Chapter 1: Introduction

A robot is a device that performs specified tasks autonomously. The mention of robots to most conjures images of clumsy, mechanical, walking humanoids. Of course, the vast majority of robots are used in an industrial setting. These manufacturing robots execute simple, repetitive motions to facilitate their great variety of functions. The unifying feature among all of these industrial robots is the repeatable motion of its actuated arms.

1.1 Point-to-point Positioning

Robots are now commonplace in all sorts of manufacturing facilities. They serve a variety of tasks, including drilling, painting, and assembling parts. Certain automated functions like painting require that robotic arms follow a specified path. However, most motions required of industrial robots only stipulate a beginning and final position. This important subset of robotic motions is dubbed point-to-point motions.

In some applications, the precise positioning of robotic links in critical to the quality of the manufactured product. Boeing, for instance, requires that some of its drilling operations be within a very tight position tolerance on its aircraft. Without a robot to achieve precision positioning, elaborate rigs need to be constructed to hold the work piece in a calibrated position. This is not only more expensive than using a precision point-to-point robot, it is also significantly slower.
1.2 Traditional Controllers

The most commonly used controller for point-to-point positioning is the Proportional Controller. This control scheme essentially applies a control effort (often a voltage) to the actuators that is proportional to the difference between the current and target position, known as the error. Proportional control systems have been adapted to include integral and derivative terms as well, creating a Proportional-Integral-Derivative (PID) control system.

1.3 Friction

As the required positioning tolerance become tighter for a point-to-point robot, the effects of friction become more pronounced on the robot’s performance. In many cases, friction causes the robot arm or link to come to rest outside of the desired position tolerance. Measures can be taken to counteract the effects of friction, but most of these measures require a precise estimate of the friction force in order to be effective.

1.4 Pulse-Width Control

Pulse-Width Control (PWC) is a control scheme that was designed to deal with friction and poor estimates thereof. By applying discrete pulses instead of a continuous control effort, the arm of the robot is tapped into position. PWC has been shown to be very effective in overcoming problems that arise as a result of friction [1].
1.5 Adaptation and Tables

As successful as PWC is in overcoming friction, work is now being done to improve the accuracy of the applied pulses and thus reduce the amount of time necessary to achieve the desired tolerance. Currently, two methods of increasing the accuracy of pulses are being pursued. The first of which are lookup tables, which are tables of calibrated data which can be used to “look up” the necessary pulse widths. The second of which is adaptation, where the control system “fixes” the tables over time so they are more accurate. Both of these methods have been shown to improve the effectiveness of PWC.

1.6 Blanketing

Currently, the adaptation scheme being used only adapts tables in a localized manner. High resolution tables have very little ability to adapt properly as a result of this limitation. Eliminating this problem could significantly improve the speed of PWC positioning. The aim of this thesis is to develop a method of “blanketing” the current adaptation scheme so it is not as localized.
Chapter 2: Background Research

2.1 Traditional Controllers

The most common control system applied to industrial robotics is the Proportional Controller (P control). A proportional controller applies control effort proportional to the error in the system. The traditional P controller is often modified to include an integral or derivative term to decrease the error at steady state or improve the time required to converge on steady state. These appended terms to a P controller are reclassified as a Proportional-Integral-Derivative Controller (PID).

In the case of point-to-point position control, friction and flexibility can produce undesirable effects on performance. While the implications of flexibility in control systems is significant, the primary focus of this honors thesis is friction. Upon experimental investigation in one-dimensional cases, friction was determined to cause unsatisfactory steady state errors and instabilities. This inadequate performance is due in large part to the inherent unpredictability of friction. This property also hinders control methods such as friction compensation, a constant fourth term added to the PID formula that is designed to exactly cancel the friction force.

2.2 Pulse-Width Control

Yang and Tomizuka [1] proposed an alternative control law, dubbed Pulse-Width Control, that was designed to provide effective positioning with poor estimates of friction. Pulse-width control (PWC) applies discrete, square-wave pulses with varying
time-widths instead of a continuous, time-varying voltage to position the robotic link. The amplitude of the pulse is of sufficient amplitude to ensure that the link breaks the force of static friction. The pulse-width is calculated to accelerate the link and allow it to coast into the desired position. Equation (1) is derived from a simple block sliding on a surface with kinetic friction.

\[
t_p = \frac{2mf_c |e|}{f_p(f_p - f_c)} = K \sqrt{|e|} 
\]

The pulse-width, \( t_p \), is a function of the system error, \( e \), link mass, \( m \), Coulomb friction force, \( f_c \), and the force amplitude of the pulse, \( f_p \). These parameters can be simply lumped together as a constant gain, \( K \), with the pulse-width proportional to the root of the error.

If the friction estimate is too high or low, the link overshoots or undershoots the target respectively. Once the link comes to rest, a new pulse is calculated and applied and the process repeats. As a result of the greater-than-static-friction amplitude pulse, the link will always be moved by a subsequent pulse and never truly sticks with a steady state error. More importantly, the system performs well with poor estimates of friction.

### 2.3 Multiple Degrees of Freedom

Systems with multiple degrees of freedom lead to more complications. The intuitive expansion of PWC to multiple degree-of-freedom systems is to calibrate one gain for each degree of freedom. When dynamic coupling is present between the links, the dynamics of one link in motion affects the result of the pulse in other links. In
systems with significant coupling, simulations show a dramatic increase in convergence 
time in some cases and instability in others.

To incorporate coupling in the calibration, a look-up table was implemented 
instead of a single gain per link [10]. Tabular Pulse-Width Control (Tabular PWC), 
interpolates the required pulse-widths for each link from a set of tables with axes 
denoting the desired displacements for each link

Simulations of Tabular PWC are extremely effective in cases where two-gain 
PWC was not [10]. Similar to PWC in the one-dimensional case, Tabular PWC’s 
performance does not radically decline with a somewhat poorly calibrated table. 
Experimentation with the Parker robot showed superior performance to dual-gain PWC, 
even with low resolution tables (5 x 5).
Chapter 3: Literature Review

Traditionally, proportional control is used to position most robotic systems. To minimize steady-state error and decrease convergence time, integral and derivative terms are often added to the model. Proportional, integral, derivative (PID) control, however, can lead to large convergence times, unsatisfactory steady-state errors and instabilities. There are three traditional methods of altering PID control that reduce the effects of friction on performance: dithering, high-servo gains, and model-based friction compensation [11].

In a dithered system, a high-frequency signal is introduced in the control signal. This oscillating control effort, when analyzed in the pre-sliding friction model, actually reduces the coefficient of friction by as much as 66% [11]. The effect of the friction reduction reduces the steady-state error of a traditional PID controller. This method, however, can be a strain on the actuators as they must change direction with high frequency when the net velocity approaches zero.

Increasing servo gains is a very simple method of overcoming friction. The increased gains will provide a greater control effort at given errors that should overcome friction. However, once static friction is broken, the heightened control effort causes a large amount of overshoot which can lead to instability. In the best case, ramping up gains reduces the total steady state error at the risk of instability and large convergence times.
Friction-compensation is a model-based technique where friction is opposed by feed-forward cancellation. An estimated value of the friction force is simply added to the control effort and theoretically cancels the effects of friction. Such a technique requires an accurate estimate of the friction coefficient in order to be successful. High friction estimates can lead to long convergence times and instability in severe cases.

Pulse-Width Control in point-to-point positioning in robotic systems was originally investigated by Yang and Tomizuka [1] in 1988. This paper utilized a single mass model for the robotic link and established a PWC gain that is proportional to the square root of the error. Their controller implemented an adaptive algorithm to adjust the gain. Yang and Tomizuka showed that PWC causes the link to converge on the goal position.

Rathbun, Berg, and Buffinton investigated the effects of flexibility on PWC in 2002 [5]. A three degree-of-freedom gantry style robot was used for experimentation. However, traditional PWC yielded a limit cycle response when system flexibility was introduced. After examining the effects of very small pulses, it was found that a linearly decreased gain fits the data more closely than a square root. Rathbun, Berg, and Buffinton implemented a piecewise-linear gain to accommodate this effect. The gain adjustment eliminated the limit cycle response. Pongpunwattana performed a similar experiment with a one-degree of freedom, flexible, revolute jointed robot. Their results were similar to Rathbun, Berg, and Buffinton, noting that poor friction estimates may still yield a limit cycle.
Buffinton, Perkins, Beal, and Berg [11] in 2005 investigated multiple-link, coupled, robotic systems. A two-degree of freedom, revolute jointed robot was simulated in MATLAB to assess the effects of coupling. Using a lookup table that incorporates the effects of both pulses on both links, it was found that there was a significant improvement over the Yang and Tomizuka model. While the Yang and Tomizuka model could lead to limit cycles or instability when coupling is present, Tabular PWC shows stability and faster convergence time with multiple links. Furthermore, the implementation of a lookup table makes piecewise-linear gains unnecessary, as they are accounted for in table calibration. Perkins, Buffinton, and Berg [3] tested Tabular PWC using the robot developed by Parker [7]. Using a two-degree of freedom system, one prismatic and one translational link, Tabular PWC positioned the robot within an unprecedented tolerance of one motor encoder count.

Schwab [9] in 2006 developed a method of tabular adaptation for the pulse-width lookup tables. This method, dubbed Proportional Adaptation, was tested in simulation with an intentionally flawed look-up table. It was determined that Proportional Adaptation causes the table to converge to the correct table over time and decreases the total number of pulses required to achieve the target position.
Chapter 4: Models Utilized

4.1 One-Link Rigid Model

The one-link rigid mass model was originally utilized by Yang and Tomizuka [1]. This model simulates a block sliding along a surface with friction as shown in Figure 4.1.

\[ F - \mu mg = ma , \]  

with \( \mu = \mu_s \) when the block is at rest, and \( \mu = \mu_k \) when \( v > 0 \).

4.2 Two-Link Rigid Model with One Revolute and Prismatic Joint

The following model is used to simulate the behavior of the Parker robot [7], shown in Figure 4.2. The Parker robot was built to test the effect of flexibility on robotic motions, but has been recently used as an experimental test-bed for PWC.
Three rigid bodies A, B, and C, are used to model the mass distribution of the system. Body A rotates about a fixed vertical axis (the revolute joint), while Body B translates along an axis fixed in A (the prismatic joint). Body C is rigidly attached to one end of B. The coordinates $q_1$ and $q_2$ describe the configuration of joints. The coordinate $q_1$ measures the angle between a fixed line and a line fixed in A, as shown, while $q_2$ measures the translational distance between point P and the end of Body B that is attached to Body C. Actuators are connected to points O and P, and thus are points of input to the system. These parameters are visualized in figure 4.3
The masses of A, B, and C are denoted $m_A$, $m_B$, and $m_C$, respectively, and their moments of inertia about vertical lines passing through their mass centers are denoted $I_A$, $I_B$, and $I_C$, respectively. The parameters measured from the Parker experimental robot are given in Table 4.1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_A$</td>
<td>0.125 m</td>
</tr>
<tr>
<td>$L_T$</td>
<td>0.25 m</td>
</tr>
<tr>
<td>$L_P$</td>
<td>0.25 m</td>
</tr>
<tr>
<td>$L_C$</td>
<td>1 m</td>
</tr>
<tr>
<td>$m_A$</td>
<td>1 kg</td>
</tr>
<tr>
<td>$m_B$</td>
<td>1 kg</td>
</tr>
<tr>
<td>$m_C$</td>
<td>0.5 kg</td>
</tr>
<tr>
<td>$I_A$</td>
<td>0.05 kg-m²</td>
</tr>
<tr>
<td>$I_B$</td>
<td>0.1 kg-m²</td>
</tr>
<tr>
<td>$I_C$</td>
<td>0.01 kg-m²</td>
</tr>
</tbody>
</table>

Table 4.1: List of Two-Link Model Parameters
The dynamical equations of motion governing the system of Figure 1 are given below.

These equations were formulated using Kane’s method [6]:

\[
T_A -(m_A L_A^2 + I_A)\ddot{u}_1 - \{m_B[(L_p + q_2 - L/2)^2 + L_T^2] + I_B\} \ddot{u}_1 - \\
2m_B u_1 u_2 (L_p + q_2 - L/2) + m_B L_T \ddot{u}_2 - \{m_c[(L_p + q_2 + l_c)^2 + L_T^2] + I_c\} \ddot{u}_1 = 0,
\]

\[
2m_B u_1 u_2 (L_p + q_2 - L/2) + m_B L_T \ddot{u}_2 + \{m_c[(L_p + q_2 + L_c)^2 + L_T^2] + I_c\} \ddot{u}_1 = 0,
\]

\[
F_{A/B} + m_B L_T \ddot{u}_1 - m_B \ddot{u}_2 + m_B (L_p + q_2 - L/2) \dddot{u}_1 = 0,
\]

\[
m_c L_T \dddot{u}_1 - m_c \dddot{u}_2 + m_c (L_p + q_2 + L_c) \dddot{u}_1 = 0,
\]

where \(T_A\) is the torque applied to body A, \(F_{A/B}\) is the force applied to B at P by an actuator fixed in A, \(u_1\) and \(u_2\) are the generalized speeds of the links, and \(\ddot{u}_1\) and \(\ddot{u}_2\) are the generalized accelerations.
Chapter 5: Theory

5.1 Tabular Pulse-Width Control

Traditional Pulse-Width Control applies a pulse to the robotic link whose width is proportional to the square root of the position error. However, in many cases, the dynamics of a system do not follow the single mass model. Practical robots experience phenomena like backlash that are not accounted for in traditional PWC. Also, as shown by Rathbun, Berg and Buffinton [2], a linearly decreased gain is shown to more accurately yield the desired link displacement for very small motions. All of these factors that interfere with the standard, single-mass model give reason to pursue a non-model based method of determining pulse widths, namely, a look-up table.

5.2 One Degree-of-freedom Robots

The most elementary implementation of Tabular Pulse-Width Control arises in a system with a single link. Pulse-Width Control with such a system has a single input, the pulse-width, and a single output, the link displacement. Table 5.1 shows an example of a one-dimensional table for a one degree-of-freedom system. Note that the root of the displacement is used instead of actual displacement. The root of the error more accurately reflects the underlying single-mass model, theoretically making linear interpolation between points more accurate.
Table 5.1: Sample one-dimensional table for a one degree-of-freedom system

<table>
<thead>
<tr>
<th>Root Error (mm)$^{1/2}$</th>
<th>0.000</th>
<th>0.100</th>
<th>0.141</th>
<th>0.173</th>
<th>0.020</th>
<th>0.224</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pulse-Width (s)</td>
<td>0.000</td>
<td>0.046</td>
<td>0.085</td>
<td>0.121</td>
<td>0.159</td>
<td>0.186</td>
</tr>
</tbody>
</table>

Instead of calibrating a gain for the controller, a table is generated by applying calibration pulses to the system. Once the table is generated, desired pulse-widths are interpolated from the table. Complicated dynamics and system irregularities such as backlash do not affect the accuracy of a look-up table due its non-model-based nature.

5.3 Multiple Degree-of-freedom Robots

When extending PWC to multiple degrees of freedom, the simplest modification would be to add an additional one-dimensional table for each additional degree of freedom. However, this method does not take into account dynamic coupling from multiple links. To fully describe the coupling effects of the motion of each link on every other link for an N-link system in tabular form requires N tables of N-dimensions. For instance, a two degree-of-freedom robot requires two tables with two dimensions. As figure 5.1 shows, the axis values indicate the square root of the error for each link. The values in table one and two are the required pulse-widths for link one and two respectively to achieve the root displacement indicated on the axes.
5.4 “Forward” and “Reverse” Tables

The table setup in figure 5.1 is designed for calculating the desired pulse width using simple, linear interpolation. Given a set of error roots for the system, a two-dimensional interpolation is used on both tables to obtain the correct pulse-width values. However, practical concerns become apparent when attempting to generate these tables using an experimental system. For PWC, the pulse-width is an input argument and the resulting displacement of the link to which it is applied is an output argument. The tabular setup shown in figure 5.1, however, arranges the error roots on its axes (the table “input”) and the pulse widths as the table values (the table “output”). The difference is highlighted in equation 1 where $pw_1$ and $pw_2$ designate the pulse widths applied to links 1 and 2 and $disp_1$ and $disp_2$ are the displacements of links 1 and 2. Simply put, the calibration routine for the look-up table example in figure 5.1 requires that pulse widths
(an output parameter) be found to match the given displacements (an input parameter),
while the robot control system requires pulse-widths be an input parameter to obtain the
resulting displacements as output parameters.

\[
\begin{align*}
[\text{disp}_1, \text{disp}_2] &= \text{RobotControlSystem}(pwl, pw2) \\
[pwl, pw2] &= \text{CalibrationRoutine}(\text{disp}_1, \text{disp}_2)
\end{align*}
\]

**Equation 1: Functional representation of parametric incongruity between the
system dynamics and required calibration routine**

To accurately generate these tables without an accurate model for its behavior, calibration
pulses must be exhaustively applied until the correct combination of displacements
occurs. Figure 5.2 visually illustrates the difficulty and impracticality of generating a
table in this manner in a hypothetical two degree-of-freedom system.

![Figure 5.2: Visualization of exhaustive calibration method of look-up tables](image)

17
Since the arrangement of parameters in the robotic system is immutable, this issue gives reason to develop a new setup of table parameters that is more calibration friendly. A logical fix is to reverse the arrangement of the axis and table values, namely placing pulse widths as axis values and displacement roots as table values. Reversing the placement of the input and output variables on the tables allows for pulse-widths values to be input parameters in both the robot control system. Congruity ensues for the output displacements as well, as shown by functional representation in equation 2.

\[
\begin{align*}
[\text{disp}_1, \text{disp}_2] &= \text{RobotControlSystem}(\text{pw}_1, \text{pw}_2) \\
[\text{disp}_1, \text{disp}_2] &= \text{NewCalibrationRoutine}(\text{pw}_1, \text{pw}_2)
\end{align*}
\]

Equation 2: Functional representation of parametric congruity between the control system and new required calibration routine

Now that displacement roots are required as outputs from the calibration routine, only a single pulse needs to be applied of the specified set of pulse widths to calibrate a table point. This new table setup with a reversed parameter arrangement has been dubbed a “reverse” table. Consequently, the previously conceived table in figure 5.1 is henceforth referred to as a “forward” table. A graphical representation of a reverse table setup for a degree-of-freedom robot is shown below in figure 5.3.
5.5 Reverse Interpolation

While ease of calibration is a key advantage when using reverse tables, similar incompatibilities in parameters arise when attempting to interpolate pulse widths from a reverse table. Forward tables have the advantage of a simple interpolation scheme to extract pulse widths since interpolation takes axis values as input parameters (the desired displacements in the case of forward tables) and an interpolated table value as an output parameter (the required pulse width in the case of forward tables). Reverse tables, having switched the position of these parameters, are not compatible with traditional interpolation schemes. Once again, a graphical representation is shown to illustrate this lack of compatibility in figure 5.4.
Figure 5.4: Graphical representation of the mismatch between reverse tables and traditional interpolation schemes

The advantages of using a reverse table are significant enough to warrant investigation into a new interpolation scheme. A “reverse” interpolation method would accept table values as inputs and yield corresponding axis values. Conversely, traditional tabular interpolation schemes are referenced as “forward” interpolation for clarification.

A reverse interpolation scheme has been developed based on simple principles.

The two degree-of-freedom case is used in the subsequent reverse interpolation examples. A reverse interpolation scheme for a two degree-of-freedom robot must yield a set of two
pulse widths that give a desired set of displacements for each of the two robotic links. These two pulse widths are calculated in three steps.

The first two steps of the reverse interpolation process build arrays of pulse widths which each satisfy one of the two desired displacements. To build the first of these arrays, the desired displacement of link 1 is interpolated along each row of the table containing link 1 displacements (table 1). For each row of table 1 (a row is the direction of constant pw2 values), a pw2 value is interpolated that corresponds to the desired link 1 displacement value. These values of pw1 and pw2 are stored as a data point in a corresponding array, dubbed array 1. Figure 5.5 visually demonstrates this process for a sample two degree-of-freedom system.

A second array is then generated in a similar manner using the table containing link 2 displacements (table 2). Each column of table 2 (a column is the direction of constant pw1 values) is traversed and a pw2 value is interpolated for each column. The
resulting interpolated pulse-width combinations are stored in array 2.

![Diagram of interpolation process]

**Figure 5.5: Illustration of interpolation process to generate arrays**

The final step in the reverse interpolation process is the intersection of the two generated arrays. Array 1 contains sets of pulse widths that result in the desired displacement of link 1, and array 2 contains sets of pulses that result in the desired displacement of link 2. The intersection point of these two arrays yields a set of pulses that produces both desired displacements. Figure 5.6 shows a graphical representation of the array generation and intersection process.
Step 1: Interpolate displacement values for link 1 along each row of table 1, storing each set of interpolated values into array 1

Step 2: Interpolate displacement values for link 2 along each column of table 2, storing each set of interpolated values into array 2

Step 3: Plot arrays and calculate intercept to yield interpolated pulse widths

**Figure 5.6: Graphical representation of the “reverse” interpolation process for a two degree-of-freedom robot**

This reverse interpolation scheme has two immediate methods of implementation in Tabular PWC. One version of Tabular PWC is to generate a reverse table and use reverse-interpolation to calculate pulse-widths as needed instead of using a forward table.
An alternative method of implementing Tabular PWC requires generating a reverse table and using reverse-interpolation to populate a forward-table. Henceforth, this forward table is used as the controller’s look-up table as well as simple forward interpolation. Both methods allow for a simple method of table calibration as well as a viable system of interpolation.

5.6 Adaptation

A number of factors contribute to the need to adapt calibrations over time. Environmental variations during normal operations can adversely affect the accuracy of initial calibrations. These inaccuracies increase system convergence time and can lead to instabilities in extreme cases. Even in the absence of environmental variation, robots can have entirely different properties in different areas of their workspace.

5.6.1 Proportional Adaptation

Proportional Adaptation, developed by Schwab [9], was shown to be an effective method of adjusting the table to new parameters. The basic purpose of this adaptation scheme is to receive the result of an applied pulse and adjust the lookup table such that a subsequent interpolation yields the result of the applied pulse. In concept, if each table cell is changed to match the result of the experimental system, then the table itself will more closely model the system after many adaptations.

5.6.2 1D Proportional Adaptation
A one-dimensional table provides the simplest example of the method. The proportional adaptation algorithm (PAA) alters the closest points in the table surrounding the new data point. In one dimension, PAA alters the two table points adjacent to the new data point. In the following example, a pulse width of $t_e$ is applied to a one degree-of-freedom robot to produce a link displacement of $x_e$. The value of $t_e$ is bordered by $t_1$ and $t_2$ in the lookup table whose corresponding displacement values are $x_1$ and $x_2$. In figure 5.7, $\hat{x}_e$ is the actual displacement resulting from the pulse width $t_e$ and $\hat{x}_1$ and $\hat{x}_2$ are the adapted table values as a result of PAA. As is the goal of adaptation, a subsequent interpolation of the adapted points yields the experimentally attained value of $\hat{x}_e$.

Figure 5.7: Visual Representation of One-Dimensional Proportional Adaptation
Proportional Adaptation gets its name from the proportion in which the surrounding table points are changed. The actual values obtained from the interpolated table as a result of the adaptation is defined as follows:

\[
\hat{x}_e = \hat{x}_1 + \frac{\hat{x}_2 - \hat{x}_1}{t_2 - t_1} (t_e - t_1).
\]  (5)

The table values are updated as a function of the change of the interpolated value as well as their proximity to the interpolated value along the horizontal axis. Algebraically, the expressions for the updated table values are

\[
\hat{x}_1 = x_1 + C \left( \frac{t_2 - t_e}{t_2 - t_1} \right) [\hat{x}_e - x_e] \quad \text{and}
\]

\[
\hat{x}_2 = x_2 + C \left( \frac{t_2 - t_e}{t_2 - t_1} \right) [\hat{x}_e - x_e],
\]  (6)

where \( C \) is given by

\[
C = \frac{\hat{L}^2}{\hat{L}_1^2 + \hat{L}_2^2}.
\]  (8)

When computing \( C \), \( \hat{L} \) is the total distance between \( t_1 \) and \( t_2 \), \( \hat{L}_1 \) is the distance between \( t_1 \) and \( t_e \), and \( \hat{L}_2 \) is the distance between \( t_e \) and \( t_2 \).

### 5.6.3 2D Proportional Adaptation

Adapting a two-dimensional table with PAA is similar to the aforementioned one-dimensional adaptation. When adapting a forward table, as done by Schwab, figure 5.8 represents the parameters involved in adapting a two degree-of-freedom system.
Figure 5.8: Visual representation of a two-dimensional, forward table undergoing PAA

In this example, the subscripts \( t \) and \( r \) refer to a hypothetical translational and rotational links in a two DOF robot. It is also important to note that only half of the interpolation is occurring in this example since a two DOF system has two tables and the process is simply repeated for the other pulse-width table. A pulse of duration \( t_e \) is applied to one of the links. Displacements \( \hat{x}_{re} \) and \( \hat{x}_{te} \) result from the two pulses. Error values \( x_{r1}, x_{r2}, x_{t1}, \) and \( x_{t2} \) are along the table axes that surround the actual displacements, and \( t_{11}, t_{12}, t_{21}, \) and \( t_{22} \) are their corresponding pulse-widths. The pulse width corresponding to the actual position errors will therefore be different than the applied pulse width, and is designated as \( \hat{t}_e \). The pulse width \( \hat{t}_e \) can be found with the expression:

\[
\hat{t}_e = \frac{1}{\hat{A}} (\hat{A}_{22} t_{11} + \hat{A}_{21} t_{12} + \hat{A}_{12} t_{21} + \hat{A}_{11} t_{22}),
\]

where \( \hat{A} = \hat{A}_{22} + \hat{A}_{21} + \hat{A}_{12} + \hat{A}_{11} \), and the areas are defined by

\[
\hat{A}_{11} = \text{abs}[(x_{re} - x_{r1})(x_{te} - x_{t1})],
\]

\[
\text{(10)}
\]
In two dimensions, the constant $C$ is given by the expression:

$$C = \frac{\hat{A}^2}{\hat{A}_{11}^2 + \hat{A}_{12}^2 + \hat{A}_{21}^2 + \hat{A}_{22}^2}. \quad (14)$$

The values of $t_{11}$, $t_{12}$, $t_{21}$, and $t_{22}$ in this table are adapted such that a consequent interpolation of displacements $\hat{x}_{re}$ and $\hat{x}_{te}$ yields the pulse width, $\hat{t}_e$. The adaptation equations are given as:

$$\hat{t}_{11} = t_{11} + C \left( \frac{\hat{A}_{22}}{A} \right) [\hat{t}_e - \hat{t}_e], \quad (15)$$

$$\hat{t}_{12} = t_{12} + C \left( \frac{\hat{A}_{21}}{A} \right) [\hat{t}_e - \hat{t}_e], \quad (16)$$

$$\hat{t}_{21} = t_{21} + C \left( \frac{\hat{A}_{12}}{A} \right) [\hat{t}_e - \hat{t}_e], \quad (17)$$

$$\hat{t}_{22} = t_{22} + C \left( \frac{\hat{A}_{11}}{A} \right) [\hat{t}_e - \hat{t}_e]. \quad (18)$$

As mentioned previously, to complete the adaptation, the same procedure should be applied to the other table of pulse widths. Schwab [9] has shown that PAA is very effective when implemented in PWC in decreasing the number of pulses required to attain the desired position tolerance as well as causing an incorrect table to converge to a correct one.
5.7 Blanketing

A disadvantage of PAA is that it currently only affects one cell at a time. For tables with large resolutions, this inability to affect large regions can significantly slow the convergence of the table to new parameters. An alternative to the adaptation scheme to allow for the effect of a single adaptation to spread to adjacent cells would allow for larger resolution tables to converge more quickly. Figure 5.9 shows this proposed effect on a 2-D table and shows how the spreading of the adaptation can resemble a “blanket” being “stretched”. This “stretchy blanket” aesthetic is why the method is dubbed “blanketing”.

![Figure 5.9: A sample adaptation with and without a blanketing effect](image)

Several blanketing concepts were considered and tested with thought experiments. After some trial and error, it was determined that a method should be devised using simple precepts. These precepts, if adhered to, should yield precisely the desired distributed adaptive effect.
5.7.1 **Blanketing Precept 1**

The effect of the blanketing algorithm should be to affect table values closer to the initial adaptation to a greater degree than distant cells. As the distance between the table value experiencing the blanketing effect and the initially adapted cell increases, the effect on the blanketed cell should approach zero. The motivation for this precept is the notion of relevance of data. A data point closer to the adapted cell is considered more “relevant” than a point further away.

5.7.2 **Blanketing Precept 2**

A blanketed data point should be affected by some fraction of the change in the originally adapted cell. Should no adaptation occur in the adapted cell, then no change would propagate to the blanketed point. This precept is motivated by the scenario where a theoretically “perfect” table is put through a cycle of adaptation and blanketing. In such a case, no adaptation should occur and thus no blanketing effect should take effect either. While this precept may seem obvious, some potentially viable methods of blanketing failed this precept.

5.7.3 **Blanketing Precept 3**

The effect on the blanketed table value should always be in the same direction (sign) as the initial change creating by the initial adaptation. In a real robotic system, parameters like friction are likely to vary in a uniform direction. For instance, a loss of lubrication over time in a joint would require larger pulses to obtain the same
displacement. There is little motivation for an adaptation effect that pulls higher data points down and lower data points up. Without this precept, blanketed points essentially “gravitate” toward the new data point from both sides. Figure 5.10 illustrates how adapting blanketed points in the opposite direction of the original change can lead to regions of incorrect adaptation.

Blanketing Example without Precept 3

Blanketing Example with Precept 3

Figure 5.10: Example demonstrating the rationale for precept 3.

5.7.4 Blanketing Functions
Using these three precepts, a preliminary blanketing algorithm was developed. PAA is used to determine the initial change in the table, $\Delta y$. The blanketed adaptation is given by the following expression:

$$\Delta y_b = \Delta y \cdot B(\Delta x, \sigma) .$$

(19)

The value of the blanketed change is given by $\Delta y_b$, $B$ is a chosen blanketing function, $\Delta x$ is the distance between the blanketed point and the initially adapted point, and $\sigma$ is a constant denoting the width of the blanketing function. The blanketing function, $B(\Delta x, \sigma)$, must follow the first precept in that it must decay to zero as $\Delta x$ increases and approach a unity value when $\Delta x$ approaches zero. Two likely options for the blanketing function are an exponential decay and a Gaussian distribution. As shown in figure 5.11, both functions approach one as $\Delta x$ approaches zero and decay to zero as $|\Delta x|$ grows large.

Figure 5.11: Plot of sample blanketing functions
Ultimately, the Gaussian curve was selected for a number of reasons. The exponential decay would form a cusp in the table after each adaptation. The Gaussian “bell curve”, however, has the effect of diminishing marginal returns as the distance from zero increases. Such a property is compatible with an adaptation application because points close to the original adaptation are changed by a similar value. This change decreases with an increasing slope until the distance separating the points reaches $\sigma$, an inflection point. This inflection point has benefits as well as $\sigma$ actually indicates a property that has relevance to the function width. Conversely, $\sigma$ for the exponential decay function indicates when $B(\Delta x, \sigma)$ decays to $1/e$, a property that has little bearing or meaning to an adaptation scheme. The equation for this bell curve is given by:

$$B(\Delta x, \sigma) = e^{\frac{-\Delta x^2}{2\sigma^2}}.$$ (20)

When initially tested, this blanketing function showed favorable convergence properties. Upon closer investigation, this three precept blanketing resulted in some instability points in the adapted table. Figure 5.12 shows how two adaptations in close proximity can constructively interfere and create even larger table error.
5.7.5 Blanketing Precept 4

The fourth precept of blanketing stabilizes this interference by capping the maximum change allowable for blanketed points. A blanketed point can not be changed beyond a ceiling value set by the height of the original adaptation. This precept is similar to the behavior of a circus tent in which supports are being set up. This circus tent analogy is demonstrated in figure 5.13. Adding a second support post to the tent, analogous to a second adaptation to the table, does not cause any of the tent material to overshoot the second post.
Including the fourth precept into the blanketing model requires a capping function

\[ \Delta y_c = C(\Delta y, \Delta y_0) . \]

This capping function, \( C \), is defined by:

\[ \Delta y_c = 0 \text{ when } \text{sign}(\Delta y) \neq \text{sign}(\Delta y_0), \]  
\[ \Delta y_c = \Delta y \text{ when } \Delta y > \Delta y_0, \]  
\[ \Delta y_c = \Delta y_0 \text{ for all else}, \]
where $\Delta y_c$ is the capped change for the blanketed point, $\Delta y$ is the change of the originally adapted point and $\Delta y_0$ is the difference in height between the originally adapted point and the blanketed point. The blanketing and capping functions are then combined as follows:

$$\Delta y_b = B(C(\Delta y, \Delta y_0), \sigma).$$  \hspace{1cm} (24)

### 5.7.6 Blanketing in 2-D

Altering the blanketing equations for two dimensions requires two additional considerations. Both a composite distance and multiple $\sigma$ values need to be calculated. A composite distance in two-dimensions results from a simple Pythagorean analysis. Multiple $\sigma$ values are required because different axes may have different units or different behaviors that warrant a different blanketing width. Developing a formula for a composite $\sigma$ value, $\sigma_c$, would allow for the same blanketing formula to be used in multiple degree-of-freedom applications. This $\sigma_c$ value can be calculated by the following formula:

$$\sigma_c = \left( \frac{d_1}{d_c} \right)^2 \sigma_1 + \left( \frac{d_2}{d_c} \right)^2 \sigma_1 \sigma_r^2,$$

$$d_c = \sqrt{d_1^2 + (d_2 \sigma_r)^2},$$

$$\sigma_r = \frac{\sigma_1}{\sigma_2},$$

where $d_1$ and $d_2$ are the distances between the originally adapted point and the blanketed point on axes 1 and 2, and $\sigma_1$ and $\sigma_2$ are the blanketing width values for axes 1 and 2. These 2-D composite values, $\sigma_c$ and $d_c$, are substituted into the blanketing equation.
instead of \( \sigma \) and \( \Delta x \). The resulting blanketing effect of two successive adaptations is demonstrated visually in figure 5.15.

Figure 5.15: Two sample adaptations of an originally flat table
Chapter 6: Experimental Results

The eventual goal of blanketing is to reduce the convergence time of tabular adaptation and to make high resolution tables feasible to implement. While the ultimate objective is to test the blanketing algorithm in the PWC setting, initial tests are performed with a simple test function in one-dimension. This test should indicate the general effectiveness of the adaptation scheme for various parameters.

6.1 One-Dimensional, Non-PWC Tests

The blanketing scheme was tested at various table resolutions and $\sigma$ values. The key factor to monitor when examining the adaptation is the sum of the squares of the table error. This indicates the quality of the adaptation and measures how close the adapted tables approximate the true values. In particular, the characteristic values being examined are the steady state values of the table error and time constant of the convergence. The time constant indicates the number of adaptations required for the table error to traverse 63% of the difference between the initial and final value. The steady state value is the approximate value at which the table error has stopped decreasing. In these tests, the blanketing width is set to a percentage of the length of the table.

In these tests, random points are selected for adaptation along the 1-D table. While several table shapes were tested to determine the validity of the tested methods, one was used consistently for the statistical runs. All of the other functions used had very
similar results, however, this function, shown in figure 6.1, was chosen as a representative example.

![Test Function for 1-D Blanketing Simulations](image)

**Figure 6.1: Sample function used in 1-D blanket tests**

Many adaptations are made (seven times the resolution of the table) so that the PAA can reach steady state. In figure 6.2, PAA is shown along with blanketing with and without the capping function (precept 4).
Figure 6.2: Graph showing the performance of PAA, blanketing without capping, and blanketing with capping

This plot shows how blanketing without capping can become unstable. It is sufficient information to conclude that blanketing without capping should not be used. Also, blanketing with capping converges significantly faster than PAA. Eventually, around 300 adaptations, PAA has a lower table error than blanketing with capping.

Several more tests over more blanket widths are averaged together to obtain statistical data on convergence times and steady state error. Figure 6.3 shows the convergence time of the table error for various blanket widths.
There is a clear downward trend of the convergence time with these blanket width percentages (BWP). Also, time constant multiples of two or more also decrease until some threshold where they greatly increase. This rapid increase in the higher time constants indicates that the average value oscillates significantly and never converges within the required threshold. Steady-state error was also computed these BWP values as shown in figure 6.4.
Figure 6.4: Steady State Error versus BWP

Mean-steady-state error (SSE) has a very strong increasing trend with increasing BWP. Also, the two different table resolutions yielded nearly similar results. It is evident that there is a clear tradeoff between SSE and convergence time with various values of BWP.

6.2 2-D Blanketing with PWC

Based on the 1-D results, the blanketing algorithm with capping accelerates the convergence of tables for certain range of values of BWP. Using the 2-D expansion of blanketing, the same blanketing algorithm will now be applied to a PWC application. In these tests, 20 random motions are attempted using pulse-width control. Pulses are
applied for each motion until the links are within tolerance, adapting the table after each pulse. Figure 6.5 shows the average table error and number of pulses

Figure 6.5: Average table error and pulse counts over 20 random motions

Figure 6.5 shows a case where the BWP assigned (10%) is slightly too large, since the table error and number of applied pulses slowly climb over time. Figure 6.6 shows a second case where a BWP of 5% is used instead of a 10% BWP.
Figure 6.6: Average table error and pulse counts for a 5% BWP

At a 5% BWP, the error of the tables now monotonically decreases. Now, further reducing the value of BWP to 2.5% yields the results in figure 6.7.
Once again, the table error monotonically decreases, but the amount to which the table error converges is greater than the 5% BWP before. When further reducing the BWP value to zero, the results in figure 6.8 were found.
Figure 6.8: Average table error and pulse counts for a 0% BWP, approximating PAA

Once again, the table error decreased monotonically, but again converged upon a larger value than the previous BWP value of 2.5%.
Chapter 7: Conclusions

Blanketing adaptations with the proposed blanketing algorithm has been shown to be effective in decreasing the convergence time of the adapting tables. Depending on the value of BWP, the table error can monotonically decrease or can result in instability of the table. As BWP increases, convergence times and time constants tend to decrease. Consequently, this decrease in convergence time most always results in an increase in the mean SSE of the table.

Since there is a clear tradeoff in increasing the blanketing effect, BWP, it stands to reason that a variable or “smart” BWP could be applied. When the table is undergoing major changes, a larger BWP would be helpful to more quickly converge on a more correct table. However, as the mean steady state error is being approached, the BWP could be decreased, thus decreasing the mean steady state error. Such a method would yield the best qualities of both large and small values of BWP.

Implementing a variable BWP, while simple in concept, runs into immediate practical issues. For one, in theory, BWP should decrease as table error decreases. In reality, there is no way that table error can be measured since no perfect table can be known in an experimental system. One potential method is to measure the changes in the table over time and use that as an indicator of the table quality (tables that are changing quickly are less correct than ones changing slowly).

As expected, the trends in table error value very closely follow trends in pulse counts. This is a good result since the premise of tabular adaptation in PWC is based on
the fact that smaller table errors will result in fewer applied pulses. The similarities in
trends between errors and pulse counts give further credence to the idea that blanketing
can help decrease the number of pulses applied by PWC over time.
Chapter 8: Future Work

8.1 Blanketing

Potential avenues of future work exist inside and outside of blanketed adaptation. For one, methods of implementing variable BWP’s could be implemented in simulation to optimize convergence time and mean steady-state error. Also, the proposed method of blanketed adaptation has yet to be tested outside of simulation. While a perfect table is simple to compare to in simulation, results in a real system are only the number of pulses to achieve the correct tolerance. If comparable results are measured using the Parker [7] robot, then more credence would be given to blanketing.

8.2 Adapting Reverse Tables

Adapting a reverse-table has been an elusive goal in PWC research. It has been found that reverse-interpolation is not compatible with PAA, which is designed to adapt a table that is “forward-interpolated”. Alternative methods of adapting reverse-tables have been developed and tested with minimal success. The possibility exists to simply generate a forward table from the reverse table, adapt it using PAA or some blanketed form of it, then reverse-interpolate the reverse table. While this is very computationally inefficient, this method could be performed over larger time intervals.

8.3 Flexibility
While friction has been the subject of discussion in this thesis, link flexibility can be a great cause of concern for PWC. Since PWC applies square-wave pulses, and square-waves have the tendency to excite many modes of vibration, tight tolerance positioning can be adversely affected. Since it is required for PWC to wait for steady-state after each pulse is applied, vibrations can severely lengthen this time and increase the total convergence time.

8.4 Perfect Controller Analysis

All robotic systems, whether by friction, flexibility, backlash, imperfect signal amplification, or feedback quantization error, fundamentally have variability in the system that cannot be perfectly predicted. Thus, if a pulse were sent to the motors on any given robot multiple times, the resulting displacement would be slightly different each pulse. By statistically modeling the resulting variability for a given pulse, the performance of a theoretically “perfect”, pulse-width controller can be modeled in simulation. Using this statistical model, the performance of the real system can be compared to a theoretical best performance. This can act as a base-line test for all forms of Adaptive PWC.
Chapter 9: Bibliography


Chapter 10: Appendix

BLANKET_STANFORD_2D_STAT.m

%BLANKET_STANFORD_2D_STAT
%applies 2-D Blanketing Algorithm to Adaptation

% Application of Adaptive PWC to a two rigid link Stanford-like manipulator using
% the Euler integration method. Constant multiplier adaptive
% algorithm updates delta t’s in tables. Dynamics of the
% motors and transmissions are included.

% System parameters assigned based on Parker robot.

% Last revised on May 30, 2006.

clear all
close all
clc

num_samples = 5;

sigma_prop1 = 0.025;
sigma_prop2 = 0.025;

PLOTS_ON = 0;
BLANKET_ON = 1;
PAUSES_ON = 0;
EXTRA_PLOT_ON = 0;
zero_barrier_on = 1;

sigma1 = 2*0.0548*sigma_prop1;
sigma2 = 2*0.0173*sigma_prop2;

% Assign values to system constants.
LA = 0.1;    % Distance to A* (estimate as of 5/30/06)
LT = 0.125;  % Distance to prismatic joint
LP = 0.09375; % Distance to point P
L = 0.75;    % Length of link B
LC = 0;      % Distance to C*
mA = 2;      % Mass of link A (estimate as of 5/29/06)
mB = 0.695;  % Mass of link B
mC = 0;      % Mass of link C
IA = 0.042;  % Moment of inertia of link A
IB = 0.033;  % Moment of inertia of link B
IC = 0;      % Moment of inertia of link C
Kg1 = 60;    % Transmission ratio for rotational joint
Kg2 = 10; % Transmission ratio for translational joint
Jm = 0.0000367; % Motor moment of inertia
Jg1 = 0.00018; % Effective moment of inertia for rotational joint transmission
dsp = 0.0381; % Pitch in m/rad of translational joint transmission
Kt = 0.06778; % Motor torque constant
Ka = 2.2; % Amplifier gain in amps/volt

c = 1.5; % Coefficient of the actual output torque over estimated friction

V1 = 0.31665; % Voltage required to move revolute joint
V2 = 0.09535; % Voltage required to move prismatic joint

TcA = Kg1*Kt*Ka*V1; % Coulomb friction torque applied to link A
TA = c*TcA; % Magnitude of torque pulse applied to link A
TcAest = TcA; % Estimated Coulomb friction torque applied to link A
TsA = TcA; % Maximum static friction torque applied to link A

FcAB = 2*Kg2/dsp*Kt*Ka*V2; % Coulomb friction force between link A and link B
FAB = c*FcAB; % Magnitude of torque pulse applied to link B
FcABest = FcAB; % Estimated Coulomb friction force between link A and link B
FsAB = FcAB; % Maximum static friction torque between link A and link B

% Formats the parameter vector to be passed to called functions
parameter_vector = [LA, LT, LP, L, LC, mA, mB, mC, IA, IB, IC, Kg1, Kg2, Jm, Jg1, dsp, Kt, Ka, c, TA, TcA, TcAest, TsA, FAB, FcAB, FcABest, FsAB];

maxerror1 = 0.003; % The symmetric range of values for rotational error in the table
% i.e. maxerror1 = 5 => [-5, -4, ..., 0, ..., 5]
maxerror2 = 0.0003; % The symmetric range of values for translational error in the table

numvalues1 = 29; % determines the number of columns in the table (resolution of rotational values)
umvalues2 = 29; % determines the number of rows in the table (resolution of translational values)

% Changes any even number num_values to an odd number so the symmetric spacing
% always includes the zero case
numvalues1 = numvalues1 + 1 - mod(numvalues1, 2);
numvalues2 = numvalues2 + 1 - mod(numvalues2, 2);

E1table = linspace(maxerror1*-1, maxerror1, numvalues1);
E2table = linspace(maxerror2*-1, maxerror2, numvalues2);

% Generates delta t tables
[Deltat1tableExact, Deltat2tableExact] = Build_PWC_Table(E1table, E2table, parameter_vector);

for r = 1:numvalues2
    for m = 1:numvalues1
        if Deltat1tableExact(r,m) <= 0.0001
            Deltat1tableExact(r,m) = 0;
        end
    end
end
if Deltat2tableExact(r,m) <= 0.0001
    Deltat2tableExact(r,m) = 0;
end
end
end

E1roottable = sign(E1table).*sqrt(abs(E1table));
E2roottable = sign(E2table).*sqrt(abs(E2table));

Extrapval1 = 1.01*max(E1roottable);
Extrapval2 = 1.01*max(E2roottable);

c = 1.5;

for r = 1:numvalues2
    for m = 1:numvalues1
        if (r == 1 | r == numvalues2) | (m == 1 | m == numvalues1)
            Deltat1tableInitial(r,m) = Deltat1tableExact(r,m);
            Deltat2tableInitial(r,m) = Deltat2tableExact(r,m);
        else
            Deltat1tableInitial(r,m) = c*Deltat1tableExact(r,m);
            Deltat2tableInitial(r,m) = c*Deltat2tableExact(r,m);
        end
    end
end

% Deltat1tableInitial = c*Deltat1tableExact;
% Deltat2tableInitial = c*Deltat2tableExact;

for sample = 1:num_samples
    seed_value = sample*100;

    Deltat1table = Deltat1tableInitial;
    Deltat2table = Deltat2tableInitial;

    zscale_rot = max(max(abs(Deltat1tableExact - Deltat1tableInitial)));
    zscale_trans = max(max(abs(Deltat2tableExact - Deltat2tableInitial)));

    % Introduce constants to minimize the number of calculations.
    linflag = 1;  % Set equal to 1 to include nonlinearities; otherwise 0.

    rot_SumSquares = 0;
    trans_SumSquares = 0;

    for r = 1:numvalues2
        for m = 1:numvalues1
            if Deltat1tableExact(r,m) ~= 0
...
\begin{verbatim}
rot_PercentErr(r,m) = abs(Deltat1tableInitial(r,m) - Deltat1tableExact(r,m))/abs(Deltat1tableExact(r,m));
rot_SumSquares = rot_SumSquares + rot_PercentErr(r,m)^2;
end

if Deltat2tableExact(r,m) ~= 0
  trans_PercentErr(r,m) = abs(Deltat2tableInitial(r,m) - Deltat2tableExact(r,m))/abs(Deltat2tableExact(r,m));
  trans_SumSquares = trans_SumSquares + trans_PercentErr(r,m)^2;
end
end

RotationalError(1) = rot_SumSquares;
TranslationalError(1) = trans_SumSquares;

dt = 0.00001;           % Integration time step
time_final = 2.0;        % Final time
pepsA = 0.00003;        % Angular position error tolerance for link A
pepsB = 0.000003;        % Position error tolerance for link B
uepsA = 0.0001;         % Angular speed error tolerance for link A
uepsB = 0.0001;         % Speed error tolerance for link B
Eeps = 0.000005;        % System energy stopping tolerance

% Pre-allocate space for faster execution.

t = zeros(1,round(time_final/dt)+1);       % Time
q1 = zeros(1,round(time_final/dt)+1);      % Angular position of link A
q2 = zeros(1,round(time_final/dt)+1);      % Angular position of link B
u1 = zeros(1,round(time_final/dt)+1);      % Angular velocity of link A
u2 = zeros(1,round(time_final/dt)+1);      % Angular velocity of link B
u1dot = zeros(1,round(time_final/dt));     % Angular acceleration of link A
u2dot = zeros(1,round(time_final/dt));     % Angular acceleration of link B
TO = zeros(1,round(time_final/dt));        % Torque applied at O
FP = zeros(1,round(time_final/dt));        % Force applied at P
TfO = zeros(1,round(time_final/dt));       % Frictional torque at O
FfP = zeros(1,round(time_final/dt));       % Frictional force at P
Energy = zeros(1,round(time_final/dt));    % Total energy of the system

% Initialize time and state variables.

t(1) = 0;
q1(1) = 0;
q2(1) = 0;
u1(1) = 0;
u2(1) = 0;

tpend1=0;           % Pulse end-time for torque at O
tpend2=0;           % Pulse end-time for force at P
directionO=1;       % Pulse direction indicator for torque at O
\end{verbatim}
directionP = 1;  % Pulse direction indicator for force at P

Energy(1) = 0.5*((IA+IB+IC+mA*LA^2)*u1(1)^2 + mB*((-LT*u1(1)+u2(1))^2 + (LP+q2(1)-L/2)^2)*u1(1)^2) ...
+ mC*((-LT*u1(1)+u2(1))^2 + (LP+q2(1)+LC)^2)*u1(1)^2) ...
+ Kg1^2*(Jm+Jg1)*u1(1)^2 + 4*Kg2^2*Jm/dsp^2*u2(1)^2);

pulsecountmax = 20;

if(PLOTS_ON)
    figure(1)
    set(1, 'Position', [4 32 1018 668])
end

% seed_value = 0;

for runs = 1:20

    % qf1 = 0.0015*sign(rand-0.5) + (2*rand-1)*0.0015;
    % qf2 = 0.00015*sign(rand-0.5) + (2*rand-1)*0.00015;

    [rand_value1, seed_value] = RandPredict(seed_value);
    [rand_value2, seed_value] = RandPredict(seed_value);

    qf1 = 0.5*(2*rand_value1-1)*maxerror1*0.75;
    qf2 = 0.5*(2*rand_value2-1)*maxerror2*0.75;

    qf1data(runs) = qf1*180/pi;
    qf2data(runs) = qf2;

    Deltat1Error = abs(Deltat1tableExact - Deltat1table);
    Deltat2Error = abs(Deltat2tableExact - Deltat2table);

    if(PLOTS_ON)
        subplot(2,2,1)
        surf(E1roottable, E2roottable, Deltat1Error)
        shading interp
        axis([min(E1roottable) max(E1roottable) min(E2roottable) max(E2roottable) 0 zscale_rot 0 zscale_rot])
        title('Error between Exact and Current Rotational Pulse Width Table')
        xlabel('rotational error root')
        ylabel('translational error root')
        zlabel('exact/actual pulse widths error (sec)')
        subplot(2,2,2)
        surf(E1roottable, E2roottable, Deltat2Error)
    end

57
shading interp
axis([min(E1roottable) max(E1roottable) min(E2roottable) max(E2roottable) 0
zscale_trans 0 zscale_trans])
title('Error between Exact and Current Translational Pulse Width Table')
xlabel('rotational error root')
ylabel('translational error root')
zlabel('exact/actual pulse widths error (sec)')
end

pulsecount(runs) = 1;

if(PLOTS_ON)
    subplot(2,1,2)
    plot(pulsecount(runs), 1, 'rp')
    axis([0 20 0 2])
    text(1, 1.5, num2str(runs))
    text(1, 0.5, num2str(pulsecount(runs)))
    title('Pulse Count')
    xlabel('Pulses')
    pause(0.5)
end

poserr1 = qf1 - q1(1);
poserr2 = qf2 - q2(1);

SqrtPoserr1 = sqrt(abs(poserr1));
directionO = sign(poserr1);
if (sqrt(abs(poserr1)) <= max(E1roottable)) & (sqrt(abs(poserr2)) <= max(E2roottable))
    tp1 = interp2(E1roottable, E2roottable, Deltat1table, sign(poserr1)*sqrt(abs(poserr1)),
sign(poserr2)*sqrt(abs(poserr2)), 'linear');
else
    tp1 = Extrapval1;
end
tpend1=t(1)+tp1;
lastposerr1 = poserr1;

SqrtPoserr2 = sqrt(abs(poserr2));
directionP = sign(poserr2);
if (sqrt(abs(poserr1)) <= max(E1roottable)) & (sqrt(abs(poserr2)) <= max(E2roottable))
    tp2 = interp2(E1roottable, E2roottable, Deltat2table, sign(poserr1)*sqrt(abs(poserr1)),
sign(poserr2)*sqrt(abs(poserr2)), 'linear');
else
    tp2 = Extrapval2;
end
tpend2=t(1)+tp2;
lastposerr2 = poserr2;
q1old = q1(1);
q2old = q2(1);

i = 1;

while ((Energy(i) > Eeps) | (abs(poserr1) > pepsA) | (abs(poserr2) > pepsB)) & (pulsecount(runs) < pulsecountmax)
    if Energy(i) < Eeps              % Check if the system has stopped
        % Application of PWC at O.
        if abs(poserr1) > pepsA  |  abs(poserr2) > pepsB     % Check if link A or link B have
            reached the desired orientation
                if t(i) >= tpend1 &  t(i) >= tpend2 % Check if time has exceeded the pulse
                    end-time
        end
    end
    if i > 1
        poserr1net = q1(i) - q1old;
        q1old = q1(i);
        SqrtPoserr1net = sqrt(abs(poserr1net));

        poserr2net = q2(i) - q2old;
        q2old = q2(i);
        SqrtPoserr2net = sqrt(abs(poserr2net));

        % If net displacements are within the ranges of the tables, find their
        % locations in the tables
        if (SqrtPoserr1net <= max(E1roottable)) & (SqrtPoserr2net <= max(E2roottable))
            for j = 1:numvalues1-1
                if (E1roottable(j) <= sign(poserr1net)*SqrtPoserr1net) & (sign(poserr1net)*SqrtPoserr1net <= E1roottable(j+1))
                    column = j;
                    new_xr1 = E1roottable(j);
                    new_xr2 = E1roottable(j+1);
                end
            end
            for j = 1:numvalues2-1
                if (E2roottable(j) <= sign(poserr2net)*SqrtPoserr2net) & (sign(poserr2net)*SqrtPoserr2net <= E2roottable(j+1))
                    row = j;
                end
            end
        end
    end

end
new_xt1 = E2roottable(j);
new_xt2 = E2roottable(j+1);
end
end

%Define Areas

new_xre = sign(poserr1net)*SqrtPoserr1net;
new_xte = sign(poserr2net)*SqrtPoserr2net;

newA11 = abs((new_xte - new_xt1)*(new_xre - new_xr1));
newA12 = abs((new_xt1 - new_xte)*(new_xre - new_xr2));
newA21 = abs((new_xre - new_xr1)*(new_xt2 - new_xte));
newA22 = abs((new_xt2 - new_xte)*(new_xr2 - new_xre));

Ahat = newA11 + newA12 + newA21 + newA22;

A_sum_squared = newA11^2 + newA12^2 + newA21^2 + newA22^2;

Constant = Ahat^2/A_sum_squared;

%Rotational Table

rot_tr11 = Deltat1table(row, column);
rot_tr12 = Deltat1table(row, column + 1);
rot_tr21 = Deltat1table(row + 1, column);
rot_tr22 = Deltat1table(row + 1, column + 1);

rot_tre_est = (newA22*rot_tr11 + newA21*rot_tr12 + newA12*rot_tr21 +
newA11*rot_tr22)/Ahat;

weight1 = 0.0;
weight2 = 1.0;

rot_tr11new = weight1*Deltat1table(row, column) + weight2*(rot_tr11 +
Constant*newA22/Ahat*(tp1 - rot_tre_est));
rot_tr12new = weight1*Deltat1table(row, column + 1) + weight2*(rot_tr12 +
Constant*newA21/Ahat*(tp1 - rot_tre_est));
rot_tr21new = weight1*Deltat1table(row + 1, column) + weight2*(rot_tr21 +
Constant*newA12/Ahat*(tp1 - rot_tre_est));
rot_tr22new = weight1*Deltat1table(row + 1, column + 1) + weight2*(rot_tr22 +
Constant*newA11/Ahat*(tp1 - rot_tre_est));

if(~BLANKET_ON)
    Deltat1table(row, column) = rot_tr11new;
    Deltat1table(row, column + 1) = rot_tr12new;
    Deltat1table(row + 1, column) = rot_tr21new;
    Deltat1table(row + 1, column + 1) = rot_tr22new;
else

d1_up_left = rot_tr11new - rot_tr11;
d1_up_right = rot_tr12new - rot_tr12;
d1_down_left = rot_tr21new - rot_tr21;
d1_down_right = rot_tr22new - rot_tr22;

Deltat1table = Stretch2DTable(Deltat1table, E1roottable, E2roottable, row, column, d1_up_left, d1_up_right, d1_down_left, d1_down_right, sigma1, sigma2, zero_barrier_on);

% rot_tr11new
% Deltat1table(row, column)
% rot_tr12new
% Deltat1table(row, column+1)
% rot_tr21new
% Deltat1table(row+1, column)
% rot_tr22new
% Deltat1table(row+1, column+1)
% disp('next adaptation')

end

% Translational Table

trans_tr11 = Deltat2table(row, column);
trans_tr12 = Deltat2table(row, column + 1);
trans_tr21 = Deltat2table(row + 1, column);
trans_tr22 = Deltat2table(row + 1, column + 1);

trans_tre_est = (newA22*trans_tr11 + newA21*trans_tr12 + newA12*trans_tr21 + newA11*trans_tr22)/Ahat;

trans_tr11new = weight1*Deltat2table(row, column) + weight2*(trans_tr11 + Constant*newA22/Ahat*(tp2 - trans_tre_est));
trans_tr12new = weight1*Deltat2table(row, column + 1) + weight2*(trans_tr12 + Constant*newA21/Ahat*(tp2 - trans_tre_est));
trans_tr21new = weight1*Deltat2table(row + 1, column) + weight2*(trans_tr21 + Constant*newA12/Ahat*(tp2 - trans_tre_est));
trans_tr22new = weight1*Deltat2table(row + 1, column + 1) + weight2*(trans_tr22 + Constant*newA11/Ahat*(tp2 - trans_tre_est));

if(~BLANKET_ON)
    Deltat2table(row, column) = trans_tr11new;
    Deltat2table(row, column + 1) = trans_tr12new;
    Deltat2table(row + 1, column) = trans_tr21new;
    Deltat2table(row + 1, column + 1) = trans_tr22new;
else
    d2_up_left = trans_tr11new - trans_tr11;
    d2_up_right = trans_tr12new - trans_tr12;
    d2_down_left = trans_tr21new - trans_tr21;
    d2_down_right = trans_tr22new - trans_tr22;
end
Deltat2table = Stretch2DTable(Deltat2table, E1roottable, E2roottable, row, column, d2_up_left, d2_up_right, d2_down_left, d2_down_right, sigma1, sigma2, zero_barrier_on);

end

for j = row:row+1
    for k = column:column+1
        if Deltat1table(j,k) < 0
            Deltat1table(j,k) = 0;
        end

        if Deltat2table(j,k) < 0
            Deltat2table(j,k) = 0;
        end
    end
end

if(EXTRA_PLOT_ON == 1)
    Deltat1table(:,fix(numvalues1/2)+1) = 0;
    Deltat2table(fix(numvalues2/2)+1,:) = 0;

    figure(2)
    subplot(2,2,1)
    surf(E1roottable, E2roottable, Deltat1table);

    subplot(2,2,2)
    surf(E1roottable, E2roottable, Deltat2table);

    figure(2)
    subplot(2,2,3)
    surf(E1roottable, E2roottable, abs(Deltat1table - Deltat1tableExact));

    subplot(2,2,4)
    surf(E1roottable, E2roottable, abs(Deltat2table - Deltat2tableExact));

    pause

    figure(1)
end
end

% Interpolate the pulse width for torque at 0.

if (sqrt(abs(poserr1)) <= max(E1roottable)) & (sqrt(abs(poserr2)) <= max(E2roottable))
    tp1 = interp2(E1roottable, E2roottable, Deltat1table, sign(poserr1)*sqrt(abs(poserr1)), sign(poserr2)*sqrt(abs(poserr2)), 'linear');
else
tp1 = Extrapval1;
end
directionO=sign(poserr1);
tpend1=t(i)+tp1;

% Interpolate the pulse width for force at P.

if (sqrt(abs(poserr1)) <= max(E1roottable)) & (sqrt(abs(poserr2)) <= max(E2roottable))
    tp2 = interp2(E1roottable, E2roottable, Deltat2table, sign(poserr1)*sqrt(abs(poserr1)), sign(poserr2)*sqrt(abs(poserr2)), 'linear');
else
    tp2 = Extrapval2;
end
directionP=sign(poserr2);
tpend2=t(i)+tp2;
pulsecount(runs) = pulsecount(runs) + 1;
if(PLOTS_ON)
    subplot(2,1,2)
    plot(pulsecount(runs), 1, 'rp')
    axis([0 20 0 2])
    text(1, 1.5, num2str(runs))
    text(1, 0.5, num2str(pulsecount(runs)))
    title('Pulse Count')
    xlabel('Pulses')
    pause(0.1)
endif

% Check if time is less than the pulse end-time for torque at O.

if t(i) < tpend1
    TO(i) = directionO*TA;
else
    TO(i) = 0;
end

% Check if time is less than the pulse end-time for force at P.

if t(i) < tpend2
    FP(i) = directionP*FAB;
else
    FP(i) = 0;
end

% Establish the direction of the friction torque at O.
if \( u_1(i) > 0 \)
\[
T_{fO}(i) = T_cA;
\]
else if \( u_1(i) < 0 \)
\[
T_{fO}(i) = -T_cA;
\]
end

% Establish the direction of the friction force at P.

if \( u_2(i) > 0 \)
\[
F_{fP}(i) = F_{cAB};
\]
else if \( u_2(i) < 0 \)
\[
F_{fP}(i) = -F_{cAB};
\]
end

% Use the equations of motion to calculate the accelerations of links A % and B.

\[
X_{11} = m_A*L_A^2 + I_A + m_B*((L_P+\text{linflag}*q_2(i)-L/2)^2 + L_T^2) + I_B + 
\]
\[
m_C*((L_P+\text{linflag}*q_2(i)+L_C)^2 + L_T^2) + I_C ... 
\]
\[
+ K_g1^2*(J_m+J_g1);
\]
\[
X_{12} = -(m_B+m_C)*L_T;
\]
\[
X_{21} = X_{12};
\]
\[
X_{22} = m_B + m_C + 4*K_g2^2*J_m/dsp^2;
\]
\[
Y_1 = -2*u_1(i)*u_2(i)*(m_B*(L_P+q_2(i)-L/2)+m_C*(L_P+q_2(i)+L_C))*\text{linflag} + T_O(i) - T_{fO}(i);
\]
\[
Y_2 = (m_B*(L_P+q_2(i)-L/2)+m_C*(L_P+q_2(i)+L_C))*u_1(i)^2*\text{linflag} + F_P(i) - F_{fP}(i);
\]
\[
X = [X_{11} \, X_{12}; \, X_{21} \, X_{22}];
\]
\[
Y = [Y_1; \, Y_2];
\]
\[
U = X\backslash Y;
\]
\[
u_1\text{dot}(i) = U(1);
\]
\[
u_2\text{dot}(i) = U(2);
\]

% Check if link A has stopped and link B is still moving.

if \( \text{abs}(u_1(i)) < \text{uepsA} \) & \( \text{abs}(u_2(i)) > \text{uepsB} \)

% Calculate the acceleration of link B assuming link A has stopped.

\[
u_2\text{dotstop} = \frac{(F_P(i) - F_{fP}(i))/(m_B+m_C+4*K_g2^2*J_m/dsp^2)}{;}
\]

% Calculate the friction torque with link A stopped.

\[
T_{fO}(i) = T_O(i) + (m_B+m_C)*L_T*u_2\text{dotstop};
\]

% Check if the required friction torque on link A is less than the % maximum static friction torque.

if \( \text{abs}(T_{fO}(i)) < T_sA \)
% Establish the angular speed and acceleration of link A and
% acceleration of link B with link A stopped.

u1(i) = 0;
u1dot(i) = 0;
u2dot(i) = u2dotstop;

else

% Establish the direction of the friction torque at O.

if u1(i) > 0
    TfO(i) = TcA;
elseif u1(i) < 0
    TfO(i) = -TcA;
else
    TfO(i) = sign(TfO(i))*TsA;
end
end

% Check if both the links have stopped.

elseif abs(u1(i)) < uepsA & abs(u2(i)) < uepsB

% Calculate the friction torque at O and friction force at P with both
% links stopped.

TfO(i) = TO(i);
FfP(i) = FP(i);

% Check if the required friction torque on link A is less than the
% maximum static friction torque.

if abs(TfO(i)) < TsA

% Establish the angular speed and acceleration of link A.

u1(i) = 0;
u1dot(i) = 0;
else

% Establish the direction of the friction torque at O.

if u1(i) > 0
    TfO(i) = TcA;
elseif u1(i) < 0
    TfO(i) = -TcA;
else
    TfO(i) = sign(TfO(i))*TsA;
end
end
% Check if the required friction force on link B is less than the
% maximum static friction force.

if abs(FfP(i)) < FsAB

% Establish the speed and acceleration of link B.

u2(i) = 0;
u2dot(i) = 0;
else

% Establish the direction of the friction force at P.

if u2(i) > 0
    FfP(i) = FcAB;
elseif u2(i) < 0
    FfP(i) = -FcAB;
else
    FfP(i) = sign(FfP(i))*FsAB;
end
end

% Check if link B has stopped relative to A (link B is moving along
% with A).

elseif abs(u2(i)) < uepsB & abs(u1(i)) > uepsA

% Calculate the angular acceleration of link A assuming link B has
% stopped relative to A.

u1dotstop = (TO(i) - TfO(i)) / (mA*LA^2 + IA + mB*((LP+linflag*q2(i)-L/2)^2 + LT^2) ... + IB + mC*((LP+q2(i)+LC)^2 + LT^2) + IC + Kg1^2*(Jm+Jg1));

% Calculate the friction force at P with link B stopped relative to
% link A.

FfP(i) = FP(i) + (mB+mC)*LT*u1dotstop + linflag*(mB*(LP+q2(i)-L/2)+mC*(LP+q2(i)+LC))*u1(i)^2;

% Check if the required friction force on link B is less than the
% maximum static friction force.

if abs(FfP(i)) < FsAB

% Establish the angular speed of link A and speed and acceleration of
% link B.

u1dot(i) = u1dotstop;
u2(i) = 0;
u2dot(i) = 0;
else

% Establish the direction of the friction force at P.

if u2(i) > 0
    FfP(i) = FcAB;
elseif u2(i) < 0
    FfP(i) = -FcAB;
else
    FfP(i) = sign(FfP(i))*FsAB;
end
end
end

% Use Euler's method to calculate generalized speeds and generalized
% coordinates of the two links.

i = i + 1;

u1(i) = u1(i-1)+u1dot(i-1)*dt;
u2(i) = u2(i-1)+u2dot(i-1)*dt;

q1(i) = q1(i-1)+u1(i-1)*dt;
q2(i) = q2(i-1)+u2(i-1)*dt;

t(i)=t(i-1)+dt;

poserr1=qf1-q1(i);
poserr2=qf2-q2(i);

% Calculate the total system energy.

Energy(i) = 0.5*((IA+IB+IC+mA*LA^2)*u1(i)^2 + mB*((-LT*u1(i)+u2(i))^2 + (LP+q2(i)-L/2)*u1(i)^2) ... 
+ mC*((-LT*u1(i)+u2(i))^2 + (LP+q2(i)+LC+q2(i))^2) ... 
+ Kg1^2*(Jm+Jg1)*u1(i)^2 + 4*Kg2^2*Jm/dsp^2*u2(i)^2);
end

rot_SumSquares = 0;

for r = 1:numvalues2
    for m = 1:numvalues1
        if Deltat1tableExact(r,m) ~= 0
            rot_PercentErr(r,m) = abs(Deltat1table(r,m) - Deltat1tableExact(r,m))/abs(Deltat1tableExact(r,m));
            rot_SumSquares = rot_SumSquares + rot_PercentErr(r,m)^2;
        end
    end
end
end
trans_SumSquares = 0;

    for r = 1:numvalues2
        for m = 1:numvalues1
            if Deltat2tableExact(r,m) ~= 0
                trans_PercentErr(r,m) = abs(Deltat2table(r,m) - Deltat2tableExact(r,m))/abs(Deltat2tableExact(r,m));
                trans_SumSquares = trans_SumSquares + trans_PercentErr(r,m)^2;
            end
        end
    end

RotationalError(runs+1) = rot_SumSquares;
TranslationalError(runs+1) = trans_SumSquares;

% Plot the results.

minTO = -TA;
maxTO = TA;

minFP = -FAB;
maxFP = FAB;

minq1 = min(q1*57.3);
maxq1 = max(q1*57.3);

if(PLOTS_ON)
    subplot(2,2,3)
    plot([0,t(i)], [qf1*180/pi,qf1*180/pi], 'r--', t(1:i), q1(1:i)*180/pi, 'b', t(1:i), 0.5*(maxq1-minq1)+(maxq1-minq1)*(TO(1:i)-0.5*(maxTO-minTO))/(maxTO-minTO), 'k--', t(1:i), 0.5*(maxq1-minq1)+(maxq1-minq1)*(FP(1:i)-0.5*(maxFP-minFP))/(maxFP-minFP), ':k')
    title('Orientation of Link A')
    axis([0 t(i) minq1-0.1*(maxq1-minq1) maxq1+0.1*(maxq1-minq1)])
    xlabel('t (sec)')
    ylabel('q1 (deg)')
    legend('Desired orientation', 'Actual orientation', 'Actuator torque at O', 'Actuator force at P', 0)

    minq2 = min(q2);
    maxq2 = max(q2);

    subplot(2,2,4)
    plot([0,t(i)], [qf2,qf2], 'r--', t(1:i), q2(1:i), 'b', t(1:i), 0.5*(maxq2-minq2)+(maxq2-minq2)*(TO(1:i)-0.5*(maxTO-minTO))/(maxTO-minTO), 'k--', t(1:i), 0.5*(maxq2-minq2)+(maxq2-minq2)*(FP(1:i)-0.5*(maxFP-minFP))/(maxFP-minFP), ':k')
    title('Position of Link B')
    xlabel('t (sec)')
ylabel('q2 (m)')
axis([0 t(i) minq2-0.1*(maxq2-minq2) maxq2+0.1*(maxq2-minq2)])
legend('Desired position', 'Actual position', 'Actuator torque at O', 'Actuator force at P', 0)
end

flag = 0;
if flag == 1

    minu1 = min(u1);
    maxu1 = max(u1);

    figure(5)
    plot(t(1:length(TO)), u1(1:length(TO)), t(1:length(TO)), (maxu1-minu1)*(TO-0.5*(maxTO+minTO))/(maxTO-minTO), '--k')
    title('Angular Speed of Link A')
    xlabel('t (sec)')
    ylabel('u1 (rad/s)')
    axis([0 time_final minu1-0.1*(maxu1-minu1) maxu1+0.1*(maxu1-minu1)])
    legend('Angular speed', 'Actuator torque at O', 0)

    minu2 = min(u2);
    maxu2 = max(u2);

    figure(6)
    plot(t(1:length(TO)), u2(1:length(TO)), t(1:length(FP)), (maxu2-minu2)*(FP-0.5*(maxFP+minFP))/(maxFP-minFP), ':k')
    title('Speed of Link B')
    xlabel('t (sec)')
    ylabel('u2 (m/s)')
    axis([0 time_final minu2-0.1*(maxu2-minu2) maxu2+0.1*(maxu2-minu2)])
    legend('Speed', 'Actuator force at P', 0)

    figure(7)
    plot(t(1:length(TO)), TO)
    title('Actuator Torque Exerted at O')
    xlabel('t (sec)')
    ylabel('TO (N-m)')
    axis([0 time_final minTO-0.1*(maxTO-minTO) maxTO+0.1*(maxTO-minTO)])

    figure(8)
    plot(t(1:length(FP)), FP)
    title('Actuator Force Exerted at P')
    xlabel('t (sec)')
    ylabel('FP (N)')
    axis([0 time_final minFP-0.1*(maxFP-minFP) maxFP+0.1*(maxFP-minFP)])

    figure(9)
    plot(t(1:length(TfO)), TfO, t(1:length(TO)), TO, '--k')
    title('Friction Torque at O')
xlabel('t (sec)')
ylabel('TfO (N-m)')
axis([0 time_final minTO-0.1*(maxTO-minTO) maxTO+0.1*(maxTO-minTO)])
legend('Friction Torque at O', 'Actuator Torque at O', 0)

figure(10)
plot(t(1:length(FfP)), FfP, t(1:length(FP)), FP, ':k')
title('Friction Force at P')
xlabel('t (sec)')
ylabel('FfP (N)')
axis([0 time_final minFP-0.1*(maxFP-minFP) maxFP+0.1*(maxFP-minFP)])
legend('Friction force at P', 'Actuator force at P', 0)

minEnergy = min(Energy);
maxEnergy = max(Energy);

figure(11)
plot(t(1:length(Energy)), Energy, t(1:length(TO)), (maxEnergy-minEnergy)*(TO-0.5*(maxTO+minTO))/(maxTO-minTO), '--k',
     t(1:length(FP)), (maxEnergy-minEnergy)*(FP-0.5*(maxFP+minFP))/(maxFP-minFP), ':k')
title('System Energy')
xlabel('t (sec)')
ylabel('Energy (Joules)')
axis([0 time_final minEnergy-0.1*(maxEnergy-minEnergy) maxEnergy+0.1*(maxEnergy-minEnergy)])
legend('System Energy', 'Actuator torque at O', 'Actuator force at P', 0)

end

%     if(~PAUSES_ON)
%         pause(2)
%     else
%         pause
%     end
end

if(PLOTS_ON)
 subplot(2,1,1)
 plot([1:runs], RotationalError(1:runs), 'bo-', [1:runs], TranslationalError(1:runs), 'r-')
 title('Sum of Squares of Error for each Run')
 xlabel('Run')
 ylabel('Sum of Squares of Error')
 maxy = [RotationalError TranslationalError];
 axis([1 runs 0 max(maxy)])
 legend('Rotational table Error', 'Translational table Error', 0)
 subplot(2,1,2)

end
plot([1,runs], [3,3], '--r', [1:runs], pulsecount(1:runs), 'bo-')
title('Number of Pulses for each Run')
xlabel('Run')
ylabel('Number of Pulses')
axis([1 runs 0 max(pulsecount)+1])

end

RotErrorMatrix(sample,:) = RotationalError(1:runs);
TransErrorMatrix(sample,:) = TranslationalError(1:runs);
PulseMatrix(sample,:) = pulsecount(1:runs);
sample
end

MeanRotError = mean(RotErrorMatrix);
MeanTransError = mean(TransErrorMatrix);
MeanPulses = mean(PulseMatrix);

figure(1);
subplot(2,1,1);
plot([1:runs], MeanRotError, 'bo-', [1:runs], MeanTransError, 'ro-')
subplot(2,1,2);
plot([1:runs], MeanPulses, 'kx-')

subplot(2,1,1)
ylabel('Sum Squares of the Error')
xlabel('Motion Number')
subplot(2,1,2)
ylabel('Average Number of Pulses')
xlabel('Motion Number')

xlabel('Motion Number')
title('Average Pulses vs. Motion Number')
subplot(2,1,1)
title('Table Error vs. Motion Number')
legend('Rotational Table Error', 'Translational Table Error')
clf; % Tests various blanketing methods using a sample test function to model

sigma_percentage = 0.05;
x_min = 0;
x_max = 4;
num_points = 500;
decay = (x_max-x_min)*sigma_percentage;

runs = 7*num_points;

x = linspace(x_min, x_max, num_points);
y_actual = testfunction(x);

y_table_initial = testfunction_initial(x);

x_table = linspace(x_min, x_max, num_points);
y_table_normal = y_table_initial;
y_table_blanket = y_table_initial;
y_table_blanket_cap = y_table_initial;

plot(x,y_actual,'k:',x_table,y_table_normal,'bx--',x_table,y_table_blanket,'rx--')

error_normal = zeros(1,runs);
error_blanket = zeros(1,runs);
error_blanket_cap = zeros(1,runs);

for m = [1:runs]
    x_rand = rand*(x_max-x_min)+x_min;
y_rand_actual = testfunction(x_rand);
y_table_normal = adapt_normal(x_table, y_table_normal, x_rand, y_rand_actual);
y_table_blanket = adapt_blanket(x_table, y_table_blanket, x_rand, y_rand_actual,decay);
y_table_blanket_cap = adapt_blanket_capping(x_table, y_table_blanket_cap, x_rand, y_rand_actual,decay);

    figure(1);
    plot(x,y_actual,'k:',x_table,y_table_normal,'bx--',x_table,y_table_blanket,'rx--',x_table,y_table_blanket_cap,'gx--',x_rand,y_rand_actual,'kx')


```matlab
sum_normal = 0;
sum_blanket = 0;
sum_blanket_cap = 0;

for p = 1:num_points
    sum_normal = (y_table_normal(p)-testfunction(x_table(p)))^2 + sum_normal;
    sum_blanket = (y_table_blanket(p)-testfunction(x_table(p)))^2 + sum_blanket;
    sum_blanket_cap = (y_table_blanket_cap(p)-testfunction(x_table(p)))^2 + sum_blanket_cap;
end

error_normal(m) = sum_normal;
error_blanket(m) = sum_blanket;
error_blanket_cap(m) = sum_blanket_cap;

% figure(2)
% plot([1:runs],error_normal,'bx-',[1:runs],error_blanket,'rx-',[1:runs],error_blanket_cap,'gx-')
%
% pause(0.1)
end

figure(1);
plot(x,y_actual,'k:',x_table,y_table_normal,'bx--',x_table,y_table_blanket,'rx--
',x_table,y_table_blanket_cap,'gx--',x_rand,y_rand_actual,'kx')

% figure(2)
% plot([1:runs],error_normal,'k.-',[1:runs],error_blanket,'ko-',[1:runs],error_blanket_cap,'kx-')
% legend('PAA','Blanketing (no capping)','Blanketing with Capping');
% title('Blanketing Effect on Table Error in 1-D')
% xlabel('Number of Adaptations')
% ylabel('Sum of Squares of Table Error')

figure(2)
plot([1:runs],error_normal,'k.-',[1:runs],error_blanket_cap,'kx-')
legend('PAA','Blanketing');
title('Blanketing Effect on Table Error in 1-D')
xlabel('Number of Adaptations')
ylabel('Sum of Squares of Table Error')
```

73
BLANKET_TEST_STAT.m

% Statistical Blanketing Testing Script

clc;
close all;
clear;

stat_runs = 50;
sigma_percentage = 0.25;

x_min = 0;
x_max = 4;
num_points = 25;
decay = (x_max-x_min)*sigma_percentage;

runs = 10*num_points;

for z = [1:stat_runs]
    x = linspace(x_min, x_max, num_points);
    y_actual = testfunction(x);
    y_table_initial = testfunction_initial(x);
    x_table = linspace(x_min, x_max, num_points);
    y_table_normal = y_table_initial;
    y_table_blanket = y_table_initial;
    y_table_blanket_cap = y_table_initial;
    plot(x,y_actual,'k:',x_table,y_table_normal,'bx--',x_table,y_table_blanket,'rx--')
    error_normal = zeros(1,runs);
    error_blanket = zeros(1,runs);
    error_blanket_cap = zeros(1,runs);
    error_normal_stat = zeros(1,runs);
    error_blanket_stat = zeros(1,runs);
    error_blanket_cap_stat = zeros(1,runs);
    for m = [1:runs]
        x_rand = rand*(x_max-x_min)+x_min;
        y_rand_actual = testfunction(x_rand);
        y_table_normal = adapt_normal(x_table, y_table_normal, x_rand, y_rand_actual);
        y_table_blanket = adapt_blanket(x_table, y_table_blanket, x_rand, y_rand_actual, decay);
        y_table_blanket_cap = adapt_blanket_capping(x_table, y_table_blanket_cap, x_rand, y_rand_actual, decay);
    end
end

% figure(1);
sum_normal = 0;
sum_blanket = 0;
sum_blanket_cap = 0;

for p = 1:num_points
    sum_normal = (y_table_normal(p)-testfunction(x_table(p)))^2 + sum_normal;
    sum_blanket = (y_table_blanket(p)-testfunction(x_table(p)))^2 + sum_blanket;
    sum_blanket_cap = (y_table_blanket_cap(p)-testfunction(x_table(p)))^2 + sum_blanket_cap;
end

error_normal(m) = sum_normal;
error_blanket(m) = sum_blanket;
error_blanket_cap(m) = sum_blanket_cap;

error_normal_stat(m) = error_normal_stat(m) + sum_normal;
error_blanket_stat(m) = error_blanket_stat(m) + sum_blanket;
error_blanket_cap_stat(m) = error_blanket_cap_stat(m) + sum_blanket_cap;

error_all_normal(z,m) = sum_normal;
error_all_blanket_cap(z,m) = sum_blanket_cap;

% figure(2)
% plot([1:runs],error_normal,'bx-',[1:runs],error_blanket,'rx-',[1:runs],error_blanket_cap,'gx-')

% pause(0.1)
end

z
end

average_normal = error_normal_stat/stat_runs;
average_blanket = error_blanket_stat/stat_runs;
average_blanket_cap = error_blanket_cap_stat/stat_runs;

average_normal = mean(error_all_normal);
average_blanket_cap = mean(error_all_blanket_cap);

value_at_tau_norm = average_normal(1)*exp(-1);
value_at_tau1_blanket = average_blanket_cap(1)*exp(-1);

tau_norm = interp1(average_normal,[1:runs],value_at_tau_norm);
tau1_blanket = interp1(average_blanket_cap,[1:runs],value_at_tau1_blanket)

value_at_tau2_blanket = average_blanket_cap(1)*exp(-2);
tau2_blanket = interp1(average_blanket_cap,[1:runs],value_at_tau2_blanket)
\[
\text{value}_\text{at}_\tau_3\text{blanket} = \text{average}_\text{blanket}_\text{cap}(1) \times \exp(-3);
\]
\[
\tau_3\text{blanket} = \text{interp1(average}_\text{blanket}_\text{cap},[1:\text{runs}],\text{value}_\text{at}_\tau_3\text{blanket})
\]

\[
\text{value}_\text{at}_\tau_4\text{blanket} = \text{average}_\text{blanket}_\text{cap}(1) \times \exp(-4);
\]
\[
\tau_4\text{blanket} = \text{interp1(average}_\text{blanket}_\text{cap},[1:\text{runs}],\text{value}_\text{at}_\tau_4\text{blanket})
\]

\[
\text{final}_\text{value}_\text{normal} = \text{mean(average}_\text{normal(ceil(0.8*\text{runs}):\text{runs}))/average}_\text{normal(1)};
\]
\[
\text{final}_\text{value}_\text{blanket} = \text{mean(average}_\text{blanket}_\text{cap(ceil(0.8*\text{runs}):\text{runs}))/average}_\text{normal(1)}
\]

\[
\text{overtake}_\text{time} = \text{interp1(average}_\text{normal},[1:\text{runs}],\text{final}_\text{value}_\text{blanket}*\text{average}_\text{normal(1)})
\]
\[
\text{std}_\text{blanket} = \text{std(average}_\text{blanket}_\text{cap(ceil(0.8*\text{runs}):\text{runs}))}
\]

\[
\text{plot([1:\text{runs}],average}_\text{normal,'k.-',[1:\text{runs}],average}_\text{blanket}_\text{cap,'kx-'}',[1,\text{runs}],[\text{final}_\text{value}_\text{blanket}*average}_\text{blanket}_\text{cap(1)},\text{final}_\text{value}_\text{blanket}*average}_\text{blanket}_\text{cap(1)}],[\text{overtake}_\text{time},\text{overtake}_\text{time}],[0,\text{average}_\text{normal(1)}],'b--'
\]
\[
,\text{tau1}_\text{blanket},\text{tau1}_\text{blanket},[0,\text{average}_\text{normal(1)}],'k:'\text{tau2}_\text{blanket},\text{tau2}_\text{blanket},[0,\text{average}_\text{normal(1)}],'k:'\text{tau3}_\text{blanket},\text{tau3}_\text{blanket},[0,\text{average}_\text{normal(1)}],'k:'\text{tau4}_\text{blanket},\text{tau4}_\text{blanket},[0,\text{average}_\text{normal(1)}],'k:');
\]
\[
\text{legend('PAA','Blanketing');}
\]
\[
\text{title('Blanketing Effect on Table Error in 1-D')}
\]
\[
\text{xlabel('Number of Adaptations')}
\]
\[
\text{ylabel('Sum of Squares of Table Error')}
\]
function [adapted_table1] = Stretch2DTable(initial_table, axis1, axis2, row, col, d11, d12, d21, d22, sigma1, sigma2, zero_barrier_on)

adapted_table1 = initial_table;

adapted_table1(row, col) = initial_table(row, col) + d11;
adapted_table1(row, col+1) = initial_table(row, col+1) + d12;
adapted_table1(row+1, col) = initial_table(row+1, col) + d21;
adapted_table1(row+1, col+1) = initial_table(row+1, col+1) + d22;

s = size(initial_table);
length1 = s(1);
length2 = s(2);

if(sigma2 == 0 & sigma1 ~= 0)
sigma_ratio = 10000;
elseif(sigma2 == 0 & sigma1 == 0)
sigma_ratio = 1;
else
    sigma_ratio = sigma1/sigma2;
end

column = col;
current_row = row;
current_column = column;

value1 = axis1(current_row);
value2 = axis2(current_column);
new_val1 = adapted_table1(current_row, current_column);
% new_val2 = adapted_table2(current_row, current_column);
% delta1 = new_val1 - initial_table(current_row, current_column);
% delta2 = new_val2 - initial_table(current_row, current_column);

quadrant = GetQuad(value1, value2);

for m = [current_row:-1:1]
    for n = [current_column:-1:1]
        dist1 = axis1(m) - value1;
        dist2 = axis2(n) - value2;
        dist = sqrt(dist1^2 + (sigma_ratio*dist2)^2);
        sigma_comp = (dist1/dist)^2*sigma1 + (dist2/dist)^2*sigma1*sigma_ratio^2;
        % sigma_comp = sigma1*sigma2^2/((sigma2*dist1)^2+(sigma1*dist2)^2);
    end
end
delta1_init = new_val1 - adapted_table1(m,n);
%    delta2_init = new_val2 - adapted_table2(m,n);

current_quad = GetQuad(axis1(m), axis2(n));

if((m ~= current_row | n ~= current_column) & (quadrant == current_quad | zero_barrier_on ~= 1))
    adapted_table1(m,n) = initial_table(m,n) + Blanket(dist, delta1, delta1_init, sigma_comp);
%       adapted_table2(m,n) = adapted_table2(m,n) + Blanket(dist, delta2, delta2_init, sigma);
end
end
end
end
end

current_row = row;
current_column = column+1;

value1 = axis1(current_row);
value2 = axis2(current_column);
new_val1 = adapted_table1(current_row, current_column);
%    new_val2 = adapted_table2(current_row, current_column);
delta1 = new_val1 - initial_table(current_row, current_column);
%    delta2 = new_val2 - table2(current_row, current_column);
quadrant = GetQuad(value1, value2);

for m = [current_row:-1:1]
    for n = [current_column:1:length2]
        dist1 = axis1(m) - value1;
dist2 = axis2(n) - value2;
dist = sqrt(dist1^2 + (sigma_ratio*dist2)^2);
        sigma_comp = (dist1/dist)^2*sigma1 + (dist2/dist)^2*sigma1*sigma_ratio^2;
%           sigma_comp = sigma1*sigma2^2/((sigma2*dist1)^2+(sigma1*dist2)^2);
delta1_init = new_val1 - adapted_table1(m,n);
%       delta2_init = new_val2 - adapted_table2(m,n);
current_quad = GetQuad(axis1(m), axis2(n));

        if((m ~= current_row | n ~= current_column) & (quadrant == current_quad | zero_barrier_on ~= 1))
            adapted_table1(m,n) = initial_table(m,n) + Blanket(dist, delta1, delta1_init, sigma_comp);
%               adapted_table2(m,n) = adapted_table2(m,n) + Blanket(dist, delta2, delta2_init, sigma);
        end
    end
end
end

78
current_row = row+1;
current_column = column;

value1 = axis1(current_row);
value2 = axis2(current_column);
new_val1 = adapted_table1(current_row, current_column);
% new_val2 = adapted_table2(current_row, current_column);
delta1 = new_val1 - initial_table(current_row, current_column);
% delta2 = new_val2 - table2(current_row, current_column);

quadrant = GetQuad(value1, value2);

for m = [current_row:1:length1]
    for n = [current_column:-1:1]
        dist1 = axis1(m) - value1;
        dist2 = axis2(n) - value2;
        dist = sqrt(dist1^2 + (sigma_ratio*dist2)^2);
        sigma_comp = (dist1/dist)^2*sigma1 + (dist2/dist)^2*sigma1*sigma_ratio^2;
        % sigma_comp = sigma1*sigma2^2/((sigma2*dist1)^2+(sigma1*dist2)^2);
        delta1_init = new_val1 - adapted_table1(m,n);
        % delta2_init = new_val2 - adapted_table2(m,n);
        current_quad = GetQuad(axis1(m), axis2(n));
        if((m ~= current_row | n ~= current_column) & (quadrant == current_quad | zero_barrier_on ~= 1))
            adapted_table1(m,n) = initial_table(m,n) + Blanket(dist, delta1, delta1_init, sigma_comp);
            % adapted_table2(m,n) = adapted_table2(m,n) + Blanket(dist, delta2, delta2_init, sigma);
        end
    end
end

current_row = row+1;
current_column = column+1;

value1 = axis1(current_row);
value2 = axis2(current_column);
new_val1 = adapted_table1(current_row, current_column);
% new_val2 = adapted_table2(current_row, current_column);
delta1 = new_val1 - initial_table(current_row, current_column);
% delta2 = new_val2 - table2(current_row, current_column);

quadrant = GetQuad(value1, value2);

for m = [current_row:1:length1]
for n = [current_column:1:length2]
    dist1 = axis1(m) - value1;
    dist2 = axis2(n) - value2;
    dist = sqrt(dist1^2 + (sigma_ratio*dist2)^2);
    sigma_comp = (dist1/dist)^2*sigma1 + (dist2/dist)^2*sigma1*sigma_ratio^2;
    % sigma_comp = sigma1*sigma2^2/((sigma2*dist1)^2+(sigma1*dist2)^2);

    delta1_init = new_val1 - adapted_table1(m,n);
    % delta2_init = new_val2 - adapted_table2(m,n);

    current_quad = GetQuad(axis1(m), axis2(n));

    if((m ~= current_row | n ~= current_column) & (quadrant == current_quad | zero_barrier_on ~= 1))
        adapted_table1(m,n) = initial_table(m,n) + Blanket(dist, delta1, delta1_init, sigma_comp);
        % adapted_table2(m,n) = adapted_table2(m,n) + Blanket(dist, delta2, delta2_init, sigma);
    end
end
end
function y_adapted = adapt_blanket_capping(x_table, y_table, x, y, decay)

m = 1;
while x_table(m+1) < x
    m = m + 1;
end

y_adapted = y_table;

L1 = abs(x-x_table(m));
L2 = abs(x-x_table(m+1));
L = L1 + L2;
C = L^2/(L1^2 + L2^2);

interp1(x_table,y_table,x);

ye = interp1(x_table,y_table,x);

y_adapted(m) = y_adapted(m) + C*(x_table(m+1)-x)/(x_table(m+1)-x_table(m))*(y-ye);

delta1 = y_adapted(m) - y_table(m);
delta2 = y_adapted(m+1) - y_table(m+1);

x1 = x_table(m);
x2 = x_table(m+1);

y1_init = y_table(m);
y2_init = y_table(m+1);

for n = [m-1:-1:1]
    dx = x_table(n) - x1;
    dy_init = y1_init - y_table(n);
    y_adapted(n) = y_adapted(n) + Blanket(dx, delta1, dy_init, decay);
end

for n = [m+2:1:length(x_table)]
    dx = x_table(n) - x2;
    dy_init = y2_init - y_table(n);
    y_adapted(n) = y_adapted(n) + Blanket(dx, delta2, dy_init, decay);
end
ADAPT_NORMAL.m

function y_adapted = adapt_normal(x_table, y_table, x, y)

m = 1;
while x_table(m+1) < x
    m = m + 1;
end

y_adapted = y_table;

L1 = abs(x-x_table(m));
L2 = abs(x-x_table(m+1));
L = L1 + L2;
C = L^2/(L1^2 + L2^2);

interp1(x_table,y_table,x);

ye = interp1(x_table,y_table,x);

y_adapted(m) = y_adapted(m) + C*(x_table(m+1)-x)/(x_table(m+1)-x_table(m))*(y-ye);
y_adapted(m+1) = y_adapted(m+1) + C*(x-x_table(m))/(x_table(m+1)-x_table(m))*(y-ye);

BELL_CURVE.m

function p = BellCurve(x, std)

p = exp(-(x.^2)/(2*std^2));

BLANKET.m

function dy_blanket = Blanket(dx, dy_adapt, dy_initial, std)

% dy_blanket = BlanketFunction(dy_adapt, dx, std);
dy_blanket = BlanketFunction(CappingFunction(dy_adapt, dy_initial), dx, std)
BLANKETFUNCTION.m

```matlab
function [delta_y] = BlanketFunction(delta_y_adapted, delta_x, std)

if(std == 0 & delta_x ~= 0)
    delta_y = 0;
elseif(delta_x == 0)
    delta_y = delta_y_adapted;
else
    delta_y = delta_y_adapted*BellCurve(delta_x, std);
end
```

GETQUAD.m

```matlab
function quadrant = GetQuad(value1, value2)

if(value1 >= 0 & value2 >= 0)
    quadrant = 1;
elseif(value1 <= 0 & value2 >= 0)
    quadrant = 2;
elseif(value1 >= 0 & value2 <= 0)
    quadrant = 3;
else
    quadrant = 4;
end
```

BUILD_PWC_TABLE.m

```matlab
% FUNCTION: BUILD_PWC_TABLE
% USAGE: [tmatrix1, tmatrix2] = Build_PWC_Table(e1vector, e2vector, parameter_vector)
% PURPOSE: Returns a table of delta t values that correspond to the input
% errors e1 and e2 for a particular movement axis
% INPUT: e1vector - a vector of rotational position errors, e2vector - a vector of translational position errors, parameter_vector - a vector of system specs
% OUTPUT: tmatrix1 - a matrix that contains the delta t values for rotational pulses, tmatrix2 - a table of delta t values for translational pulses

function [tmatrix1, tmatrix2] = Build_PWC_Table(e1vector, e2vector, parameter_vector)

for m = [1:length(e2vector)]
    for n = [1:length(e1vector)]
        [tmatrix1(m,n), tmatrix2(m,n)] = Retrieve_PWC_Stanford_Widths(e1vector(n), e2vector(m), parameter_vector);
    end
end
```
RETRIEVE_PWC_STANFORD_WIDTHS

% Calculation of delta t's for two rigid link Stanford-like manipulator for
% "perfect" motion.

% FUNCTION: [dt1, dt2] = Retrieve_PWC_Stanford_Widths(e1, e2)
% PURPOSE: Returns delta t's given a rotational error e1 and translational
% error e2
% INPUTS: e1 - rotational error, e2 - translational error, parameter_vector
% - a vector of system specifications
% parameter_vector format:
% parameter_vector = [LA, LT, LP, L, LC, mA, mB, mC, IA, IB, IC, Kg1, Kg2,
% Jm, Jg1, dsp, Kt, Ka, coef, TA, TcA, TcAest, TsA, FAB, FcAB, FcABest,
% FsAB];

% OUTPUTS: dt1: delta t for rotational pulse, dt2: delta t for
% translational pulse

% Last revised on May 29, 2006.
% Last revised on December 29th, 2005.
% June 9th, 2005: Revised equations to include motor and component
% characteristics
% June 10th, 2005: Revised into function from script
% December 29th, 2005: Added dt1 = 0 and dt2 = 0 conditions at end of
% function
% May 29, 2006: Deleted Ke1 and Ke2 from parameter_vector

function [dt1, dt2] = Retrieve_PWC_Stanford_Widths(e1, e2, parameter_vector)

LA = parameter_vector(1);       % Distance to A*
LT = parameter_vector(2);       % Distance to prismatic joint
LP = parameter_vector(3);       % Distance to point P
L = parameter_vector(4);        % Length of link B
LC = parameter_vector(5);       % Distance to C*
mA = parameter_vector(6);       % Mass of link A
mB = parameter_vector(7);       % Mass of link B
mC = parameter_vector(8);       % Mass of link C
IA = parameter_vector(9);       % Moment of inertia of link A
IB = parameter_vector(10);      % Moment of inertia of link B
IC = parameter_vector(11);      % Moment of inertia of link C
Kg1 = parameter_vector(12);     % Gear ratio of harmonic drive
Kg2 = parameter_vector(13);     % Gear ratio of worm
Jm = parameter_vector(14);      % Motor moment of inertia
Jg1 = parameter_vector(15);      % Gear moment of inertia
dsp = parameter_vector(16);      % Displaement (pitch) of rack
Kt = parameter_vector(17);      % Motor torque constant
Ka = parameter_vector(18);      % Power amplifier volts-to-amps gain
coef = parameter_vector(19);    % Coefficient of the actual output torque over estimated friction
TA = parameter_vector(20); % Magnitude of torque pulse applied to link A
TcA = parameter_vector(21); % Coulomb friction torque applied to link A
TcAest = parameter_vector(22); % Estimated Coulomb friction torque applied to link A
TsA = parameter_vector(23); % Maximum static friction torque applied to link A
FAB = parameter_vector(24); % Magnitude of torque pulse applied to link B
FcAB = parameter_vector(25); % Coulomb friction force between link A and link B
FcABest = parameter_vector(26); % Estimated Coulomb friction force between link A and link B
FsAB = parameter_vector(27); % Maximum static friction torque between link A and link B

q2bar = 0; % L/2 - LP;

m11 = mA*L^2 + IA + mB*((LP+q2bar-L/2)^2 + LT^2) + mC*((LP+q2bar+LC)^2 + LT^2) + IC + Kg1^2*(Jm + Jg1);
m12 = -(mB+mC)*LT;
m21 = m12;
m22 = mB + mC + 4*Kg2^2*Jm/(dsp^2);

D = m11*m22 - m12*m21;

% Change signs for motions in other directions.

Ff1 = sign(e1 + eps)*TcA; % sign(e1+eps)*Kg1*Kt*Ka*V1;
Ff2 = sign(e2 + eps)*FcAB; % sign(e2+eps)*2*Kg2/dsp*Kt*Ka*V2;

F1 = Ff1 * coef; % sign(e1 + eps)*TA;
F2 = Ff2 * coef; % sign(e2 + eps)*FAB;

F1bar = (m22*F1 - m12*F2)/D;
Ff1bar = (m22*Ff1 - m12*Ff2)/D;
F2bar = (-m21*F1 + m11*F2)/D;
Ff2bar = (-m21*Ff1 + m11*Ff2)/D;

u1dot = F1bar - Ff1bar;
u2dot = F2bar - Ff2bar;

% Check if link A is initially stopped and only B will move.

u2dotstop = (F2 - Ff2)/m22;
Ff1stop = F1 - m12*u2dotstop;

if abs(Ff1stop) < abs(Ff1)
u1dot = 0;
u2dot = u2dotstop;
end

% Check if link B is initially stopped relative to A (B moves with A).

u1dotstop = (F1 - Ff1)/m11;
Ff2stop = F2 - m21*u1dotstop;
if abs(Ff2stop) < abs(Ff1)
    u1dot = u1dotstop;
    u2dot = 0;
end

errormin = 50;

dt1min = 1;
dt2min = 1;

grid1start = 0.1;
grid2start = 0.1;

% tic
for count = 1:1
    for n = 1:4
        for i = 1:20
            for j = 1:20

                dt1 = dt1min-grid1start/10^(n-2) + i*grid1start/10^(n-1);
                dt2 = dt2min-grid2start/10^(n-2) + j*grid2start/10^(n-1);

                if dt1 < dt2

                    q1star = 0.5*u1dot*dt1^2;
                    q2star = 0.5*u2dot*dt1^2;
                    u1star = u1dot*dt1;
                    u2star = u2dot*dt1;

                    F1barprime = -m12*F2/D;
                    F2barprime = m11*F2/D;

                    tf1 = -u1star/(F1barprime - Ff1);
                    if dt1+tf1 < dt2 & tf1 > 0

                        q1doublestar = q1star + u1star*tf1 + 0.5*(F1barprime - Ff1)*tf1^2;
                        q2doublestar = q2star + u2star*tf1 + 0.5*(F2barprime - Ff2)*tf1^2;
                        u2doublestar = u2star + (F2barprime - Ff2)*tf1;

                        F2barprime = F2/m22;
                        Ff2barprime = Ff2/m22;

                        q2triplestar = q2doublestar + u2doublestar*(dt2-dt1-tf1) + 0.5*(F2barprime - Ff2barprime)*(dt2-dt1-tf1)^2;
                        u2triplestar = u2doublestar + (F2barprime - Ff2barprime)*(dt2-dt1-tf1);

                        tf2 = u2triplestar/F2barprime;

                end
            end
        end
    end
end
q2quadstar = q2triplestar + u2triplestar*tf2 - 0.5*Ff2barprime*tf2^2;

error = abs(e1-q1doublestar) + abs(e2-q2quadstar);

else

q1doublestar = q1star + u1star*(dt2-dt1) + 0.5*(F1barprime - Ff1bar)*(dt2-dt1)^2;
q2doublestar = q2star + u2star*(dt2-dt1) + 0.5*(F2barprime - Ff2bar)*(dt2-dt1)^2;

tf1 = u1doublestar/Ff1bar;
tf2 = u2doublestar/Ff2bar;

if tf2 > tf1 & tf1 > 0

q1triplestar = q1doublestar + u1doublestar*tf1 - 0.5*Ff1bar*tf1^2;
q2triplestar = q2doublestar + u2doublestar*tf1 - 0.5*Ff2bar*tf1^2;
u2triplestar = u2doublestar - Ff2bar*tf1;

Ff2barprime = Ff2/m22;

tf2 = u2triplestar/Ff2barprime;

q2quadstar = q2triplestar + u2triplestar*tf2 - 0.5*Ff2barprime*tf2^2;

error = abs(e1-q1triplestar) + abs(e2-q2quadstar);

else

q1triplestar = q1doublestar + u1doublestar*tf2 - 0.5*Ff1bar*tf2^2;
u1triplestar = u1doublestar - Ff1bar*tf2;
q2triplestar = q2doublestar + u2doublestar*tf2 - 0.5*Ff2bar*tf2^2;

Ff1barprime = Ff1/m11;

tf1 = u1triplestar/Ff1barprime;

q1quadstar = q1triplestar + u1triplestar*tf1 - 0.5*Ff1barprime*tf1^2;

error = abs(e1-q1quadstar) + abs(e2-q2triplestar);

end

elseif dt2 < dt1

q1star = 0.5*u1dot*dt2^2;
\begin{align*}
q_{2\text{star}} &= 0.5u_{2\text{dot}}dt^2; \\
u_{1\text{star}} &= u_{1\text{dot}}dt; \\
u_{2\text{star}} &= u_{2\text{dot}}dt; \\
F_{1\text{barprime}} &= m_{22}F_1/D; \\
F_{2\text{barprime}} &= -m_{21}F_1/D; \\
t_2 &= -u_{2\text{star}}(F_{2\text{barprime}} - F_{f2\text{bar}}); \\
\text{if } dt_2 + t_2 < dt_1 \& t_2 > 0 \\
q_{1\text{doublestar}} &= q_{1\text{star}} + u_{1\text{star}}t_2 + 0.5(F_{1\text{barprime}} - F_{f1\text{bar}})t_2^2; \\
u_{1\text{doublestar}} &= u_{1\text{star}} + (F_{1\text{barprime}} - F_{f1\text{bar}})t_2; \\
q_{2\text{doublestar}} &= q_{2\text{star}} + u_{2\text{star}}t_2 + 0.5(F_{2\text{barprime}} - F_{f2\text{bar}})t_2^2; \\
F_{1\text{barprime}} &= F_1/m_{11}; \\
F_{f1\text{barprime}} &= F_{f1}/m_{11}; \\
q_{1\text{triplestar}} &= q_{1\text{doublestar}} + u_{1\text{doublestar}}(dt_1 - dt_2 - t_2) + 0.5(F_{1\text{barprime}} - F_{f1\text{barprime}})(dt_1 - dt_2 - t_2)^2; \\
u_{1\text{triplestar}} &= u_{1\text{doublestar}} + (F_{1\text{barprime}} - F_{f1\text{barprime}})(dt_1 - dt_2 - t_2); \\
t_1 &= u_{1\text{triplestar}}/F_{f1\text{barprime}}; \\
q_{1\text{quadstar}} &= q_{1\text{triplestar}} + u_{1\text{triplestar}}t_1 - 0.5F_{f1\text{barprime}}t_1^2; \\
error &= \text{abs}(e_1 - q_{1\text{quadstar}}) + \text{abs}(e_2 - q_{2\text{doublestar}}); \\
\text{else} \\
q_{1\text{doublestar}} &= q_{1\text{star}} + u_{1\text{star}}(dt_1 - dt_2) + 0.5(F_{1\text{barprime}} - F_{f1\text{bar}})(dt_1 - dt_2)^2; \\
u_{1\text{doublestar}} &= u_{1\text{star}} + (F_{1\text{barprime}} - F_{f1\text{bar}})(dt_1 - dt_2); \\
q_{2\text{doublestar}} &= q_{2\text{star}} + u_{2\text{star}}(dt_1 - dt_2) + 0.5(F_{2\text{barprime}} - F_{f2\text{bar}})(dt_1 - dt_2)^2; \\
u_{2\text{doublestar}} &= u_{2\text{star}} + (F_{2\text{barprime}} - F_{f2\text{bar}})(dt_1 - dt_2); \\
t_1 &= u_{1\text{doublestar}}/F_{f1\text{bar}}; \\
t_2 &= u_{2\text{doublestar}}/F_{f2\text{bar}}; \\
\text{if } t_2 > t_1 \& t_1 > 0 \\
q_{1\text{triplestar}} &= q_{1\text{doublestar}} + u_{1\text{doublestar}}t_1 - 0.5F_{f1\text{bar}}t_1^2; \\
q_{2\text{triplestar}} &= q_{2\text{doublestar}} + u_{2\text{doublestar}}t_1 - 0.5F_{f2\text{bar}}t_1^2; \\
u_{2\text{triplestar}} &= u_{2\text{doublestar}} - F_{f2\text{bar}}t_1; \\
F_{f2\text{barprime}} &= F_{f2}/m_{22}; \\
t_2 &= u_{2\text{triplestar}}/F_{f2\text{barprime}}; \\
q_{2\text{quadstar}} &= q_{2\text{triplestar}} + u_{2\text{triplestar}}t_2 - 0.5F_{f2\text{barprime}}t_2^2; \\
\end{align*}
error = abs(e1-q1triplestar) + abs(e2-q2quadstar);

else

q1triplestar = q1doublestar + u1doublestar*tf2 - 0.5*Ff1bar*tf2^2;
u1triplestar = u1doublestar - Ff1bar*tf2;
q2triplestar = q2doublestar + u2doublestar*tf2 - 0.5*Ff2bar*tf2^2;

Ff1barprime = Ff1/m11;

tf1 = u1triplestar/Ff1barprime;
q1quadstar = q1triplestar + u1triplestar*tf1 - 0.5*Ff1barprime*tf1^2;
error = abs(e1-q1quadstar) + abs(e2-q2triplestar);
end
end
else

q1star = 0.5*u1dot*dt1^2;
q2star = 0.5*u2dot*dt2^2;
u1star = u1dot*dt1;
u2star = u2dot*dt2;

tf1 = u1star/Ff1bar;
tf2 = u2star/Ff2bar;

if tf2 > tf1 & tf1 > 0

q1doublestar = q1star + u1star*tf1 - 0.5*Ff1bar*tf1^2;
q2doublestar = q2star + u2star*tf1 - 0.5*Ff2bar*tf1^2;
u2doublestar = u2star - Ff2bar*tf1;

Ff2barprime = Ff2/m22;

tf2 = u2doublestar/Ff2barprime;
q2triplestar = q2doublestar + u2doublestar*tf2 - 0.5*Ff2barprime*tf2^2;
error = abs(e1-q1doublestar) + abs(e2-q2triplestar);
else

q1doublestar = q1star + u1star*tf2 - 0.5*Ff1bar*tf2^2;
u1doublestar = u1star - Ff1bar*tf2;
q2doublestar = q2star + u2star*tf2 - 0.5*Ff2bar*tf2^2;

end
Ff1barprime = Ff1/m11;

tf1 = u1doublestar/Ff1barprime;

q1triplestar = q1doublestar + u1doublestar*tf1 - 0.5*Ff1barprime*tf1^2;

error = abs(e1-q1triplestar) + abs(e2-q2doublestar);

if (error < errormin)
    dt1minij = dt1;
    dt2minij = dt2;
    errormin = error;
end

dt1min = dt1minij;
dt2min = dt2minij;
end
end

end
end

%toc;

dt1 = dt1min;
if dt1 <= grid1start/10^2 + eps
dt1 = 0;
end

dt2 = dt2min;
if dt2 <= grid2start/10^2 + eps
dt2 = 0;
end
CAPPINGFUNCTION.m

function delta_y_cap = CappingFunction(delta_y_adapt, delta_y_initial)

if(sign(delta_y_initial + delta_y_adapt) ~= sign(delta_y_adapt))
    delta_y_cap = 0;
elseif(abs(delta_y_initial + delta_y_adapt) < abs(delta_y_adapt))
    delta_y_cap = delta_y_adapt + delta_y_initial;
else
    delta_y_cap = delta_y_initial;
end

delta_y_cap;

RANDPREDICT.m

function [rand_value, seed_value] = RandPredict(seed_value)

rand_value = mod((19562.7.*seed_value.*(10000-500-
    seed_value)+500+(seed_value+5).^2),10000)/10000;
seed_value = seed_value + 1;