2007

The PHANTOM Omni as an under-actuated robot

John Albert Beckman
Iowa State University

The PHANTOM Omni as an under-actuated robot

by

John Albert Beckman

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Mechanical Engineering

Program of Study Committee:
Greg Luecke, Major Professor
Qingze Zou
Nicola Elia

Iowa State University

Ames, Iowa

2007

UMI Number: 1447474

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF CODES

CHAPTER 1. OVERVIEW

With the increased use of virtual environments in analyzing science and engineering problems, methods for interaction with objects has become of utmost importance. While the graphics used in such environments has greatly increased to near real-life quality in some applications, it is very important to give the user accurate tactile information about the objects in the environment. These interaction are accomplished using haptic devices, one such being the PHANTOM Omni (from here on to be referred to simply as the PHANTOM) made by Sensable, Incorporated. This small desktop device has the ability to apply Cartesian forces (in the x, y, or z directions) and also has an easy to use programmable interface utilizing C++.

1.1 Past methods

Many interactions programmed with the PHANTOM are created based in a graphical nature, partly because the structure of the code for controlling the device is very close to the structure of an OpenGL based graphical program. For example, a simple OpenGL program can be created to draw a cube on the screen centered at $(0, 0, 0)$ with a side length of one. In a similar manner, the PHANTOM can be programmed to interperet walls where the edges of the cube are, giving the user the sensation of either feeling a cube located in space or being stuck inside a cube of a given size. When using both graphic and haptic interaction, it is important to have the objects aligned in space. For example, if a cube is drawn centered at $(0, 0, 0)$ and the haptic interaction believes the cube is centered at $(50, 0, 0)$, the resulting program will confuse the user, since it sees one object but feels a completely different object.

1.2 New methods

Instead of using the graphical method of using the PHANTOM to interact with objects, a method using vector math and classical control theory will be explored. As an explanatory example of this method, suppose an infinite cylinder exists in the virtual environment (perhaps a post or support that extends beyond the important area of the environtment). In order to explain where the cylinder is located, three pieces of information are needed: its diameter or radius (a scalar), the direction in which the axis of the cylinder points (a vector) and a point in space that lies on the axis of the cylinder (a vector). With these three quantities, all points on the surface of the cylinder can be determined. The location of the haptic device is readily determined, as can its position relative to the cylinder. Using a simple proportional controller, an algorithm for moving the end of the PHANTOM to the nearest point on the surface can be created. The PHANTOM will be able to move along the surface of the cylinder with no resistance, but as soon as it is moved away from the surface, the controller will force it back (similar to a spring force).

Even though this type of interaction is useful, it does not give a full range of tactile experiences. The PHANTOM can sense six degrees-of-freedom—x-, y- and z-position as well as its orientation about these same axes—but can only actuate on three of them (the positioning degrees of freedom). The other measured degrees of freedom can be used to influence the position of the PHANTOM. Two method will be looked at for accounting this. The first method is a more graphical approach. Two points on the surface of the virtual object: the closest physical point (which is what the previous constraint did) and the point that has the same outward normal as the orientation of the tip of the PHANTOM. A Bezier curve is used to connect these two points and by varying and "influence" factor, the point the PHANTOM is forced toward a point somewhere in between the two calculated points. The second method is a more general, control theory approach. By using the concept of a virtual manipulator, the PHANTOM is force toward a virtual robot. For example, a planar circle constraint can be thought of as a one revolute joint (a pivot/hinge) robot with a given arm length (the radius). By applying adequate theory, the position of the virtual robot can be moved to the "closest"

point the physical robot. The "closest" point is not uniquely defined—it is a point that tries to minimize the combination of position error and orientation error. These two errors can be weighted to have one count more that the other, if desired.

1.3 Structure of this report

A basic knowledge of linear algebra is needed to understand most of this report. Not in terms of theorems about vector spaces or special matrix properties, but simply addition, subtraction, multiplation and other basic operations operations (dot product, cross product, determinant, etc). In case some of these operations are not clear, consult Appendix A, containing a brief overview of these points. Since the main method of constraining the PHANTOM uses classical control theroy, Appendix B contains a very simplified version of these concepts. It discusses system response and basic controller structure.

In order to examine the PHANTOM as a robot, kinematics associated with robots are presented. Next, an overview of the PHANTOM device itself is given, including the forward kinematics (find the position and orientation of tip based on the joint angles), the inverse kinematics (determine the joint angles based on the position and orientation of the tip) and an overview of how to program the PHANTOM. The next section develops the spatial constraints for various shapes (chosen to represent several possible robot configurations) and gives code necessary to implement surface following. Finally, the constraints accounting for the three un-actuated degrees-of-freedom are taking into account, both in the theory behind them and the implementation into code.

CHAPTER 2. ROBOT KINEMATICS

In order to adequately describe robots, a method for expressing robot displacements and orientations is needed. The method used involves 4×4 homogeneous transformation matrices. Once a firm grasp of expressing positions and orientations is obtained, a convention for positioning reference frames on the robot is discussed. Once reference frames are established on the robot, the forward kinematics of the robot can be determined—the set of equations that relate the robot's position and orientation to the values of joint angles/displacements. For a more in depth look at kinematics, see [2]. For a more in depth look at the design of kinematic systems (designing a linkage/robot to travel between desired points), see [7].

This section is intended as a review of the mathematical theory behind kinematic calculations

2.1 Expressing positions and orientations

Before robots can be understood, consider the simpler case of describing two arbitrarily placed coordinate frames. Consider figure 2.1, depicting two sets of coordinate axes.

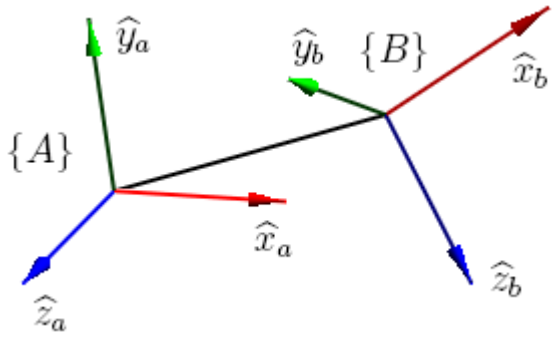

Figure 2.1 Two arbitrary frames

To find the position of frame $\{B\}$ relative to frame $\{A\}$, one must simply measure the coordinates of the origin of $\{B\}$ with respect to the unit-length axes of $\{A\}$. That is, the location of the origin of $\{B\}$ with respect to coordinate system $\{A\}$ can be defined as

$$ {}^{A}P_{Borigin} = \begin{bmatrix} {}^{A}X_B \\ {}^{A}Y_B \\ {}^{A}Z_B \end{bmatrix} \tag{2.1} $$

Where the quantities ${}^{A}X_B$, ${}^{A}Y_B$, and ${}^{A}Z_B$ are the lengths along the axes in frame $\{A\}$.

In a similar manner, the orientation of frame $\{B\}$ with respect to frame $\{A\}$ can also be determined with respect to the axes of frame $\{A\}$. In this case, each axis of frame $\{B\}$ must be specified with respect to the x, y, and z axes in frame $\{A\}$. The orientation of frame $\{B\}$ with respect to $\{A\}$ can be written in matrix form as

$$ {}^{A}R_B = \begin{bmatrix} \widehat{X}_b \cdot \widehat{X}_a & \widehat{Y}_b \cdot \widehat{X}_a & \widehat{Z}_b \cdot \widehat{X}_a \\ \widehat{X}_b \cdot \widehat{Y}_a & \widehat{Y}_b \cdot \widehat{Y}_a & \widehat{Z}_b \cdot \widehat{Y}_a \\ \widehat{X}_b \cdot \widehat{Z}_a & \widehat{Y}_b \cdot \widehat{Z}_a & \widehat{Z}_b \cdot \widehat{Z}_a \end{bmatrix} \tag{2.2} $$

The dot products in equation (2.2) obtain the length of the projection of each axis of frame $\{B\}$ on the axes of frame $\{A\}$. Instead of this representation, the orientation can be thought of as the combination of rotations frame $\{B\}$ undergoes about the three axes of frame $\{A\}$ (x, y and z) to achieve the desired orientation.

When rotating around a single axis, equation (2.2) simplifies considerably. For example, a rotation about $\widehat{X}_a$ by an angle θ can be expressed as the matrix

$$ R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.3} $$

Notice that the $(3,3)$ element of equation (2.3) is a 1, signifying that the z axis of frame $\{B\}$ is pointed in the same direction as that of frame $\{A\}$. In a similar manner, the orientations when rotated about the x or y axes can be expressed as

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \tag{2.4}$$

or

$$R_y(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \tag{2.5}$$

In equation (2.4) is seen in the $(1,1)$ element, denoting that the x axis of $\{B\}$ points in the same direction as the x axis of frame $\{A\}$. Likewise in equation (2.5), the one in element $(2,2)$ inplies that the y axis of $\{B\}$ points in the same direction as the y axis of $\{A\}$.

The 3×3 rotation matrix presented in equation (2.2) can be thought of as a rotation about $\widehat{X}_a$ by an angle α, then a rotation about $\widehat{Y}_a$ by an angle β followed by a final rotation about $\widehat{Z}_a$ by an angle θ. This final rotation can be expressed as

$$\begin{aligned} {}^A_B R_{XYZ}(\alpha,\beta,\theta) &= R_Z(\theta)\times R_Y(\beta)\times R_X(\alpha) \\ &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \\ &= \begin{bmatrix} c\theta c\beta & -s\theta c\alpha - c\theta s\beta s\alpha & -s\theta s\alpha + c\theta s\beta c\alpha \\ s\theta c\beta & c\theta c\alpha - s\theta s\beta s\alpha & c\theta s\alpha + s\theta s\beta c\alpha \\ -s\beta & -c\beta s\alpha & c\beta c\alpha \end{bmatrix} \end{aligned} \tag{2.6}$$

Care must be taken when performing these multiplications, for matrix multiplication is not communative, as demonstrated in equation (2.7)

$$\begin{aligned} {}^A_B R_{ZYX}(\theta,\beta,\alpha) &= R_Z(\alpha)\times R_Y(\beta)\times R_X(\theta) \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} c\beta c\alpha & -c\beta s\alpha & s\beta \\ -s\theta s\beta c\alpha + c\theta s\alpha & s\theta s\beta s\alpha + c\theta c\alpha & s\theta c\beta \\ -c\theta s\beta c\alpha - s\theta s\alpha & c\theta s\beta s\alpha - s\theta c\alpha & c\theta c\beta \end{bmatrix} \end{aligned} \tag{2.7}$$

However, it should be noted that another rotation scheme is possible. Instead of rotating about the fixed axes of frame $\{A\}$, it is possible to rotate about the axes of moving frame $\{B\}$.

This is commonly referred to as an Euler angle move.

2.2 Operations on spatial representations

Now that it is known how to express one frame relative to another, it is important to be able to make connections between frames. In order to understate the kinematics of robots, a method must be used to account for a serial chain of frames (as is the case in robots). A widely used method is that of the 4×4 homogeneous matrix transformation. This matrix accounts for both the displacement of one frame relative to another (such as in equation (2.1)) and the orientation (as in equation (2.2)). This information is combined into a single 4×4 matrix which has the form

$$ {}^{A}_{B}T = \begin{bmatrix} {}^{A}_{B}R & {}^{A}P_{B\,origin} \\ 0\ 0\ 0 & 1 \end{bmatrix} \tag{2.8} $$

As a simple example, suppose frame $\{B\}$ has the same orientation as frame $\{A\}$ but is transposed by $\begin{bmatrix} p_x & p_y & p_z \end{bmatrix}^T$. The 4×4 homogeneous transform for this translation would be given by

$$ {}^{A}_{B}T = \begin{bmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.9} $$

As a comparable example, suppose the origin of frame $\{B\}$ is coincident with the origin of frame $\{A\}$, but is rotated 30^o about the Z_a axis. This homogeneous transformation would be given by

$$ {}^{A}_{B}T = \begin{bmatrix} 0.866 & -0.5 & 0 & 0 \\ 0.5 & 0.866 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.10} $$

In a similar manner, these two types can be combined, which would describe a frame located at $\begin{bmatrix} p_x & p_y & p_z \end{bmatrix}^T$ and rotated 60^o about the X_a axis:

$$ {}^{A}_{B}T = \begin{bmatrix} 1 & 0 & 0 & p_x \\ 0 & 0.5 & -0.866 & p_y \\ 0 & 0.866 & 0.5 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.11} $$

The 4×4 homogeneous transform provides a simple, easy to program method for performing spatial manipulations. To illustrate this scenario, let frame $\{A\}$ be fixed and let frame $\{B\}$ be moving with respect to $\{A\}$. Furthermore, suppose there is a point or vector ${}^{B}P$ in frame $\{B\}$ whose position is needed to be known with respect to frame $\{A\}$, such as shown in figure 2.2 below. This can be easily computed using

$$ {}^{A}P = \begin{bmatrix} {}^{A}p_x \\ {}^{A}p_y \\ {}^{A}p_z \\ 1 \end{bmatrix} = \begin{bmatrix} {}^{A}_{B}R & {}^{A}P_{B\,origin} \\ 0\ 0\ 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{B}p_x \\ {}^{B}p_y \\ {}^{B}p_z \\ 1 \end{bmatrix} \tag{2.12} $$

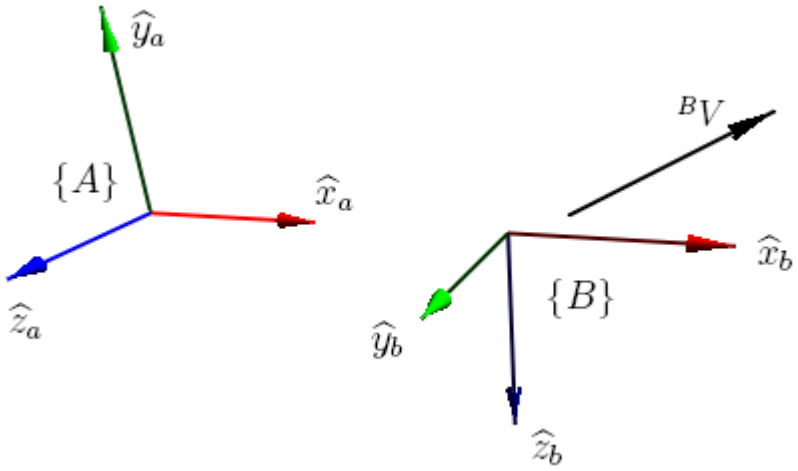

Figure 2.2 Two arbitrary frames and a vector

2.3 Kinematics of Serial Chains

A serial chain robot is simply a collection of coordinate frames that progress along the links of a robot. That being the case, mapping vectors from the outter-most frame to the base of the robot can be handled similarly to that of equation (2.12) For example, figure 2.3 shows a serial robot consisting of three frames: a base, an intermediate and an outermost.

Suppose the components the vector ${}^{C}\vec{V}$, whose components are known in frame $\{C\}$, are to be determined in frame $\{A\}$. This is identical to having arbitrary spatial in space, but they

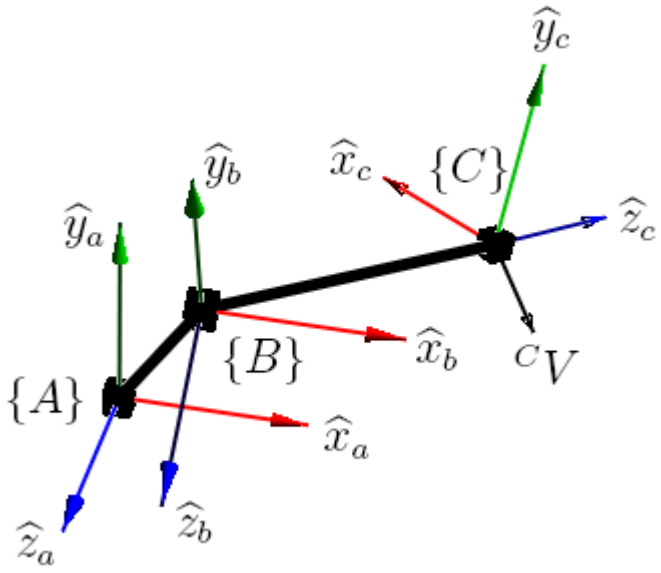

Figure 2.3 A generic serial chain robot

are just associated with links of a robot in this case. In order to perform this mapping, ${}^{C}\vec{V}$ must first be mapped into frame $\{B\}$, as seen in (2.13).

$$
{}^{B}V = \begin{bmatrix} {}^{B}V_x \\ {}^{B}V_y \\ {}^{B}V_z \\ 1 \end{bmatrix} = \begin{bmatrix} {}^{B}_{C}R & {}^{B}P_{C\,origin} \\ 0\ 0\ 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{C}V_x \\ {}^{C}V_y \\ {}^{C}V_z \\ 1 \end{bmatrix} \tag{2.13}
$$

$$
{}^{B}V = {}^{B}_{C}T\,{}^{C}V
$$

Next, this vector in $\{B\}$ can be mapped into frame $\{A\}$.

$$
{}^{A}V = \begin{bmatrix} {}^{A}V_x \\ {}^{A}V_y \\ {}^{A}V_z \\ 1 \end{bmatrix} = \begin{bmatrix} {}^{A}_{B}R & {}^{A}P_{B\,origin} \\ 0\ 0\ 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{B}V_x \\ {}^{B}V_y \\ {}^{B}V_z \\ 1 \end{bmatrix} \tag{2.14}
$$

$$
{}^{A}V = {}^{A}_{B}T\,{}^{B}V
$$

The expression for ${}^{B}\vec{V}$ from equation (2.13) can be substituted into equation (2.14).

$$\begin{aligned} {}^{A}V &= {}^{A}_{B}T\,{}^{B}_{C}T\,{}^{C}V \\ {}^{A}V &= {}^{A}_{C}T\,{}^{C}V \end{aligned} \tag{2.15}$$

Notice that the frame designating subscripts and superscripts cancel from lower left to upper right (i.e. The subscript "'B"' of ${}^{A}_{B}T$ cancels with the superscript "'B"' of ${}^{B}_{A}T$). If these do not cancel out, the resulting calculation will not be in the desired frame. The resulting transform ${}^{A}_{C}T$ describes the position and orientation of frame $\{C\}$ in frame $\{A\}$.

2.4 Denavit and Hartenberg (DH) Notation

Once general spatial kinematics are understood, a method of consistency when describing robots is needed. One of the most widely used methods is known as Denavit-Hartenberg notation, named after two early researchers in analytic design of linkages in the 1950s and 60s (see [4]). They created a standardized method of analyzing links of mechanisms (since robots had not emerged yet) involving standandized measurements. Before these parameters can be determined, frames must be fixed to the robot (or mechanism).

2.4.1 Frame attachement procedure and DH parameters

The number of frames required for a given robot (or mechanism) is easily found as the number of joints plus one. This extra frame represents the base of the robot, in which all positions and orientations are ultimately found with respect to. Although its position is arbitrary, it is usually helpful to place this base frame at the at-rest position of the first joint (i.e. when the joint has zero displacement). For the intermediate (before the end of the robot) links in the serial chain, frames are affixed such that the $\hat{Z}$ axis of the frame is coincident with the joint axis of the link. For revolute joints, this means $\hat{Z}$ is along the axis of rotation, and for prismatic joints this means $\hat{Z}$ is along the direction of movement. The $\hat{X}$ axis of the frame points along the common normal between the two joint axes.

With these standardized frame attachments, the paramters describing them can be determined. Consider figure 2.4, depicting an arbitrary revolute linkage. Four parameters can be

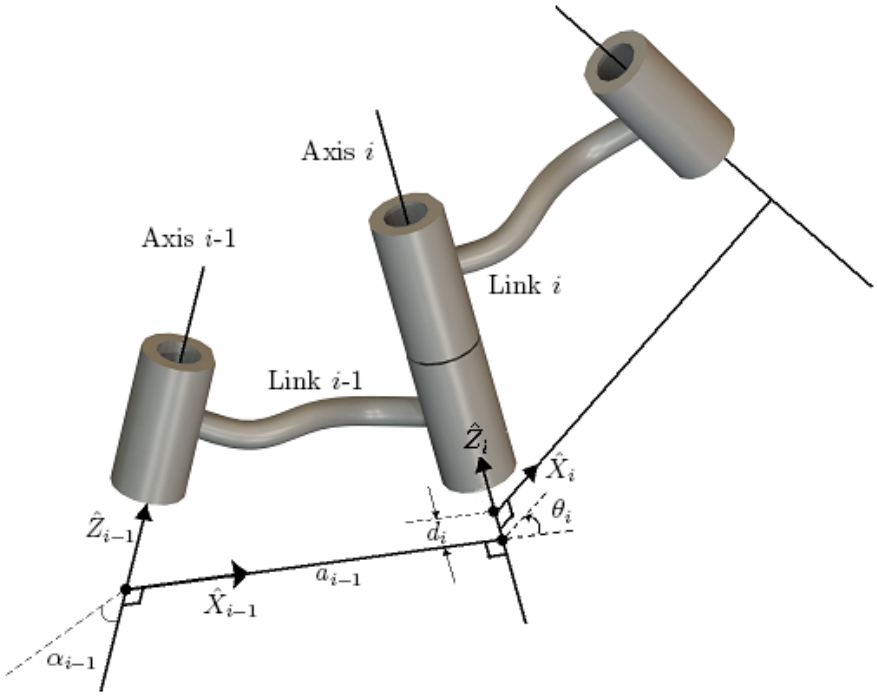

Figure 2.4 Arbitrary links in a serial chain

used to adequately describe the position and orientation of frame i with respect to *i-1*. These four parameters are the link length (a_i), the link twist (α_i), the joint offset (d_i) and the joint angle (θ_i).

These four parameters are known as the DH parameters and each measure is defined by convention. The link length a_i is defined as the distance from $\hat{Z}_i$ to $\hat{Z}_{i+1}$ measured along $\hat{X}_i$. This distance is the lenght along the common normal of the joint axes. The link twist α_i is the angle from $\hat{Z}_i$ to $\hat{Z}_{i+1}$ measured along $\hat{X}_i$. Given a link, this parameter measures the angle between the two joints connecting it to other links. The joint offset d_i is the distance from $\hat{X}_{i-1}$ to $\hat{X}_i$ masured along $\hat{Z}_i$—the distance along $\hat{Z}_i$ the two frames are offset. Finally, the joint angle θ_i is the angle between $\hat{X}_{i-1}$ and $\hat{X}_i$ measured along $\hat{Z}_i$. The DH paramters can be arranged into a table, as seen in the sample table 2.1 The number of rows in this table will correspond with the number of joints in the robot (or mechanism).

Table 2.1 Example DH parameter table

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	0	θ_1
2	90^o	0	d_2	0
3	0	0	L_2	θ_3
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

2.4.2 Link transforms

With the DH parameters determined, a transform must be determined to represent frame i in terms of frame *i-1*. This transform can be thought of as the product of two separate transforms: a link transform and a joint transform. Each of these transforms are classified as screw transforms, since they contain a translation along and rotation about an axis. For the case of the link transform, frame i is located a distance a_{i-1} along and rotated an angle α_{i-1} around axis $\hat{X}_{i-1}$. Expressing this as a 4×4 homogenous transform

$$X\left(a_{i-1},\, \alpha_{i-1}\right) = \begin{bmatrix} 1 & 0 & 0 & a_{i-1} \\ 0 & \cos\alpha_{i-1} & -\sin\alpha_{i-1} & 0 \\ 0 & \sin\alpha_{i-1} & \cos\alpha_{i-1} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.16}$$

In a similar analysis, the joint transform can be a similarly described as a displacement along $\hat{Z}_i$ a distance d_i and a rotation about $\hat{Z}_i$ an angle θ_i. In matrix form

$$Z\left(d_i,\, \theta_i\right) = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & 0 \\ \sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.17}$$

The two expression in (2.16) and (2.17) can be combined into a single expression

$$
\begin{aligned}
T\left(\alpha_{i-1},\, a_{i-1},\, d_i,\, \theta_i\right) &= X\left(\alpha_{i-1},\, a_{i-1}\right) Z\left(d_i,\, \theta_i\right) \\
&= \begin{bmatrix} 1 & 0 & 0 & a_{i-1} \\ 0 & \cos\alpha_{i-1} & -\sin\alpha_{i-1} & 0 \\ 0 & \sin\alpha_{i-1} & \cos\alpha_{i-1} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & 0 \\ \sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_{i-1} \\ \cos\alpha_{i-1}\sin\theta_i & \cos\alpha_{i-1}\cos\theta_i & -\sin\alpha_{i-1} & -d_i\sin\alpha_{i-1} \\ \sin\alpha_{i-1}\sin\theta_i & \sin\alpha_{i-1}\cos\theta_i & \cos\alpha_{i-1} & d_i\cos\alpha_{i-1} \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
\tag{2.18}
$$

Because of the length of this transform, it is often written as

$$
T\left(\alpha_{i-1},\, a_{i-1},\, d_i,\, \theta_i\right) = \begin{bmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ c\alpha_{i-1}s\theta_i & c\alpha_{i-1}c\theta_i & -s\alpha_{i-1} & -d_i s\alpha_{i-1} \\ s\alpha_{i-1}s\theta_i & s\alpha_{i-1}c\theta_i & c\alpha_{i-1} & d_i c\alpha_{i-1} \\ 0 & 0 & 0 & 1 \end{bmatrix}
\tag{2.19}
$$

where "s" stands for the sine function and "c" stands for the cosine function. At times, even the "θ" is omitted, leaving just the index of which angle to use.

2.5 Inverse kinematics

Inverse kinematics is, like its name suggests, the opposite of the forward kinematics described up to this point. In forward kinematics, the joint values of the robot were known and the resulting position and orientation was to be determined. In inverse kinematics, the position and orientation of the robot is known and the joint values must be determined. Since each robot is created differently (different dimensions, different frame descriptions, etc), there is no one method for determining the joint values.

One thing common to all inverse kinematic calculations is looking for terms to solve for a given joint variable. For example, consider a simple, planar, one joint robot which has forward

kinematics

$$T(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & L_1\cos\theta \\ \sin\theta & \cos\theta & 0 & L_1\sin\theta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.20}$$

When this transform is known, the angle θ can be found using a four quandrant arctangent, called "atan2" the C++ and Matlab programming languages, using T_{11} and T_{21}:

$$\theta = \tan^{-1}\frac{T_{21}}{T_{11}} \tag{2.21}$$

As the number of joints in the robot increases, the difficulty in finding a closed-form solution (one where general equations can be written, instead of using numeric approximations) increases.

CHAPTER 3. PHANTOM OMNI

The PHANTOM Omni is six degree-of-freedom haptic manipulator. Originally developed as a thesis project at MIT, the PHANTOM was designed for use in "touching" and manipulating three-dimensional data. Besides being able to read the position values for it's six revolute joints, the PHANTOM also contains three motors attached to the first three joints, allowing for spacial forces (forces controlling the position of the tip of the stylus in cartesian space). It is because of this force application that researchers in virtual environments have used the PHANTOM to interact with virtual objects.

Figure 3.1 The PHANTOM Omni

The important thing about this particular 6R robot design is the segmentation of position and orientation—the first three joints are used to position the end stylus and the remaining three joints are used solely for finding the orientation. These final three joints all intersect at the same point, and can be interpretted as a spherical joint. This layout is similar to the PUMA 560 robot, a 6R industrial robot developed by Unimation in 1978. The major difference between these two robots (apart from their sizes) is that the PHANTOM contains only three

drive motors. This means that only the position of the stylus can be set by the computer. The PUMA, on the other hand, has motors on all of its joints, allowing for arbitary positioning and orientating.

3.1 Kinematics

3.1.1 Forward

In order to describe the kinematics of the PHANTOM, the frames locations must be determined. Figure 3.2 shows the PHANTOM with the frame axes fixed onto it.

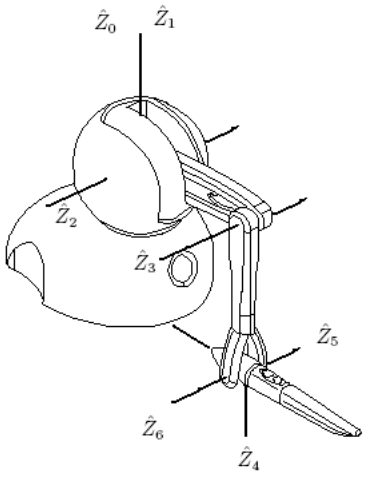

Figure 3.2 The PHANTOM with frames axes

While it is useful to see where the frames fit on the actual robot, this can prove to be cumbersome to analyze. Figure 3.3 shows just the axes with the respective DH parameters.

From figure 3.3, the DH parameters can be determined, as listed in table 3.1. The other important quantity is to determine the transform between the workframe ("w") and frame zero. This is determined by observation and measurement

$$ {}^{w}_{0}T = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 30mm \\ 1 & 0 & 0 & -176mm \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.1} $$

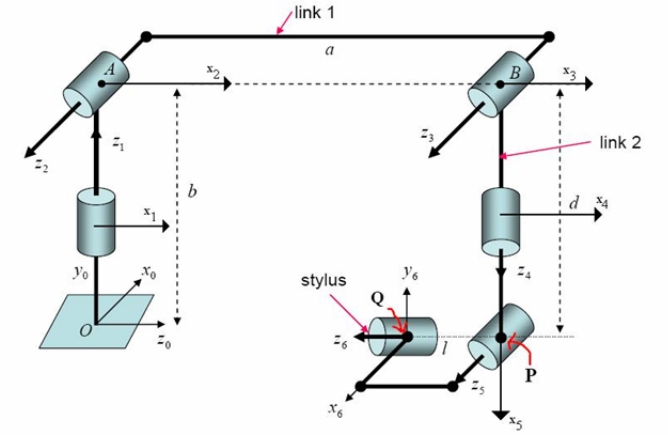

Figure 3.3 The axes of the PHANTOM

Table 3.1 PHANTOM Omni DH table

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	0	θ_1
2	90^o	0	0	θ_2
3	0	127.508mm	0	θ_3
4	90^o	0	149.352mm	θ_4
5	-90^o	0	0	θ_5
6	90^o	0	0	θ_6

With the DH parameters and this relation to the workspace known, it is possible to formulate the forward kinematics. This begins by writing out the matrix equation for the position of the end as a function of the six joint variables

$$ {}^{w}_{6}T = {}^{w}_{0}T\,{}^{0}_{1}T\,{}^{1}_{2}T\,{}^{2}_{3}T\,{}^{3}_{4}T\,{}^{4}_{5}T\,{}^{5}_{6}T \tag{3.2} $$

Because of the configuration of this device, it is possible to separate equation (3.2) into two distinct parts: a portion which positions the end, and a portion which orients the end.

$$ {}^{w}_{6}T = {}^{w}_{0}T \left[R\left(\theta_1,\, \theta_2,\, \theta_3\right) \right] \left[W\left(\theta_4,\, \theta_5,\, \theta_6\right) \right] \tag{3.3} $$

In equation (3.3), $R\left(\theta_1,\, \theta_2,\, \theta_3\right)$ is the position function and $W\left(\theta_4,\, \theta_5,\, \theta_6\right)$ is the orientation function (often called the wrist movement function). Each of these expressions also have the

other DH parameters (link length, link twist, joint offset) in them, but since they do not change, they are internal elements.

In order to compute these matrices, a combination of link transforms and screw transforms are used. The positioning function R needs to include both link lengths (a_2 and d_4) along with the first three angles. The orientating function W must contain only the last three joint angles. By separating the operations of the screw matrices (they are, after all, a combination of a translation and a rotation), expressions can be determined

$$\begin{aligned} R\left(\theta_1,\, \theta_2,\, \theta_3\right) &= {}^{0}_{1}T\,{}^{1}_{2}T\,{}^{2}_{3}T\,TX\left(\alpha_3,\, a_3\right) Z\left(d_4, 0\right) \\ W\left(\theta_4,\, \theta_5,\, \theta_6\right) &= Z\left(0, \theta_4\right)\,{}^{4}_{5}T\,{}^{5}_{6}T \end{aligned} \tag{3.4}$$

In equation (3.4), what would have been ${}^{3}_{4}T$ was broken into its screw matrix components, allowing for θ_4 to be separated from the position element d_4. Substituting in the values from the DH table into the expression yields the matrices

$$R\left(\theta_1,\, \theta_2,\, \theta_3\right) = \begin{bmatrix} c_1c_2c_3 - c_1s_2s_3 & s_1 & c_1c_2s_3 + c_1s_2c_3 & -\left(-c_1c_2s_3 - c_1s_2c_3\right)d_4 + c_1c_2a_2 \\ s_1c_2c_3 - s_1s_2s_3 & -c_1 & s_1c_2s_3 + s_1s_2c_3 & -\left(-s_1c_2s_3 - s_1s_2c_3\right)d_4 + s_1c_2a_2 \\ s_2c_3 + c_2s_3 & 0 & s_2s_3 - c_2c_3 & -\left(-s_2s_3 + c_2c_3\right)L_2 + s_2L_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

$$W\left(\theta_4,\, \theta_5,\, \theta_6\right) = \begin{bmatrix} c_4c_5c_6 - s_4s_6 & -c_4c_5c_6 - s_4c_6 & c_4s_5 & 0 \\ s_4c_5c_6 + c_4s_6 & -s_4c_5s_6 + c_4c_6 & s_4s_5 & 0 \\ -s_5c_6 & s_5s_6 & c_5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.6}$$

The complete forward kinematics are found as

$$\begin{aligned} {}^{w}_{6}T\left(\theta_1,\, \theta_2,\, \theta_3,\, \theta_4,\, \theta_5,\, \theta_6\right) &= {}^{w}_{0}T\, R\left(\theta_1,\, \theta_2,\, \theta_3\right) W\left(\theta_4,\, \theta_5,\, \theta_6\right) \\ &= \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{3.7}$$

where

$$
\begin{aligned}
r_{11} &= (s_1c_2c_3 - s_1s_2s_3)(c_4c_5c_6 - s_4s_6) - c_1(s_4c_5c_6 + c_4s_6) - (s_1c_2s_3 + s_1s_2c_3)s_5c_6 \\
r_{12} &= (s_1c_2c_3 - s_1s_2s_3)(-c_4c_5s_6 - s_4c_6) - c_1(-s_4c_5s_6 + c_4c_6) + (s_1c_2s_3 + s_1s_2c_3)s_5s_6 \\
r_{13} &= (s_1c_2c_3 - s_1s_2s_3)c_4c_5 - c_1s_4s_5 + (s_1c_2s_3 + s_1s_3c_3)c_5 \\
p_x &= -(-s_1c_2s_3 - s_1s_2c_3)d_4 + s_1c_2a_2 \\
\\
r_{21} &= (s_2c_3 + c_2s_3)(c_4c_5c_6 - s_4s_6) - (s_2s_3 - c_2c_3)s_5c_6 \\
r_{22} &= (s_2c_3 + c_2s_3)(-c_4c_5s_6 - s_4c_6) + (s_2s_3 - c_2c_3)s_5s_6 \\
r_{23} &= (s_2c_3 + c_3s_3)c_4s_5 + (s_2s_3 - c_2c_3)c_5 \\
p_y &= -(-s_2s_3 + c_2c_3)d_4 + s_2a_2 + 30 \\
\\
r_{31} &= (c_1c_2c_3 - c_1s_2s_3)(c_4c_5c_6 - s_4s_6) + s_1(s_4c_5c_6 + c_4s_6) - (c_1c_2s_3 + c_1s_2c_3)s_5c_6 \\
r_{32} &= (c_1c_2c_3 - c_1s_3s_3)(-c_4c_5s_6 - s_4c_6) + s_1(-s_4c_5s_6 + c_4s_6) + (c_1c_2s_3 + c_1s_2c_3)s_5s_6 \\
r_{33} &= (c_1c_2c_3 - c_1s_2s_3)c_4s_5 + s_1s_4s_5 + (c_1c_2s_3 + c_1s_3c_3)c_5 \\
p_z &= -(-c_1c_2s_3 - c_1s_3c_3)d_4 + c_1c_2a_2 - 176
\end{aligned}
\tag{3.8}
$$

3.1.2 Inverse

Computing inverse kinematics of the PHANTOM Omni, and serial robotic chains in general, requires more work than simply evaluating single valued expressions (like in the forward kinematics). As this method is examined, the possiblility for branch discontinuities occur. A branch discontinuity arrises from the math functions being manipulated. An example of a brach discontinuity is taking the arccosine of a number—it is possible to have two different angles that yield the same cosine.

Before analysis can begin, the transform from workspace to frame {0} must be undone. This is accomplished by multiplying by its inverse.

$$T = {}^{w}_{0}T^{-1}T_{given} \tag{3.9}$$

Furthermore, let this new matrix be labeled as seen in equation (3.10)

$$T\left(\theta_1,\,\theta_2,\,\theta_3,\,\theta_4,\,\theta_5,\,\theta_6\right)=\begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.10}$$

The first step is to determine the first three joint angles, which uniquely determine the position, designated as the ordered triple $(p_x,\,p_y,\,p_z)$. By analyzing the forward kinematics an expression for θ_1 can be determined.

$$\theta_1 = atan2\left(-p_x,\,p_y\right) \pm \cos^{-1}\left(\frac{d_3}{\sqrt{p_x^2+p_x^2}}\right) \tag{3.11}$$

Notice the $\pm$ in front of the arccosine term in equation (3.11). This means there are two possible solutions for this angle, which then also help determine other values. This is the first branch discontinuity of the inverse kinematics.

Using a similar method for finding θ_1, an expression for θ_3 can be determined

$$\theta_2 = atan2\left(-L_2, 0\right) \pm \cos^{-1}\left(\frac{\frac{A}{2L_1}}{\sqrt{L_2^2}}\right) \tag{3.12}$$

where

$$A = p_x^2 + p_y^2 + p_z^2 - L_1^2 - L_1^2 \tag{3.13}$$

Once again, the arccosine function has two solutions. Each of these solutions is not dependent on the value for θ_1, meaning that there are now four possible combinations of θ_1 and θ_3. With θ_1 and θ_3 known, they can be substituted back into previous undetermined expressions to help solve for θ_2. First let

$$\begin{aligned} A &= \cos\left(\theta_1\right) p_x + \sin\left(\theta_1\right) p_y \\ B &= L_1 - L_2 \sin\theta_3 \\ C &= L_2 \cos\theta_3 \end{aligned} \tag{3.14}$$

These expression come from the expression for p_x, p_y and p_z in equation (3.8). From these expressions, the unknown terms $\sin\theta_2$ and $\cos\theta_2$ can be solve for, giving

$$\begin{aligned} \sin\theta_2 &= \frac{CA + Bp_z}{-A^2 - p_z^2} \\ \cos\theta_2 &= \frac{B}{A} + \left(\frac{p_z}{A}\right)\left(\frac{CA + Bp_z}{-A^2 - p_z^2}\right) \end{aligned} \tag{3.15}$$

To get the singular value of θ_2, use the four quadrant arctangent as before

$$\theta_2 = atan2\left(\sin\theta_2,\, \cos\theta_2\right) \tag{3.16}$$

Note that for a given combination of θ_1 and θ_3, there will be only one solution for θ_2—meaning there are still only four solutions to the inverse kinematics.

The next step in solving the inverse kinematics of the PHANTOM Omni is the isolation of the wrist transform $W\left(\theta_4,\, \theta_5,\, \theta_6\right)$. As shown in equation (3.7), the forward kinematics are a product of a positioning transform and an orientation transform. By multiplying the inverse of the positioning transform (which is a function of θ_1, θ_2 and θ_3) by the given transform in equation (3.10), just the wrist transform is left, as seen in equation (3.17).

$$R^{-1}T = W$$

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} & 0 \\ b_{21} & b_{22} & b_{23} & 0 \\ b_{31} & b_{32} & b_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c_4c_5c_6 - s_4s_6 & -c_4c_5c_6 - s_4c_6 & c_4s_5 & 0 \\ s_4c_5c_6 + c_4s_6 & -s_4c_5s_6 + c_4c_6 & s_4s_5 & 0 \\ -s_5c_6 & s_5s_6 & c_5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.17}$$

In equation (3.17), it is clearly evident that $\cos\theta_5 = b_{33}$, meaning θ_5 can take values of

$$\theta_5 = \pm\cos^{-1} b_{33} \tag{3.18}$$

Since the original transform was multiplied by the inverse of the position transform (which contained the first three angle values, and four possible solution sets), each angle set will alter the values in the "B" matrix (see equation (3.17)). This means the value for b_{33} used in computing θ_5 in equation (3.18) will change depending on the solution, which doubles the number of solution sets for the inverse kinematics to eight total.

The remaining two angles can be found easily. First, find θ_4 by noticing the $(1,3)$ and $(2,3)$ elements of the wrist transform W. Since θ_5 is now known, the arctangent will return the single value for θ_4

$$\theta_4 = atan2\left(\frac{b_{23}}{-\sin\theta_5}, \frac{b_{13}}{-\sin\theta_5}\right) \tag{3.19}$$

Similarly, elements $(3,1)$ and $(3,2)$ contain only θ_6 and the known θ_5

$$\theta_6 = atan2\left(\frac{b_{32}}{-\sin\theta_5}, \frac{b_{31}}{\sin\theta_5}\right) \tag{3.20}$$

With these last two angles, they are uniquely defined by the four quadrant arctangent. By follwing all the possible branchs, there are eight possible solutions, detailed in figure 3.4. Because of this many possible solutions, care must be taken when selecting the desired angle. On the case of the actual device, there are limits the joints can rotate to, which reduce the number of solutions available.

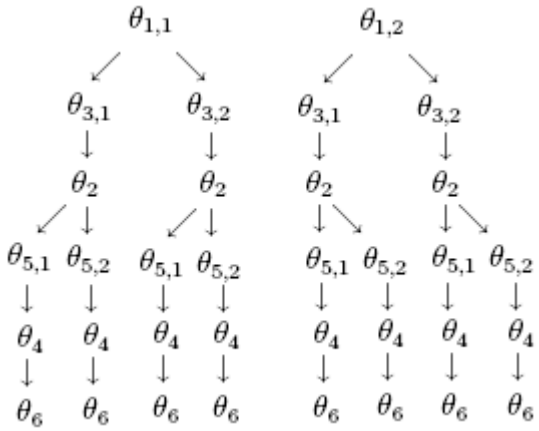

Figure 3.4 The eight solution sets for joint angles

3.2 Programming overview

The PHANTOM Omni can be programmed with the C++ language. There are two basic program types: a console based application or a graphical, OpenGL-based application. In either case, the code controlling the PHANTOM would be evaluated the same way. The haptic interface to the PHANTOM can be interpretted as another object to be rendered by the program, meaning that the computer will first process the haptic callback function and then the graphic callback function. Because of this similarity in operation, a graphical based program was chosen for development. Having graphical capabilities allows for expansion in the future for manipulating objects and seeing what happens. As a starting point, one of the included programs (which came bundled with the PHANTOM from Sensable Technologies) was used as a shell into which all the desired functionality was included.

The first thing to be done in the program is to initialize the haptic device. This involves determining which type of haptic device is attached (Sensable makes several different models in the PHANTOM line), enabling force output and adding the haptic rendering instance to the flow of the program. This code is shown below, in the function called "initHD". This function is called from a general initialization function, which initializes the OpenGL graphical interface.

Code 3.1 Haptic device initialization code

```
1  void initHD()
2  {
3      //  Built-in class to store error messages
4      HDErrorInfo error;
5
6      //  Store the handle of the device
7      ghHD = hdInitDevice(HD_DEFAULT_DEVICE);
8
9      //  Check to see if the device initialized
10     if (HD_DEVICE_ERROR(error = hdGetError()))
11     {
12         //  If it gets here, no devices found (or can not be initialized)
13         //  so exit program
14         exit(-1);
15     }
16
17     //  Enable the ability to output forces
18     hdEnable(HD_FORCE_OUTPUT);
19
20     //  Register the haptic rendering callback function (``touchScene'')
21     //  and give it a high priority
22     hUpdateDeviceCallback = hdScheduleAsynchronous(
23         touchScene, 0, HD_MAX_SCHEDULER_PRIORITY);
24
25     hdStartScheduler();
26
27     //  Check to see if the rendering instance was added to the scheduler
28     if (HD_DEVICE_ERROR(error = hdGetError()))
29     {
30         //  If scheduler can not be initialized, exit program,
31         //  since the haptics will not work
32         exit(-1);
33     }
34 }
```

Now that the PHANTOM Omni has been properly initialized, it is possible to integrate haptic rendering into the program. Since a graphical program is being used, the function which handles the drawing of the scene must have knowledge of the state of the device (position, transform, etc) so that it can interact with the graphical elements. Shown below is the beginning of this drawing function. "HapticDisplayState" on line four is a defined data structure which will store the current state of the device.

This next code showns the storing the device state for the use of the graphics portion of the program. Depending on what the graphical portion of code is doing, other device parameters can be stored. For example, if the program is to simply drawing a box with the position and

Code 3.2 Integrating the device into OpenGL

```
void drawScene()
{
    //  Used to store the state of the device
    HapticDisplayState state;

    // Obtain a thread-safe copy of the current haptic display state.
    hdScheduleSynchronous(copyHapticDisplayState, &state,
                          HD_DEFAULT_SCHEDULER_PRIORITY);


    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
     ...
    other OpenGL commands
     ...
}
```

orientation of the end of the PHANTOM, only the position and transform would be needed. On the other hand, if the program is to showing time histories of the position, joint angles, and velocity, those values would have to be stored for graphing.

Code 3.3 Storing the device state

```
HDCallbackCode HDCALLBACK copyHapticDisplayState(void *pUserData)
{
    //  Points to global variable that stores the state of the device
    HapticDisplayState *pState = (HapticDisplayState *) pUserData;

    //  Get the device position, velocity and transform and store it
    //  more device values can be stored, as long as they are
    //  define in the data structure
    hdGetDoublev(HD_CURRENT_POSITION, pState->position);
    hdGetDoublev(HD_CURRENT_VELOCITY, pState->velocity);
    hdGetDoublev(HD_CURRENT_TRANSFORM, pState->transform);

    return HD_CALLBACK_DONE;
}
```

With the device integrated into the graphical portion of the code, the constraints on the device must be accounted for. All of these are placed inside their own callback function. This function behaves in much the same way as the display callback function for OpenGL: the main program loops continually, refreshing the graphics and, in this case, the haptic interaction. This function can be quite long, depending on what type of constraint is programmed, so only the basic shell is shown in code 3.4

Code 3.4 Haptic rendering

```
HDCallbackCode HDCALLBACK touchScene(void *pUserData)
{
    //  Variables for the current and previous button values
    int currentButtons, lastButtons;

    //  The force and then the control law gains
    hduVector3Dd force (0, 0, 0);

    //  Control gains
    hduVector3Dd kp (10, 10, 10);

    //  Depending on the constraint, a different number of variables will be needed
    //  At the very least, the current position and a desired position are needed
    hduVector3Dd position;
    hduVector3Dd desiredPos (0, 0, 0);

    //  Used to store the maximum continuous force capable of being produced, and any errors
    HDdouble forceClamp;
    HDErrorInfo error;

    //  All haptic rendering must be in a frame declaring which haptic device to use
    hdBeginFrame(ghHD);

      //  Get the current and previous button values
      hdGetIntegerv(HD_CURRENT_BUTTONS, &currentButtons);
      hdGetIntegerv(HD_LAST_BUTTONS, &lastButtons);

      //  Get the current position
      hdGetDoublev(HD_CURRENT_POSITION, position);
      //  Other possible values to retrieve that might be useful include:
      //  HD_CURRENT_JOINT_ANGLES, HD_CURRENT_GIMBAL_ANGLES, and HD_CURRENT_VELOCITY

      // Detect button state transitions
      if ((currentButtons & HD_DEVICE_BUTTON_1) != 0 &&
          (lastButtons & HD_DEVICE_BUTTON_1) == 0)
      {
          //  If something should be done when a button is first pressed, it can be done here
      }
      else if ((currentButtons & HD_DEVICE_BUTTON_1) == 0 &&
               (lastButtons & HD_DEVICE_BUTTON_1) != 0)
      {
          // If something should be done when a button is first released, it can be done here
      }


      //Put contraints here

      //  Get the maximum continuous force possible for the device
      hdGetDoublev(HD_NOMINAL_MAX_CONTINUOUS_FORCE, &forceClamp);

      //  Limit force output to the maximum if it is exceeded
      if (hduVecMagnitude(force) > forceClamp)
      {
            hduVecNormalizeInPlace(force);
            hduVecScaleInPlace(force, forceClamp);
      }

      //  Set the force on the device
      hdSetDoublev(HD_CURRENT_FORCE, force);

    hdEndFrame(ghHD);

    //  Check for errors with haptic code
    if (HD_DEVICE_ERROR(error = hdGetError()))
    {
        //  If there is a problem with the force itself, stop the force
        if (hduIsForceError(&error))
        {
            hdSetDoublev(HD_CURRENT_FORCE, (0,0,0));
        }
        else
        {
            exit(-1);      // This is likely a more serious error, so exit
        }
    }

    return HD_CALLBACK_CONTINUE;
}
```

With the device now set up to run correctly, some special functions have be created by the manufacturers (Sensable, Inc.) to ease the programming. At any point during the program, values of different device parameters can be accessed with various commands, depending on the variable type being retreived. These commands were already used in Codess 3.1 through 3.4 without being described.

Code 3.5 hdGet commands

```
hdGetDoublev(parameter name, variable to store value in)
hdGetIntegerv(parameter name, variable to store value in)
hdGetLongv(parameter name, variable to store value in)
```

Some of the possible parameter names and what they return are listed in table 3.2. In addition to the current values stored in these locations, the device also stores the previous value. An example of this functionality would be to check whether a button has been just pressed or just released, which could be used to start or end an effect on the PHANTOM. These past values are referenced nearly the same way, but by replacing "CURRENT" with "LAST" in the parameter name (for example, "HD_LAST_BUTTONS" instead of "HD_CURRENT_BUTTONS"). Care must be taken to use the correct version of the "hdGet" command. For example, the values stored in "HD_CURRENT_ENCODER_VALUES" is a "long" variable and would then require "hdGetLongv".

Table 3.2 PHANTOM Omni "hdGet" parameters

Parameter name	*Value it returns*
HD_CURRENT_POSITION	device position in workspace coordinates, in millimeters
HD_CURRENT_VELOCITY	device velocity in workspace coordinates, in millimeters per second
HD_CURRENT_TRANSFORM	end effector transform
HD_CURRENT_JOINT_ANGLES	angles of first three joints in radians
HD_CURRENT_ENCODER_VALUES	the raw values returned from the digital encoders on all six joints
HD_CURRENT_BUTTONS	which (if any) of the two buttons are pressed

Interestingly, only one of the angles returned by HD_CURRENT_JOINT_ANGLES matches the convection established in the development of the kinematics for the device. In the case of θ_1, the positive datum established ealier is actually the negative direction in the joint-space

of the device. Of the three angles returned, θ_2 is the same in both cases. Computations must be done to determine the DH-defined angle θ_3. Figure 3.5 shows the angle measures returned by the device (designated with the subscript "p") and the DH angles. Three things must be done to find θ_3. First, compute $\alpha = 90^o - \theta_{2p}$, determined from the right triangle formed by with hypoteneus L_1. Second, compute $\beta = 180^o - \alpha - \theta_{3p} = 90^o + \theta_{2p} - \theta_{3p}$. Finally, compute $\theta_3 = 90^o - \beta = \theta_{3p} - \theta_{2p}$

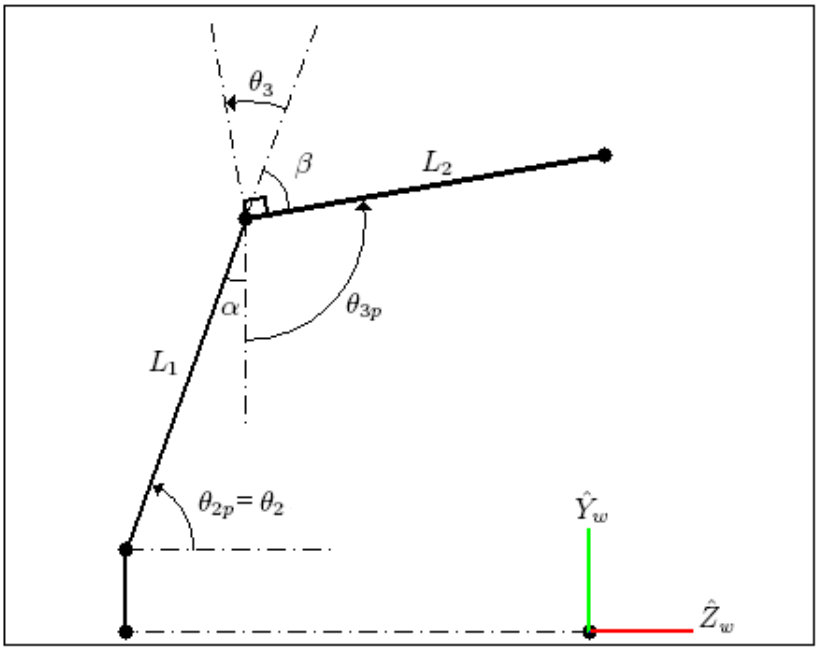

Figure 3.5 Differences in angle conventions

CHAPTER 4. PHANTOM SPATIAL CONSTRAINTS

As a starting point for controlling the PHANTOM Omni, different forms of constraints were tested. Two different methods for controlling the PHANTOM are possible: a cartesian space scheme and a joint-space scheme. In the cartesian method, the constraint can be established in the work space of the device itself (in x, y, z coordinates). The joint-space scheme involves working the inverse kinematics of the robot to detering the current joint values as well as the desired values. A control loop is then applied around the joint itself, making it reach its desired value. It is much easier to program in the cartesian space (the sample programs supplied by the manufacturer use this method) because of the branch defects that might occur when calculating the inverse kinematics.

All of these constraints are able to be used only simple proportional control. It is possible to incorporate a derivative control loop into the controller design since the velocity of the stylus tip can be read from the PHANTOM device interface. This can be problematic, however, because there is no precise way of finding the derivative control gains for a desired operation of the device.

This section covers the development of these equations and how they are generalized, along with the C++ code needed to implement them. The information is organized by the constraint type: theory and mathematics behind the constraint followed by the code and results of using the constrant. As will be seen, the challenge with the constraints is determining the desired point of the stylus. Once it is found, adding control is simple.

4.1 Points

Theory

Constaining the stylus of the PHANTOM to a single point is by far the easiest type of constraint. The type of contraint has zero degrees of freedom, but by using this type of contraint, fixed springs can be modeled. Consider a three-dimensional case, as depicted in figure 4.1 below. When the stylus is not located at the desired position, there will be an error vector between the desired position and the current position, designated by the dashed line. By simply multiplying this vector by corresponding elements of a proportional gain vector K_p (by having a vector of gain values, it would be possible to have one dimension of motion—for example, in the x direction—have a higher force applied than the others). This means the force applied by the PHANTOM's three motors will cause the stylus to move toward the desired position, with force components as shown in equation (4.1).

$$F = K_p\vec{e} \qquad \begin{cases} F_x = K_{px}\,(x_d - x) & +K_{ex}\,(\dot{x}_d - \dot{x}) \\ F_y = K_{py}\,(y_d - y) & +K_{ey}\,(\dot{y}_d - \dot{y}) \\ F_z = K_{pz}\,(z_d - z) & +K_{ez}\,(\dot{z}_d - \dot{z}) \end{cases} \tag{4.1}$$

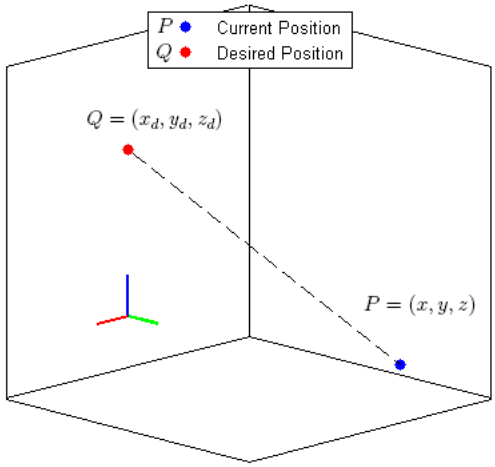

Figure 4.1 Point constraint

Coding and results

Coding this constraint is as simple as the theory behind it, as can be seen in Code 4.1. The proportional gain Kp was created as a vector to allow different directions to have different control effects. This could be used, for example, if the PHANTOM was to be used to model the feel of a material that has different compression ratios in different direction. As a test of this functionality, the desired point was chosen to be (0, 0, 0). Figure 4.2 shows both the distance the stylus is away from this point and the magnitude of the force applied to the stylus in workspace coordinates.

Code 4.1 Point constraint code

```
1 force=kp*(desiredPos-state.position); {optional: +ke*(desiredVel - state.velocity)};
```

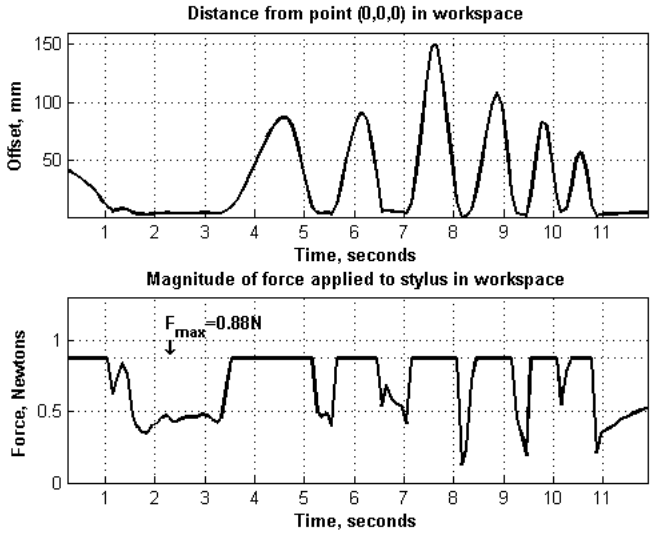

Figure 4.2 Results for point constraint

By examining the displacement and force in a single direction, in this case the x-direction (see figure 4.3, it can be seen that the components of the force are acting in the correct direction. Notice for a positive displacement, the force is negative, showing that the constraint is indeed forcing the stylus back towards the desired point. This is similar in the other directions, showing that the constraint is working as intended.

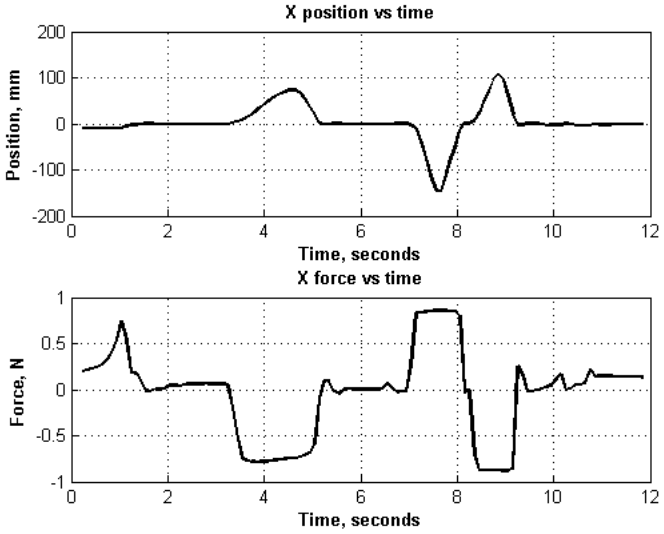

Figure 4.3 Point constraint in the x-direction

4.2 Spheres

Theory

Constraining the PHANTOM's stylus to a sphere closely follows from the analysis of a point. A circle can be defined with two parameters: the center point ($C = (x_c, y_c, z_c)$) and the radius (R). In order to find the closest point on the surface of the sphere, a vector $\vec{V}$ must be constructed connecting the center point C with the current stylus location P, as shown in figure 4.4. Thus

$$\vec{V} = P - C \tag{4.2}$$

The desired position Q=(x_q, y_q, z_q) is seen to be a distance R from C along this vector. By normalizing $\vec{V}$ and then multiplying by the radius, the point Q can be determined

$$Q = C + \frac{R}{\left|\vec{V}\right|}\vec{V} \tag{4.3}$$

With the desired position now calculated, the control law can be applied to the position error between this point and the current stylus position:

$$F = K_p\vec{e} \quad \begin{cases} F_x = K_{px}\left(Q_x - P_x\right) & +K_{ex}\left(\dot{x}_d - \dot{x}\right) \\ F_y = K_{py}\left(Q_y - P_y\right) & +K_{ey}\left(\dot{y}_d - \dot{y}\right) \\ F_z = K_{pz}\left(Q_z - P_z\right) & +K_{ez}\left(\dot{z}_d - \dot{z}\right) \end{cases} \tag{4.4}$$

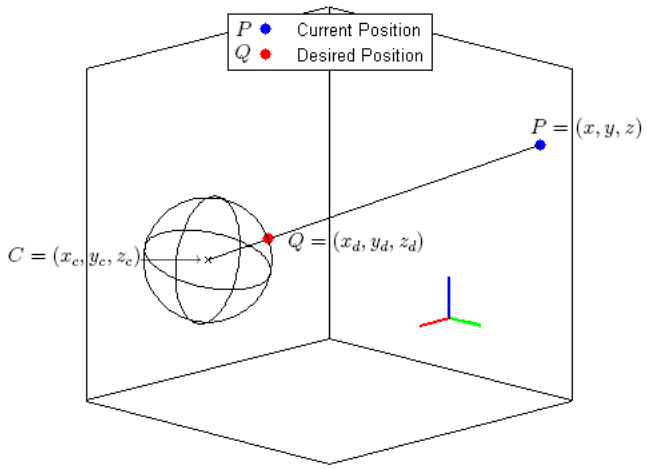

Figure 4.4 Sphere constraint

Coding and results

The coding of the spherical constraint follows closely from the theory and is, in actuality, a simple modification of the point constraint code.

Code 4.2 Sphere constraint code

```
1  // compute distance from the current stylus position to the center of the sphere
2  posvect=state.position-circleCenter;
3
4  // compute the magnitude of this vector
5  posvectmag=posvect.magnitude();
6
7  //  find the desired position
8  desiredPos=circleCenter+circleRad*posvect/posvectmag;
9
10 //  calculate force as gain(s) times error(s)
11 force=kp*(desiredPos-state.position);   {optional:  +ke*(desiredVel - state.velocity)};
```

As a test of this constraint, a sphere located at $(0, 0, 0)$ was created with a radius of 50mm. Figure 4.5 shows the results of this test. Figure 4.6 show the time histories of the distance

from $(0, 0, 0)$ and the magnitude of the force applied. Note that this force is a magnitude and does not indicate direction.

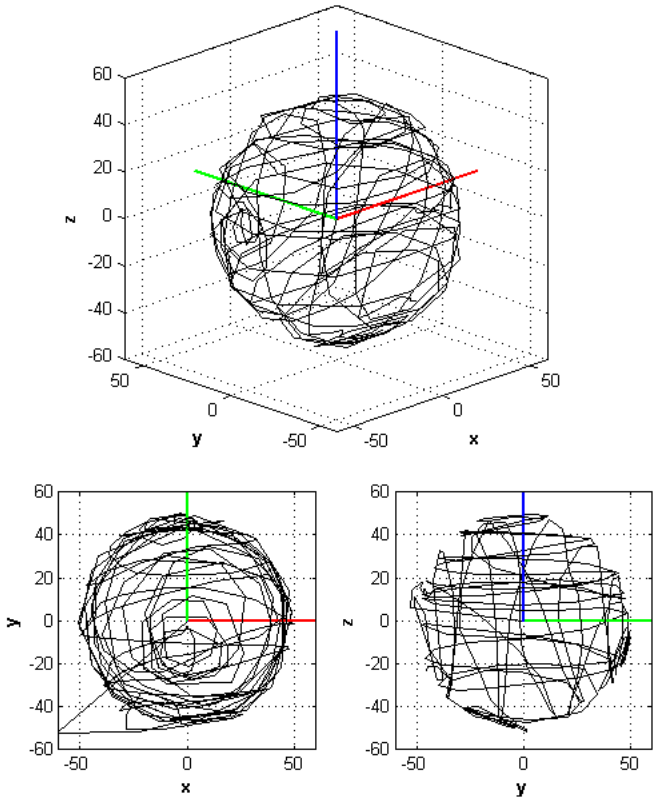

Figure 4.5 Results of the sphere constraint

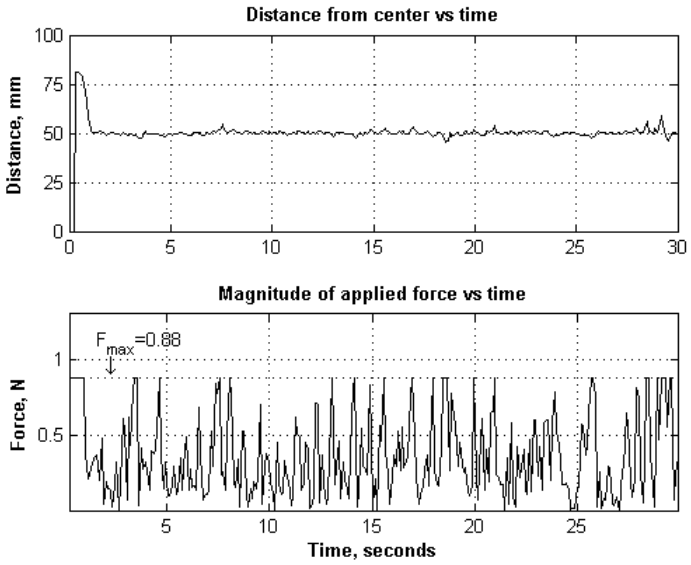

Figure 4.6 Results of the sphere constraint

4.3 Lines

Theory

Securing the stylus to a line is another straight-forward constraint. A line can be thought of a as single degree of freedom robot joint—a prismatic joint. Lines can be broken down into three groups:

1. lines on the coordinate axes
2. lines parallel to one of the coordinate axes
3. lines not parallel to any of the coordinate axes (arbitary lines)

The goal is to develop a generic approach to constraining to any of these line types. In order to adequately describe a line, only two quantities must be known: the direction a line travels and a single point on the line. For example, the x-axis can be described by the point $P_0 = (0, 0, 0)$ and the vector $\vec{V} = [1, 0, 0]$. Note that the vector is not required to be of unit magnitude. Addressing the third class of lines will also constrain the first two types as well.

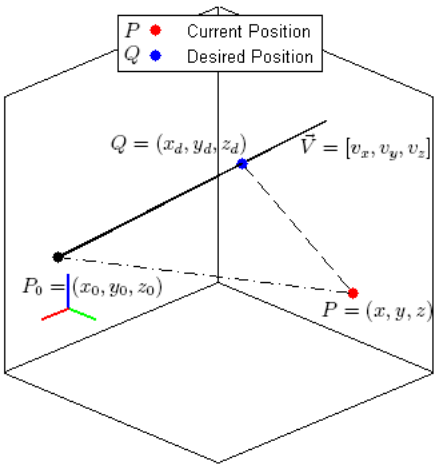

Figure 4.7 Line constraint

Consider figure 4.7. The point P_0 is the point on the line. Any point on the line is simply the addition of a scaled version of the line vector $\vec{V}$ to the point (P_0). By varying the scaling factor a from $-\infty$ to ∞, all points on the line can be found.

$$P_{on\ line} = P_0 + a\vec{V} \tag{4.5}$$

The distance between a point and a line is the length along an orthogonal line passing through the point. In order to find the distance this desired point is away from the point on the line, a triangle must be constructed. Connecting the point on the line P_0 to the current stylus position P forms the hypoteneus of this triangle, with the nearest point on the line (Q) forming the third vertex. If $\angle QP_0P$ is said to equal θ, the distance along the line from P_0 to Q is a scalar multiple of the $\vec{V}$ from P_0, which is also related to the hypoteneus $P - P_0$ by the cosine function.

$$|P_0 - Q| \equiv a\left|\vec{V}\right| = |P_0 - P|\cos\theta \tag{4.6}$$

Here, the || symbols denote magnitudes of vectors. The cosine can be found using the dot product of $\vec{V}$ with $P_0 - P$

$$\vec{V}\cdot[P_0 - P] = \left|\vec{V}\right||P_0 - P|\cos\theta \qquad \rightarrow \qquad \cos\theta = \frac{\vec{V}\cdot[P_0 - P]}{\left|\vec{V}\right||P_0 - P|} \tag{4.7}$$

Now, by substituting $\cos\theta$ from (4.7) into (4.6) and solving for the unknown scaling factor a

$$\begin{aligned} a &= \frac{|P_0 - P|}{\left|\vec{V}\right|}\frac{\vec{V}\cdot[P_0 - P]}{\left|\vec{V}\right||P_0 - P|} \\ a &= \frac{\vec{V}\cdot[P_0 - P]}{\left|\vec{V}\right|^2} \end{aligned} \tag{4.8}$$

Now, this factor can be substitued into (4.5)

$$Q = P_0 + \frac{\vec{V}\cdot[P_0 - P]}{\left|\vec{V}\right|^2}\vec{V} = (Q_x, Q_y, Q_z) \tag{4.9}$$

Now that the desired point has been determined, the control law can be applied to the stylus position.

$$F = K_p\vec{e} \qquad \begin{cases} F_x = K_{px}(Q_x - P_x) & +K_{ex}(\dot{x}_d - \dot{x}) \\ F_y = K_{py}(Q_y - P_y) & +K_{ey}(\dot{y}_d - \dot{y}) \\ F_z = K_{pz}(Q_z - P_z) & +K_{ez}(\dot{z}_d - \dot{z}) \end{cases} \tag{4.10}$$

Coding and results

The C++ code written for the line constraint is shown below. As with many programs, the names of the variables used where created to be descriptive of the value being stored, instead of matching the variable names used in the theory section.

Code 4.3 Line constraint code

```
1  //  Vector from the position to the point on the line
2  posvect=state.position-pointonline;
3
4  //  Dot product of
5  AdotP=linevector.dotProduct(posvect);
6
7  //  Find magnitude of the vector that is the line
8  linevectormag=linevector.magnitude();
9
10 //  Find the magnitude of the position vector
11 posvectmag=posvect.magnitude();
12
13 //  Solve for cosine theta as dot product divided by product of magnitude
14 costheta=AdotP/(linevectormag*posvectmag);
15
16 //  Compute the line scaling factor for the nearest point
17 linescale=posvectmag*costheta/linevectormag;
18
19 // Compute the desired (nearest) point on the line
20 desiredPos=linevector*linescale;
21
22 //  Force is proportional to error
23 force=kp*(desiredPos-posvect);
```

As a test of the arbitary line, the point on the line line was chosen to be $P_0 = (0, 30, 0)$ and the direction of the line was chosen to be $\vec{V} = [-1, -2, 1]$. Figure 4.8 shows the collection of all points the stylus travelled during a run of this constraint code. In addition to this data, the point P_0 and the line vector $vecV$ were drawn on the plot for comparison. At the start of the program, the data returned from the program have the stylus position as $(0, 0, 0)$. After this first sample, the position returned by the code becomes wherever the stylus actually is.

As the force constraining the stylus is applied, the stylus moves toward the constraint, as can be seen in the figure.

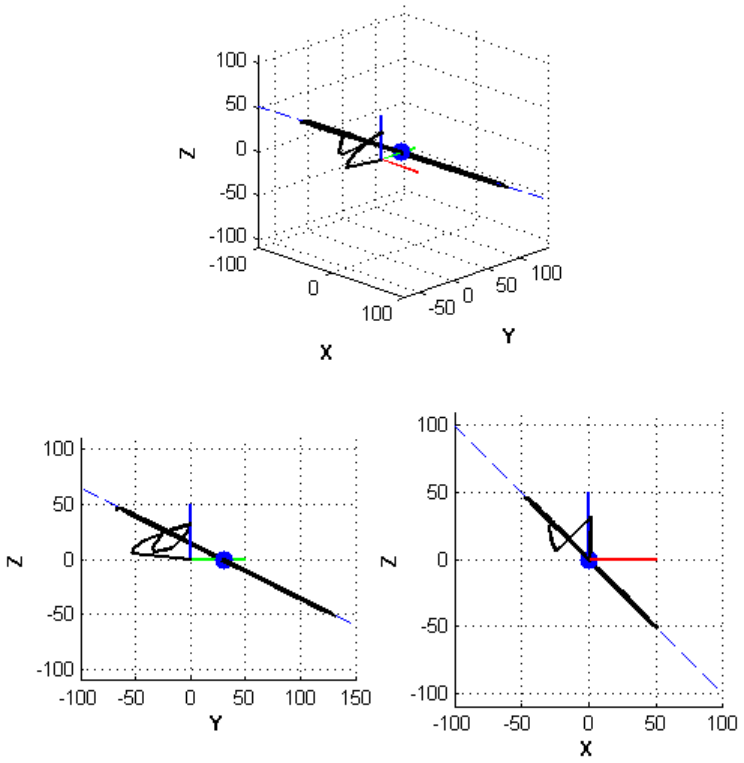

Figure 4.8 Results for arbitrary line

4.4 Planes

Theory

Planes serve as a meaningful application of arbitrarily oriented surfaces. Many objects encountered in the world are a collection of planar faces. This type of constrant has two degrees of freedom—for example, moving in the x and/or y direction in a plane—and can be thought of as a $2P$ serial robot. To describe a plane, two parameters are needed: a point in the plane, and the normal vector to the plane. Figure 4.9 depicts the general case of an arbitrarily oriented (but infinite) plane.

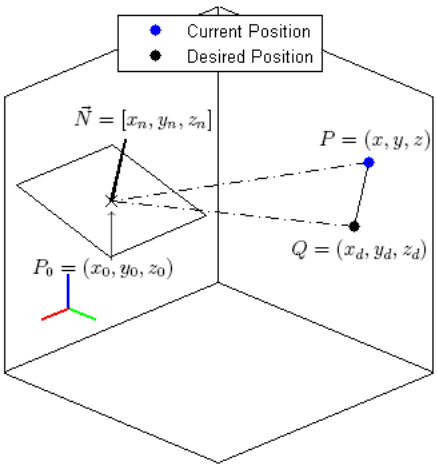

Figure 4.9 Plane constraint

In order to determine the desired point in the plane, the distance the stylus is from the plane must be determined. This is accomplished using a similar to an arbitrary line. Begin by constructing a vector connecting P and P_0. By taking the dot product of this vector with the normal, the angle between them can be determined:

$$\vec{N} \cdot [P - P_0] = \left|\vec{N}\right| |P - P_0| \cos\theta \tag{4.11}$$

Next, a right triangle is constructed with points P, P_0 and a third point P_3 lying an unknown distance along $\vec{N}$. By using the definition of the cosine

$$|P_3 - P_0| \equiv a\vec{N} = |P - P_0| \cos\theta \tag{4.12}$$

Now, substituting the cosine from (4.11) into equation (4.12) and then solving for the unknown scaling factor a

$$a = \frac{\vec{N} \cdot [P - P_0]}{\left|\vec{N}\right|^2} \tag{4.13}$$

Now that the scalilng factor is known, the desired stylus location can be calculated

$$Q = P - a\vec{N} \tag{4.14}$$

With the desired point calculated, a control law can be used:

$$F = K_p\vec{e} \qquad \begin{cases} F_x = K_{px}(Q_x - P_x) & +K_{ex}(\dot{x}_d - \dot{x}) \\ F_y = K_{py}(Q_y - P_y) & +K_{ey}(\dot{y}_d - \dot{y}) \\ F_z = K_{pz}(Q_z - P_z) & +K_{ez}(\dot{z}_d - \dot{z}) \end{cases} \tag{4.15}$$

Coding and results

The code for this constraint is shown below. As with the previous constriants, the code for this contraints comes from simply coding the equations developed in the theory section.

Code 4.4 Plane constraint code

```
1  //   Find the vector from the known point on the plane to the current position
2  posvect=state.position-pointonline;
3
4  //   Compute dot product of new position vector and plane normal vector
5  AdotP=posvect.dotProduct(linevector);
6
7  //   Find magnitude of the normal
8  linevectormag=linevector.magnitude();
9
10 //   Compute desired position
11 desiredPos=state.position-(AdotP/linevectormag)*linevector;
12
13 // Compute force from control law
14 force=kp*(desiredPos-posvect);      {optional: +ke*(desiredVel-state.velocity);}
```

This constraint, as demonstrated in the theory section and the code, is defined by a point on the plane and the planar normal. Because of the mobility of the PHANTOM, adequate testing of this constraint is somewhat difficult. The point on the plane needs to be somewhat near the origin and the normal must be oriented such that more than only a small portion of the plane can be touched. Taking this into account, a test was created using the point $(0, 0, 0)$ and the normal $[1, 1, 1]$. There results are shown in figure 4.10

4.5 Circles

Theory

Another simple yet important constraint for the PHANTOM is a circle. A circle provides a single degree of freedom to the user, in this case a revolute joint. There are two cases for circles:

1. circle lies in in a plane parallel to one of the principle coordinate planes

2. circle lies in an arbitrarily oriented plane

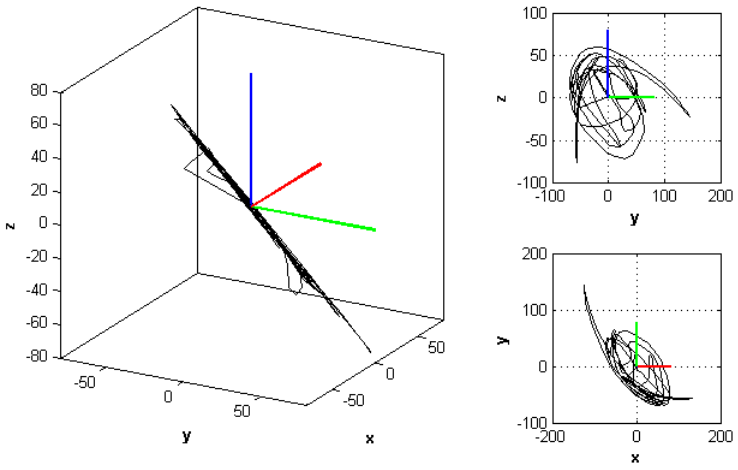

Figure 4.10 Results for arbitary plane

It is apparent that if a method can be created to fill the second case, it will satisfy the first. In order to define a circle in space, three things must be known: where the center is located, the radius of the circle, and the normal vector to the plane the circle is located it. This is depicted in figure 4.11.

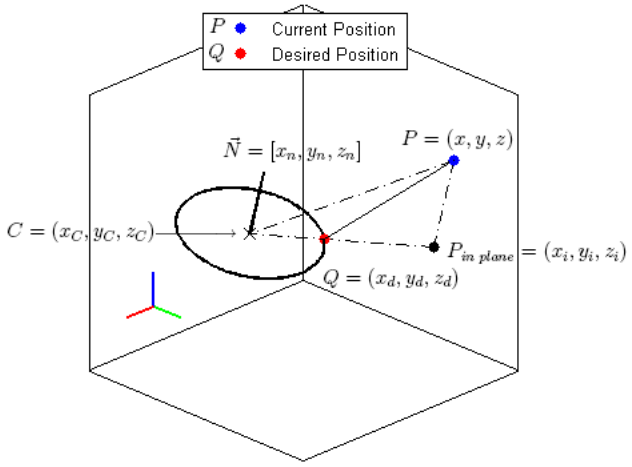

Figure 4.11 Circular constraint

In order to create this constraint, a few parameters must be defined. Let the center of the circle be defined as $C = (x_c, y_c, z_c)$ and let its radius be R. The planar normal shall be defined as $\vec{N} = [x_n, y_n, z_n]$ Let the current position of the stylus be $P = (x, y, z)$. The first thing to

be determined is the in-plane projection of the point, $P_{in\ plane}$. The method for this begins by finding the scaling factor of $\vec{N}$, just like in the case of an arbitrary line. The angle between vector $[P-C]$ and $\vec{N}$ can be found using the dot product:

$$\vec{N} \cdot [P-C] = \left|\vec{N}\right| |P-C| \cos\theta \tag{4.16}$$

Like with an arbitrary line, a scaling factor can be found such that

$$a\left|\vec{N}\right| = |P-C| \cos\theta \tag{4.17}$$

The value for $\cos\theta$ from equation (4.16) can be substituted into equation (4.16), which can then be solved for the scaling factor a:

$$a = \frac{\vec{N} \cdot [P-C]}{\left|\vec{N}\right|^2} \tag{4.18}$$

With this scaling factor, the in plane point $P_{in\ plane}$ can be found as;

$$P_{in\ plane} = P - a\vec{N} \tag{4.19}$$

Next, the coordinates of the desired point on the circle must be determined. To find this, begin by forming vector $\vec{V}$ connecting points P and C. The point Q is a distance R along a normalized $\vec{V}$ vector away from C:

$$Q = C + \frac{R}{\left|\vec{V}\right|}\vec{V} \tag{4.20}$$

Now that this desired point has be calculated, a control law can be applied.

$$F = K_p\vec{e} \qquad \begin{cases} F_x = K_{px}(x - x_d) & +K_{ex}(\dot{x}_d - \dot{x}) \\ F_y = K_{py}(y - y_d) & +K_{ey}(\dot{y}_d - \dot{y}) \\ F_z = K_{pz}(z - z_d) & +K_{ez}(\dot{z}_d - \dot{z}) \end{cases} \tag{4.21}$$

Coding and results

The code for this constraint, as the theory shows, can be essentially divided into two parts. First, the nearest in-plane point must be determined (the plane constraint code) and then the vector from this position to the center of the circle is determined. The desired point is located along this vector, a distance of the radius away from the center. This code is shown in Code 4.5.

Code 4.5 Circle constraint code

```
1  //  Vector from current position to circle center
2  posvect=state.position-circleCenter;
3
4  //  Magnitude of this vector
5  posvectmag=posvect.magnitude();
6
7  //  linevector is the planar normal. Find its magnitude
8  linevectormag=linevector.magnitude();
9
10 //  Dot product of the position vector and the planar normal
11 AdotP=linevector.dotProduct(posvect);
12
13 //  Find the scaling factor of the normal vector
14 linescale=AdotP/(linevectormag*linevectormag);
15
16 //  Find the nearest point in the same plane as the circle
17 newP=state.position-linescale*linevector;
18
19 //  Compute vector beween the in-plane point and the circle center
20 errvect=newP-circleCenter;
21
22 //  Find magnitude of this distance
23 errvectmag=errvect.magnitude();
24
25 //  Compute the desired position in relation to the circle center
26 desiredPos=circleCenter+(circleRad/errvectmag)*errvect;
27
28 //  Calculate force as gain(s) times error(s)
29 force=kp*(desiredPos-state.position);   {optional: +ke*(desiredVel-state.velocity);}
```

As a test of this contraint, the center was chosen to be $C = (10, 20, 30)$, the radius $R = 40$ and the axis $\vec{A} = [-1, 1, -1]$. These results are shown in figure 4.12.

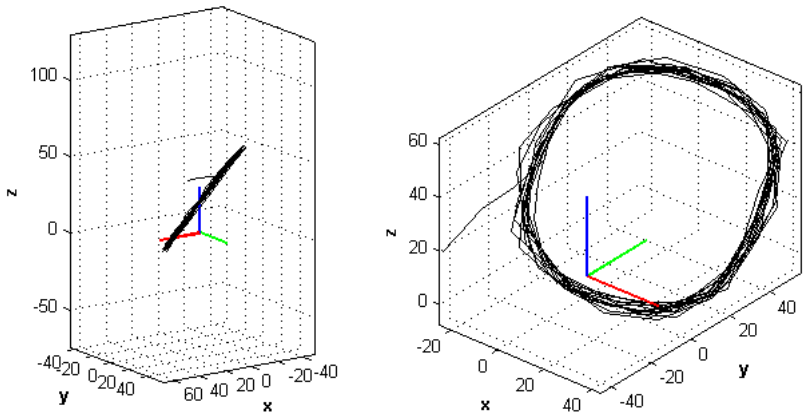

Figure 4.12 Circular constraint results

4.6 Cylinders

Theory

As with some of the line constraint, a cylinder can be classified into one of three groups:

1. cylinders whose axis lies on one of the coordinate axes
2. cylinders whose axis is parallel to one of the coordinate axes
3. cylinders whose axis lies along an arbitrary line in space

As with deriving the arbitrary line constraint, the third type of cylinder will cover the first two types. The difficult thing when developing this constraint is to allow the stylus to slide along any part of the cylinder. Part of the constraint is part of an arbitrary circle constraint to figure out which direction the cylinder is pointing and a second part is an arbitrary line constraint, allowing the stylus to move up and down the infinite length of the cylinder. In this regard, this constraint can be thought of as the collection of all lines parallel to the axis of the cylinder at a fixed distance (the radius).

In order to define a cylinder in space, three quantities are needed: the center of the circular portion of the cylinder, its radius and a vector describing the direction of the axis of the

cylinder (this can be thought of as the planar normal of the arbitrary circle constraint). Figure 4.13 shows this general setup. The cylinder is defined by C, the center, $\vec{A}$, the axis, and R, the radius. The ultimate goal of the constraint is to find the nearest point on the surface of the cylinder. First, however, the closest point on the defined circle must be found (like already mentioned, this constraint can be described as the arbitary circle constraint with no force applied in the direction of the planar normal). This is identical to finding the off-plane distance in the planar constraint or arbitrary circle constraint. This involves first using the dot product, where θ is defined as the angle between $\vec{N}$ and $[P - C]$.

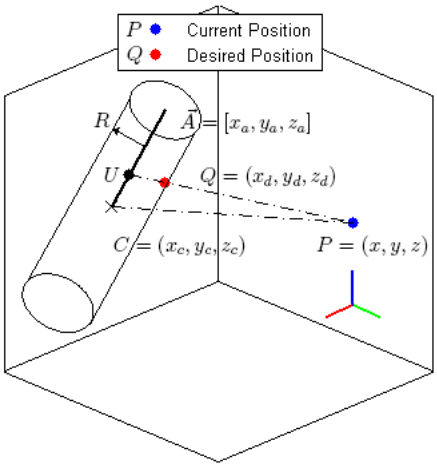

Figure 4.13 The arbitrary cylinder constraint

$$\vec{A} \cdot [P - C] = \left|\vec{A}\right| |P - C| \cos\theta \tag{4.22}$$

Next, the properties of right triangles give rise to the next equation in determining the off-plane distance

$$a \left|\vec{A}\right| = |P - C| \cos\theta \tag{4.23}$$

Equation (4.23) can be solved for $\cos\theta$ and substituted into equation (4.23). Solving for the unknown scaling factor a

$$a = \frac{\vec{A} \cdot [P - C]}{\left|\vec{A}\right|^2} \tag{4.24}$$

With this scaling factor, the nearest point on the axis of the cylinder can be found as

$$U = C + a\vec{A} \tag{4.25}$$

Next, a vector from P to U can be constructed as $\vec{E} = [P - U]$. This vector is in the plane of the circular portion of the cylinder. By normalizing this vector and multiplying by the radius of the cylinder, the nearest point on the surface of the cylinder can be found.

$$\begin{aligned} Q &= U + \frac{R}{\left|\vec{E}\right|}\vec{E} \\ &= C + a\vec{A} + \frac{R}{\left|\vec{E}\right|}\vec{E} \end{aligned} \tag{4.26}$$

Coding and results

At this point, it has been seen that coding the constraints simply involves naming quantities derived in the theory section of a constraint to variable names and then doing the required operations. The code for the cylinder constraint was developed in a similar way.

To test this constraint, the point on the axis was chosen to be $C = (30, 0, 30)$, the radius $R = 30$ and the vector of the axis $A = [-1, 1, -1]$. These results are shown in figure 4.14.

Code 4.6 Cylinder constraint code

```
1  // Vector from known point on axis to current stylus position
2  posvect=state.position-pointonline;
3
4  // Magnitude of this vector
5  posvectmag=posvect.magnitude();
6
7  // Magnitude of the axis vector
8  linevectormag=linevector.magnitude();
9
10 // Dot product of position vector and the axis vector
11 AdotP=posvect.dotProduct(linevector);
12
13 // Find cosine of the included angle
14 costheta=AdotP/(posvectmag*linevectormag);
15
16 // Find the line scaling factor
17 linescale=posvectmag*costheta/linevectormag;
18
19 // Compute the nearest point on the axis of the cylinder
20 point1=pointonline+linescale*linevector;
21
22 // Calculate error vector from current stylus position to nearest point on axis
23 errvect=state.position-point1;
24
25 // Find the magnitude of the error
26 errvectmag=errvect.magnitude();
27
28 // The desired position is a distance of the radius away from the axis along the error vector
29 desiredPos=point1+errvect*(circleRad/errvectmag);
30
31 //  Compute the force from the control law
32 force=kp*(desiredPos-state.position);      {optional: +ke*(desiredVel-state.velocity)};
```

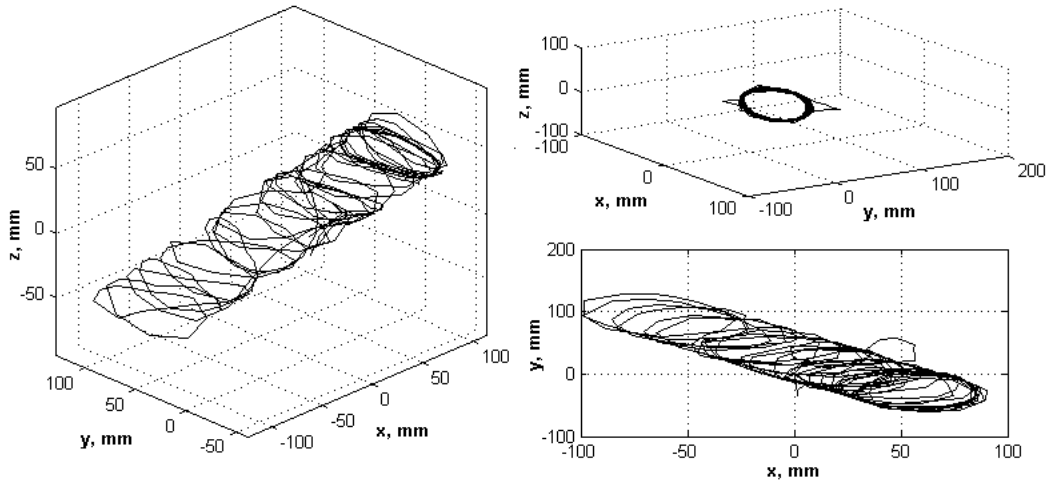

Figure 4.14 Results for an arbitrary cylinder

CHAPTER 5. ADVANCED SPATIAL CONSTRAINTS

In the previous section, constraints were developed for the PHANTOM Omni based solely on the sensed position at the end of the device. While following contours serves an important purpose, it is not using the PHANTOM Omni to its full potential—these constraints only use three of the six degrees of freedom the PHANTOM is capable of determining. Consider the case of constraining to a planar circle, as shown in figure 5.1. Three stylus position orientation combinations are shown. Using the previous constraints, the stylus would move to the nearest point on the curve.

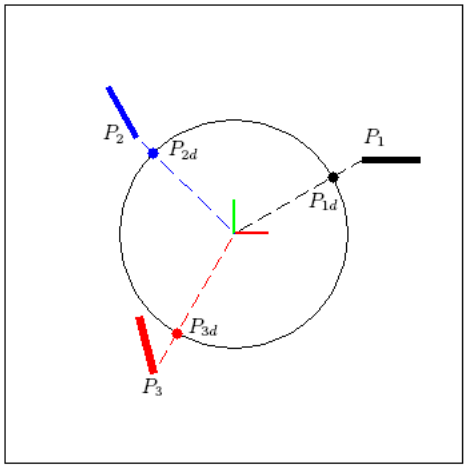

Figure 5.1 Circle constraint

Instead, the orientation can be used in determining the desired point of the stylus. By incorporating these other degrees of freedom into the constraint, the PHANTOM can be used to allow the user to "feel" different things not normally felt with the PHANTOM. As a demonstra-

tion of this functionality, a planar circle has been used. As opposed to the previous constraints, two desired locations are calculated: the closest physical point to the stylus position (this is the previously developed constraint), and the point that has the same orientation. An "influence" factor is used to determine the weighting each of these plays in determining that actual point the stylus is drawn to. By setting ranging this value between zero and one, the desired point of the stylus is determined. This new constraint can be thought of in two ways: a Bezier curve connecting the two points and secondly a virtual robot setup. Both are explained below.

5.1 Constraint as Bezier curve

Consider figure 5.2, showing the current stylus position (P) as well as two other points: P_o showing the point on the circle whose normal has the same orientation angle as the stylus and P_p being the closest point on the circle to the stylus position (this is where the stylus would go if the previously developed constraints were used). P_o can be envisioned as having 0% orientation error and 100% position error. In a similar manner, P_p can be said to have 0% position error and 100% orientation error. Suppose it is desired for the stylus to move somewhere between these two points (for example, to the point with 50% position error and 50% orientation error). As an added constraint on this movement, the desired point will represent the shortest of the two distances between the points (for example, if $\alpha = 0^o$ and $\beta = 90^o$, the resulting angle will be $0^o \leq \theta \leq 90^o$ instead of $90^o \leq \theta \leq 360^o$).

To begin determination of this constraint, the angles α and β must be determined. For this development, the circle to be rotated about is assumed to be in the XY plane (similar development can be done if the circle is in other planes). To determine both angles, the current transform of the PHANTOM is needed. The direction the stylus is pointing (the stylus is the Z axis in frame six) can be found by examining the transform of the device, as seen in equation (5.1). When the stylus is close to being in the XY plane, the ${}^w_6\hat{Z}_z$ component will be close to zero. This is also true if the circle to follow is in another plane (if the circle was in the XZ

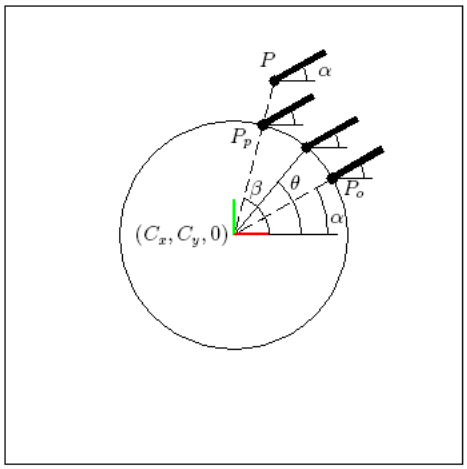

Figure 5.2 Depicting position- and orientation-based angles

plane, the ${}^{w}_{6}\hat{Z}_y$ component would be close to zero).

$$\begin{bmatrix} {}^{w}_{6}\hat{X}_x & {}^{w}_{6}\hat{Y}_x & {}^{w}_{6}\hat{Z}_x & P_x \\ {}^{w}_{6}\hat{X}_y & {}^{w}_{6}\hat{Y}_y & {}^{w}_{6}\hat{Z}_y & P_y \\ {}^{w}_{6}\hat{X}_z & {}^{w}_{6}\hat{Y}_z & {}^{w}_{6}\hat{Z}_z & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.1}$$

With the orientation of the stylus known, the two angles α and β can be determined. The orientation angle α can be found as

$$\alpha = atan2\left({}^{w}_{6}\hat{Z}_y,\, {}^{w}_{6}\hat{Z}_x\right) \tag{5.2}$$

The angle β is found in a similar way. For this angle, the location of the center must be known.

$$\beta = atan2\left(P_x - C_x,\, P_y - C_y\right) \tag{5.3}$$

Since the "atan2" function is begin used for both of these angles, the angle returned will be on the interval $-\pi < \theta \leq \pi$. The maximum magnitude the difference of these two angles can take is 180^o (if the difference between the two is greater than 180^o, rotating in the opposite direction will be less than 180^o, which is desired. To control where the desired point is, let there be an "influence" factor called t. The angle of this new point can be found as

$$\begin{aligned} \theta &= \alpha + t\,(\beta - \alpha) \\ \theta &= (1 - t)\,\alpha + (t)\,\beta \end{aligned} \tag{5.4}$$

5.2 Constraint as virtual manipulator

Consider the planar setup as shown in 5.3. In this case, the PHANTOM Omni is assumed to be operating in a plane where the first joint angle θ_1 is held constant at zero degrees. This means that the PHANTOM is already constrained to the YZ plane.

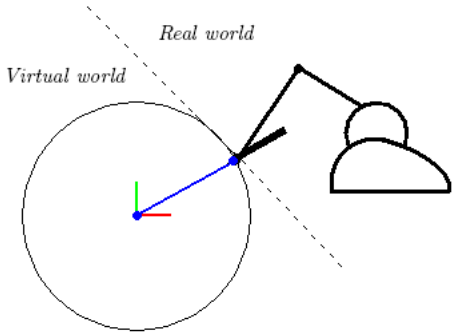

Figure 5.3 Planar virtual manipulator setup

Now, a slight difference in naming convention will be made. Instead of calling this the YZ plane, a coordinate system will be established calling this the XY plane, which will follow the normal pictorial description of the plane (positive x axis to right, positive y axis is up), as seen in figure 5.4. This is new coordinate system can be converted to the operating space of the PHANTOM by using a desired transform. When looking at 5.4, system also can be analyzed as a planar four-bar mechanism. The "ground" link connecting the frame of the real robot to the frame of the virtual robot is known (in fact, the location of the virtual frame is hard-coded in the code relative to the position of the frame of the real robot)

First, the kinematics of the real world robot can be determined. The results are with

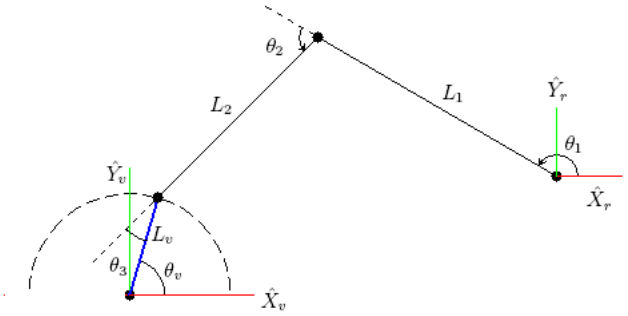

Figure 5.4 Planar virtual manipulator schematic

respect to the first joint of the real robot.

$$\begin{Bmatrix} x \\ y \\ \theta \end{Bmatrix} = \begin{Bmatrix} L_1 \cos\theta_1 + L_2 \sin(\theta_1 + \theta_2) \\ L_1 \sin\theta_1 + L_2 \cos(\theta_1 + \theta_2) \\ \theta_1 + \theta_2 + \theta_3 \end{Bmatrix} = \Phi(\theta_1, \theta_2, \theta_3) \tag{5.5}$$

Next, the Jacobian of this robot configuration will be calculated. The Jacobian expresses the Cartesian velocity of the robot in terms of the joint velocities or, in a similar manner, relates the joint torques to the force exerted by the robot in Cartesian space (or the force resisted by a given set of torques). Begin by taking the derivative of the kinematic equations presented in (5.5).

$$\begin{Bmatrix} \dot{x}_r \\ \dot{y}_r \\ \dot{\theta}_r \end{Bmatrix} = \begin{Bmatrix} -L_1 \sin\theta_1 \dot{\theta}_1 + L_2 \cos(\theta_1 + \theta_2)\left(\dot{\theta}_1 + \dot{\theta}_2\right) \\ L_1 \cos\theta_1 \dot{\theta}_1 - L_2 \sin(\theta_1 + \theta_2)\left(\dot{\theta}_1 + \dot{\theta}_2\right) \\ \dot{\theta}_1 + \dot{\theta}_2 + \dot{\theta}_3 \end{Bmatrix} \tag{5.6}$$

Next, arrange equation (5.6) into matrix form

$$\begin{Bmatrix} \dot{x}_r \\ \dot{y}_r \\ \dot{\theta}_r \end{Bmatrix} = \begin{bmatrix} -L_1 \sin\theta_1 + L_2 \cos(\theta_1 + \theta_2) & L_2 \cos(\theta_1 + \theta_2) & 0 \\ L_1 \cos\theta_1 - L_2 \sin(\theta_1 + \theta_2) & -L_2 \sin(\theta_1 + \theta_2) & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{Bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \end{Bmatrix} \tag{5.7}$$

$$= J_R(\theta_1, \theta_2, \theta_3)\,\dot{\Theta}_R$$

In a similar manner, the kinematics of the virtual robot can be determined

$$\begin{Bmatrix} x_v \\ y_v \\ \theta_v \end{Bmatrix} = \begin{Bmatrix} L_v \cos\theta_v \\ L_v \sin\theta_v \\ \theta_v \end{Bmatrix} \tag{5.8}$$

Taking the derivative of the kinematics in equation (5.8) and factoring out the one angle θ_V gives the Jacobian

$$\begin{aligned} \begin{Bmatrix} \dot{x}_v \\ \dot{y}_v \\ \dot{\theta}_v \end{Bmatrix} &= \begin{bmatrix} -L_V \sin\theta_V \\ L_V \cos\theta_V \\ 1 \end{bmatrix} \dot{\theta}_v \\ &= J_V(\theta_V)\,\Theta_V \end{aligned} \tag{5.9}$$

The idea behind this constraint is to position the virtual manipulator such that that its position represents the "nearest" position to the real robot. Since three degrees of freedom are be measured in this planar case (x position, y position and orientation angle θ), the "nearest" position is not simply defined—there exists the point that has the least position error (the nearest point from the previous constraints) and there is a point that has the least orientation error (in actuality, it will have zero orientation error since the virtual manipulator can have any orientation in the operational plane).

When thinking about this constraint as a virtual robot, the idea of the force exertion of the robot comes to mind. Suppose the stylus of the PHANTOM is being pushed downward, as shown in figure 5.5.

Free body diagrams were constructed of both the virtual and real robots. As seen on the free body diagram for the virtual manipulator, the external force (i.e. the force applied from the real robot, which is the force applied by the user) can be broken into radial and angular components about the single degree of freedom of the virtual robot. In order for the virtual robot to resist the external force, only the radial component of the force must be balanced. Using basic trigonometry, this can be found to be $F_{v\theta} = F\sin\theta$. By summing the moments about the pivot, the virtual robot torque can be found as

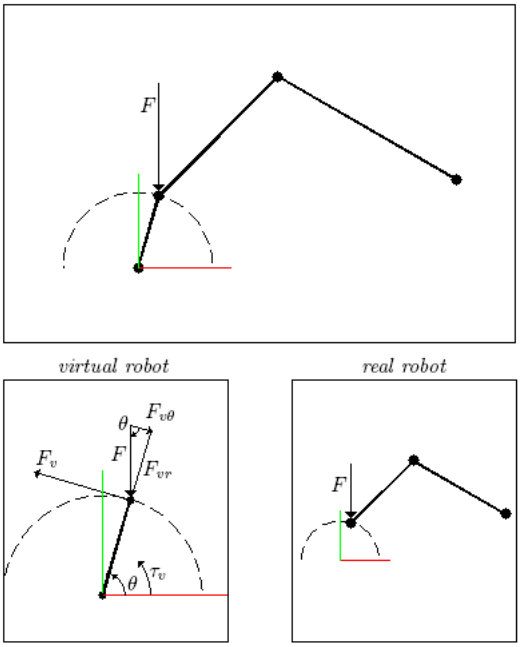

Figure 5.5 Free body diagrams of the virtual and real robots

$$\begin{aligned} \sum M &= \tau_v - F_{v\theta} \\ &= \tau_v - F \sin\theta \end{aligned} \tag{5.10}$$

For the static case, $\sum M = 0$ and the virtual torque is seen to be equal to the angular component of the external force. Now, remembering an expression from the analysis of robots, a relation between the joint torque and the Cartesian force is found to be

$$\tau_v = J_v^T F \tag{5.11}$$

Remember, the Jacobian J_v is a function of θ_v. In order to determine the force that a given virtual manipulator is capable of withstanding, equation (5.11) must be solved for F. In the general case, this is not an easy feat—in this simple case, F is a 3×1 column vector representing Cartesian forces, the dimensions of τ_v depend on the number of joints in the robot, and J_v is probably not a square matrix. In order to accomplish this, a Moore-Penrose matrix

inverse can be used (also referred to as a pseudo-inverse). This method finds the least-squares fit for F. By applying this method to equation (5.11), F can be found as

$$F = J_v \left(J_v^T \, J_v \right)^{-1} \tau_v \tag{5.12}$$

This give a Cartesian force in the range space of the virtual manipulator (it accounts only for the forces provided by the joint torques/forces and not forces provided by the structure of the virtual manipulator). To help make the calculation in equation (5.12) account for these type of effects, a weighting matrix can be included, resulting in the general form

$$F = W_A \, J_v \left(J_v^T \, W_A \, J_v \right)^{-1} \tau_v \tag{5.13}$$

In order to continue this analysis, consider the error experienced by the system: $e = X_V - X_R$. Both positions X_V and X_R are functions of the respective joint angles. A linearization of these terms can be found using a Taylor sears expansion

$$\begin{aligned} X_v &= \Phi_{v0} + \left.\frac{\partial \Phi_v}{\partial \theta_{v1}}\right|_{\theta_v = \theta_{v0}} \Delta\theta_{v1} + \left.\frac{\partial \Phi_v}{\partial \theta_{v2}}\right|_{\theta_v = \theta_{v0}} \Delta\theta_{v2} + \cdots + \textit{higher order terms} \\ X_r &= \Phi_{r0} + \left.\frac{\partial \Phi_r}{\partial \theta_{r1}}\right|_{\theta_r = \theta_{r0}} \Delta\theta_{r1} + \left.\frac{\partial \Phi_r}{\partial \theta_{r2}}\right|_{\theta_r = \theta_{r0}} \Delta\theta_{r2} + \cdots + \textit{higher order terms} \end{aligned} \tag{5.14}$$

By neglecting the higher order terms of equation (5.14), it is noticed that the partial derivative terms represent simply the Jacobian of the manipulator. It is then possible to solve this equation to find the necessary changes in the current position of the virtual manipulator to minimize the error.

$$e \approx \Phi_{v0} + J_v \Delta\theta_v - \Phi_{r0} - J_r \underbrace{\Delta\theta_r}_{\substack{do \\ not \\ move}} \tag{5.15}$$

In order to minimize the error, the virtual manipulator will be moved. This is easier to accomplish than moving the PHANTOM, since the virtual manipulator is only software. This

means that the $\Delta\theta_r$ term in equation (5.15) will be zero. Also, by definition, this method drives the error to zero, resulting in an equation that relates the error to the movement required of the virtual manipulator.

$$\begin{aligned} 0 = \Phi_{v0} + J_v \Delta\theta_v - \Phi_{r0} = X_r - X_v + J_v \Delta\theta_v \\ X_r - X_v = -J_v \Delta\theta_v \end{aligned} \tag{5.16}$$

Since there might not be a single "nearest" position of the virtual manipulator when using this setup, the weighted pseudo-inverse is used once again.

$$\Delta\theta_v^* = \left(J_{v0}^T W_e J_{v0}\right)^{-1} J_{v0}^T W_e \left(X_r - X_{v0}\right) \tag{5.17}$$

This small change is then added to the current joint variables of the virtual manipulator, giving $\theta_{v\,new} = \theta_{v0} + \Delta\theta_v^*$. The new position of the virtual manipulator is then found as $X_v = \Phi_v(\theta_v)$. By iterating this procedure until the error is small enough, the virtual manipulator can be position as close as possible.

For the simple case already presented in figure 5.3, a closed form solution can be found. The Jacobian for the virtual manipulator was already found as

$$J_v = \begin{bmatrix} -L\sin\theta_v \\ L\cos\theta_v \\ 1 \end{bmatrix} \tag{5.18}$$

This can be substituted into equation (5.17), along with choosing a weighing matrix of W_e as

$$W_e = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & w_3 \end{bmatrix} \tag{5.19}$$

Substituting:

$$\begin{aligned} \Delta\theta_v^* &= \left(J_{v0}^T W_e J_{v0}\right)^{-1} J_{v0}^T \left(X_r - X_v\right) \\ &= \left(\begin{bmatrix} -L_v\sin\theta_v & L_v\cos\theta_v 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & w_3 \end{bmatrix} \begin{bmatrix} -L_v\sin\theta_v \\ L_v\cos\theta_v \\ 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} -L_v\sin\theta_v & L_v\cos\theta_v 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & w_3 \end{bmatrix} \begin{Bmatrix} x_r - x_v \\ y_r - y_v \\ \theta_r - \theta_v \end{Bmatrix} \\ \Delta\theta_v^* &= \theta_r - \theta_v \end{aligned} \tag{5.20}$$

With this specific choice for W_e, the error of the system is all located in the angle portion of the virtual manipulator ($e = \begin{bmatrix} 0 & 0 & \theta_r - \theta_v \end{bmatrix}^T$).

The next step is to develop the control law to be used by the PHANTOM. Since three errors are known (x-position, y-position, and angular orientation), a classic controller can be created using proportional control (and derivative control if more damping is desired in the system). Looking at equation (5.13), the cartesian force provided by the virtual manipulator is known. Suppose a force F_H (the hand force provided by the user) is applied, the sum of these forces gives the motion-producing force on the system:

$$\begin{aligned} f_{motion} &= F_H - F \\ &= F_H - J_v \left(J_V^T J_v \right)^{-1} J_v^T F_H \\ &= \left(I - J_v \left(J_V^T J_v \right)^{-1} J_v^T \right) F_H \end{aligned} \tag{5.21}$$

Now, letting F_H be governed by $K_p e + K_d \dot{e}$, the control law takes the form

$$\tau_h = J_r^T \left(I - J_v \left(J_V^T J_v \right)^{-1} J_v^T \right) (K_p e + K_d \dot{e}) \tag{5.22}$$

Overall, the equation of the system looks like

$$H_{eq} \ddot{\theta} + C_{eq} \dot{\theta} = \left(I - J_v \left(J_V^T J_v \right)^{-1} J_v^T \right) (K_p e + K_d \dot{e}) - J_r^T f_H \tag{5.23}$$

where H_{eq} is an equivalent inertia of the system and C_{eq} is an equivalent damping of the system. The PHANTOM was designed to have low inertia and damping so $H_{eq} = C_{eq} \approx 0$. Equation (5.23) can then easily be rearranged (ignoring the derivative portion of the control law) into equation (5.24). Notice the conversion to the weighted pseudo-inverse calculation and the omission of the derivative control.

$$f_H = \left(I - W_A J_v \left(J_V^T W_A J_v \right)^{-1} J_v^T \right) (K_p e) \tag{5.24}$$

For the specific case developed, a weighing matrix must be chosen:

$$W_A = \begin{bmatrix} w_1 & 0 & 0 \\ 0 & w_2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{5.25}$$

Now that all the quantities are known, they can be substituted into equation (5.24). Being sure to take the appropriate transposes and inverses of the virtual manipulator Jacobian, the equation results in

$$f_H = \begin{Bmatrix} f_x \\ f_y \\ M_z \end{Bmatrix} = \begin{bmatrix} 1 - \frac{w_1 \sin^2\theta_v}{w_1 \sin^2\theta_v + w_2 \cos^2\theta_v} & \frac{w_1 \sin\theta_v \cos\theta_v}{w_1 \sin^2\theta_v + w_2 \cos^2\theta_v} & \frac{w_1 \sin\theta_v}{L_v\left(w_1 \sin^2\theta_v + w_2 \cos^2\theta_v\right)} \\ \frac{w_2 \cos\theta_v \sin\theta_v}{w_1 \sin^2\theta_v + w_2 \cos^2\theta_v} & 1 - \frac{w_2 \cos^2\theta_v}{w_1 \sin^2\theta_v + w_2 \cos^2\theta_v} & \frac{w_2 \cos\theta_v}{L_v\left(w_1 \sin^2\theta_v + w_2 \cos^2\theta_v\right)} \\ 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} e_x \\ e_y \\ e_\theta \end{Bmatrix} \tag{5.26}$$

Now, for a simple example of the weighing, let $w_1 = w_2 = 1$. By substituting these weights:

$$\begin{Bmatrix} f_x \\ f_y \\ M_z \end{Bmatrix} = \begin{bmatrix} 1 - \frac{\sin^2\theta_v}{\sin^2\theta_v + \cos^2\theta_v} & \frac{\sin\theta_v \cos\theta_v}{\sin^2\theta_v + \cos^2\theta_v} & \frac{\sin\theta_v}{L_v\left(\sin^2\theta_v + \cos^2\theta_v\right)} \\ \frac{\cos\theta_v \sin\theta_v}{\sin^2\theta_v + \cos^2\theta_v} & 1 - \frac{\cos^2\theta_v}{\sin^2\theta_v + \cos^2\theta_v} & \frac{\cos\theta_v}{L_v\left(\sin^2\theta_v + \cos^2\theta_v\right)} \\ 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} e_x \\ e_y \\ e_\theta \end{Bmatrix} \tag{5.27}$$

Next, several trigonometric identities can be substituted to simplify this even further, namely

$$\begin{aligned} \sin^2(\theta) + \cos^2(\theta) &= 1 \\ 2\sin(\theta)\cos(\theta) &= \sin(2\theta) \end{aligned} \tag{5.28}$$

resulting in equation (5.29)

$$\begin{Bmatrix} f_x \\ f_y \\ M_z \end{Bmatrix} = \begin{Bmatrix} \cos^2(\theta_v)\, k_p e_x + \frac{1}{2}\sin(2\theta_v)\, k_p e_y + \frac{\sin(\theta_v)}{L_v} k_p e_\theta \\ \frac{1}{2}\sin(2\theta_v)\, k_p e_x + \sin^2(\theta_v)\, k_p e_y + \frac{\cos(\theta_v)}{L_v} k_p e_\theta \\ 0 \end{Bmatrix} \tag{5.29}$$

The proportional gains (the K_p's) can be chosen somewhat arbitrarily—the gains for the cartesian (x- and y-errors) were set the same, but the angle gain was set much larger (500 times larger). With only a small gain, twisting the stylus in the plane had very little effect,

so by increasing its gain, the slight number computer will correspond to a much larger force acting on the stylus.

5.3 Coding and Results

Bezier Curve Method

The first step in coding this constraint is to determine the angles associated with the points P_p and P_o. As explained previously, the orientation of the stylus can be determined by noticing that the z-axis of frame 6 points along the stylus. Because of this, the transform returned by the device query gives the three components of the z-axis in the workspace. By using the desired elements of the returned array (the returned array is stored as a one-dimensional data variable, so by using equation (5.30), the correct elements can be determined), the orientation angle can be determined. In the example of working in the XY plane, the elements needed are $T[8]$ and $T[9]$. Using the notation of before, this angle is α, and can be found as $\alpha = atan2\,(T[9],\, T[8])$.

$$\begin{bmatrix} {}^{w}_{6}\hat{X}_x & {}^{w}_{6}\hat{Y}_x & \boxed{{}^{w}_{6}\hat{Z}_x} & P_x \\ {}^{w}_{6}\hat{X}_y & {}^{w}_{6}\hat{Y}_y & {}^{w}_{6}\hat{Z}_y & P_y \\ {}^{w}_{6}\hat{X}_z & {}^{w}_{6}\hat{Y}_z & {}^{w}_{6}\hat{Z}_z & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} T[0] & T[4] & \boxed{T[8]} & T[12] \\ T[1] & T[5] & T[9] & T[13] \\ T[2] & T[6] & T[10] & T[14] \\ T[3] & T[7] & T[11] & T[15] \end{bmatrix} \tag{5.30}$$

The second angle is determined by the vector formed by the center of the circle and the current stylus position, as described earlier. Because "atan2" is a four quadrant arc-tangent, correct signs of the angles are returned. However, there is still the possibility that the two angles are separated by more than 180^o. This can only occur is one of the angles is negative (since atan2 returns angles from $-\pi < \theta \leq \pi$, one angle would have to be negative for the difference between the two angles to be greater than 180^o or π radians). This can be checked easily and if it is the case, 360^o (2π radians) can be added to the small angle, leaving the angle located at the same location on the unit circle, but larger so that the correct desired angle can be computed. As example of this, consider figure 5.6.

In this example, $\alpha = \frac{-5\pi}{6}$, $\beta = \frac{\pi}{2}$, and the "influence factor $t = 0.4$. The distance from α to

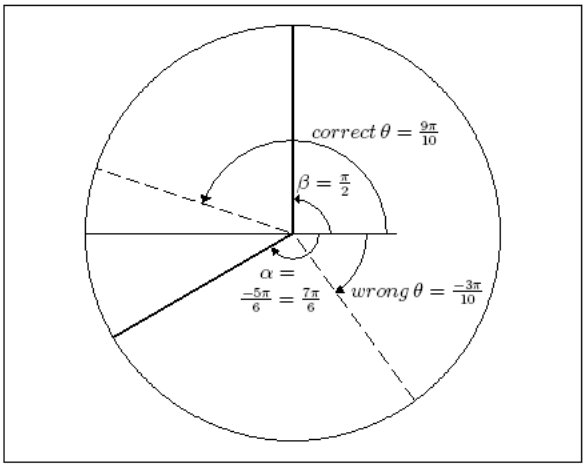

Figure 5.6 Choosing the right angle for combination constraint

β is $\frac{\pi}{2} + \frac{5\pi}{6} = \frac{4\pi}{3}$, which is greater than π radians. If the angle θ was to be computed without altering α, it would give

$$\begin{aligned}
\theta &= \alpha + t\,(\beta - \alpha) \\
\theta &= \frac{-5\pi}{6} + \frac{4}{10}\left(\frac{\pi}{2} - \frac{-5\pi}{6}\right) \\
&= \frac{-5\pi}{6} + \frac{4}{10}\left(\frac{8\pi}{6}\right) \\
&= \frac{-5\pi}{6} + \frac{8\pi}{15} \\
&= \frac{-3\pi}{10}
\end{aligned} \tag{5.31}$$

This angle, as already seen in figure 5.6 to be on the incorrect (i.e. long) path from α to β. By adjusting α by 2π, this will adequately adjust the resultant angle θ

$$\begin{aligned}
\theta &= \alpha + t\,(\beta - \alpha) \\
\theta &= \frac{7\pi}{6} + \frac{4}{10}\left(\frac{\pi}{2} - \frac{7\pi}{6}\right) \\
&= \frac{7\pi}{6} + \frac{4}{10}\left(\frac{-4\pi}{6}\right) \\
&= \frac{7\pi}{6} - \frac{4\pi}{15} \\
&= \frac{9\pi}{10}
\end{aligned} \tag{5.32}$$

So, by simply checking if the angular difference between the two angles is greater than 180^o (π radians) the correct angle is easily determined. The actual code used for this constraint is shown in Code 5.1. As already mentioned, this code can be altered to allow for other principle planes with ease. For these cases, the correct components must be used when computing the arctangents (for example, if the XZ plane is being using, the first angle "phi" would need to be computed as "phi=atan2(state.transform[10],state.transform[8])").

Figure 5.7 shows three time histories for a trial set of data. The top line is the angle α, representing the angle that corresponds to the point on the circle with the same outward normal has the same orientation of the stylus. The bottom line represents the angle β, the angle from the positive x-axis to the nearest point on the circle. On this particular plot, the middle line (denoted with the many small "x" marks) represents the calculated position based a Bezier parameter value of 0.5 (incidentally halfway between α and β). During time when all three lines are close to each other, the stylus of the PHANTOM is oriented normal to the circle (most predominantly between eight and nine seconds).

Virtual Manipulator Method

Since the coding of the Bezier curve method was already performed, applying the virtual manipulator code is fairly straight-forward. A closed form solution for the desired problem was computed, so this was coded instead of an iterative solution. As already described, the z-axis of frame {6} is along the stylus and is used to determine the orientation of the stylus. The code for this constraint can be seen in code 5.2. The third component of the the force vector

Code 5.1 Orientation based XY planar circle code

```
1  // find orientation angle based on current transform
2  float phi=atan2(state.transform[9],state.transform[8]);
3
4  //  Calculate point on the sphere that has the same normal orientation
5  point1[0]=circleCenter[0]+circleRad*cos(phi);
6  point1[1]=circleCenter[1]+circleRad*sin(phi);
7  point1[2]=circleCenter[2];
8
9  //  Compute the vector from the current stylus position to the center of the circle, then find its magnitude
10 posvect=state.position-circleCenter;
11 posvectmag=sqrt(((float)posvect[0])*((float)posvect[0])+
12                  ((float)posvect[1])*((float)posvect[1]));
13
14 //  Compute desired point, manually set z-dimension to where ever the circle lies
15 desiredPos=circleRad*posvect/posvectmag;
16 desiredPos[2]=circleCenter[2];
17
18 //  find angle of same orientation from positive x-axis
19 float alpha=atan2((point1[1]-circleCenter[1]),(point1[0]-circleCenter[0]));
20
21 //  find angle of the nearest point from positive x-axis
22 float beta=atan2((desiredPos[1]-circleCenter[1]),(desiredPos[0]-circleCenter[0]));
23
24 //  Adjust to get0 < θ < pi
25 if (abs(beta-alpha)>PI)
26 {
27   alpha=alpha+2*PI;
28 }
29
30 // Introduce the influence factor. Ranges from 0 to 1
31 // r2=0 - just orientation, r2=1 - just position
32 float r2=0.5;
33
34 //  Compute the desired angle as combination of the two previous angles with the influence factor
35 float theta=alpha+r2*(beta-alpha);
36
37 // Compute the desired position as the point on the circle with the desired angle
38 desiredPos[0]=circleCenter[0]+circleRad*cos(theta);
39 desiredPos[1]=circleCenter[1]+circleRad*sin(theta);
40 desiredPos[2]=circleCenter[2];
41
42 // Calculate force
43 force=kp*(desiredPos-state.position);      {optional: +ke*(desiredVel-state.velocity);}
```

("force[2]") simply pulls the stylus to the plane where the virtual manipulator is operating.

Figure 5.8 shows the results of using this code. As can be seen in the code, the proportional gains associated with error in the x- and y-directions are only 0.5 whereas the gain associated with the angular error is 250. The image shows the desired circle of rotation (the pivot of the virtual manipulator is at the center of this circle). Also shown are the stylus (small, thick circle and thick line), the nearest point on the circle (the square), the tangent at that location (dashed line) and a representative force vector (traveling in the direction of the tangent, not to scale).

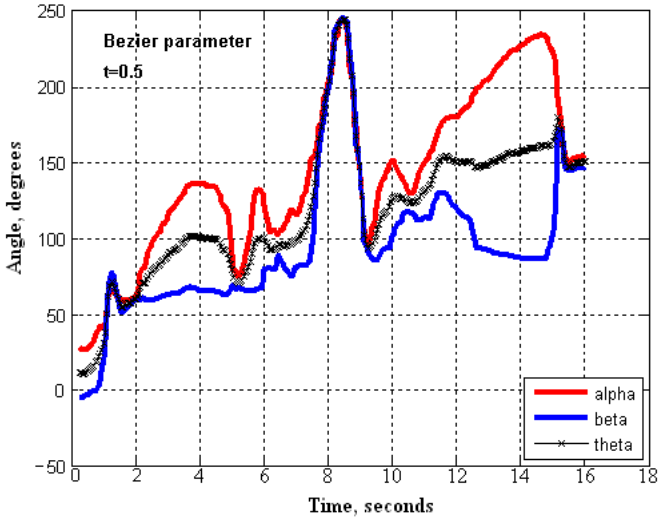

Figure 5.7 Bezier curve method angle histories

Code 5.2 Virtual manipulator in XY plane code

```
1
2  thetaR=atan2(state.transform[9],state.transform[8]);
3  thetaV=atan2(state.position[1],state.position[0]);
4
5  Perror[0]=(pivotLoc[0]+Lv*cos(thetaV))-state.position[0];
6  Perror[1]=(pivotLoc[1]+Lv*sin(thetaV))-state.position[1];
7  Perror[2]=thetaV-thetaR;
8
9  fx=((cos(thetaV))*(cos(thetaV))*0.5*Perror[0]+(sin(thetaV))*
10      (cos(thetaV))*0.5*Perror[1])+(250*Perror[2]*sin(thetaV)/Lv);
11 fy=((cos(thetaV))*(sin(thetaV))*0.5*Perror[0]+(sin(thetaV))*
12      (sin(thetaV))*0.5*Perror[1])+(-250*Perror[2]*cos(thetaV)/Lv);
13
14 force[0]=fx;
15 force[1]=fy;
16 force[2]=-kp[2]*state.position[2];
```

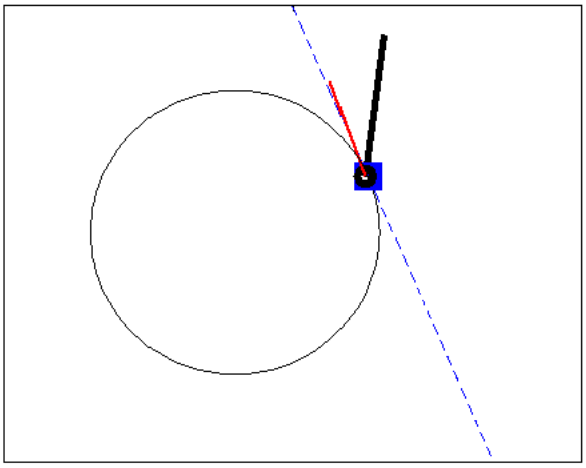

Figure 5.8 Virtual manipulator results

5.4 Orientation based cylinder

A simple extension of the Bezier curve-based constraint is to extend it to a cylinder. In actuality, this constraint is very simple to code. Given a simple planar cylinder in the XY plane, for example, the force component in the z-direction should be zero (just as was the case in the regular cylinder constraint developed earlier). This means the only real difference between coding this cylinder and circle above is to change the force component along the axis of the cylinder

Code 5.3 Orientation based XY planar cylinder code

```
1  // find orientation angle based on current transform
2  float phi=atan2(state.transform[9],state.transform[8]);
3
4  //  Calculate point on the sphere that has the same normal orientation
5  point1[0]=circleCenter[0]+circleRad*cos(phi);
6  point1[1]=circleCenter[1]+circleRad*sin(phi);
7  point1[2]=circleCenter[2];
8
9  //  Compute the vector from the current stylus position to the center of the circle, then find its magnitude
10 posvect=state.position-circleCenter;
11 posvectmag=sqrt(((float)posvect[0])*((float)posvect[0])+
12                  ((float)posvect[1])*((float)posvect[1]));
13
14 //  Compute desired point, manually set z-dimension to where ever the circle lies
15 desiredPos=circleRad*posvect/posvectmag;
16 desiredPos[2]=circleCenter[2];
17
18 //  find angle of same orientation from positive x-axis
19 float alpha=atan2((point1[1]-circleCenter[1]),(point1[0]-circleCenter[0]));
20
21 //  find angle of the nearest point from positive x-axis
22 float beta=atan2((desiredPos[1]-circleCenter[1]),(desiredPos[0]-circleCenter[0]));
23
24 //  Adjust to get0 < θ < pi
25 if (abs(beta-alpha)>PI)
26 {
27   alpha=alpha+2*PI;
28 }
29
30 // Introduce the influence factor. Ranges from 0 to 1
31 // r2=0 - just orientation, r2=1 - just position
32 float r2=0.5;
33
34 //  Compute the desired angle as combination of the two previous angles with the influence factor
35 float theta=alpha+r2*(beta-alpha);
36
37 // Compute the desired position as the point on the circle with the desired angle
38 desiredPos[0]=circleCenter[0]+circleRad*cos(theta);
39 desiredPos[1]=circleCenter[1]+circleRad*sin(theta);
40 desiredPos[2]=circleCenter[2];
41
42 // Calculate force
43 force=kp*(desiredPos-state.position);      {optional: +ke*(desiredVel-state.velocity);}
44 force[2]=0;      // Set z-force to zero since it is along the axis of the cylinder
```

Because of the complexity of attempting to create meaningful figures of three-dimensional objects (not to mention the time varying nature of the user input to the PHANTOM's stylus), a plot of this constraint is not provided.

CHAPTER 6. SUMMARY AND DISCUSSION

In this report, the PHANTOM Omni was explored as a robot instead of a three-dimensional data analysis tool. The basic, position-based constraints presented in Chapter 4 provided a basic stepping stone into the world of haptic interactions. These interactions were expanded to include the unactuated degrees of freedom (the degrees responsible for spatial orientation), allowing for more complex interactions and a "springy" feel for the stylus.

The first use of these other degrees of freedom used the theory of a Bezier curve to determine the desired position of the stylus. Two angle were calculated based on the user's input via the stylus: the in plane orientation angle of the stylus and the angle along the desired circle to the nearest point. By finding the shortest angle between those two angles, the desired point can be placed anywhere along the arc connecting them. While this provided some interaction not normally found on the PHANTOM, it is not simple to adjust this to other scenarios (consider expanding this to a sphere where two points are found, once again the nearest physical point and the point with the same outward normal as the stylus). These other interactions would require complex math analysis and programming.

The next method explored was the use of a virtual manipulator. The virtual manipulator model for interaction provides a more powerful framework for more advanced haptic interactions. By integrating an iterative method to find the "nearest" position of a given virtual manipulator, nearly any constraint system can be determined. Since this method only requires the input of the Jacobian of the virtual robot, no additional math would have to be performed, allowing more complicated systems to be constructed (consider again the sphere described before—this represents a 2R robot with an easy to calculate Jacobian). In theory, this method of control does not need an underactuated robot like the PHANTOM. Robots

like the higher end PHANTOM Desktop (a six degree-of-freedom sensing and actuating robot made by Sensable) or even a PUMA 560 should be able to be constrained with this method.

For future work, it would be interesting to implement the iterative method for the virtual manipulator and to try several of the more advanced constraints. Another further step in this research would be to calibrate the interaction so that a given sensory response can be approximated. This type of calibration would allow the user of the device to differentiate between multiple inputs (as an example, suppose the PHANTOM is used to model the flexibility of several different cantilever beams), allowing for more interactive engineering design.

APPENDIX A. REVIEW OF LINEAR ALGEBRA

This appendix is intended as a brief refresher of linear algebra, particularly vector and matrix operations. For a more complete discussion of linear algebra, see [1].

Defining vectors and matrices

A vector is an ordered sequence of numbers. In this regard, a vector can be thought of as one–dimensional representation of data. In many branches of sciences and engineering, vectors have three components — corresponding to the three priciple spatial axes. A vector represents a magnitude and direction of a given quantity. Examples of vectors include: displacements, velocities, accelerations, forces, fluid flow, among many others. Vectors are different from scalar quantities in the regard that a scalar is just the number (for example, a speed of $3\frac{m}{s}$) and a vector contains a direction (compared to $3\frac{m}{s}$ along the x-axis). A vector can be defined as shown in A.1.

$$\vec{\mathbf{a}} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \tag{A.1}$$

The magnitude of the vector can be calculated as

$$|\vec{\mathbf{a}}| = \sqrt{a_1^2 + a_2^2 + a_3^2} \tag{A.2}$$

On the other hand, a matrix is a collection of data arranged into rows and columns, making the matrix a two–dimensional representation of data. These matrices can have arbitrary size, as demonstrated in (A.3). This is said to be an $m \times n$ matrix.

$$\mathbf{b} = \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m,1} & b_{m,2} & \cdots & b_{m,n} \end{bmatrix} \tag{A.3}$$

Adding, subtracting and scalar multiplication

Adding and subtracting

When performing addition or subtraction of vectors, the lengths of the two (or more) vectors must be the same — that is, they contain the same number of elements. In the simple case of two three-element long vectors **a** and **b**, their sum is found as

$$\begin{aligned} \vec{\mathbf{a}} &= \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \\ \vec{\mathbf{b}} &= \begin{bmatrix} b_x & b_y & b_z \end{bmatrix} \\ \vec{\mathbf{a}} + \vec{\mathbf{b}} &= \begin{bmatrix} a_x + b_x & a_y + b_y & a_z + b_z \end{bmatrix} \end{aligned} \tag{A.4}$$

The processe of matrix addition or substraction closely follows the rule for vector addition or subtraction. Since the dimension of the one–dimensional vectors must match in order to add/subtract, both dimensions of the two–dimensional matrices must match to be able to add/subtract them, as demonstrated in equation (A.5).

$$\begin{aligned} \mathbf{A} &= \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \\ \mathbf{B} &= \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \end{bmatrix} \\ \mathbf{A} + \mathbf{B} &= \begin{bmatrix} a_{1,1} + b_{1,2} & a_{1,2} + b_{1,2} & a_{1,3} + b_{1,3} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} & a_{2,3} + b_{2,3} \end{bmatrix} \end{aligned} \tag{A.5}$$

Scalar multiplication

Scalar multiplication of a vector or matrix involves multiplying each element within the vector or matrix by the scalar factor, as shown in (A.6), a simple example of a scalar multiplying a vector.

$$\begin{aligned} \vec{\mathbf{a}} &= \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \\ c\vec{\mathbf{a}} &= \begin{bmatrix} ca_x & ca_y & ca_z \end{bmatrix} \end{aligned} \tag{A.6}$$

A scalar multiplying a matrix is handled similarly, where the scalar multiplies every element of the matrix.

$$\begin{aligned} \mathbf{B} &= \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \end{bmatrix} \\ c\mathbf{B} &= \begin{bmatrix} cb_{1,1} & cb_{1,2} & cb_{1,3} \\ cb_{2,1} & cb_{2,2} & cb_{2,3} \end{bmatrix} \end{aligned} \tag{A.7}$$

Vector and matrix multiplication

The dot product

The first method of multiplying two vectors is known as the dot product (or the inner product, depending on the text consulted). Like with adding vectors, the dot product requires that the dimension of the vectors are the same. The dot product is then defined as

$$\begin{aligned} \vec{\mathbf{x}} &= \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \\ \vec{\mathbf{y}} &= \begin{bmatrix} y_1 & y_1 & \cdots & y_n \end{bmatrix} \\ \vec{\mathbf{a}} \cdot \vec{\mathbf{b}} &= x_1 y_1 + x_2 y_2 + \cdots + x_n y_n = \sum_{i=1}^{n} x_i y_i \end{aligned} \tag{A.8}$$

An alternate way of defining the dot product has a more geometric interpretation:

$$\vec{\mathbf{x}} \cdot \vec{\mathbf{y}} = \|\vec{\mathbf{x}}\| \, \|\vec{\mathbf{y}}\| \cos(\theta) \tag{A.9}$$

where θ is the angle between the two vectors

The cross product

In contrast to the dot product, the cross product of two vectors returns a vector instead of scalar. This vector, it happens, is orthogonal to the two initial vectors — that is, the resultant vector is the normal to the plane formed by the initial vectors. The cross product can be defined as

$$\vec{\mathbf{x}} \otimes \vec{\mathbf{y}} = \|\vec{\mathbf{x}}\| \, \|\vec{\mathbf{y}}\| \sin(\theta) \tag{A.10}$$

Like with the dot product, θ is the in-plane angle between the two initial vectors. For the case of two three-dimensional vectors, the cross product can be written in two other ways. But first, these different operations must be discussed.

Matrix multiplication

As seen above, vector multiplication requires that both vectors have the same length. When multiplying matrices, only the inner dimension of the matrices must match. For example, if $\mathbf{a}$ is a (2×4) matrix and $\mathbf{b}$ is a (4×3) matrix, the multiplication is possible because $(2 \times 4) \times (4 \times 3)$ has the same inner dimension. This also allows the size of the result to be seen — (2×3) — because the inner dimesions cancel each other out. Generally, matrix multiplication can be defined as $\mathbf{A_{m,n}} \times \mathbf{B_{n,p}} = \mathbf{C_{m,p}}$ whose $(i,j)^{th}$ entry is the dot product of the i^{th} row of $\mathbf{A}$ with the j^{th} column of $\mathbf{B}$.

As example of this multiplication consider the following

$$\begin{aligned}
\mathbf{A} &= \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \end{bmatrix} \\
\mathbf{AB} &= \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \end{bmatrix} \\
\mathbf{AB} &= \begin{bmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + a_{1,3}b3,1 & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} + a_{1,3}b3,2 \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} + a_{2,3}b3,1 & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} + a_{2,3}b3,2 \end{bmatrix}
\end{aligned} \tag{A.11}$$

Looking at multiplication in this manner allows a generalization of the vector operations described. If a vector is horizontal, it can be thought of as a $(1 \times n)$ matrix. Likewise, if a vector is aligned vertically, it can be thought of as $(n \times 1)$ matrix. With this new knowledge, the vector cross product can be rewritten as the multiplication of a (3×3) matrix with a (3×1) vector, resulting in a (3×1) vector, as follows in equation (A.12)

$$\begin{aligned}
\vec{\mathbf{x}} &= \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \\
\vec{\mathbf{y}} &= \begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} \\
\vec{\mathbf{x}} \otimes \vec{\mathbf{y}} &= \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \\
\vec{\mathbf{x}} \otimes \vec{\mathbf{y}} &= \begin{bmatrix} x_2 y_3 - x_3 y_2 \\ x_3 y_1 - x_1 y_3 \\ x_2 y_1 - x_1 y_2 \end{bmatrix}
\end{aligned} \tag{A.12}$$

Other useful operations

The determinant

Without going into too much theoretical linear algebra, the determinant of a matrix relates a singular value (or expression) to a matrix. Given a matrix A, the determinant has several different notations:

$$\det A = |A| \tag{A.13}$$

As a simple example, consider the 2×2 matrix

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \tag{A.14}$$

The determinant of this can be found as

$$\det A = \begin{vmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{vmatrix} = a_{1,1}a_{2,2} - a_{1,2}a_{2,1} \tag{A.15}$$

As a more complex example, consider the 3×3 matrix

$$B = \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} \tag{A.16}$$

In order to find the determinant of this matrix, a co-factor expanion is used

$$\begin{aligned}\det B &= \begin{vmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{vmatrix} = b_{1,1} \begin{vmatrix} b_{2,2} & b_{2,3} \\ b_{3,2} & b_{3,3} \end{vmatrix} - b_{1,2} \begin{vmatrix} b_{2,1} & b_{2,3} \\ b_{3,1} & b_{3,3} \end{vmatrix} + b_{1,3} \begin{vmatrix} b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \end{vmatrix} \\ &= b_{1,1}\left(b_{2,2}b_{3,3} - b_{2,3}b_{3,2}\right) - b_{1,2}\left(b_{2,1}b_{3,3} - b_{2,3}b_{3,1}\right) \\ &\quad + b_{1,3}\left(b_{2,1}b_{3,2} - b_{2,2}b_{3,1}\right)\end{aligned} \tag{A.17}$$

The result of equation (A.17) is remarkable similar to the result of equation (A.12). With a small alteration, the cross product can then be written as:

$$\begin{aligned}\vec{\mathbf{x}} &= \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \\ \vec{\mathbf{y}} &= \begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} \\ \vec{\mathbf{x}} \otimes \vec{\mathbf{y}} &= det \begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{bmatrix} \\ &= \mathbf{i}\left(x_2y_3 - x_3y_2\right) + \mathbf{j}\left(x_3y1 - x_1y_3\right) + \mathbf{k}\left(x_1y_2 - x_2b_1\right)\end{aligned} \tag{A.18}$$

which then can be written in standard vector form as

$$\vec{\mathbf{x}} \otimes \vec{\mathbf{y}} = \begin{bmatrix} x_2y_3 - x_3y_2 & x_3y1 - x_1y_3 & x_1y_2 - x_2b_1 \end{bmatrix} \tag{A.19}$$

The transpose

The last operation covered in this review is the transpose. The transpose is used to reorder the elements of a vector (or matrix). As an example of using the transpose, a column vector can be transposed to become a horizontal vector. Given a vector (or matrix) $\mathbf{A}$, the transpose is denoted $\mathbf{A}^T$. A general formula for the transpose states that the $\left(i^{th}, j^{th}\right)$ element of $\mathbf{A}$ becomes the $\left(j^{th}, i^{th}\right)$ element of $\mathbf{A}^T$. Equation (A.20) shows how to use the transpose on a vector and a matrix.

$$\mathbf{A} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \end{bmatrix}$$
$$\mathbf{A}^T = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \qquad \mathbf{B}^T = \begin{bmatrix} b_{1,1} & b_{2,1} \\ b_{1,2} & b_{2,2} \\ b_{1,3} & b_{2,3} \end{bmatrix} \tag{A.20}$$

Vector normalization

The process of vector normalization involves the scaling of the vector such that its magnitude is unity. A scaled version of this vector is simply the original divided by its magnitude

$$\begin{aligned} \vec{\mathbf{a}} &= \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \\ |\vec{\mathbf{a}}| &= \sqrt{a_x^2 + a_y^2 + a_z^2} \\ \hat{\mathbf{a}} &= \frac{\vec{\mathbf{a}}}{|\vec{\mathbf{a}}|} \end{aligned} \tag{A.21}$$

The "'hat"' notation is usually adopted to denote vectors of unit length.

APPENDIX B. REVIEW OF CONTROL THEORY

This section provides a brief review of the concepts of classical controls theory. Main concepts will be system modeling in time and s-domains, time domain specifications and basic control structure. For a more in-depth look at classical control theory, see [8].

System modeling

The first step in modeling a system is to find the equation of motion. These exact equations depend on the system in question, but generalities can be developed. Many systems of practical importance can be approximated by either first or second order systems (that is, function of at most the first or second derivative of the desired variable). If the equations do not fall into these categories, it is possible to use approximations and simplifications to get the system into a first or (more likely the case) second order system. If certain *modes* of movement behave much faster than the others, their part of the response can be neglected to the slower ones.

First order systems

A first order system is characterized by a first order differential equation, such as (for example) seen in equation (B.1) (the reason for the "a" before the forcing function $f(t)$ will become apparent it a bit).

$$\dot{x} + ax = a\, f(t) \tag{B.1}$$

With many control applications, the Laplace transform is used to analyze systems. The Laplace transform is defined as

$$\mathscr{L}[f(t)] = F(s) = \int_{0-}^{\infty} f(t)\, e^{-st} dt \tag{B.2}$$

Ignoring the initial conditions of the system, a transfer function for the system shown in equation (B.1) can be found

$$\begin{aligned} \mathscr{L}\left[\dot{x}+ax\right] &= \mathscr{L}\left[a\,f\left(t\right)\right] \\ X\left(s+a\right) &= a\,F\left(s\right) \\ T\left(s\right) = \frac{X\left(s\right)}{F\left(s\right)} &= \frac{a}{s+a} \end{aligned} \tag{B.3}$$

When analyzing transfer functions, the final value theorem can be used. This states that the steady state value the system approaches can be found by setting "s" to zero in the transfer function. In mathematical symbols

$$\lim_{t\to\infty} f\left(t\right) = \lim_{s\to 0+} F\left(s\right) \tag{B.4}$$

By applying the final value theorem to equation (B.3), the system is seen to become one at steady state. This is easily verified by simulating the systing, shown in figure B.1.

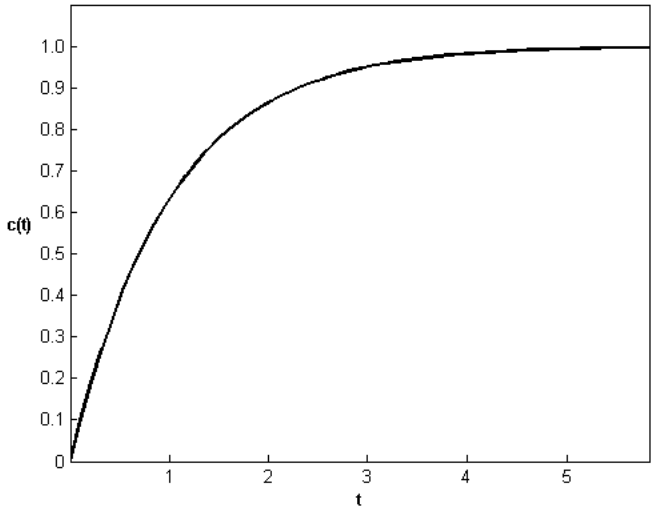

Figure B.1 First order system response

If the system equation is changed slightly from (B.1) to a slightly different:

$$\dot{x}+ax = f\left(t\right) \tag{B.5}$$

Taking the Laplace transform gives a new transfer function

$$\begin{aligned} \mathscr{L}\left[\dot{x}+ax\right] &= \mathscr{L}\left[f\left(t\right)\right] \\ X\left(s+a\right) &= F\left(s\right) \\ T\left(s\right) = \frac{X\left(s\right)}{F\left(s\right)} &= \frac{1}{s+a} \end{aligned} \tag{B.6}$$

Notice the only difference between equations (B.1) and (B.5) is the term "a" multiplying the forcing function. This will cause the output to reach $\frac{1}{a}$ instead of reaching one. Likewise, if the "a" multiplying the forcing function in (B.1) was changed to "3a", the output would reach a magnitude of three. These cases are easily seen using equation (B.4).

Second order systems

A second order system consists of a system with the second derivative of a desired variable, as demonstrated in equation (B.7)

$$a\ddot{x}+b\dot{x}+cx = f\left(t\right) \tag{B.7}$$

As with the first order system, the Laplace transform can be taken (once again ignoring initial conditions) to give a transfer function

$$\begin{aligned} \mathscr{L}\left[a\ddot{x}+b\dot{x}+cx\right] &= \mathscr{L}\left[f\left(t\right)\right] \\ X\left(as^2+bs+c\right) &= F\left(s\right) \\ T\left(s\right) = \frac{X\left(s\right)}{F\left(s\right)} &= \frac{1}{as^2+bs+c} \end{aligned} \tag{B.8}$$

The denominator of this equation is called the characteristic equation of the system—it contains the dynamics of the physical system as shown in equation (B.7) From this equation, a slight generalization can be created to describe an arbitrary second order system. The two parameters that adequately describe a second order system are the natural frequency (ω_n) and the damping ratio (ζ). A damping ratio of zero represents an un-damped system (it will oscillate forever). A damping ratio of one is considered critically damped. Damping ratios from zero to one are classified as under-damped, and damping ratios above one are classified as over-damped A general equation can be formed as

$$T = \frac{C\omega_n^2}{s^2+2\zeta\omega_n s+\omega_n^2} \tag{B.9}$$

In equation (B.9), the factor "C" is the steady state gain of the system, determined using the final value theorem. A plot of this general function, for a variety of damping ratios, is shown in figure

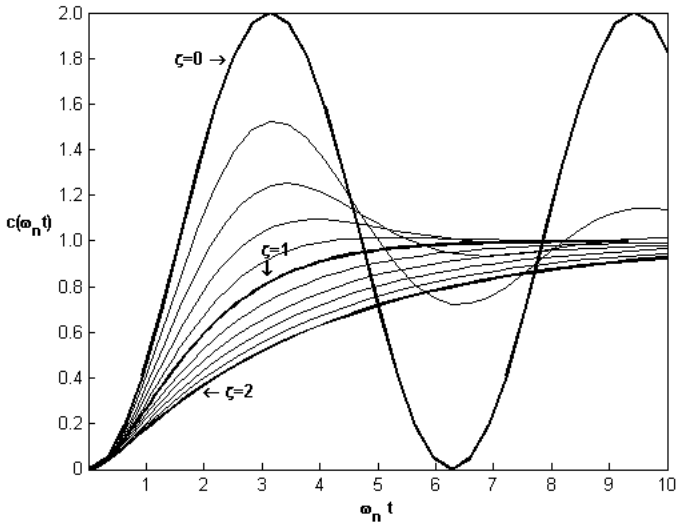

Figure B.2 Generic second order responses

Higher order simplification

As previously mentioned, higher order systems can often be simplified to a second order system. As an example, consider a third order transfer function given by

$$T(s) = \frac{5}{(s+1)(s+2)(s+30)} \tag{B.10}$$

In this form, the roots of the characteristic equation are easily seen as -1, -2 and -30. The root of -30 is greatly larger (and thus, faster) than the other two roots, that it is desired to see its effect on the overall dynamics. Employing a partial fraction expansion of equation (B.10) gives

$$T(s) = \frac{-\frac{5}{29}}{s+1} + \frac{\frac{5}{28}}{s+2} + \frac{\frac{5}{812}}{s+30} \tag{B.11}$$

From equation (B.11), the fast root of -30 can be easily neglected, leaving just a second order system, as seen in equation (B.12).

$$T(s) = \frac{-\frac{5}{29}}{s+1} + \frac{\frac{5}{28}}{s+2} \tag{B.12}$$

By simulating both of these systems, the results can be seen in figure B.3. It is seen that the error between these two systems is very small (at most 3×10^{-3}). Because of this, it is seen that the fast root of the system can be neglected to simplify the model. It is also possible to isolate different dynamic modes in the equations for a system. When dealing with control systems in state space (matrix) form, operations could be performed to cause the different modes to be isolated from each other. For example, it is might be possible to isolate the second order dynamics governing the roll angle (side-to-side) of an automobile from the second order dynamics governing the pitch angle (front-to-back) of the same automobile.

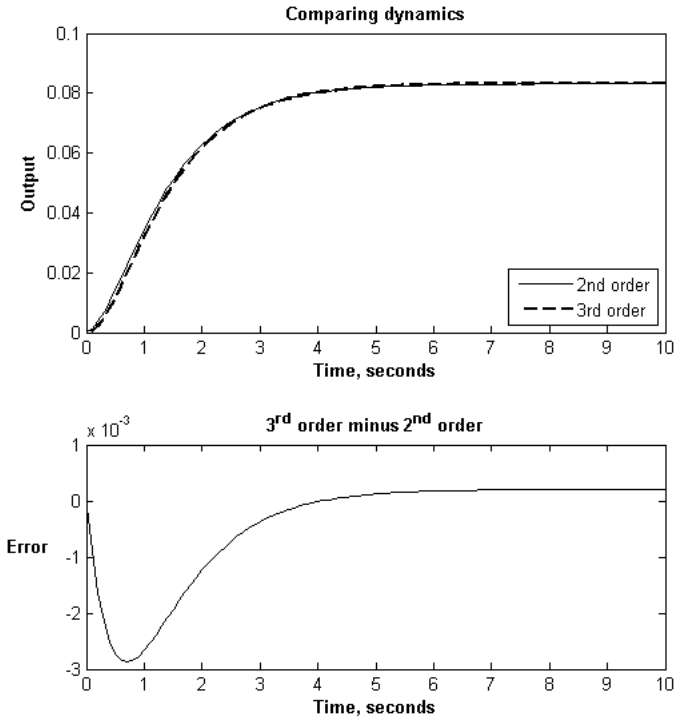

Figure B.3 Comparing third and second order responses

Time domain specifications

One of the common inputs to a system is a step input. Such an input occurs when something is applied to a system starting at one time and remaining "on" for the rest of time (there is much theory about classical control systems as linear, time-invariant systems). Examples of such an input would be turning a voltage source on at a given time in an electrical system or by adding mass to a mechanical system. First and second order systems can be generalized.

First order systems

Consider again the system found in equation (B.5). When applying a step input, the response of the generic system can be seen in figure B.4. A first order system experiences an exponential increate to its final value. Two parameters have been drawn on the figure that help to characterize the response: the rise time (T_r) and the settling time (T_s). The rise time is the time it takes the system to move from 10% of its final movement to 90% of its final movement (these values vary depending on the text consulted). The settling time is the time when the system is within 2% of its final value (once again, this number changes with the text used).

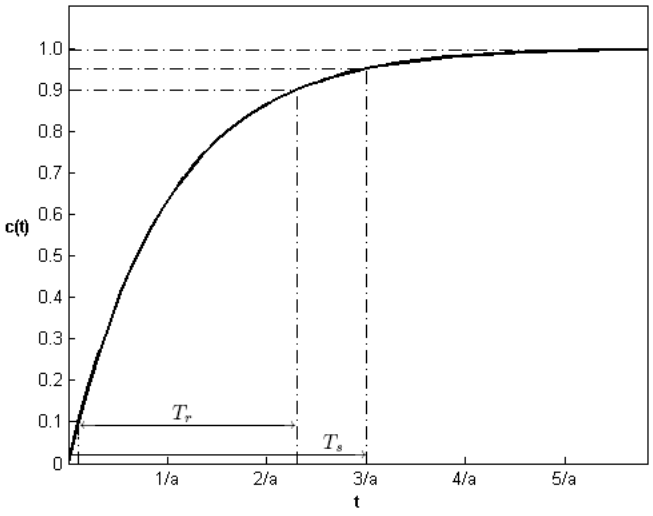

Figure B.4 First order system step response

Second order systems

There are several time-bases characteristics important to the analysis of a second order system. An example response is shown in figure B.5. As with the first order system, a second order system has a rise time and a settling time. For the most part, the definitions are the same as for first order systems with one slight difference. Because of the oscillatory nature of a second order system, it is possible that the system approaches the steady state value from either above or below (unlike just below, as depicted earlier in figure B.4 for the first order system). Both of these boundaries need to be watched to determine if the system stays within them. In addition to rise and settling times, the second order system has the peak time and the overshoot. Peak time, like its name implies is the time at which the system reaches its maximum value. The overshoot is the ammount above steady state the system reaches at the peak. This value is often expressed as a percentage (figure B.5 has roughly 38% overshoot).

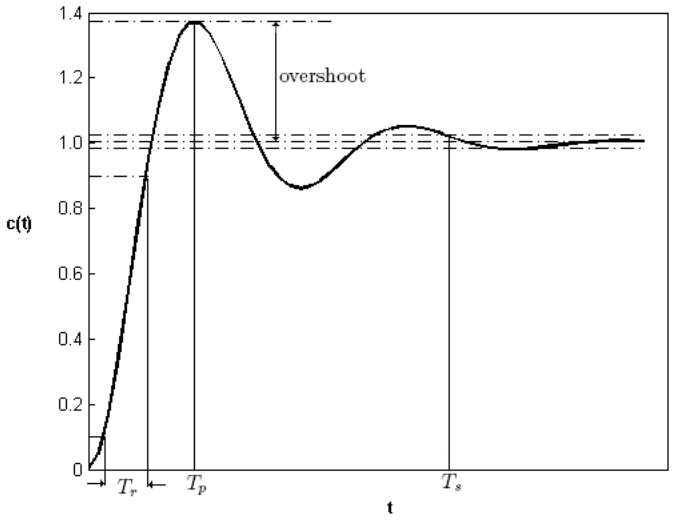

Figure B.5 Second order system step response

Basic control structures

There are many types of control structures. They range from the easily implemented proportional controller, to the phase lag and phase lead controllers used in frequency response

(see [9]), and ultimately to the difficult observer-based full-state feedback of state-space systems (see [3]). While the more advanced controllers do allow better control of complex systems, only three simple cases will be discussed here: proportional control, proportional plus derivative control and finally proportional, integral and derivative control. With all controllers, the input to the system input is to be defined by some sort of error in the measure of the system as compared to some reference signal. The examples presented will use mechanical systems, using position as the variable of interest.

Proportional (P)

The easiest control structure to use is a simple proportional controller. This controller looks at the difference between the position and a desired position. The input to the system is, like its name implies, proportional to this difference, as seen in equation (B.13).

$$F_{input} = K_p \left(x_{desired} - x_{measured} \right) \tag{B.13}$$

Proportional and derivative (PD)

PD control begins with the basic structure of the proportional controller but adds a derivative term, as seen in equation (B.14). This term often acts to slow the system down as it approaches the desired system (that is, the $\dot{x}_{desired}$ term is zero, since the system is to be at a given position at rest). Likewise, if the system is moving away from the desired position, this term will work to slow the system down.

$$F_{input} = K_p \left(x_{desired} - x_{measured} \right) + K_e \left(\dot{x}_{desired} - \dot{x}_{measured} \right) \tag{B.14}$$

Proportional, integral and derivative (PID)

The PID controller is perhaps one of the most implemented controllers. It combines the proportional and derivative elements of the previous two with an integral term. This term looks at the cumulative error of the system. This integral element is especially useful to correct for

steady state errors—if the system normally reaches only 0.8 when trying to reach 1.0, this integral term will grow large with time, causing the system to eventually reach the desired value.

$$F_{input} = K_p \left(x_{desired} - x_{measured}\right) + K_e \left(\dot{x}_{desired} - \dot{x}_{measured}\right) + K_i \int_0^t \left(x_{desired} - x_{measured}\right) \tag{B.15}$$

BIBLIOGRAPHY

[1] Stephen Andrilli and David Hecker. *Elementary Linear Algebra.* Harcourt Academic Press, second edition, 1999.

[2] John J. Craig. *Introduction to Robotics: Mechanics and Control.* Pearson Prentice Hall, third edition, 2005.

[3] Gene F. Franklin, J. David Powell, and Michael Workman. *Digital Control of Dynamic Systems.* Addison Wesley Longman, Inc., third edition, 1998.

[4] Richard S. Hartenberg and Jacques Denavit. *Kinematic Synthesis of Linkages.* McGraw-Hill, 1964.

[5] F.S. Hill, Jr. *Computer Graphics using OpenGL.* Prentice Hall, second edition, 2001.

[6] Jose San Martin and Gracian Trivino. A study of the manipulability of the phantom omni haptic interface. In *3rd Workshop in Virtual Reality Interactions and Physical Simulation*, pages http://www.eg.org/EG/DL/PE/vriphys/vriphys06/127–128.pdf.abstract.pdf, 2006.

[7] J. Michael McCarthy. *Geometric Design of Linkages*, volume 11 of *Interdisciplinary Applied Mathematics: Systems and Control.* Springer-Verlag, 2000.

[8] Norman S. Nise. *Control Systems Engineering.* John Wiley & Sons, Inc., fourth edition, 2004.

[9] Charles L. Phillips and H. Troy Nagle. *Digital Control System Analysis and Design.* Prentice Hall, third edition, 1995.

[10] Sensable Technologies. Specifications for the phantom omni haptic device. *http://www.sensable.com/documents/documents/PHANTOM_Omni_Spec.pdf*, 2004. Accessed June 26, 2007.