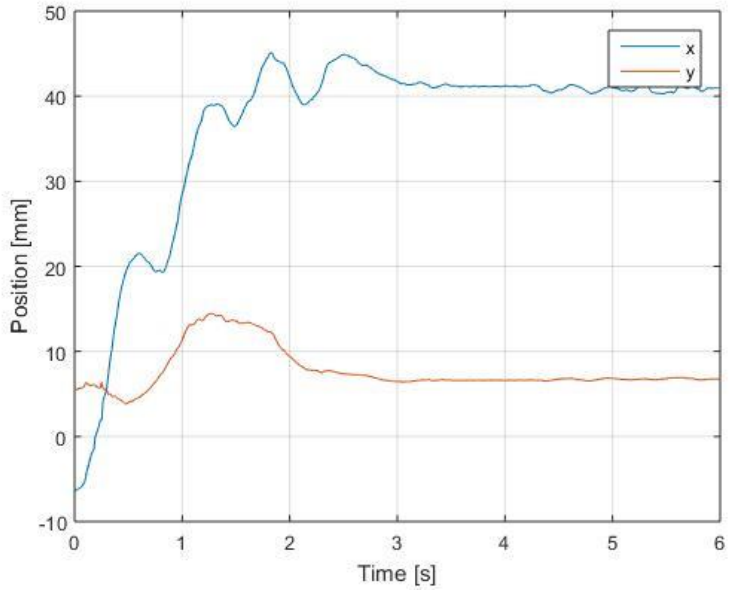


Figure 7.5: Experimental results for the ball position for a given step input

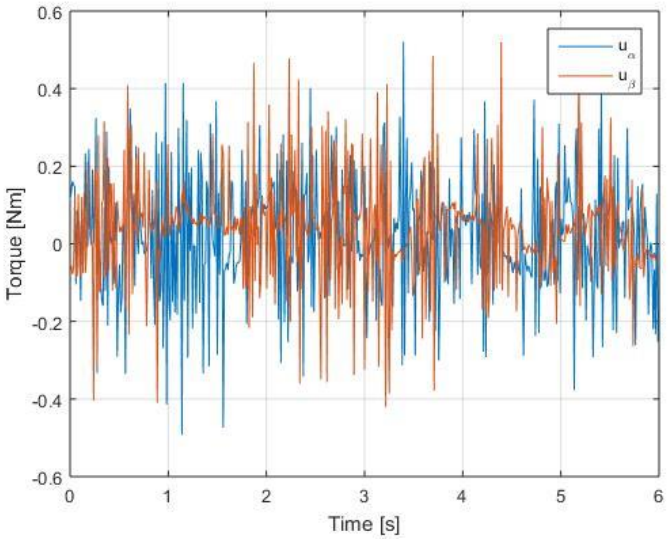


Figure 7.6: Corresponding output torque of the motors

It can be seen that the actual settling time of the system is around 3 seconds, which is quite good, and the overshoot is not too heavy. This performance comparable to what is achieved by the simulations with the specified pole locations. However, the torque output is very different in the sense that it is fluctuating heavily and in magnitude much bigger than the torques achieved in simulating. This can be explained by the mentioned backlash that makes the motor response behave like this. Other than that, the step response looks satisfactory and it can also be seen that there are heavy zero-dynamics taking place within this system by the observing oscillatory parts of the position output. This comes

mainly from the fact that the entire system is under-actuated meaning the torques do not have a direct influence on the ball position causing zero-dynamics to exist within this system.

Overall, the system is quite jerky due to the mechanical design, so future improvements could definitely be made in this regard. In addition, the linkage from motor to plate could be redesigned in a sense that one could use cams or differently designed link setups in order to improve system performance. Another aspect that could be utilized is vision feedback by a camera that might yield more accurate results for the position of the ball than the touchpad. What can be definitely be improved also is the backlash of the motor by choosing motors that utilize different sets of gears.

8 Appendix

8.1 Used Pins of F28335 Control Board

F28335 Pin	Connected with	Purpose
A0	Touchpad Y1	Reading of X-position
A1	Touchpad X1	Reading of Y-position
GPIO 00	H-bridge 2	PWM signal
GPIO 02	H-bridge 1	PWM signal
GPIO 01	Touchpad Y1	3.3V
GPIO 05	Touchpad X1	3.3V
GPIO 04	Touchpad Y2	Tri-state/GND
GPIO 03	Touchpad X2	Tri-state/GND
GPIO 24	Channel A motor 2	Encoder channel A
GPIO 25	Channel B motor 2	Encoder channel B
GPIO 20	Channel A motor 1	Encoder channel A
GPIO 21	Channel B motor 1	Encoder channel B

Table 8.1: Used Pins of F28335 Control Board

8.2 C Code

Extensive C code was written using the CodeComposer Suite from Texas Instruments. The code was written in order to implement the control algorithms, interface the sensors and collect data from the chips used. Since I was using two different chips, I composed two separate main files.

8.2.1 MSP 430 Code

The code using the MSP 430 uses more code that directly interacts with register because these were set up manually.

```
/*******************************************************************************
MSP430G2553 Project Creator

GE 423  - Dan Block
        Spring(2012)

        Written(by) : Maximilian Sieb
College of Engineering Control Systems Lab
University of Illinois at Urbana-Champaign
 *******************************************************************************/
```

```
#include "msp430g2553.h"
#include "UART.h"



char newprint = 0;
unsigned long timecnt = 0;


//[b,a]=butter(3,0.2,'low') for MATLAB

int ADCcnt = 0;    //used to find out how much time lies between two ADC samplings
per direction (came out to be 4ms)
int ADCflag = 0;



long saveenc1[10];
char datacnt = 0;

int y_ref = 573; //zero value is 573  Y
int x_ref = 538; //zero value is 538    X

int e_x = 0;
int e_y = 0;

float tau_1 = 0;          //torques
float tau_2 = 0;
float Kpx1 = 1;           //gains   //1
float Kpy1 = 1;
float Kdx1 = 0.001;
float Kdy1 = 0.001;
float Kpx2 = 0.001;       //gains //0.006
float Kpy2 = 0.001;
float Kdx2 = 0.1;
float Kdy2 = 0.1;
float theta1ref = 0;
float theta2ref = 0;
//float h1 = 0;
//float h2 = 0;
//float h3 = 0;
//float h4 = 0;

char flagMotor = 0; //flag to guide the code through the motor/PWM control
char flagPad = 0; //flag to guide the code through the ADC sampling
int LSstate = 0; //flag to guide the code through the encoder sampling
char new_xdata = 0; //adc data flag
char new_ydata = 0;
char new_encdata = 0; //encoder data flag
int ii = 0;
char refFlag = 0;
char flag = 1;

long enc1reading = 0;           //raw encoder readings
long enc2reading = 0;
float enc1vel = 0;
float enc2vel = 0;
```

```
float enc1vel_old = 0;
float enc2vel_old = 0;
float enc1vel_old2 = 0;
float enc2vel_old2 = 0;
long enc1reading_old = 0;
long enc2reading_old = 0;


int SPIbyte1 = 0;
int SPIbyte2 = 0;
int SPIbyte3 = 0;
int SPIbyte4 = 0;
int SPIbyte5 = 0;



 void main(void) {

       WDTCTL = WDTPW + WDTHOLD;                 // Stop WDT

       if (CALBC1_16MHZ ==0xFF || CALDCO_16MHZ == 0xFF) while(1);

       DCOCTL = CALDCO_16MHZ;    // Set uC to run at approximately 16 Mhz
       BCSCTL1 = CALBC1_16MHZ;

       // Initialize Port 1
       P1SEL &= ~0x01;  // See page 42 and 43 of the G2553's datasheet, It shows
that when both P1SEL and P1SEL2 bits are zero
       P1SEL2 &= ~0x01; // the corresponding pin is set as a I/O pin.  Datasheet:
http://coecsl.ece.illi-
nois.edu/ge423/datasheets/MSP430Ref_Guides/msp430g2553datasheet.pdf
       P1REN = 0x0;  // No resistors enabled for Port 1
       P1DIR |= 0x1; // Set P1.0 to output to drive LED on LaunchPad board.  Make
sure shunt jumper is in place at LaunchPad's Red LED
       P1OUT &= ~0x01;  // Initially set P1.0 to 0

       /*Set up pins for motor PWM signal
        P2.2 - TA1.1
        P2.4 - TA1.2
        */
       P2DIR |= (BIT2 + BIT4);                                          //set pins
to output
       P2SEL |= (BIT4+BIT2);                           // P2.2 and 2.4  option
       P2SEL2 &= ~(BIT4+BIT2);                        // P2.2 TA1.1 and P2.4 TA1.2
option

       //set P2.0 as output GPIO for debugging to check duration of timer function
       P2DIR |= (BIT0);                                          //set pins to out-
put
       P2SEL &= ~(BIT0);                           // P2.2 and 2.4  option
       P2SEL2 &= ~(BIT0);                        // P2.2 TA1.1 and P2.4 TA1.2 op-
tion
       P2OUT &= ~BIT0;

       TA1CCR0 = 800;                                          // PWM carrier
frequency - 20kHz
       TA1CCR1 = 400;                              //TA1CCR1 PWM duty cycle   -
50% (equals zero speed)
       TA1CCTL1 = OUTMOD_7;                        // TA1CCR1 reset/set
```

```
      TA1CCR2 = 400;                                // TA1CCR2 PWM duty cycle - 50%
(equals zero speed
      TA1CCTL2 = OUTMOD_7;                          // TA1CCR2 reset/set
      TA1CTL = TASSEL_2 + MC_1;                    // SMCLK and up-mode

      /***********/
      /*Set up GPIO for touchpad
      P1.0 - Y1
      P2.1 - Y2
      P1.3 - X1
      P2.5 - X2
       */

      //set pins up as GPIO
      P1SEL &= ~(BIT0+BIT3);
      P1SEL2 &= ~(BIT0+BIT3);
      P2SEL &= ~(BIT1  +BIT5);
      P2SEL2 &= ~(BIT1  +BIT5);

      ADC10CTL0 = SREF_0 + ADC10SHT_0 + ADC10IE + ADC10ON; //initialize ADC10 con-
trol register 0
      ADC10CTL1 = (SHS_0 + ADC10DIV_0) & ~ADC10DF;       //initialize ADC10 con-
trol register 1, INCH will be selected later in code


      /***********/
      /*Set up GPIO for encoders
      P1.4 - SMCLK
      P2.6 - SS enc1
      P2.7 - SS enc2
      P1.5 - UCB0CLK
      P1.6 - UCB0 SOMI
      P1.7 - UCBO SIMO
       */

      //set up P1.4 as SMCLK for the LS7366 preprocessing (fcki port of the
LS7366)
      P1DIR |= BIT4;
      P1SEL |= BIT4;
      P1SEL2 &= ~BIT4;

      //set up P2.6 and P2.7 as SS pins as GPIO
      P2SEL &= ~(BIT6  +BIT7);
      P2SEL2 &= ~(BIT6  +BIT7);
      P2DIR |= (BIT6 + BIT7); //set as output
      P2OUT |= (BIT6 + BIT7);

      //Set up pins for SPI (SCK,SIMO and SOMI)
      P1SEL |= (BIT5 + BIT6  + BIT7);                          //Set up P1SEL and
P1SEL2 for SPI
      P1SEL2 |= (BIT5  + BIT6 + BIT7);

      //set up SPI for encoder communication
      UCB0CTL0 = UCCKPH+UCMSB + UCMST + UCSYNC;  // 3-pin, 8-bit SPI master, sets
up USCI
      UCB0CTL0 &= ~(UCCKPL);          //Polarity and Phase equal to 0 (see LS7366
data sheet)
      //UCB0CTL0 &= ~ UC7BIT;
      UCB0CTL1 = UCSSEL_2 + UCSWRST;                    // SMCLK
```

```
	UCB0BR0 = 40;                                          // divide clock by 80 to
achieve a SCK of 200kHz (try 16 without delay cycles)
	UCB0BR1 = 0;                                      //
	//UCB0MCTL = 0;                                     // No modulation
	UCB0CTL1 &= ~UCSWRST;                             // **Initialize USCI state ma-
chine**

	//LS7366 initialization
	/******/
	P2OUT &= ~(BIT6+BIT7); //pull both chip selects low
	IFG2 &= ~UCB0RXIFG;//clear interrupt flag
	UCB0TXBUF = ((unsigned)0x20); //CLR Count both chips

	while((IFG2&UCB0RXIFG) == 0) {} //poll on flag register because interrupt
routine are disabled during main
	IFG2 &= ~UCB0RXIFG;

	__delay_cycles(100); //delay clock cycles because chip select was being
pulled high while the last clock has not finished yet (clock frequency too low)
	P2OUT |= (BIT6+BIT7); //end instruction - pull chip select back high
	SPIbyte1 = UCB0RXBUF; //clear RX buffer

	P2OUT &= ~(BIT6+BIT7);
	UCB0TXBUF = ((unsigned)0x88); //WR to MDR0

	while((IFG2&UCB0RXIFG) == 0) {}
	IFG2 &= ~UCB0RXIFG;
	SPIbyte1 = UCB0RXBUF; //write second byte to MDR0
	UCB0TXBUF = ((unsigned)0x83);

	while((IFG2&UCB0RXIFG) == 0) {}
	IFG2 &= ~UCB0RXIFG;
	__delay_cycles(100);
	P2OUT |= (BIT6+BIT7); //second byte has been transmitted - end instuction
	SPIbyte1 = UCB0RXBUF;
	P2OUT &= ~(BIT6+BIT7);
	UCB0TXBUF = ((unsigned)0x90);  //write to MDR1

	while((IFG2&UCB0RXIFG) == 0) {}
	IFG2 &= ~UCB0RXIFG;
	//P2OUT |= (BIT6+BIT7);
	SPIbyte1 = UCB0RXBUF; //write second byte to MDR1
	UCB0TXBUF = 0x00;

	while((IFG2&UCB0RXIFG) == 0) {}
	IFG2 &= ~UCB0RXIFG;
	__delay_cycles(100);
	P2OUT |= (BIT6+BIT7);
	SPIbyte1 = UCB0RXBUF; //initialization complete

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

	// Timer A Config
	TACCTL0 = CCIE;                 // Enable Periodic interrupt
	TACCR0 = 16000;                // period = 1ms
	TACTL = TASSEL_2 + MC_1; // source SMCLK, up mode


	Init_UART(9600,1);  // Initialize UART for 9600 baud serial communication
```

```
	IFG2 &= ~UCB0RXIE;
	IE2 |= UCB0RXIE;                          // Enable USCI0 RX interrupt


	_BIS_SR(GIE);             // Enable global interrupt


	while(1) {

		if(newmsg) {
			newmsg = 0;
		}

		if (newprint)  {
			//P1OUT ^= 0x1;            // Blink LED
			//UART_printf("%d \n\r",Xpos);
			//UART_printf("Hello %d\n\r",(int)(timecnt/500));
			//UART_printf("v:%d:p:%d \n\r ",(unsigned int)Xvel,Xpos);
			//UART_printf("e1:%d e2:%d\n\r ",(int)enc1vel,(int)enc1read-
ing);
			//UART_printf("e1:%d e2:%d\n\r ",(int)enc1read-
ing,(int)enc2reading);
			newprint = 0;
		}

	}
}


// Timer A0 interrupt service routine
#pragma vector=TIMER0_A0_VECTOR
__interrupt void Timer_A (void)
{

	//     if (timecnt%500 == 0) {
	//           if (refFlag) {
	//           x_ref = 200;
	//           y_ref = -200;
	//           refFlag = 0;
	//           }
	//      else {
	//           x_ref = 0;
	//           y_ref = 0;
	//           refFlag = 1;
	//      }
	//     }

	P2OUT &= ~(BIT6+BIT7);
	UCB0TXBUF = ((unsigned)0xE8);// Latch All ENCs
	LSstate = 4;
	//P2OUT |= BIT0;
	timecnt++; // Keep track of time for main while loop.
	if(ADCflag == 1) {
		ADCcnt++;
	}
	if ((timecnt%500) == 0) {
		newprint = 1;  // flag main while loop that .5 seconds have gone by.
	}
```

```
	//TOUCHPAD SETUP AND READING
	/********/
	/*
	 * the x- and y-direction will be sampled subsequently by reading ADC chan-
nel A0 and A3 in succession and setting up all pins accordingly
	 */

	//x-range: 230 to 780
	//y-range: 340 to 720

	//set up pins for X measurement

	switch (flagPad) {
	case 0:
		flagPad = 1;
		if(ADCflag == 0) {
			ADCflag = 1;
		}
		else if (ADCflag) {
			ADCflag = 2;
		}
		ADC10CTL0 &= ~(ENC);
		P1DIR &= ~BIT0;    //Input      Y1 (this voltage is 'measuring' the
x-position)
		P2DIR &= ~BIT1;           //Input/Tri-state   Y2
		P1DIR |= BIT3;         // output X1
		P2DIR |= BIT5; //Output                X2

		P1OUT |= BIT3;            //set X1 high
		P2OUT &= ~BIT5;           //set X2 low

		ADC10CTL1 = INCH_0;
		ADC10AE0 = BIT0;                      //enable A0
		ADC10CTL0 |= (ENC);


		break;

	case 1:
		flagPad = 4;
		//ADC10CTL1 = INCH_0;
		//ADC10AE0 = BIT0;                            //enable A0
		ADC10CTL0 |= (ENC + ADC10SC); //take sample

		break;
	case 2:
		flagPad = 3;
		ADC10CTL0 &= ~(ENC);
		P1DIR |= BIT0;    //Output: set to 1    Y1
		P2DIR |= BIT1;            //output                 Y2
		P1DIR &= ~BIT3;     //    input              X1   (this voltage is
'measuring' the y-position)
		P2DIR &= ~BIT5; // input/tri-state           X2

		P1OUT |= BIT0;            //set Y1 high
		P2OUT &= ~BIT1;           //set Y2 low

		ADC10CTL1 = INCH_3;
		ADC10AE0 = BIT3;                            //enable A3
		ADC10CTL0 |= (ENC); //take sample
```

```
            break;
      case 3:
            flagPad = 5;
            //ADC10CTL1 = INCH_3;
            //ADC10AE0 = BIT3;                                    //enable A3
            ADC10CTL0 |= (ENC + ADC10SC); //take sample

            break;
      case 4:
            break;
      case 5:
            break;
      default:
            break;

      }

      /*******/


      //P2OUT &= ~BIT0;
}



// ADC 10 ISR - Called when a sequence of conversions (A7-A0) have completed
#pragma vector=ADC10_VECTOR
__interrupt void ADC10_ISR(void) {


      if(flagPad == 4) {    //sample X values
            flagPad = 2;


            if(timecnt>20) {  //sometimes the touchpad position is not zeroed
                  //right now its FIR filter (subsitute Xpos with Xsample to make
it IIR) - test different performances?


            }

            //Xsample[0] = (Xsample[0]*165L)/1023;    //for IIR
            //    Xsample[0] = (Xsample[0] + Xsample[1] + Xsample[2] + Xsam-
ple[3])/4;  //for IIR

            //FIR velocity filter (try IIR with 4 values)







      } else if (flagPad == 5) {            //sample Y values
            flagPad = 0;
```

```
            //calculate velocity and position every 50ms to reduce sensitivity to
noise
            if(timecnt > 20) { //sometimes the touchpad position is not zeroed


            }



            //IIR velocity filter (try FIR with 4 values)




            if ((timecnt%2000 == 0)) {

                  if (flag) {
                        theta1ref = 100;
                        theta2ref= 100;
                        flag = 0;
                  } else {
                        theta1ref = -100;
                        theta2ref = -100;
                        flag = 1;
                  }
            }


            //for ball position control
            tau_1 =Kpy2*(theta1ref-enc1reading);// - Kdy2*enc1vel;
            tau_2 =Kpx2*(theta2ref-enc2reading);//- Kdx2*enc2vel;

      //    tau_1=(theta1ref) - Kdy2*enc1vel-Kpy2*enc1reading;
            //tau_2=(theta2ref) - Kdx2*enc2vel-Kpx2*enc2reading;

//          h1 = Kpy2*(theta1ref-enc1reading);
//          h2 = Kdy2*enc1vel;
//          h3 = Kpx2*(theta2ref-enc2reading);
//          h4 = Kdx2*enc2vel;

//          if((enc1reading > 400) || (enc1reading < -400)) {
//                tau_1 = 0;
//          } else      if((enc2reading > 400) || (enc2reading < -400)) {
//                tau_2 = 0;
//          }


      //tau_1 = 0;

            //for angular control (plate angle)
            //tau_1 = Kpy*e_y- Kdy*enc1vel;
            //tau_2 = Kpx*e_x- Kdx*enc2vel;

            //add 0.128 Nm Gravity compensation

            //if( (abs(e_y)<20) && (abs(e_x)<20) ) {
            //do nothing
            //} else {
```

```
            //convertion from Nm to register value
            if(((tau_1/0.03926+10)*40)>800) {
                  TA1CCR1 = 800;
            } else if (((tau_1/0.03926+10)*40)<0) {
                  TA1CCR1 = 0;              //motor constant is 0.03926Nm/PWM-
unit assuming unit range from -10 to 10. This needs to be scaled to register value
of 800.
            } else {
                  TA1CCR1 = (tau_1/0.03926+10)*40;
            }

            //          if(enc1reading < 600) {
            //          TA1CCR1 = 550;
            //          } else {
            //                TA1CCR1 = 400;
            //          }

            //convertion from Nm to register value
            if(((tau_2/0.03926+10)*40)>800) {
                  TA1CCR2 = 800;
            } else if (((tau_2/0.03926+10)*40)<0) {
                  TA1CCR2 = 0;
            } else {
                  TA1CCR2 = (tau_2/0.03926+10)*40 ;
            }
      }
      //                TA1CCR2 = 400;

      /********/


      if (datacnt == 10) {
            datacnt = 0;
      }
      saveenc1[datacnt] = enc1reading;
      datacnt++;
      P2OUT &= ~BIT0;


}




// USCI Transmit ISR - Called when TXBUF is empty (ready to accept another charac-
ter)
#pragma vector=USCIAB0TX_VECTOR
__interrupt void USCI0TX_ISR(void) {

      if(IFG2&UCA0TXIFG) {            // USCI_A0 requested TX interrupt
            if(printf_flag) {
                  if (currentindex == txcount) {
                        senddone = 1;
                        printf_flag = 0;
                        IFG2 &= ~UCA0TXIFG;
                  } else {
                        UCA0TXBUF = printbuff[currentindex];
```

```
                        currentindex++;
                }
        } else if(UART_flag) {
                if(!donesending) {
                        UCA0TXBUF = txbuff[txindex];
                        if(txbuff[txindex] == 255) {
                                donesending = 1;
                                txindex = 0;
                        }
                        else txindex++;
                }
        } else {  // interrupt after sendchar call so just set senddone flag
since only one char is sent
                senddone = 1;
        }

        IFG2 &= ~UCA0TXIFG;
    }

    if(IFG2&UCB0TXIFG) {      // USCI_B0 requested TX interrupt (UCB0TXBUF is
empty)

        IFG2 &= ~UCB0TXIFG;   // clear IFG
    }
}


// USCI Receive ISR - Called when shift register has been transferred to RXBUF
// Indicates completion of TX/RX operation
#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCI0RX_ISR(void) {

    if(IFG2&UCB0RXIFG) {  // USCI_B0 requested RX interrupt (UCB0RXBUF is full)

        IFG2 &= ~UCB0RXIFG;   // clear IFG

        //ENCODER INSTRUCTIONS AND READING
        /***************/
        /*
         * here, the encoders are being read every time after they have been
latched within the timer_A interrupt
         */
        switch(LSstate) {

        case 4:                                                 //read the encod-
ers
                P2OUT |= BIT0;
                P2OUT |= (BIT6+BIT7);           //end latch instruction
                SPIbyte1 = UCB0RXBUF;
                //     __delay_cycles(75);
                P2OUT &= ~BIT6;    //read first encoder
                UCB0TXBUF = ((unsigned)0x68);
                LSstate = 5;
                break;
        case 5:
                SPIbyte1 = UCB0RXBUF; //trash byte for transmitting and clear-
ing RX buffer
                UCB0TXBUF = 0x00;
                LSstate = 6;
                break;
```

```
            case 6:
                  SPIbyte2 = UCB0RXBUF; //first data byte
                  UCB0TXBUF = 0x00;
                  LSstate = 7;
                  break;
            case 7:
                  SPIbyte3 = UCB0RXBUF; //second data byte
                  UCB0TXBUF = 0x00;
                  LSstate = 8;
                  break;
            case 8:
                  SPIbyte4 = UCB0RXBUF; //third data byte
                  UCB0TXBUF = 0x00;
                  LSstate = 9;
                  break;
            case 9:
                  P2OUT |= BIT6;             //end instruction to P2.6/encoder1
                  SPIbyte5 = UCB0RXBUF; //fourth data byte
                  enc1reading = ((long)SPIbyte2<<24) | ((long)SPIbyte3<<16) |
((long)SPIbyte4<<8) | (long)SPIbyte5;
                  P2OUT &= ~BIT7;    //read second encoder
                  UCB0TXBUF = ((unsigned)0x68);
                  LSstate = 11;
                  break;
            case 11:
                  SPIbyte1 = UCB0RXBUF;
                  UCB0TXBUF = 0x00;
                  LSstate = 12;
                  break;
            case 12:
                  SPIbyte2 = UCB0RXBUF; //first data byte
                  UCB0TXBUF = 0x00;
                  LSstate = 13;
                  break;
            case 13:
                  SPIbyte3 = UCB0RXBUF; //second data byte
                  UCB0TXBUF = 0x00;
                  LSstate = 14;
                  break;
            case 14:
                  SPIbyte4 = UCB0RXBUF; //third data byte
                  UCB0TXBUF = 0x00;
                  LSstate = 15;
                  break;
            case 15:
                  P2OUT |= BIT7;             //end instruction for P2.7/encoder2
                  SPIbyte5 = UCB0RXBUF; //fourth data byte
                  enc2reading = ((long)SPIbyte2<<24) | ((long)SPIbyte3<<16) |
((long)SPIbyte4<<8) | (long)SPIbyte5;

                  //ENCODER INSTRUCTIONS AND READING
                  /*****/

                  enc1vel = (enc1reading - enc1reading_old)*1000; //maybe try
calculating delta_t, but should be 1ms if SPI is fast enough
                  enc1vel = (enc1vel + enc1vel_old + enc1vel_old2)/3;
                  enc1vel_old2 = enc1vel_old;
                  enc1vel_old = enc1vel;

                  enc2vel = (enc2reading - enc2reading_old)*1000;
```

```
                        enc2vel = (enc2vel + enc2vel_old + enc2vel_old2)/3;
                        enc2vel_old2 = enc2vel_old;
                        enc2vel_old = enc2vel;


                        enc1reading_old = enc1reading;
                        enc2reading_old = enc2reading;

                        /*****/
                        break;
                default:
                        break;
                }



                /**************/
        }

        if(IFG2&UCA0RXIFG) {  // USCI_A0 requested RX interrupt (UCA0RXBUF is full)

                //    Uncomment this block of code if you would like to use this COM
protocol that uses 253 as STARTCHAR and 255 as STOPCHAR
                /*          if(!started) {      // Haven't started a message yet
                        if(UCA0RXBUF == 253) {
                                started = 1;
                                newmsg = 0;
                        }
                }
                else { // In process of receiving a message
                        if((UCA0RXBUF != 255) && (msgindex < (MAX_NUM_FLOATS*5))) {
                                rxbuff[msgindex] = UCA0RXBUF;

                                msgindex++;
                        } else {      // Stop char received or too much data received
                                if(UCA0RXBUF == 255) {    // Message completed
                                        newmsg = 1;
                                        rxbuff[msgindex] = 255;   // "Null"-terminate the
array
                                }
                                started = 0;
                                msgindex = 0;
                        }
                }
                 */

                IFG2 &= ~UCA0RXIFG;
        }

}
```

8.2.2 F28335 Code

The code using the F28335 can transmit data to MATLAB and allows for using more variables and applying more sophisticated control algorithms.

```
#include <tistdtypes.h>
#include <coecsl.h>
#include "user_includes.h"
#include "28335_dma.h"
#include "28335_spi.h"
#include "28335_inits.h"
#include "mcbsp_com.h"
#include "i2c.h"

//void updateData(void);
//void sendData(void);

//extern volatile int new_irdata_i2c;
//extern int adc1_i2c;
//extern int adc2_i2c;
//extern int adc3_i2c;
//extern int adc4_i2c;
//extern int adc5_i2c;
//extern int adc6_i2c;
//extern int adc7_i2c;
//extern int adc8_i2c;
//extern int CompassNew;
//extern int ir1_i2c,ir2_i2c,ir3_i2c,ir4_i2c,ir5_i2c;

//extern int SPIenc_state;
//extern long SPIenc1_reading;
//extern long SPIenc2_reading;
//extern long SPIenc3_reading;
//extern long SPIenc4_reading;
//
//extern unsigned int nocomm;
//extern int newOMAPdata;
//extern int PIVEL_enable;
//extern float mcbsp_vref;
//extern float mcbsp_turn;
//extern float mcbsp_pwm1;
//extern float mcbsp_pwm2;
//extern float mcbsp_dac1;
//extern float mcbsp_dac2;
//extern long McBSP_COMerr;
//extern int McBSP_RecGoodData_ReadyTX;
//extern long McBSPb_int_count;
//extern int McBSPB_rdy;
//extern mcbsp28x_com TXBuff;
//extern mcbspL138_com RXBuff;

#pragma DATA_SECTION(xarray, ".my_arrs")
float xarray[700];
#pragma DATA_SECTION(yarray, ".my_arrs")
float yarray[700];

int arrayindex = 0;
```

```
unsigned long timeint = 0;
unsigned long noi2c = 0;
int toggleUSonic = 1;

//float gyro_x = 0;
//float gyro_y = 0;
//
//float Enc1_rad = 0;
//float Enc2_rad = 0;
//float Enc3_rad = 0;
//float Enc4_rad = 0;
//
//int switchstate = -2;
//
//int newF28335_Extra = 0;
//float F28335_Extra1 = 0.0;
//float F28335_Extra2 = 0.0;
//float F28335_Extra3 = 0.0;
//float F28335_Extra4 = 0.0;

int raw_adc_A0 = 0;
int raw_adc_A1 = 0;
int raw_adc_A2 = 0;
int raw_adc_A3 = 0;
int raw_adc_A4 = 0;
int raw_adc_A5 = 0;
int raw_adc_A6 = 0;
int raw_adc_A7 = 0;
int raw_adc_B0 = 0;
int raw_adc_B1 = 0;
int raw_adc_B2 = 0;
int raw_adc_B3 = 0;
int raw_adc_B4 = 0;
int raw_adc_B5 = 0;
int raw_adc_B6 = 0;
int raw_adc_B7 = 0;

//float omap_vref = 0;
//float omap_turn = 0;
//float omap_pwm1 = 0;
//float omap_pwm2 = 0;
//float omap_dac1 = 0;
//float omap_dac2 = 0;
//int omap_PIVEL_enable = 0;

eqep_t enc1;
eqep_t enc2;
float value_enc1 = 0;
float value_enc2 = 0;
float value_enc1_old = 0;
float value_enc2_old = 0;
float enc1vel = 0;
float enc2vel = 0;
float u1 = 0;
float u2 = 0;
float theta1ref = 0;
float theta2ref = 0;
float Xpos = 0; //2200 is origin for both values
float Ypos = 0;
```

```
float Xpos_old = 0;
float Ypos_old = 0;
float Xvel = 0;
float Yvel = 0;
float Xvel_old1 = 0;
float Yvel_old1 = 0;
float Xvel_old2 = 0;
float Yvel_old2 = 0;
//float Xref = 0;
//float Yref = 0;
float Xref = 0;
float Yref = 0;

float eY = 0;
float eX = 0;
float eX_old = 0;
float eY_old = 0;

float Kpx1 = 0.00082;//   /1.6
float Kpy1 = 0.00082;
float Kdx1 = 0.001; //   /1.6
float Kdy1 = 0.001;
float Kpx2 = 130;
float Kpy2 = 130;
float Kdx2 = 40;
float Kdy2 = 40;
float Kix = 0.37;//   /1.6
float Kiy = 0.37;
char flag = 0;
char flagPad = 0;
float Ix = 0;
float Iy = 0; //integral of error
float gcomp_linkX = 0;
float gcomp_linkY = 0;
float gcomp_ballX = 0;
float gcomp_ballY = 0;
float m_l = 0.063; //mass link
float Cpos = 0.05;
float Cneg = -0.05;

float K1 = -1.08;
float K2 = -0.135;
float K3 = 135;
float K4 = 40;
float K5 = -1.08;
float K6 = -0.135;
float K7 = 135;
float K8 = 40;
float Ki1 = 0.03;
float Ki2 = 0.03;
float kscale = 1;

//system parameters
float q = 0.046/0.03;
float g = 9.81;
//mouse ball
float m = 0.035;   //kg
//medium steel ball
//float m = 0.175;
float r = 0.01;   //m
```

```
float J_m = 4.95e-7*900;   //kg*m^2
float J_p_x = 0.00654618 + 0.12537963*0.05;
float J_p_y =  0.00724051 + 0.12537963*0.05;

float K_b = 0.3602;   //Nm/A
float K_tau = 0.3602;
float B_tilde = 0.01178;    //Nms/rad
float tau_alpha = 0;
float tau_beta = 0;


int timecnt_traj = 0;
float s = 3;
unsigned char dogReset = 0;
char dataflag = 0;
char done = 0;

float theta1ref_filt = 0;
float theta1ref_filt_old = 0;
float theta1ref_old = 0;
float theta2ref_filt = 0;
float theta2ref_filt_old = 0;
float theta2ref_old = 0;
//float coeff1num = 0.038461538461538;  //20Hz analog cut off frequency
//float coeff2den = 0.923076923076923;
float coeff1num = 0.090909090909091;//50Hz analog cut off frequency
float coeff2den = -0.818181818181818;

#define NUM_SEND_QUEUES 120
#define MAX_SEND_LENGTH 1600
#define MAX_VAR_NUM 10

#pragma DATA_SECTION(matlabLock, ".my_vars")
float matlabLock = 0;



float matlabLockshadow = 0;

// UART 1 GLOBAL VARIABLES
int    UARTsensordatatimeouterror = 0; // Initialize timeout error count
int    UARTtransmissionerror = 0;              // Initialize transmission error
count

int UARTbeginnewdata = 0;
extern SEM_Obj SEM_UARTMessageReady;
int UARTdatacollect = 0;
char UARTMessageArray[101];
int UARTreceivelength = 0;

char Main_sendingarray = 0;  // Flag to Stop terminal prints when using matlab
commands
//  Only way to clear this flag

union mem_add {
       float f;
       long i;
       char c[2];
}memloc;
```

```
union ptrmem_add {
	float* f;
	long* i;
	char c[2];
}ptrmemloc;

long* Main_address[MAX_VAR_NUM];
float Main_value[MAX_VAR_NUM];
char Main_SendArray[128];
char Main_SendArray2[128];
float Main_tempf=0;

int Main_i = 0;
int Main_j = 0;
int Main_memcount = 0;

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

//the code below is used to transmit data to MATLAB

void EchoSerialData(int memcount,char *buffer) {


	char sendmsg[256];
	int i;

	sendmsg[0] = 0x2A; // *
	sendmsg[2] = '0'; // 0
	for (i=3;i<(memcount*8+3);i++) {
		sendmsg[i] = buffer[i-3];
	}
	sendmsg[1] = i;
	serial_send(&SerialA, sendmsg, i);


}




void matlab_serialRX(serial_t *s, char data) {
	if (!UARTbeginnewdata) {// Only TRUE if have not yet begun a message
		if (42 == (unsigned char)data) {// Check for start char

			UARTdatacollect = 0;		// amount of data collected in
message set to 0
			UARTbeginnewdata = 1;		// flag to indicate we are
collecting a message
			Main_memcount = 0;
			Main_i = 0;
		}
	} else {	// Filling data
		if (0 == UARTdatacollect){
			UARTreceivelength = ((int)data)-1; // set receive length to
value of char after start char
			UARTdatacollect++;
		}else if (UARTdatacollect < UARTreceivelength){
			UARTMessageArray[UARTdatacollect-1] = (char) data;
			// If sending out float value(s), save input memory locations
and values at those addresses
```

```
                if (('0' == UARTMessageArray[0]) &&  (UARTdatacollect > 1)){

                        if (Main_i == 0) {
                                ptrmemloc.c[1] = ((UARTMessageArray[UARTdatacol-
lect-1] & 0xFF) << 8);
                        }
                        if (Main_i == 1) {
                                ptrmemloc.c[1] |= (UARTMessageArray[UARTdatacol-
lect-1] & 0xFF);
                        }
                        if (Main_i == 2) {
                                ptrmemloc.c[0] = ((UARTMessageArray[UARTdatacol-
lect-1] & 0xFF) << 8);
                        }
                        if (3 == Main_i){
                                ptrmemloc.c[0] |= (UARTMessageArray[UARTdatacol-
lect-1] & 0xFF);

                                Main_address[Main_memcount]=ptrmemloc.i;
                                Main_value[Main_memcount]=*ptrmemloc.f;

                                Main_i = 0;
                                Main_memcount++;
                        }else{
                                Main_i++;
                        }
                }
                UARTdatacollect++;
            }
            if (UARTdatacollect == UARTreceivelength){  // If input receive
length is reached
                UARTbeginnewdata = 0;     // Reset the flag
                UARTdatacollect = 0;      // Reset the number of chars col-
lected

                // Case '0' : Sending data in endian format (big-endian ad-
dress, big-endian value)
                if ('0' == UARTMessageArray[0]){
                        for (Main_i = 0;Main_i<Main_memcount;Main_i++){
                                ptrmemloc.i=Main_address[Main_i];
                                Main_SendArray[0+8*Main_i]=((ptrmem-
loc.c[1]>>8)&0xFF);
                                Main_SendArray[1+8*Main_i]=ptrmemloc.c[1]&0xFF;
                                Main_SendArray[2+8*Main_i]=((ptrmem-
loc.c[0]>>8)&0xFF);
                                Main_SendArray[3+8*Main_i]=ptrmemloc.c[0]&0xFF;
                                memloc.f=Main_value[Main_i];
                                Main_SendArray[4+8*Main_i]=((mem-
loc.c[1]>>8)&0xFF);
                                Main_SendArray[5+8*Main_i]=memloc.c[1]&0xFF;
                                Main_SendArray[6+8*Main_i]=((mem-
loc.c[0]>>8)&0xFF);
                                Main_SendArray[7+8*Main_i]=memloc.c[0]&0xFF;
                        }
                        EchoSerialData(Main_memcount,Main_SendArray);// Append
header information to send data and transmit
                        // Case '1' : Writing float value to memory address
(big-endian received address / value)
                }else if ('1' == UARTMessageArray[0]){
```

```
                        for (Main_i = 0; Main_i < (UARTreceivelength -
2)/8;Main_i++){

                                ptrmemloc.c[1] = ((UARTMessageAr-
ray[1+8*Main_i]&0xFF)<<8);
                                ptrmemloc.c[1] |= (UARTMessageAr-
ray[2+8*Main_i]&0xFF);
                                ptrmemloc.c[0] = ((UARTMessageAr-
ray[3+8*Main_i]&0xFF)<<8);
                                ptrmemloc.c[0] |= (UARTMessageAr-
ray[4+8*Main_i]&0xFF);

                                memloc.c[1] = ((UARTMessageAr-
ray[5+8*Main_i]&0xFF)<<8);
                                memloc.c[1] |= (UARTMessageAr-
ray[6+8*Main_i]&0xFF);
                                memloc.c[0] = ((UARTMessageAr-
ray[7+8*Main_i]&0xFF)<<8);
                                memloc.c[0] |= (UARTMessageAr-
ray[8+8*Main_i]&0xFF);

                                *ptrmemloc.i = memloc.i;


                        }

                        matlabLockshadow = matlabLock;
                        // Case '2' : Sending array data in following format
[char 1,char2,char3,...]
                        // [*,3+input length of array,3 (code for array receiv-
ing in Matlab),...
                        //           array(0) chars in little-endian, ... , ar-
ray(memcount) chars in little-endian]
                    }else if ('2' == UARTMessageArray[0]){
                        Main_sendingarray = 1;
                        matlabLock = 1.0;
                        matlabLockshadow = matlabLock;
                        memloc.c[1] = NULL;
                        memloc.c[0] = ((UARTMessageArray[5]&0xFF)<<8);
                        memloc.c[0] |= (UARTMessageArray[6]&0xFF);
                        Main_memcount = memloc.i;
                        ptrmemloc.c[1] = ((UARTMessageArray[1]&0xFF)<<8);
                        ptrmemloc.c[1] |= (UARTMessageArray[2]&0xFF);
                        ptrmemloc.c[0] = ((UARTMessageArray[3]&0xFF)<<8);
                        ptrmemloc.c[0] |= (UARTMessageArray[4]&0xFF);
                        Main_SendArray[0]='*';
                        Main_SendArray[1]=3+Main_memcount;
                        Main_SendArray[2]='3';

                        serial_send(&SerialA, Main_SendArray, 3);

                        for (Main_i = 0; Main_i < Main_memcount;Main_i++){
                            Main_tempf = *ptrmemloc.f;
                            memloc.f = Main_tempf;
                            Main_SendArray2[0+Main_j*4] =  (memloc.c[0]&0xFF);
                            Main_SendArray2[1+Main_j*4] =  ((mem-
loc.c[0]>>8)&0xFF);
                            Main_SendArray2[2+Main_j*4] =  (memloc.c[1]&0xFF);
                            Main_SendArray2[3+Main_j*4] =  ((mem-
loc.c[1]>>8)&0xFF);
```

```
                                memloc.c[1] = ptrmemloc.c[1];
                                memloc.c[0] = ptrmemloc.c[0];
                                memloc.i+=2;  // was plus 4
                                ptrmemloc.c[1]=memloc.c[1];
                                ptrmemloc.c[0]=memloc.c[0];
                                Main_j++;
                                if (32 == Main_j){
                                      memcpy(Main_SendArray,Main_SendArray2,128);
                                      serial_send(&SerialA, Main_SendArray, 128);
                                      Main_j = 0;
                                }
                          }
                          if (Main_j != 0){
                                serial_send(&SerialA, Main_SendArray2, (Main_mem-
count%32)*4);
                                Main_j = 0;
                          }
                          Main_sendingarray = 0;
                          // Case '3' : Write float value to memory address (big-
endian received address,
                          //           little-endian received value)
                    }else if ('3' == UARTMessageArray[0]){
                          for (Main_i = 0; Main_i < (UARTreceivelength -
2)/8;Main_i++){

                                ptrmemloc.c[1] = ((UARTMessageAr-
ray[1+8*Main_i]&0xFF)<<8);
                                ptrmemloc.c[1] |= (UARTMessageAr-
ray[2+8*Main_i]&0xFF);
                                ptrmemloc.c[0] = ((UARTMessageAr-
ray[3+8*Main_i]&0xFF)<<8);
                                ptrmemloc.c[0] |= (UARTMessageAr-
ray[4+8*Main_i]&0xFF);

                                memloc.c[1] = ((UARTMessageAr-
ray[8+8*Main_i]&0xFF)<<8);
                                memloc.c[1] |= (UARTMessageAr-
ray[7+8*Main_i]&0xFF);
                                memloc.c[0] = ((UARTMessageAr-
ray[6+8*Main_i]&0xFF)<<8);
                                memloc.c[0] |= (UARTMessageAr-
ray[5+8*Main_i]&0xFF);

                                *ptrmemloc.i = memloc.i;

                          }

                          matlabLockshadow = matlabLock;
                    }
              }
        }
}

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


//nonlinear control law as mentioned in report

void nl_control(float *tau_alpha, float *tau_beta, float x, float y, float alpha,
float beta,
```

```
            float dx, float dy, float dalpha, float dbeta) {
      x = x/1000;
      y = y/1000;
      float J_b = 2/5*m*r*r;
      float J_y_tilde = J_p_y + J_b + m*y*y;
      float J_x_tilde = J_p_x + J_b + m*x*x;
      float a1 = -(K1*x+K2*dx+K3*alpha+K4*dalpha+Ki1*Iy)*kscale;
      float a2 = -(K5*y+K6*dy+K7*beta+K8*dbeta+Ki2*Ix)*kscale;


      *tau_alpha = (q*q*(J_m + J_y_tilde/q/q)*a1 + 2*m*x*y*a2 +
                   (B_tilde*q*q+2*m*x*dx)*dalpha + (m*dx*y+m*x*dy)*dbeta +
                   m*g*x*cos(alpha))/q;

      *tau_beta = (q*q*(J_m + J_x_tilde/q/q)*a2 + 2*m*x*y*a1 +
                   (B_tilde*q*q+2*m*y*dy)*dbeta + (m*dx*y+m*x*dy)*dalpha +
                   m*g*y*cos(beta))/q;
}

//extern float v1;
//extern float v2;


void main(void)
{

      int i=0;
      //  Init and zero encoders
      init_EQEP(&enc1, EQEP1, 12000, 1, 0.0);
      init_EQEP(&enc2, EQEP2, 12000, 1, 0.0);
      EQep1Regs.QPOSCNT = 0;
      EQep2Regs.QPOSCNT = 0;

      // Initialize PWMs
      init_PWM(EPWM1);
      init_PWM(EPWM2);
      init_PWM_AS_RCSERVO(EPWM3B);  // sets up EPWM3A and EPWM3B for RCservo
      init_PWM_AS_RCSERVO(EPWM4B); // sets up EPWM4A and EPWM4B for RCservo
      init_PWM_AS_RCSERVO(EPWM5);  // sets up EPWM5A for RCservo

      // System initializations
      pre_init();

      // initialize serial port A to 115200 baud
      init_serial(&SerialA,115200,matlab_serialRX);
      // initialize serial port B to 57600 baud
      init_serial(&SerialB,57600,NULL);
      // initialize serial port C to 19200 baud
      init_serial(&SerialC,19200,NULL);

      EALLOW;  // set up LED GPIOs
      GpioCtrlRegs.GPAMUX2.bit.GPIO30 = 0;
      GpioDataRegs.GPACLEAR.bit.GPIO30 = 1;
      GpioCtrlRegs.GPADIR.bit.GPIO30 = 1;
      GpioCtrlRegs.GPAMUX2.bit.GPIO31 = 0;
      GpioDataRegs.GPACLEAR.bit.GPIO31 = 1;
      GpioCtrlRegs.GPADIR.bit.GPIO31 = 1;
      GpioCtrlRegs.GPBMUX1.bit.GPIO34 = 0;
      GpioDataRegs.GPBSET.bit.GPIO34 = 1;
      GpioCtrlRegs.GPBDIR.bit.GPIO34 = 1;
```

```
       // set up GPIO3 for amp enable or disable
       GpioCtrlRegs.GPAMUX1.bit.GPIO3 = 0;
       GpioDataRegs.GPACLEAR.bit.GPIO3 = 1;
       GpioCtrlRegs.GPADIR.bit.GPIO3 = 1;

       EDIS;

       //    init_dma_mcbsp( (Uint32)&TXBuff.darray[0],(Uint32)&RXBuff.darray[0]);
       //    InitMcbspbGpio();
       //    InitMcbspb();
       //    InitMcbspb32bit();
       init_SPI();
       InitI2CGpio();
       Init_i2c();

       // Add your inits here
       // Configure ADC (see SPRU060B)
       AdcRegs.ADCMAXCONV.all = 7;       // Convert 8 channels in SEQ1 (0-4)
       AdcRegs.ADCTRL3.bit.SMODE_SEL = 1;     // Set up simultaneous conversion
mode
       AdcRegs.ADCTRL1.bit.SEQ_CASC = 1;      // Cascaded mode
       AdcRegs.ADCCHSELSEQ1.bit.CONV00 = 0; // Setup ADCINA0 as 1st SEQ conv.
       AdcRegs.ADCCHSELSEQ1.bit.CONV01 = 1; // Setup ADCINA1 as 2nd SEQ conv.
       AdcRegs.ADCCHSELSEQ1.bit.CONV02 = 2; // Setup ADCINA2 as 3rd  SEQ conv.
       AdcRegs.ADCCHSELSEQ1.bit.CONV03 = 3; // Setup ADCINA3 as 4th  SEQ conv.
       AdcRegs.ADCCHSELSEQ2.bit.CONV04 = 4; // Setup ADCINA4 as 5th conversion
       AdcRegs.ADCCHSELSEQ2.bit.CONV05 = 5; // Setup ADCINA5 as 6th conversion
       AdcRegs.ADCCHSELSEQ2.bit.CONV06 = 6; // Setup ADCINA6 as 7th conversion
       AdcRegs.ADCCHSELSEQ2.bit.CONV07 = 7; // Setup ADCINA7 as 8th conversion

       AdcRegs.ADCTRL2.bit.INT_ENA_SEQ1 = 1;  // Enable SEQ1 interrupt (every EOS)
--pp. 2-5 (bit 11)
       AdcRegs.ADCTRL1.bit.ACQ_PS = 4;        // Current guess at the sample hold
timing --sect. 1.4
       AdcRegs.ADCTRL3.bit.ADCCLKPS = 9;      // Current guess at the ADCCLK value
4.16MHz --sect. 1.4.1
       AdcRegs.ADCTRL1.bit.CONT_RUN = 0;
       AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1;         // Reset SEQ1 see pp 2-5
       AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1;       // Clear INT SEQ1 bit see pp 2-14

       EALLOW;

       GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 0;
       GpioCtrlRegs.GPAMUX1.bit.GPIO3 = 0;
       GpioCtrlRegs.GPAMUX1.bit.GPIO4 = 0;
       GpioCtrlRegs.GPAMUX1.bit.GPIO5 = 0;
       GpioCtrlRegs.GPADIR.bit.GPIO1 = 1;
       GpioCtrlRegs.GPADIR.bit.GPIO3 = 0;
       GpioCtrlRegs.GPADIR.bit.GPIO4 = 1;
       GpioCtrlRegs.GPADIR.bit.GPIO5 = 0;
       GpioDataRegs.GPASET.bit.GPIO1 = 1;
       GpioDataRegs.GPACLEAR.bit.GPIO4 = 1;

       EDIS;
       //-----------------------------------------------------------
       PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;   // Acknowledge interrupt to PIE
       PieCtrlRegs.PIEIER1.bit.INTx6 = 1;

       IFR &= ~M_INT1;  // Make sure no ints pending.
```

```
	IER |= M_INT1; // Enable CPU Interrupt 1 for ADC


	for (i=0;i<700;i++){
		xarray[i] = 0;
		yarray[i] = 0;
	}


	// Finalize inits and start DMA/McBSP
	post_init();

	EnableDog();

}


void start_dataCollection(void) {

	// Start SPI
	AdcRegs.ADCTRL2.bit.SOC_SEQ1 = 1;
}

void adc_cisr(void) {

	//take raw ADC samples

	raw_adc_A0 = AdcMirror.ADCRESULT0;     // ADC A0 - Gyro_Y
	raw_adc_B0 = AdcMirror.ADCRESULT1;     // ADC B0 - External ADC Ch4 (no pro-
tection circuit)
	raw_adc_A1 = AdcMirror.ADCRESULT2;     // ADC A1 - Gyro_4Y
	raw_adc_B1 = AdcMirror.ADCRESULT3;     // ADC B1 - External ADC Ch1
	raw_adc_A2 = AdcMirror.ADCRESULT4;     // ADC A2 -  Gyro_4X
	raw_adc_B2 = AdcMirror.ADCRESULT5;     // ADC B2 - External ADC Ch2
	raw_adc_A3 = AdcMirror.ADCRESULT6;     // ADC A3 - Gyro_X
	raw_adc_B3 = AdcMirror.ADCRESULT7;     // ADC B3 - External ADC Ch3
	raw_adc_A4 = AdcMirror.ADCRESULT8;     // ADC A4 - Analog IR1
	raw_adc_B4 = AdcMirror.ADCRESULT9;     // ADC B4 - USONIC1
	raw_adc_A5 = AdcMirror.ADCRESULT10;    // ADC A5 -  Analog IR2
	raw_adc_B5 = AdcMirror.ADCRESULT11;    // ADC B5 - USONIC2
	raw_adc_A6 = AdcMirror.ADCRESULT12;    // ADC A6 - Analog IR3
	raw_adc_B6 = AdcMirror.ADCRESULT13;    // ADC B6 - SV1 (Daughter Card / No
Connection)
	raw_adc_A7 = AdcMirror.ADCRESULT14;    // ADC A7 - Analog IR4
	raw_adc_B7 = AdcMirror.ADCRESULT15;    // ADC B7 - SV1 (Daughter Card / No
Connection)

	SWI_post(&SWI_control);
	// Reinitialize for next ADC sequence - see SPRU060B
	AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1;         // Reset SEQ1 see pp 2-5
	AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1;       // Clear INT SEQ1 bit see pp 2-14
	//-----------------------------------------------------------
	PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;   // Acknowledge interrupt to PIE
}

void control(void) {

	if (CheckDog()==1) {
		dogReset = 60;
	}
```

```
//     if (timeint < 2000*s) {
//           Yref = 55;
//           Xref = 55 + 15*(timeint)/2000/s;
//     } else if ((timeint >= 2000*s) && (timeint < 4000*s)) {
//           Xref = 70;
//           Yref = 55 + 15*(timeint - 2000*s)/2000/s;
//     } else if ((timeint >= 4000*s) && (timeint < 8000)) {
//           Yref = 70;
//           Xref = 70 - 30*(timeint - 4000*s)/4000/s;
//     } else if((timeint >= 8000*s) && (timeint < 10000*s)) {
//           Xref = 40;
//           Yref = 70 - 30*(timeint - 8000*s)/2000/s;
//     } else if ((timeint >= 10000*s) && (timeint < 14000*s)) {
//           Xref = 40 + 30*(timeint - 10000*s)/4000/s;
//           Yref = 40;
//     } else if ((timeint >= 14000*s) && (timeint < 16000*s)) {
//           Xref = 70;
//           Yref = 40 + 15*(timeint - 14000*s)/2000/s;
//     } else if ((timeint >= 16000*s) && (timeint < 18000*s)) {
//           Yref = 55;
//           Xref = 70 - 15*(timeint - 16000*s)/2000/s;
//     } else {
//           Xref = 55;
//           Yref = 55;
//     }
//


//     if ((timeint >= 3000*s) && (timeint < 8000*s) ) {
//           Yref = 55;
//           Xref = 55 + 15*(timeint - 3000*s)/5000/s;
//     } else {
//           Yref = 55;
//           Xref = 70;
//     }

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

//Touchpad readings

switch (flagPad) { //set up Pins for samping X position
case 0:
      flagPad = 1;
      EALLOW;
      GpioCtrlRegs.GPADIR.bit.GPIO1 = 0;
      GpioCtrlRegs.GPADIR.bit.GPIO3 = 1;
      GpioCtrlRegs.GPADIR.bit.GPIO4 = 0;
      GpioCtrlRegs.GPADIR.bit.GPIO5 = 1;
      GpioCtrlRegs.GPADIR.bit.GPIO4 = 0;

      GpioDataRegs.GPACLEAR.bit.GPIO3 = 1;
      GpioDataRegs.GPASET.bit.GPIO5 = 1;
      EDIS;
      break;
case 1:
      flagPad = 2; // sample X position and convert to mm
      //Xpos = (4095 - raw_adc_A0)/40;
      Xpos = (2047 - raw_adc_A0)/2047.0*82.5; //for nl controller
      break;
```

```
case 2:
	flagPad = 3; // set up pins to sample Y position
	EALLOW;
	GpioCtrlRegs.GPADIR.bit.GPIO1 = 1;
	GpioCtrlRegs.GPADIR.bit.GPIO3 = 0;
	GpioCtrlRegs.GPADIR.bit.GPIO4 = 1;
	GpioCtrlRegs.GPADIR.bit.GPIO5 = 0;
	GpioDataRegs.GPASET.bit.GPIO1 = 1;
	GpioDataRegs.GPACLEAR.bit.GPIO4 = 1;
	EDIS;
	break;
case 3:
	flagPad = 0; //sample Y position
	Ypos =(4095 - raw_adc_A1)/40.0;
	//Ypos = (2047 - raw_adc_A1)/2047*51.5
	Ypos = (2047 - raw_adc_A1)/2047.0*51.5; //for nl controller

	if(fabsf(Ypos - Ypos_old) > 10) {
		Ypos = Ypos_old;
	}

	if(fabsf(Xpos - Xpos_old) > 10) {
		Xpos = Xpos_old;
	}


	Yvel = (Ypos - Ypos_old)*250;
	Xvel = (Xpos - Xpos_old)*250;
	Yvel = (Yvel + Yvel_old1 + Yvel_old2)/3;//moving IIR filter
	Xvel = (Xvel + Xvel_old1 + Xvel_old2)/3;
	Yvel_old2 = Yvel_old1;
	Yvel_old1 = Yvel;
	Xvel_old2 = Xvel_old1;
	Xvel_old1 = Xvel;

	Xpos_old = Xpos;
	Ypos_old = Ypos;
	eY = Yref - Ypos;
	eX = Xref - Xpos;

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


	if (fabsf(u2) > 3)  { //take care of integral wind-up
		Ix = 0.99*Ix;
	} else if (fabsf(eX) > 1) {
		Ix = Ix + (eX+eX_old)/2*0.004;
	} //else if(eX > 12) {
	//                  Ix = 0;
	//           }


	if(fabsf(u1) > 3)  {
		Iy = 0.99*Iy;
	} else if (fabsf(eY) > 1) {
		Iy = Iy + (eY+eY_old)/2*0.004;
	} //else if(eX > 12) {
	//                  Ix = 0;
	//           }
```

```
            eX_old = eX;
            eY_old = eY;
            /*******************/


            //Outer Control loop calculating the desired angle

            theta1ref = -(Kpy1*eY - Kdy1*Yvel);
            theta2ref = -(Kpx1*eX - Kdx1*Xvel);

            //filtering of reference angle using a digital filter with cutoff
frequency of analog 50Hz

            theta1ref_filt = coeff2den*theta1ref_filt_old + coeff1num*theta1ref +
coeff1num*theta1ref_old;

            theta1ref_filt_old = theta1ref_filt;
            theta1ref_old = theta1ref;

            theta2ref_filt = coeff2den*theta2ref_filt_old + coeff1num*theta2ref +
coeff1num*theta2ref_old;

            theta2ref_filt_old = theta2ref_filt;
            theta2ref_old = theta2ref;
            /********************/


            //Saturating the reference angle to prevent uncontrollable system re-
sponse and jerkyness

            if(theta1ref > 0.06) {   //0.04 works
                  theta1ref = 0.06;
            } else if (theta1ref < -0.06) {
                  theta1ref = -0.06;
            }

            if(theta2ref > 0.08) {     //0.05 works
                  theta2ref = 0.08;
            } else if (theta2ref < -0.08) {
                  theta2ref = -0.08;
            }
            /***********************/


            gcomp_linkX = g*m_l*0.03*cos(value_enc1);//gravity compensation for
links
            gcomp_linkY = g*m_l*0.03*cos(value_enc2);
            gcomp_ballX = g*Xpos/1000*m*cos(value_enc2);//same for ball
            gcomp_ballY = g*Ypos/1000*m*cos(value_enc1);

            //the overall control law representing the outer and inner loop plus
the compensational terms

                  u1 =Kpy2*(theta1ref-value_enc1) - Kdy2*enc1vel -
Kiy*Iy+g*Ypos/1000*m*cos(value_enc1)/q+g*m_l*0.03*cos(value_enc1) ; //gravity com-
pensation ball and link
                  u2 =Kpx2*(theta2ref-value_enc2) - Kdx2*enc2vel -
Kix*Ix+g*Xpos/1000*m*cos(value_enc2)/q+g*m_l*0.03*cos(value_enc2);
                  if(Xvel < 3) {
```

```
                        u1 =Kpy2*(theta1ref-value_enc1) - Kdy2*enc1vel -
Kiy*Iy+g*Ypos/1000*m*cos(value_enc1)/q+g*m_l*0.03*cos(value_enc1) ; //gravity com-
pensation ball and link
                        u2 =Kpx2*(theta2ref-value_enc2) - Kdx2*enc2vel -
Kix*Ix+g*Xpos/1000*m*cos(value_enc2)/q+g*m_l*0.03*cos(value_enc2) ;
                  } //uncomment the u1 and u2 terms if nonlinear controller below
is used

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


            //data sampling
            if ((dataflag >2) && !done) {
                  //yarray[arrayindex] = Ypos;
                  yarray[arrayindex] = u1*0.03962;
                  //xarray[arrayindex] = Xpos;
                  xarray[arrayindex] = u2*0.03962;
                  dataflag = 0;
                  if(timeint > 2000) {
                        arrayindex++;
                  }
            }

//          if((timeint > 2000) && !done) {
//                dataflag++;
//                Xref = -38;
//                Yref = 8;
//          }

            Xref = cos(1.2/1000*timeint)*20-5;
            Yref = sin(1.2/1000*timeint)*5;

            if (arrayindex == 500) {
                  //arrayindex = 0;
                  done = 1;
            }
            /********************/


            //friction compensation for both static and viscous friction
            if (enc1vel> 0.0) {
                  u1 = u1 + B_tilde*enc1vel + Cpos;
            }
            else {
                  u1 = u1 + B_tilde*enc1vel+ Cneg;
            }

            if (enc2vel> 0.0) {
                  u2 = u2 + B_tilde*enc2vel + Cpos;
            }
            else {
                  u2 = u2 + B_tilde*enc2vel + Cneg;
            }
            /*****************************/

            //nonlinear control law for the state feedback approach (uncomment
this and comment out the mentioned things above if used

            //          nl_control(&tau_alpha, &tau_beta, Xpos, Ypos,
value_enc1, value_enc2, Xvel, Yvel, enc1vel, enc2vel);
```

```
            //          u2 = tau_alpha/0.03926;
            //          u1 = tau_beta/0.03926;
            //          if (u1> 7)  u1 = 7;
            //          if (u1<-7)  u1 = -7;
            //          if (u2> 7)  u2 = 7;
            //          if (u2<-7)  u2 = -7;

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

            // Final check to make sure within range
            if (u1> 10)  u1 = 10.0;
            if (u1<-10)  u1 = -10.0;
            if (u2> 10)  u2 = 10.0;
            if (u2<-10)  u2 = -10.0;

            // Send PWM command to motors
            PWM_out(EPWM1,u1);
            PWM_out(EPWM2,u2);

            break;
      default:
            break;
      }

      value_enc1 = EQEP_read(&enc1);//sample encoder reading
      value_enc2 = EQEP_read(&enc2);



      enc1vel = (value_enc1 - value_enc1_old)*1000; //calculating velocity of an-
gles
      enc2vel = (value_enc2 - value_enc2_old)*1000;



      value_enc1_old = value_enc1;
      value_enc2_old = value_enc2;

      // Add your code here in between updateData and sendData

      // Friction compensation (alternative)
      //    if (v1> 0.0) {
      //          u1 = u1 + Vpos*v1 + Cpos;
      //    }
      //    else {
      //          u1 = u1 + Vneg*v1 + Cneg;
      //    }
      //
      //    if (v2> 0.0) {
      //          u2 = u2 + Vpos*v2 + Cpos;
      //    }
      //    else {
      //          u2 = u2 + Vneg*v2 + Cneg;
      //    }



      if ((dogReset == 0) && ((timeint%500)==0)) {
            GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;
      } else if (((timeint%50)==0) && (dogReset>0)) {
```

```
            GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;
            dogReset--;
      }


      //    if ((timeint%200)==0) {
      //          serial_printf(&SerialA, "Hello %d\n\r",(int)(timeint/500));
      //
      //    }

      timeint++;

      // Service the WatchDog
      ServiceDog();

}
```

8.3 MATLAB Code

MATLAB code was written for both determining the ideal link setup and for setting up the Simulink models.

8.3.1 Link Simulation

The linkagesystem.m file was used to determine the ideal link setup by providing graphs of the effective transmission ratio between motor and plate.

Linkagesystem.m

```
%Analysis of mechanical linkage system
syms beta, syms gamma, syms theta, syms delta, syms a, syms b
syms c, syms hp, syms ht, syms hm, syms dm, syms dp


hp = ht + sin(theta)*dp;                          %height of the connection
point (CP)
c = sqrt(((hp-hm)^2+(dp*cos(theta)-dm)^2));       %direct distance from motor
to CP
delta = asin((hp-hm)/c);
beta = delta - acos((a^2+c^2-b^2)/(2*a*c));
gamma = beta + (pi-acos((a^2+b^2-c^2)/(2*a*b)));

w =  103.4/2;
l = 164.4/2;
a = 20;
b = 70;

dp = 20;
dm = dp - a;

ht = 160;
hm = ht - b;

dif = diff(beta,theta);
theta = 0;
```

```
%%
% figure(3)


b_0 = b;
a_0 = a;
dp_0 = dp;

test = [0:0.01:pi/8];


%test plots for varying b
for ii=1:3:9


%     subplot(3,3,ii)
    a = a_0 + 10/3*(ii-1);
%     b_0 = b_0 + 10*(ii-1);
%     dm = dp - a;

%     str = sprintf('b=%f',b);

%     syms theta
%     dif_s = subs(dif);
%     plot(test,subs(dif_s,test))
%     grid on
%     title(str);
      for jj=1:3

      figure(1)

      subplot(3,3,ii+jj-1)
      dp = dp_0 + 10*(jj-1);
      dm = dp - a;
      hm = ht - b;
      str = sprintf('a=%.1f,b=%.1f,dp=%.1f',a,b,dp);

      syms theta
      thetamax(ii+jj-1)= fsolve(@(theta)(sqrt((ht + sin(theta)*dp-
hm)^2+(dp*cos(theta)-dm)^2)-a-b),0.2);
      dif_s = subs(dif);

      plot(test,subs(dif_s,test))
      ylabel('q')
      xlabel('alpha')
      grid on
      title(str);


    figure(8)
    theta = 0;
    subplot(3,3,ii+jj-1)



    beta_s = subs(beta);
    gamma_s = subs(gamma);
    line([-cos(theta)*l cos(theta)*l],[ht-sin(theta)*l ht+sin(theta)*l])
```

```
    line([0,0],[0,ht])
    line([dm dm+cos(beta_s)*a dm+cos(beta_s)*a+cos(gamma_s)*b],[hm
hm+sin(beta_s)*a hm+sin(beta_s)*a+sin(gamma_s)*b])
    title(str);
       end


end
%%
%testplots for varying a
% for ii=1:9
%
%     figure(1)
%     subplot(3,3,ii)
%     a = a_0 + 10*(ii-1);
%     dm = dp - a;
%     str = sprintf('a=%f',a);
%
%     syms theta
%     dif_s = subs(dif);
%     plot(test,subs(dif_s,test))
%     grid on
%     title(str);
%



%
% for ii=1:9
%     figure(4)
%     a = 30;
%     b = 80;
%     dp = 60;
%     dm = dp - a;
%     ht = 200;
%     hm = ht - b;
%     theta = pi/6/8*(ii-1);
%     subplot(3,3,ii)
%     beta_s = subs(beta);
%     gamma_s = subs(gamma);
%     line([-cos(theta)*l cos(theta)*l],[ht-sin(theta)*l ht+sin(theta)*l])
%     line([0,0],[0,ht])
%     line([dm dm+cos(beta_s)*a dm+cos(beta_s)*a+cos(gamma_s)*b],[hm
hm+sin(beta_s)*a hm+sin(beta_s)*a+sin(gamma_s)*b])
%     title(str);
% end
```

The maxLinkLength.m file is a helper function called within the other function.

maxLinkLength.m

```
function  fun = maxLinkLength(hp,hm,dp,dm,a,b,theta)

fun = sqrt(((hp-hm)^2+(dp*cos(theta)-dm)^2))-a-b;
end
```

8.3.2 System Simulation

This code from the systemmodel.m file is required to set up the parameters and calculate the gains for the subsequent Simulink analysis.

Systemmodel.m

```
%System Analysis - Simulink

R = 3.9;       %Armature Resistance
L = 12e-6;      %Armature Inductance
Jm = 4.95e-7 *900;       %Moment of Inertia
Bm = 1e-8;       %Viscous Friction
Kt = 0.3602;      %Torque constant
Kb = Kt;      %Back EMF constant
Ka = 1;      %Amplifier constant
m = 0.04;       %Mass of ball
Ip = 0.00714618+0.05^2*0.12537963 ;  %moment of inertia of plate
r = 0.01; %radius of ball
Ib = 2/5*m*r^2; %2/5*m; %moment of inertia of ball
B_tilde = 0.01178; %overall damping constant


g = 9.81;    %Gravity constant
q = 1.6;       %Transmission ratio

%rlocus(fsystem);
%%
system = tf([7/5*g*Kp1*Kp2],[1 Kd2 Kp2 7/5*g*Kp2*Kd1]);
Galpha = feedback(feedback(tf(1,[1 0]),Kd2)*tf(1,[1 0])*Kp2,1);
Gxvel = feedback(Galpha*-7/5*g*tf(1,[1 0]),Kd1,+1);
Gx = Gxvel*Kp1*tf(1,[1 0]);
G = feedback(Gxvel*Kp1*tf(1,[1,0]),1,+1);


%%
%System analysis of system with angle as inputs and applied linearization
%as specified in report
A=[0 1 0 0; 0 0 0 0; 0 0 0 1;0 0 0 0];
B=[0 0; -5/7*g 0;0 0;0 -5/7*g];
C = [1 0 0 0;0 0 1 0];
D=[0 0;0 0];

rank(ctrb(A,B)); %check if controllable
K = place(A,B,[-2.3 -2.3 -2.31 -2.311]);

N= inv(C*inv(-(A-B*K))*B); %scale reference input for zero steady state er-
ror
Nx = N(1,1);
Ny = N(2,2);
sys = ss((A-B*K),B*N,C,D);
t=0:0.01:10;
% lsim(sys, [ones(1001,1)*0.04,ones(1001,1)*0.02], 0:0.01:10);


%%2D

%run this section for FullStateFeedback2dNoCancellation model without inte-
grator
```

```
%%
%For pole placement, all poles but 2  will be placed
%in the far left in order to place the remaining two ones accordingly. This
%is desired due to the subsequent behaviour of a second order system.
OS=0; %Overshoot
Ts=1.12;   %Settling Time
zeta = sqrt((log(OS)^2)/(pi^2+log(OS)^2));
wn=4/(Ts*zeta);
% Re=-wn*zeta;
Re = -1/Ts;
Im=wn*sqrt(zeta^2-1); %=0
p1=Re+Im;
p2=Re-Im;
x_d = 0.04;
y_d = 0.02;
A=zeros(8,8);
A(1,2) = 1;A(2,3)=-7/5*g; A(3,4)=1;A(5,6)=1; A(6,7)=-7/5*g; A(7,8) =1;
B=zeros(8,2);
B(4,1)=q;B(8,2)=q;
C=zeros(2,8);
C(1,1)=1;C(2,5)=1;
D=0;
sys = ss(A,B,C,D);
figure(5)
t=0:0.01:10;
rank(ctrb(A,B));
f1 =1; %just a scaling factor for testing purposes
p=[f1*Re 18*f1*Re 20*f1*Re 21*f1*Re f1*Re 18*f1*Re 20*f1*Re 21*f1*Re];

K = place(A,B,p);
Kp1=K(1,1);Kd1=K(1,2);Kp2=K(1,3);Kd2=K(1,4);
N= inv(C*inv(-(A-B*K))*B);
Nx = N(1,1);
Ny = N(2,2);


%run the section below for FullStateFeedback2DwithPI model with integrators
included and for
%simple PID model
%%
A1=[A zeros(8,2); C zeros(2,2)]; %extended system matrices for introducing
the integrator states
B1=[B;zeros(2,2)];


Ts = 1.12;
f1 = 1; %just a scaling factor for testing purposes
Re = -1/Ts;
p1=[f1*Re 18*f1*Re 20*f1*Re 21*f1*Re f1*Re 18*f1*Re 20*f1*Re 21*f1*Re
15*f1*Re 15*f1*Re ]
K1 = place(A1,B1,p1)
Kp1=K1(1,1);Kd1=K1(1,2);Kp2=K1(1,3);Kd2=K1(1,4); Ki=K1(1,9);



%%
%run this for  FullStateFeedback2D_simplePIDseparateSystems model
x_d = 0.04;
y_d = 0.02;
%To convert from CodeComposer to here: gains for position feedback: CC
```

```
%gains times  1000 - for angle feedback and integrator: CC gains times
0.03962
Kp1=K1(1,1);Kd1=K1(1,2);Kp2=K1(1,3);Kd2=K1(1,4); Ki=K1(1,9);
Kp1 = -0.1; Kd1 = -0.3; Kp2 = 5.5; Kd2 = 1.18; Ki = -2;
Kp1=K(1,1);Kd1=K(1,2);
```

The plotting.m file was used to plot the Simulink results.

Plotting.m

```
% plotting
% sim('FullStateFeedback2DNoCancellation',10);
sim('FullStateFeedback2DwithPI',10);
% sim('FullStateFeedback2D_simplePID',10);
% sim('FullStateFeedback2D_simplePIDseparateSystems',15);
tt=position.Time;

%plot ball position and reference input
figure(2)
clf
subplot(2,2,1)
xx=position.Data(:,1);
yy=position.Data(:,2);
rrx=reference.Data(:,1);
rry=reference.Data(:,2);
plot(tt,xx,tt,yy,tt,rrx,'--',tt,rry,'--');
hold on
grid on
xlabel('Time [s]')
ylabel('Position [m]')
legend('x','y','x_{ref}','y_{ref}')

%plot motor control torques
% figure(2)
subplot(2,2,2)
% clf
xx=torque.Data(:,1);
yy=torque.Data(:,2);
plot(tt,xx,tt,yy)
hold on
grid on
xlabel('Time [s]')
ylabel('Torque [Nm]')
legend('u_\alpha','u_\beta')

%plot plate angles
% figure(3)
subplot(2,2,3)
% clf
xx=angleplate.Data(:,1);
yy=angleplate.Data(:,2);
plot(tt,xx,tt,yy)
hold on
grid on
xlabel('Time [s]')
ylabel('Angle [rad]')
legend('\alpha','\beta')

%plot error
% figure(4)
subplot(2,2,4)
```

```
% clf
xx=error.Data(:,1);
yy=error.Data(:,2);
plot(tt,xx,tt,yy)
hold on
grid on
xlabel('Time [s]')
ylabel('Error [m]')
legend('x_{error}','y_{error}')
```

8.4 Simulink Models

Below are screenshots of all built Simulink models. The .slx files themselves are also provided separately. The models basically simulate the system based on the equations (4.7 – 4.10) while also including the control algorithms and adequate output blocks in order to view the results.

8.4.1 PID-Controller

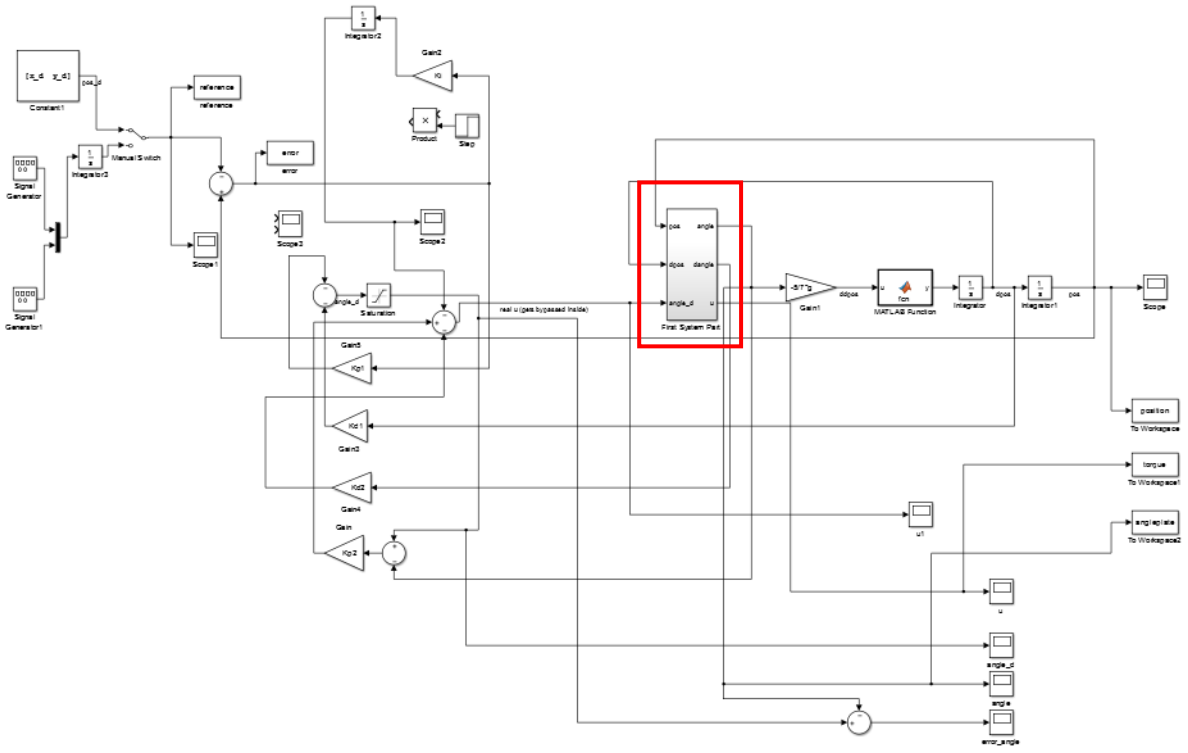


Figure 8.1: *Simulink* block diagram of PID-controller

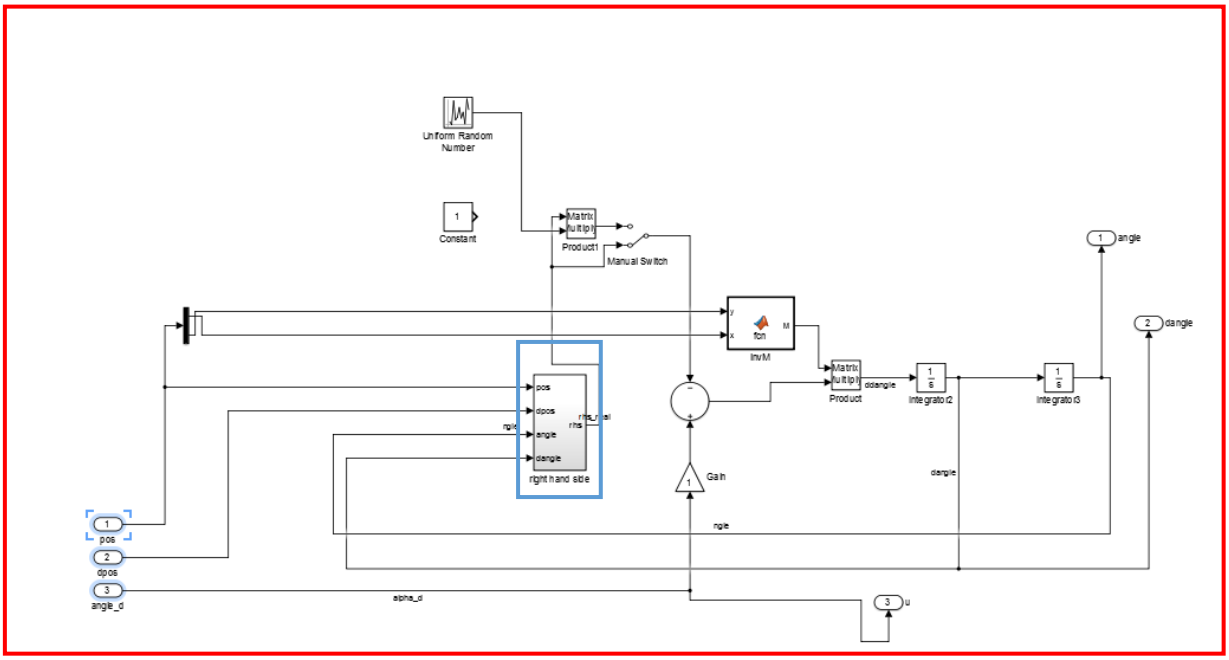


Figure 8.2: *Simulink* block diagram of the highlighted red sub-section in Figure 8.1

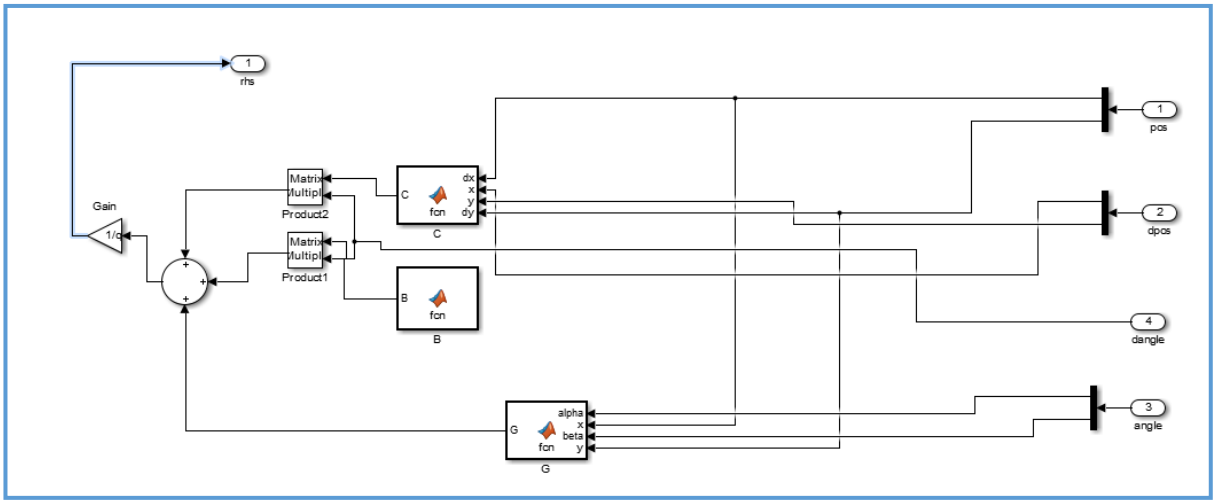


Figure 8.3: *Simulink* block diagram of the highlighted blue sub-section in Figure 8.1

8.4.2 Full State Feedback Controller

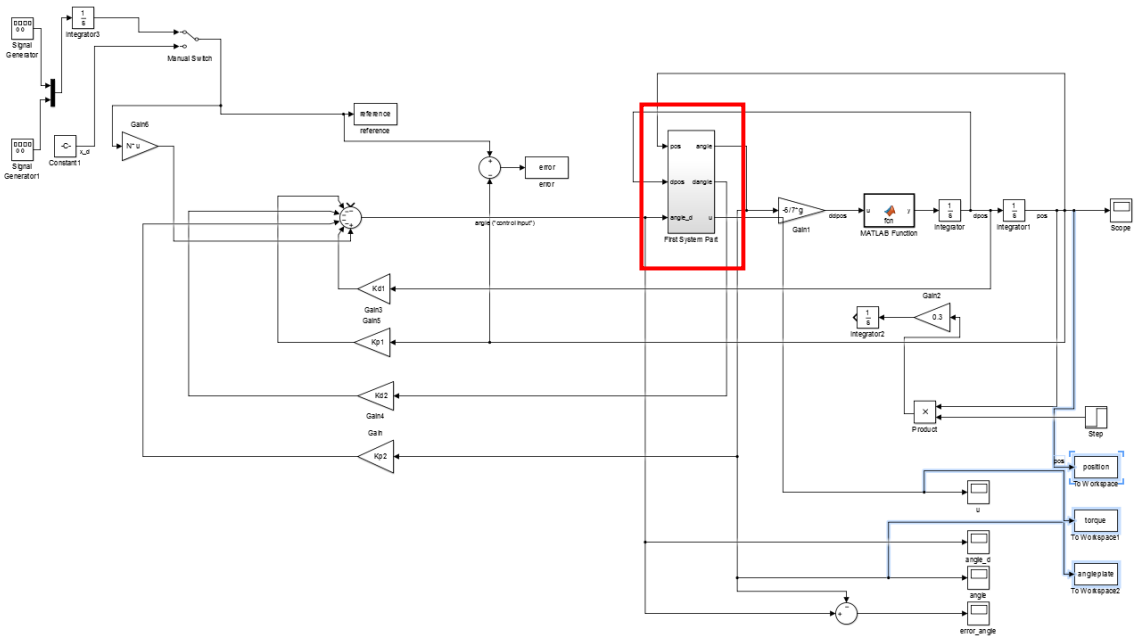


Figure 8.4: *Simulink* block diagram of the full state feedback controller

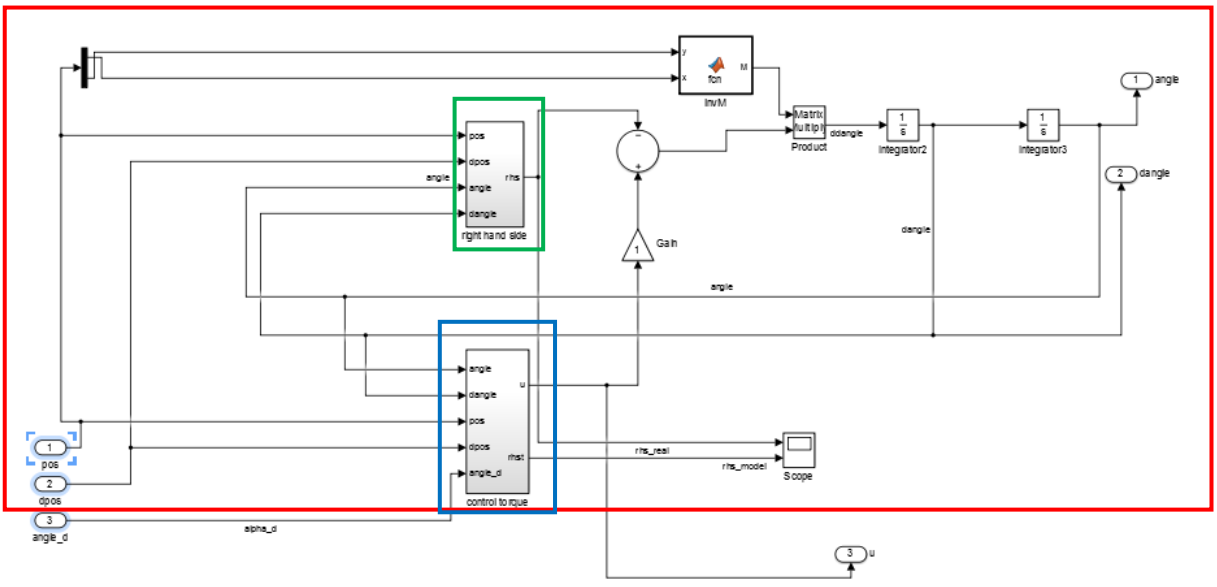


Figure 8.5: *Simulink* block diagram of the highlighted red sub-section in Figure 8.4

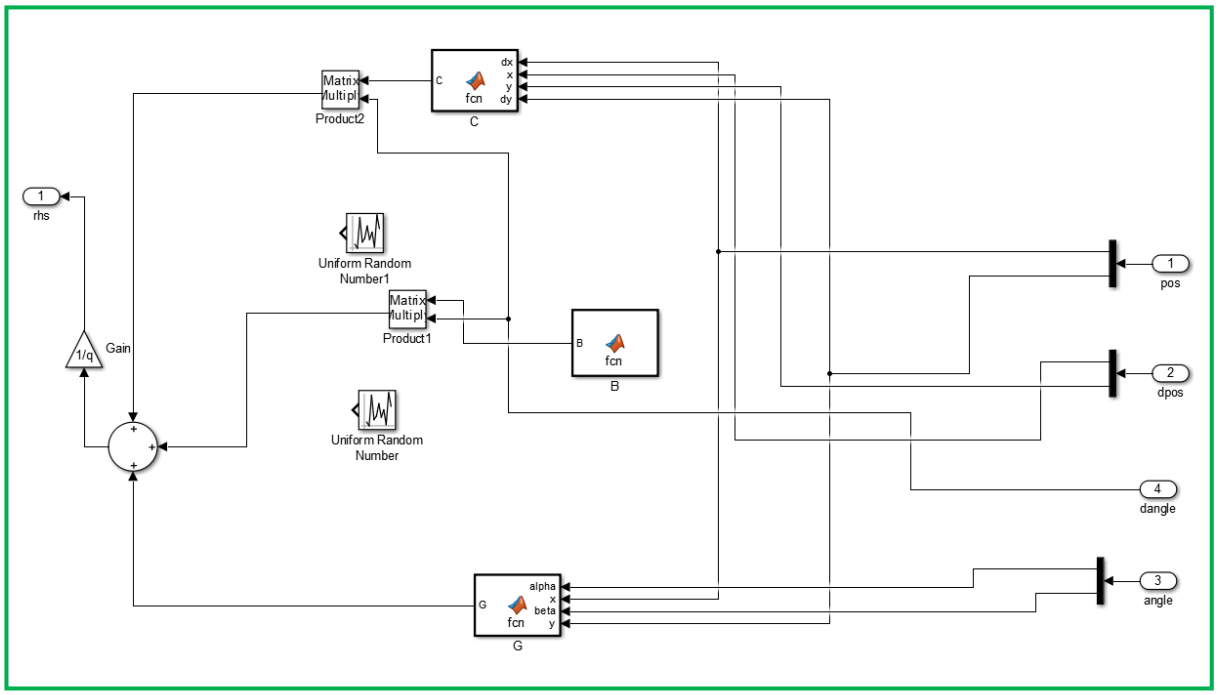


Figure 8.6: *Simulink* block diagram of the highlighted green sub-section in Figure 8.5

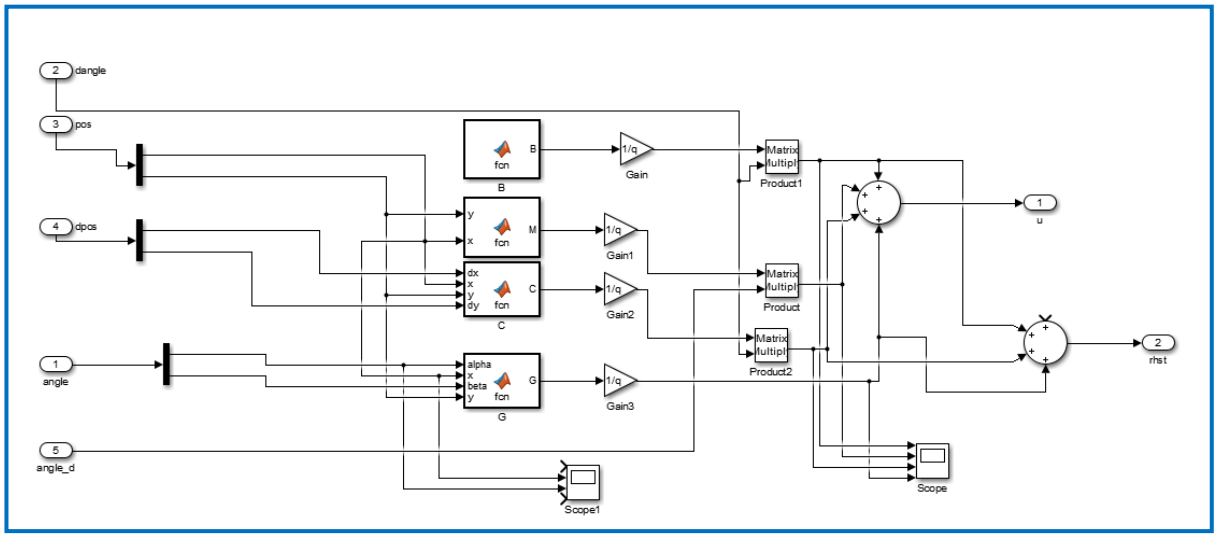


Figure 8.7: *Simulink* block diagram of the highlighted blue sub-section in Figure 8.5

8.4.3 PI-Control for Set Point Tracking

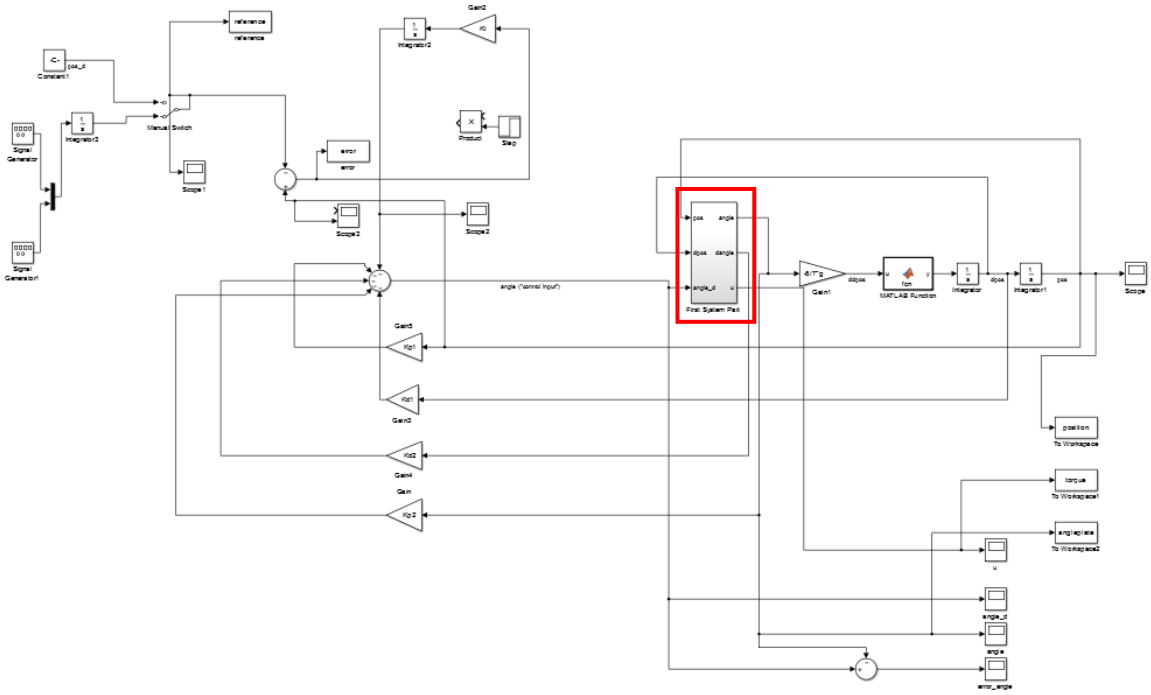


Figure 8.8: *Simulink* block diagram of the PI-controller for set point tracking

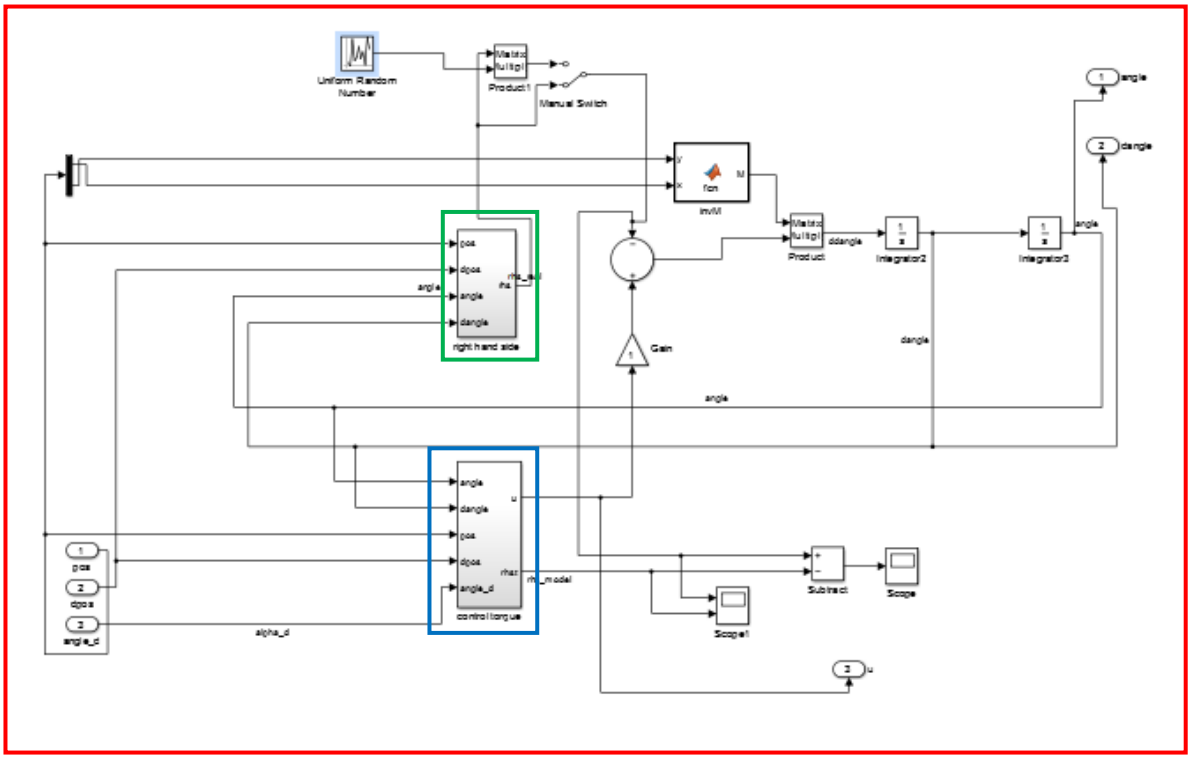


Figure 8.9: *Simulink* block diagram of the highlighted sub-section in Figure 8.8

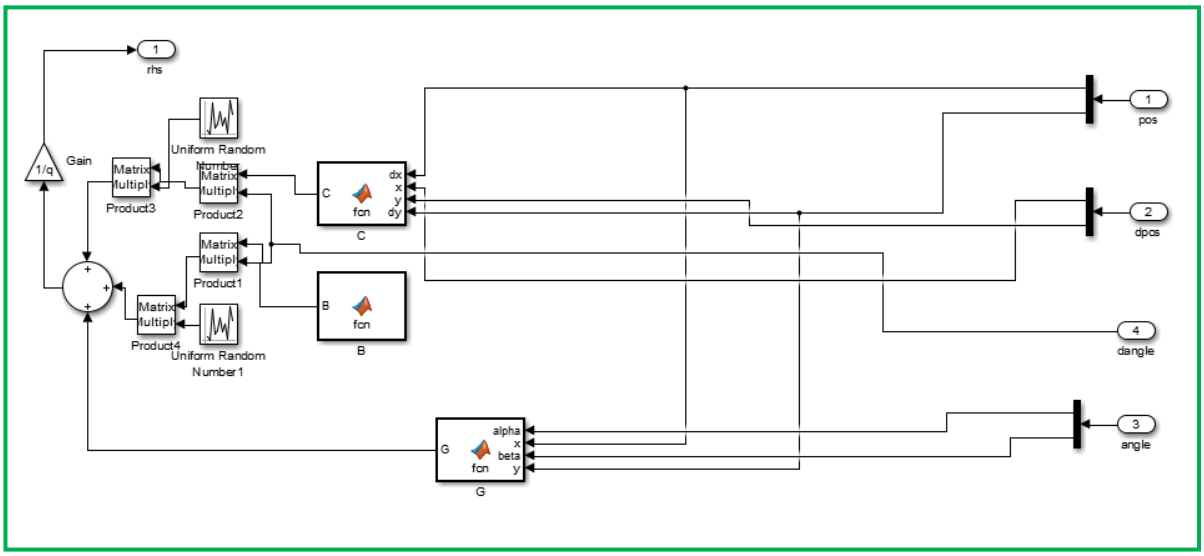


Figure 8.10: *Simulink* block diagram of the highlighted sub-section in Figure 8.8

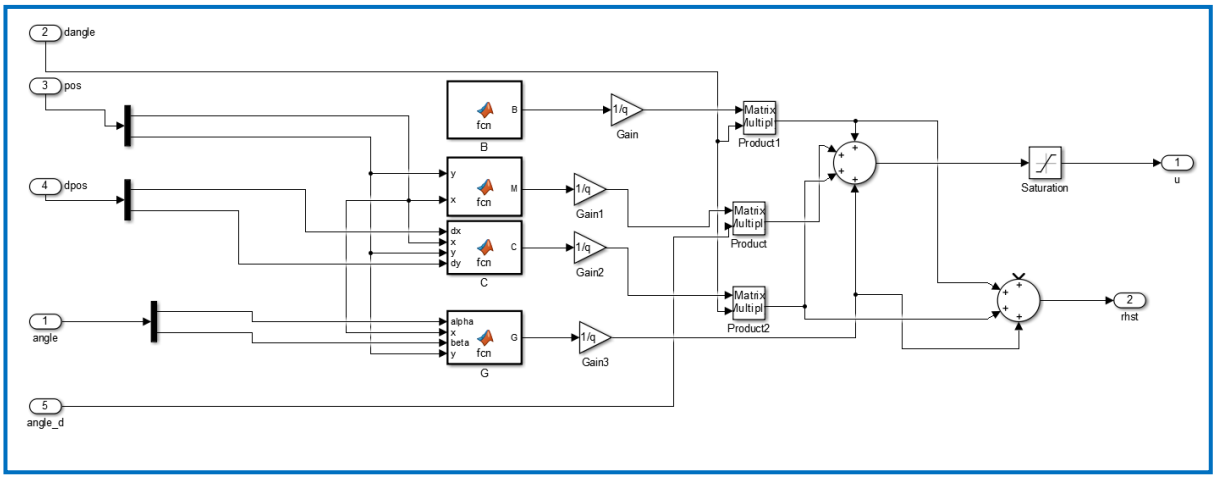


Figure 8.11 *Simulink* block diagram of the highlighted sub-section in Figure 8.8

List of Figures

List of Tables

List of Abbreviations

CAD	Computer-Aided Design
DC	Direct-Current
FIFO	First In First Out
ISR	Interrupt Service Routine
PID	Proportional-Integral-Derivative
MISO	Master-In-Slave-Out
MOSI	Master-Out-Slave-In
SPI	Serial Peripheral Interface
SS	Slave Select

List of Sources and Literature

[1] **S. Xin, S. Zheng-Shun, Z. Shi-Min**, "Fuzzy Control method for Ball and Plate System, "Computer Simulation, vol. 23, pp. 165–167, September

[2] **Dejun Lio, Yantao Tian, Huida Duan**, "Ball and Plate Control System based on sliding mode control with uncertain items observe compensation, "*2009 IEEE*

[3] **Mohammad Nokhbeh, Daniel Khashabi**, "Modelling and Control of Ball-Plate System, "*Amirkabir University of Technology, 2011*

[4] **Mark W. Spong, Seth Hutchinson, M. Vidyasagar**, „Independent Joint Control, " *Robot Modeling and Control, First Edition, p. 229ff.*

[5] **Mark W. Spong, Seth Hutchinson, M. Vidyasagar**, „Multivariable Control, "*Robot Modeling and Control, First Edition, p. 263ff.*

[6] **Michael A. Henson, Dale E. Seborg**, „Feedback Linearizing Control, " *Santa Barbara, Baton Rouge*