Enhancing the Dexterity of a Robot Hand Using Controlled Slip

by

David L. Brock

Submitted to the Department of Mechanical Engineering on May 1, 1987 in partial fulfillment of the requirements for the Degree of Master of Science in Mechanical Engineering

Abstract. Humans can effortlessly manipulate objects in their hands, dexterously sliding and twisting them within their grasp. Robots, however, have none of these capabilities, they simply grasp objects rigidly in their end effectors. To investigate this common form of human manipulation, an analysis of controlled slipping of a grasped object within a robot hand was performed. The Salisbury robot hand demonstrated many of these controlled slipping techniques, illustrating many results of this analysis.

First, the possible slipping motions were found as a function of the location, orientation, and types of contact between the hand and the object. Second, for a given grasp, the contact types were determined as a function of the grasping force and the external forces on the object. Finally, by changing the grasping force, the robot modified the constraints on the object and affect controlled slipping motions.

Thesis Supervisor: Dr. Kenneth Salisbury
Research Scientist
Laboratory of Artificial Intelligence

This empty page was substituted for a blank page in the original document.

Acknowledgments

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by Sandia National Laboratories under contract 23-2299, in part by the Systems Development Foundation, in part by the Office of Naval Research University Research Initiative Program under Office of Naval Research contract N00014-86-K-0685 and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

This empty page was substituted for a blank page in the original document.

Contents

1 Introduction **1**

1.1 Introduction 1
1.2 Overview 2
1.3 Outline 3

2 Constraint **5**

2.1 Introduction 5
2.2 Screws, wrenches, and twists 5
2.2.1 Screws 6
2.2.2 Wrenches 7
2.2.3 Twists 7
2.2.4 Transforming screw definitions 8
2.3 Contact types 11
2.4 Infinitesimal motion 11
2.4.1 Reciprocal screws 16
2.4.2 Repelling screws 16
2.4.3 Contrary screws 16
2.4.4 Permissible twist 17
2.5 Finite motions 18
2.5.1 Value of virtual coefficient 22
2.6 Multiple contacts 23
2.7 Permissible Motion 25

3 Strategies for manipulation **26**

3.1 Introduction 26
3.2 General strategy 27
3.3 Specific strategies 28

3.3.1 Gravity 28
3.3.2 Controlled accelerations 31
3.3.3 Free fingers 31
3.3.4 Other objects 32

4 Grasping force **34**
4.1 Introduction 34
4.2 Grasp force analysis 34
4.3 Two contacts 35
4.4 Three contacts 38
4.5 Four or more fingered grasps 42

5 Contact wrenches **44**
5.1 Introduction 44
5.2 Stiffness 44

6 Contact wrench/contact type relation **48**
6.1 Introduction 48
6.2 Two dimensions 48
6.3 Three dimensions 49

7 Controlled Slipping **56**
7.1 Introduction 56
7.2 Two dimensional example 56
7.2.1 Constraint 57
7.2.2 External wrench 61
7.2.3 Grasping force 64
7.2.4 Contact wrenches 65
7.2.5 Contact wrench / Contact type relation 72
7.2.6 Constraint state map 73
7.3 Three dimensional example 75

8 Implementation **83**
8.1 Introduction 83
8.2 Description of Hardware 83
8.3 Description of software 84
8.3.1 GRASP functions 86

9 Extensions and further research **97**
9.1 Introduction 97
9.2 Determining permissible motion 98
9.3 Determining constraint state 98
9.4 Global motion 99
9.5 Sensory feedback 99
9.6 Integrating manipulation techniques 101
9.7 Conclusion 102

A Coordinate frames **105**
A.1 Introduction 105
A.2 Hand frame 106
A.3 Finger frames 106
A.4 Fingertip frame 107
A.5 Contact frames 109
A.6 Grasp frame 109
A.7 Object frame 110

B Stress state in a fingertip **111**
B.1 Introduction 111
B.2 Problem definition 111
B.3 Analytic solution 112
B.4 Finite element solution 117
B.4.1 Refined mesh 121
B.5 Finite element vs. analytic 122
B.6 Solution 127
B.7 Conclusion 127

C Fingertip sensor **132**
C.1 Introduction 132
C.2 Theory 133
C.3 Design 134
C.4 Design of loadcell 136
C.4.1 Introduction 136
C.4.2 Fingertip sensor dimensions 136
C.4.3 Mechanical properties 138
C.4.4 General cantilever beam problem 139

C.4.5 Case 1 141
C.4.6 Case 2 142
C.4.7 Case 3 143
C.4.8 Case 4 146
C.4.9 Case 5 146
C.4.10 Case 6 147
C.4.11 Maximum stress 148

D Slip analysis software **155**
D.1 Introduction 155
D.2 GRASP 155

List of Figures

2.1 Transforming definitions of a screw from one coordinate frame to another 10
2.2 Contact types 12
2.3 Contact types (Continued) 13
2.4 Manipulator object interface 15
2.5 Wrench in cartesian coordinates 15
2.6 Twist defined at a contact point 18
2.7 Surface geometry in the neighborhood of the contact must considered to determine permissible motions 19
2.8 Projection of the contact trajectory onto the object surface . . 22
2.9 Expand definition of reciprocal twist 24

3.1 The wrist can be used to orient the object in a grasp so that gravity can be used to move the object 31
3.2 A force exerted on a grasped object can move it through a desired motion 34

4.1 Wrench from force through point contact 37
4.2 A pair of contacts can exert an arbitrary squeezing force . . . 40
4.3 The grasp force focus and grasp force magnitude span the three space of internal grasp solutions for three contacts . . . 44

5.1 Fingertip touching object 47
5.2 Fingertip modeled as a spring system 48

6.1 Two general bodies in contact 51
6.2 Pressure distribution in the area of contact 52
6.3 Approximation for contact type 56

7.1 Two dimensional example 57
7.2 Coordinate systems for two dimensional example 58
7.3 Graphic representation of twists in two dimensions 59
7.4 Set of unit twists may be represented by a collection of unit vectors in the twist coordinate system 60
7.5 Twists which lie in a plane in the twist coordinate system are replaced by a shaded disk 60
7.6 Twist which do not lie in a plane are replaced by sections of a sphere 61
7.7 Constraint states for the rectangular block ignoring the local surface geometry 62
7.8 Constraint states for the rectangular block including local surface geometry 63
7.9 Grasped rectangular block in a gravity field 65
7.10 Spring assemblage replaces contacts 66
7.11 Contact wrenches for fingers on block 71
7.12 Slipping criteria 74
7.13 Constraint state map 77
7.14 Fingers on a square 78
7.15 Permissible motions for square grasped on adjoining sides. . . 79
7.16 Constraint state map 80
7.17 Three fingers grasping a can 81
7.18 Coordinate systems used in the three dimensional example . . 81
7.19 Constraint state map for three fingers holding a cylinder . . . 82

8.1 Salisbury Robot Hand 84
8.2 Robot hand control scheme 85
8.3 Grasp Menu 87
8.4 Constraint state map produced by the computer 91
8.5 Robot can holding a can of soda 92
8.6 Constraint state map for robot grasping the can 93
8.7 Robot hand allowing a can to spin in its grasp 94
8.8 Robot hand spinning a can between two fingers 95
8.9 Robot hand manipulating a box 96

A.1 Hand frame 106
A.2 Finger frames 107

A.3 Phalange length and finger placement 108
A.4 Fingertip frame 108
A.5 Contact frame 109
A.6 Grasp frame 110

B.1 Robot finger in contact with a flat surface 112
B.2 Two general bodies in contact 113
B.3 Pressure distribution in the area of the contact 114
B.4 Stress gradients within a sphere due to contact with a rigid plate 116
B.5 Initial finite element mesh 117
B.6 Line on which nodal stress are compared 118
B.7 Inconsistency in calculated nodal stress σ_{yy} 119
B.8 Inconsistency in calculated nodal stress σ_{zz} 119
B.9 Inconsistency in calculated nodal stress σ_{yz} 120
B.10 Location of consistent stress 120
B.11 Stress profile on hemispherical body 121
B.12 Region of mesh refinement 122
B.13 Refined mesh 123
B.14 Expanded view of the new refined region 123
B.15 Line on which nodal stresses are compared 124
B.16 Consistency in calculated nodal stresses: σ_{yy} 124
B.17 Consistency in calculated nodal stress: σ_{zz} 125
B.18 Larger values of stress at small radial distance from the contact 125
B.19 Agreement between the finite element and analytic solutions: σ_{yy} 126
B.20 Agreement between the finite element and analytic solutions: σ_{zz} 126
B.21 Agreement between the finite element and analytic solutions: σ_{yz} 127
B.22 Stress profile in the fingertip: σ_{yy} 129
B.23 Stress profile in the fingertip: σ_{zz} 130
B.24 Stress profile in the fingertip: σ_{yz} 131

C.1 Schematic representation of the fingertip sensor 135
C.2 Various external loads 137
C.3 Fingertip dimensions 138

C.4 General cantilever beam 149
C.5 Force applied on the top of the hemisphere along the axis of the sensor 149
C.6 Leg of the loadcell under a symmetrically applied vertical load 149
C.7 Horizontal force applied to the sensor 150
C.8 Compressive members are assumed to move rigidly 150
C.9 Horizontal force applied to the bottom of the fingertip 151
C.10 Assumptions on loadcell 151
C.11 The legs of the cross under applied forces and moments 152
C.12 A leg parallel to the force 152
C.13 A leg perpendicular to the force 153
C.14 Horizontal force tangent to the surface of the fingertip 153
C.15 Assumptions on loadcell 154

Chapter 1

Introduction

1.1 Introduction

When we manipulate objects, our fingers are not always fixed to the surface. Many times we allow the objects we hold to slide or rotate at our fingertips, consciously controlling the motion of the object rather than the motion of our fingers. This *controlled slipping* technique of manipulation is not just one of the ways we can move objects, but rather a dominant form of dexterous human manipulation. For example, try putting a lid on a jar, but start with the lid top down on a table. Without thinking, we pick up the lid, spin it around between our two fingers, using the edge of the jar or another finger, and screw it on the top. Or consider the use of a pencil eraser. When we make a mistake we stop writing, flip the pencil over, push the pencil through our fingers, and erase. In both these examples and in many others, we allow objects to slide and rotate at our fingertips. Through this controlled slipping technique we can control the location and orientation of an object within our grasp.

In robotic manipulation, emphasis has been placed on producing stable grasps. The object is then moved by controlling the motion of the manipulator, assuming the object is rigidly fixed to the robot. If the object slips in the grasp, control is lost, and there are no easy forms of recovery. In addition, there are many operations which become difficult or impossible without some form of controlled slip manipulation. In this thesis, I will analyze controlled slip manipulation in a robot hand, and from this analysis predict slipping

motion of an object within a grasp. Using these predicted slipping motions, the robot hand will affect the necessary changes in the grasp to allow the object to slip in a controlled manner.

1.2 Overview

In order for a multifingered robot hand to perform dexterous operations on the environment, it must be able to acquire objects into a grasp, control the motion of the object relative to the environment and control the object within the grasp. Object acquisition is an active area of robotic research. Given a particular manipulator and an object, how can the robot grasp the object? There are numerous considerations, such as the location, size, weight, orientation, surface properties, and specific functions of the object, as well as, configuration, workspace, strength, and surface properties of the manipulator. An algorithm recently developed [Nguyuan], computes the grasp locations on an object and finger stiffnesses in the robot necessary to produce force closure grasps. The objects are modeled as polyhedrons with a certain mass and surface friction. Areas on the polyhedral surfaces are then found on which the fingertips of a multifingered hand may be placed to yield force closure grasps. The number of fingers, the surface properties of the fingertips, the stiffness of the joints, and the workspace of the hand are all taken into account when computing grasp locations. Some research in object acquisition has also been done by [Lozano-Perez]. Not only were the current constraints on the object considered, but also constraints imposed on the object by the environment, as the object is moved through its planned trajectory. The constraints both present and future are mapped onto the object surface before the object is grasped. In this way, for example, a robot gripper would not grab the end of the peg which later must be inserted into a hole. There are, however, still many areas of object acquisition not yet explored. Objects with special surfaces, such as handles or loops, non-rigid objects, like paper or foam, objects that must be moved before grasping, such as a coin on a flat table, are all common examples acquisition which are not yet possible for a robot.

The control of object motion through the coordinated control of individual manipulators is also a current area of robot research. An algorithm has recently been written [Chiu] which coordinates individual fingers to yield a

specific translations and rotations of a grasp relative to an arbitrary reference frame. The object is assumed to remain fixed relative to the grasp; therefore, specific motions of the grasp yield specific motions of the object. From practical experience, this method has worked quite well. In order to actually control the motion of the object, however, it would be necessary to sense the position and orientation of the object, since the object may have been removed, slipped, or fallen from the grasp, unknown to the robot. Real time hand-eye coordination, however, is beyond the reach of current systems because of the computational complexity. High level coordinated control of multifingered robot hands, is a relatively new area of research. Recent multifingered robot hand research, therefore, has been primarily concerned with mechanical design, actuation, transmission, and sensing.

Control of an object relative to a grasp is also a new area of research. This problem has be approached by [Tournassoud], in terms of regrasping. The object is initially grasped, then set down, released, and then regrasped. In this way the object can be reoriented relative to the grasped. The object, however, is not manipulated within the grasp. Controlled slip manipulation, however, could enhance the dexterity of a manipulator, by allowing the robot greater freedom to move the object within the grasp.

1.3 Outline

This thesis is basically an analysis of the small finite permissible motions an object may undergo in a particular grasp and how these motions may be achieved. The first chapter is the introduction. In chapter 2, the types of contacts between a fingertip and an object are enumerated. For each set of contacts and contact types, we define a *constraint state* and for each constraint state, a map of the possible object motions is produced. Determining the constraint on small finite motion, requires knowing not only is the location and orientation of contacts, but also the surface geometry in the neighborhood of the contact. Chapter 3 deals with strategies available to manipulate an object relative to a grasp. These include the use of gravity, body forces, controlled accelerations, free fingers, and other objects to move objects within the grasp. For a multifingered hand there are many ways to squeeze a grasped object. For two or three fingered grasps, there is corresponding one space or three space of possible grasp solutions. Chapter 4 presents a sim-

ple intuitive parameterization of this grasp force space. For a three fingered hand, the three space may be represented simply by a *grasp force focus* and a *grasp force magnitude*. In order to determine the type of contact that exists at each fingertip, the forces and moments transmitted through the interface must be found. In chapter 5, the set of forces and moments that exist at the contacts will be found in terms of the *contact frame*, that is, a coordinate frame set in the object and defined in terms of the contact point and outward pointing normal. Chapter 6 outlines a simple relationship between the forces and moments at the contact and the contact type. Two examples are given in chapter 7 illustrating the analyses of the previous chapters, along with an demonstration of how controlled slipping could be used to reorient an object within a grasp. Chapter 8 describes how these analyses were simulated and how some controlled slip manipulations were achieved on the Salisbury robot hand. In chapter 9, extensions of the present theories are outlined, which yield more practical and efficient techniques for controlled slip manipulation.

Chapter 2

Constraint

2.1 Introduction

The purpose of this chapter is to determine the different ways an object can move in a grasp. First, the screw system representation will be described. This representation allows a complete and homogeneous treatment of both forces and moments as well as translation and angular displacements. Second, the types of contacts which may occur between two objects will be enumerated. For each contact type, a particular set of forces and moments can be exerted through the interface. These forces and moments limit the possible motions of one object relative to another. Third, the set of infinitesimal motions possible for an object subject to a single constraint will be determined using the concept of virtual work. Fourth, the set of infinitesimal motions will be extended to include the set of small finite motions by considering the surface geometry in the neighborhood of the contact point. Fifth, a *constraint state* will be defined as an ordered list of contact types on the object. Finally, for each constraint state, the set of permissible motions for a grasped object will be found by intersecting the permissible motions of the individual contacts.

2.2 Screws, wrenches, and twists

Many of the analyses in this thesis will employ a screw system representation for forces and moments, and for infinitesimal displacements. Although

standard force and displacement vectors could be used, screw systems allow a homogeneous treatment of both forces and moments, and translational and angular displacements.

A set of forces and moments acting upon a body can be collectively called a wrench. A wrench may be described as a force along a unique line, the screw axis, and a moment about that line. Similarly, a twist can represent the infinitesimal motion of an object, an infinitesimal translation along a line and an infinitesimal rotation about that line.

2.2.1 Screws

The outline of screw systems presented here is more adequately described in [Hunt] and completely developed in [Ball]. Both the wrench and the twist are specific representations of a screw. A screw is defined by a line in three space, a screw axis, and an associated pitch about that line. A screw may also be described by a six element vector, $\mathbf{s} = [s_1, s_2, s_3, s_4, s_5, s_6]$, where $s_1, s_2, \ldots, s_6$ are the screw coordinates. The coordinates of the screw axis are

$$\begin{aligned} L &= S_1 \\ M &= S_2 \\ N &= S_3 \\ P &= S_4 - pS_1 \\ Q &= S_5 - pS_2 \\ R &= S_6 - pS_3. \end{aligned} \tag{2.1}$$

The coordinates $L, M, \ldots, R$ are known as the Plücker line coordinates of the axis, where L,M, and N are proportional to the direction cosines of the screw axis, and P,Q, and R are proportional to the moment of the line about the origin of the reference frame. The pitch of the screw is

$$p = \frac{S_1S_4 + S_2S_5 + S_3S_6}{S_1^2 + S_2^2 + S_3^2}, \tag{2.2}$$

and the magnitude of the screw is

$$m = \sqrt{S_1^2 + S_2^2 + S_3^2}, \tag{2.3}$$

unless the pitch happens to be infinite, in which case the magnitude of the screw is

$$m = \sqrt{S_4^2 + S_5^2 + S_6^2}. \tag{2.4}$$

2.2.2 Wrenches

The wrench is one interpretation of the screw and, therefore, may be defined in a similar way. The wrench may be identified, in terms of screw coordinates, by a six element vector $\mathbf{w} = [w_1, w_2, w_3, w_4, w_5, w_6]$, where w_1,w_2, and w_3 are the forces along the x,y, and z axes of an reference frame, and w_4,w_5, and w_6 are the moments about the axes of the reference frame. By replacing S_i with w_i in equation 2.1, the line coordinates of the wrench may be found,

$$\begin{aligned} L &= w_1 \\ M &= w_2 \\ N &= w_3 \\ P &= w_4 - pw_1 \\ Q &= w_5 - pw_2 \\ R &= w_6 - pw_3. \end{aligned} \tag{2.5}$$

The pitch of the wrench is given by equation 2.2,

$$p = \frac{w_1 w_4 + w_2 w_5 + w_3 w_6}{w_1^2 + w_2^2 + w_3^2}, \tag{2.6}$$

and is the ratio of the torque about the screw axis to force along it. The magnitude of the wrench from equations 2.3 or 2.4 is

$$m = \sqrt{w_1^2 + w_2^2 + w_3^2}, \tag{2.7}$$

or if the pitch is infinite, the magnitude is

$$m = \sqrt{w_4^2 + w_5^2 + w_6^2}. \tag{2.8}$$

2.2.3 Twists

As with the wrench, the twist can also be described in terms of a six element vector $\mathbf{t} = [t_1, t_2, t_3, t_4, t_5, t_6]$, where t_1,t_2, and t_3 are the rotations about the

x, y, and z axes of a reference frame and t_4,t_5, and t_6 are the displacements along the x,y, and z axes. Again, replacing S_i with t_i, the line coordinates of the twist are

$$\begin{aligned} L &= t_1 \\ M &= t_2 \\ N &= t_3 \\ P &= t_4 - pt_1 \\ Q &= t_5 - pt_2 \\ R &= t_6 - pt_3. \end{aligned} \tag{2.9}$$

The pitch of the twist is

$$p = \frac{t_1 t_4 + t_2 t_5 + t_3 t_6}{t_1^2 + t_2^2 + t_3^2}, \tag{2.10}$$

that is, the ratio of the rotation about the twist axis to the translational along the axis. The magnitude is defined by

$$m = \sqrt{t_1^2 + t_2^2 + t_3^2}, \tag{2.11}$$

or in the case where the pitch is infinite, the magnitude is

$$m = \sqrt{t_4^2 + t_5^2 + t_6^2}. \tag{2.12}$$

2.2.4 Transforming screw definitions

Although a wrench and a twist are independent of the reference frames used to define them, it is useful to be able to transform their representations from one frame to another. In general, a screw $\mathbf{s}'$ can be represented in screw coordinates, $\mathbf{s}' = [s'_1, s'_2, s'_3, s'_4, s'_5, s'_6]$, where the elements of $\mathbf{s}'$ are defined relative to a specific reference frame. Let the elements of the screw $\mathbf{s}'$ be defined relative to a specific coordinate frame $O'X'Y'Z'$. Suppose this reference frame is, in turn, defined with respect to another reference frame $OXYZ$, figure 2.1. That is, let $\mathbf{l} = [l_x, l_y, l_z]$, $\mathbf{m} = [m_x, m_y, m_z]$, and $\mathbf{n} = [n_x, n_y, n_z]$, be the unit direction vectors of the x', y', and z' axes and let $\mathbf{x} = [x, y, z]$ be the origin of the $O'X'Y'Z'$ frame all defined with respect to the $OXYZ$ frame. Suppose we now wish to define the screw $\mathbf{s}'$ in screw

coordinates relative to the $OXYZ$ frame. Let $\mathbf{s}$ be the representation of the screw in the $OXYZ$ frame, then

$$\mathbf{s} = \mathbf{T}\mathbf{s}', \tag{2.13}$$

where $\mathbf{T}$ is a linear transformation matrix given by

$$\mathbf{T} = \begin{bmatrix} \begin{bmatrix} l_x & m_x & n_x \\ l_y & m_y & n_y \\ l_z & m_z & n_z \end{bmatrix} & \mathbf{0} \\ \begin{bmatrix} l_z y - l_y z & m_z y - m_y z & n_z y - n_y z \\ l_x z - l_z x & m_x z - m_z x & n_x z - n_z x \\ l_y x - l_x y & m_y x - m_x y & n_y x - n_x y \end{bmatrix} & \begin{bmatrix} l_x & m_x & n_x \\ l_y & m_y & n_y \\ l_z & m_z & n_z \end{bmatrix} \end{bmatrix}. \tag{2.14}$$

Conversely, a screw $\mathbf{s}$ defined in the $OXYZ$ frame may also be represented in $O'X'Y'Z'$,

$$\mathbf{s}' = \mathbf{T}^{-1}\mathbf{s}, \tag{2.15}$$

where $\mathbf{T}^{-1}$ equals

$$\mathbf{T}^{-1} = \begin{bmatrix} \begin{bmatrix} l_x & l_y & l_z \\ m_x & m_y & m_z \\ n_x & n_y & n_z \end{bmatrix} & \mathbf{0} \\ \begin{bmatrix} l_z y - l_y z & l_x z - l_z x & l_y x - l_x y \\ m_z y - m_y z & m_x z - m_z x & m_y x - m_x y \\ n_z y - n_y z & n_x z - n_z x & n_y x - n_x y \end{bmatrix} & \begin{bmatrix} l_x & l_y & l_z \\ m_x & m_y & m_z \\ n_x & n_y & n_z \end{bmatrix} \end{bmatrix}. \tag{2.16}$$

Since the twist is a specific representation of a screw, it too can be defined in screw coordinates relative to different reference frames. Suppose a twist $\mathbf{t}'$ is defined relative to a frame $O'X'Y'Z'$. Again, suppose this frame is, in turn, defined with respect to another frame $OXYZ$. The representation $\mathbf{t}$ of the twist $\mathbf{t}'$ in the frame $OXYZ$ is given by

$$\mathbf{t} = \mathbf{T}\mathbf{t}', \tag{2.17}$$

where $\mathbf{T}$ is given in equation 2.14 and conversely,

$$\mathbf{t}' = \mathbf{T}^{-1}\mathbf{t}, \tag{2.18}$$

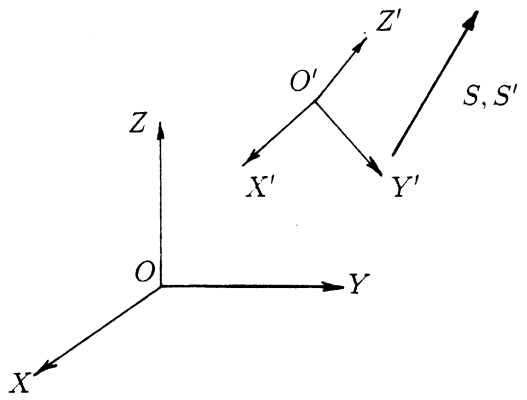


Figure 2.1: Although the screw is the same no matter which reference frame is used to define it, it is helpful to be able to transform the representation from one frame to another. Suppose a screw $\mathbf{s}'$ is defined in screw coordinates relative to a reference frame $O'X'Y'Z'$ and $O'X'Y'Z'$ is, in turn, defined relative to another frame $OXYZ$. Then the screw can also be represented in the $OXYZ$ frame by a simple linear transformation $\mathbf{s} = \mathbf{T}\mathbf{s}'$.

where $\mathbf{T}^{-1}$ is given in equation 2.16. Similarly for the wrench, given in screw coordinates relative to $O'X'Y'Z'$, its representation in $OXYZ$ is given by

$$\mathbf{w} = \mathbf{T}\mathbf{w}', \tag{2.19}$$

where $\mathbf{T}$ is given in equation 2.14. Conversely, the wrench defined in the $OXYZ$ frame may also be represented in the $O'X'Y'Z'$ by

$$\mathbf{w}' = \mathbf{T}^{-1}\mathbf{w}, \tag{2.20}$$

where $\mathbf{T}^{-1}$ is given in equation 2.16.

2.3 Contact types

The interface between a robot and an object can be characterized by a particular type of contact. For a three dimensional manipulator, there are, in

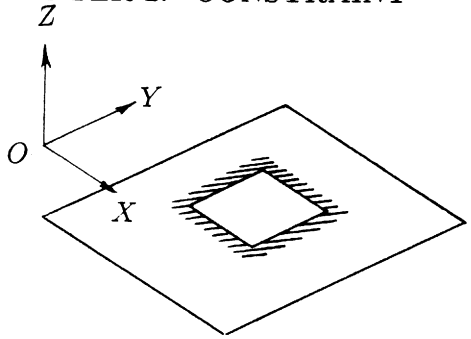
general, nine different types of contacts. These contact types are illustrated in figure 2.2, [Salisbury]. For each of these contact types, a particular set of wrenches may be exerted through the interface. The set of all possible wrenches that may be transmitted through a contact is known as a wrench system. A set of unit basis wrenches can be specified which span the entire wrench system. The nature of the contact problem, however, is inherently non-linear, since objects can both make and break contact with one another. Therefore, to describe the wrench system, it is necessary to define a set of unidirectional unit basis wrenches. Any wrench in the wrench system may then be represented by a positive linear combination of these unidirectional basis wrenches. Figure 2.2 lists, along with each contact type, the set of unidirectional unit basis wrenches whose positive linear combinations span the space of all the possible wrenches which can be transmitted through the contact.

For the analyses in this thesis, only contacts made between the fingertips of the manipulator and an object will be considered, although the techniques developed here encompass other types of contacts as well. By considering only fingertip contacts, however, the number of possible contact types is reduced to four: a soft finger contact, a point contact with friction, a point contact without friction, and no contact. For convenience, these contact types will be represented by the following numbers

1 = Soft finger contact
2 = Point contact with friction
3 = Point contact without friction
4 = No contact.

2.4 Infinitesimal motion

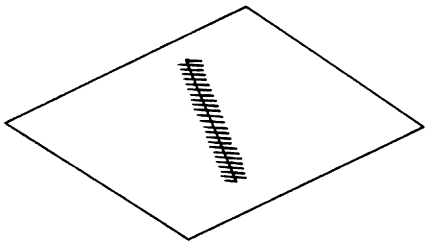
Virtual work is the work done by a wrench exerted against an arbitrary twist. By examining the sign of the virtual work, we can determine the set of infinitesimal motions which are possible for an object constrained by a contact. First a contact is characterized by a particular contact type. Then the contact is replaced by a set of unidirectional unit basis wrenches, listed in the figures 2.2 and 2.3. Then for each basis wrench, the virtual work



Unit Basis Wrenches

$w = [1,0,0,0,0,0]$ $w = [-1,0,0,0,0,0]$
$w = [0,1,0,0,0,0]$ $w = [0,-1,0,0,0,0]$
$w = [0,0,-1,0,0,0]$
$w = [0,0,0,1,0,0]$ $w = [0,0,0,-1,0,0]$
$w = [0,0,0,0,1,0]$ $w = [0,0,0,0,-1,0]$
$w = [0,0,0,0,0,1]$ $w = [0,0,0,0,0,-1]$

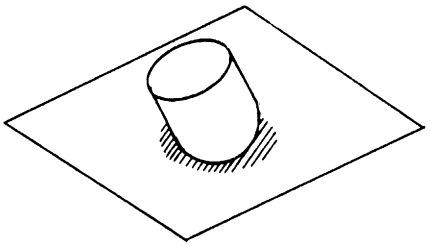
Plane contact with friction



$w = [1,0,0,0,0,0]$ $w = [-1,0,0,0,0,0]$
$w = [0,1,0,0,0,0]$ $w = [0,-1,0,0,0,0]$
$w = [0,0,-1,0,0,0]$
$w = [0,0,0,1,0,0]$ $w = [0,0,0,-1,0,0]$

$w = [0,0,0,0,0,1]$ $w = [0,0,0,0,0,-1]$

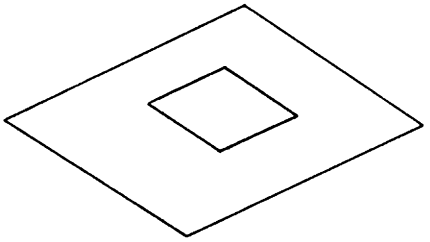
Line contact with friction



$w = [1,0,0,0,0,0]$ $w = [-1,0,0,0,0,0]$
$w = [0,1,0,0,0,0]$ $w = [0,-1,0,0,0,0]$
$w = [0,0,-1,0,0,0]$

$w = [0,0,0,0,0,1]$ $w = [0,0,0,0,0,-1]$

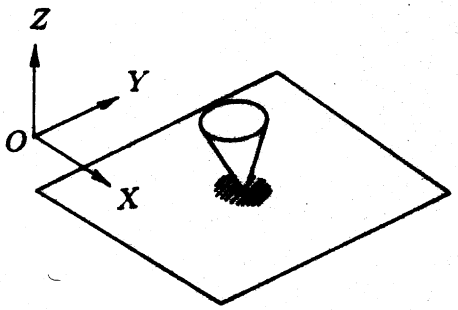
Soft finger contact



$w = [1,0,0,0,0,0]$ $w = [-1,0,0,0,0,0]$
$w = [0,0,1,0,0,0]$ $w = [0,-1,0,0,0,0]$
$w = [0,0,0,1,0,0]$ $w = [0,0,-1,0,0,0]$
$w = [0,0,0,0,1,0]$ $w = [0,0,0,-1,0,0]$
$w = [0,0,0,0,0,1]$ $w = [0,0,0,0,-1,0]$

Plane contact without friction

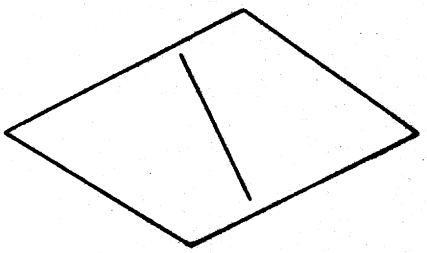
Figure 2.2: In general, there are nine different types of three dimensional contacts. Each contact type allows only certain wrenches to be transmitted through the interface. Since the contact problem is inherently non-linear, a set of unidirectional basis wrenches is defined whose positive linear combination describe the space of wrenches which can be transmitted through the interface.



Unit Basis Wrenches

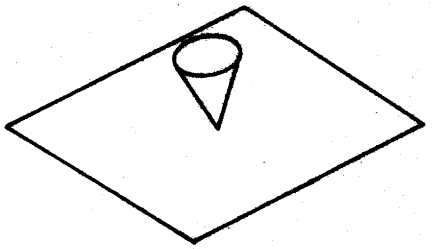
$w = [1,0,0,0,0,0]$ $w = [-1,0,0,0,0,0]$
$w = [0,1,0,0,0,0]$ $w = [0,-1,0,0,0,0]$
$w = [0,0,-1,0,0,0]$

Point contact with friction



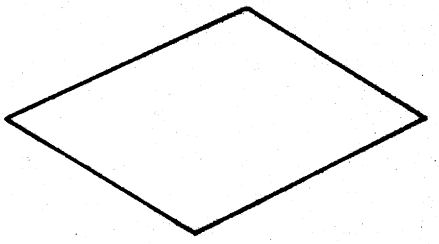
$w = [0,0,-1,0,0,0]$
$w = [0,0,0,1,0,0]$ $w = [0,0,0,-1,0,0]$

Line contact without friction



$w = [0,0,-1,0,0,0]$

Point contact without friction



No contact

Figure 2.3: Contact types (Continued)

is calculated. If the virtual work done by a twist against every unit basis wrench in the set is greater than or equal to zero, then the twist is allowed. Conversely, twists which produce negative virtual work for even a single basis wrench are disallowed.

A complete derivation of virtual work is presented in [Ball] and the derivations in this section are outlined in [Ohwovoriole]. A robot manipulator A exerts a wrench of magnitude α and pitch p_α along a screw $\mathbf{A}$ on a body B, as shown in figure 2.4. The body then undergoes a twist of amplitude β and pitch p_β along a screw $\mathbf{B}$. Define a coordinate system $OXYZ$ with the x-axis aligned with the twist axis, the z-axis the common perpendicular between the twist and wrench axes, and the y-axis perpendicular to both the x and z axes, as shown in figure 2.5. The wrench can then be decomposed into forces and moments on the cartesian coordinate system

$$\begin{aligned} F_x &= \alpha \cos\theta \\ F_y &= \alpha \sin\theta \\ F_z &= 0 \\ M_x &= \alpha(p_\alpha \cos\theta - d \sin\theta) \\ M_y &= \alpha(p_\alpha \sin\theta + d \cos\theta) \\ M_z &= 0, \end{aligned} \tag{2.21}$$

where d is the length of the mutual perpendicular between the wrench and the twist axes. The virtual work done by the wrench against the twist is

$$W = \alpha\beta \cos[(p_\alpha + p_\beta)\cos\theta - d \sin\theta]. \tag{2.22}$$

The virtual coefficient between screws $\mathbf{A}$ and $\mathbf{B}$ is defined to be

$$\omega = \cos[(p_\alpha + p_\beta)\cos\theta - d \sin\theta]. \tag{2.23}$$

The virtual coefficient is independent of both the amplitude of the twist and the intensity of the wrench. Let $(t_1, t_2, t_3, t_4, t_5, t_6)$ and $(w_1, w_2, w_3, w_4, w_5, w_6)$ be the screw coordinates of a unit twist and a unit wrench. The virtual coefficient is then

$$\omega = w_1 t_4 + w_2 t_5 + w_3 t_6 + w_4 t_1 + w_5 t_2 + w_6 t_1. \tag{2.24}$$

We will use the unit twist and unit twist from now on when evaluating the virtual coefficient. The reason for this will be apparent later when we consider the value of the virtual coefficient.

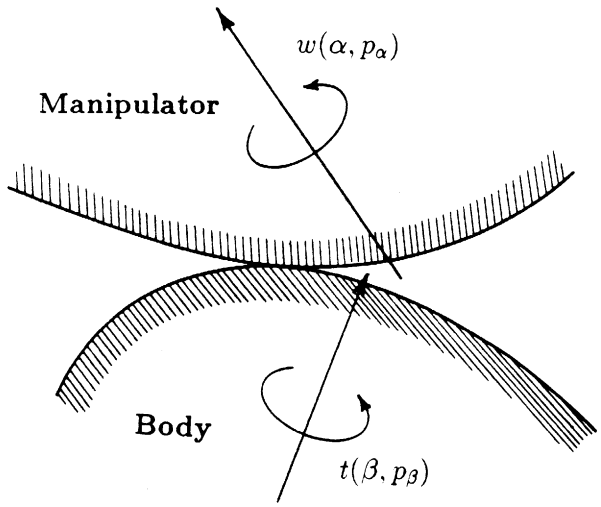


Figure 2.4: The contact forces between a manipulator A and a body B can be modeled as a wrench of w along a screw A of magnitude α and pitch p_α. The motion of the body can be described by a twist t along a screw B of amplitude β and pitch p_β.

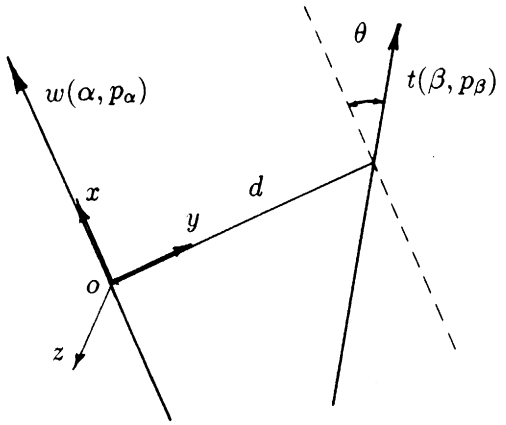


Figure 2.5: The wrench can be decomposed onto a coordinate system $OXYZ$. The z-axis is common perpendicular between the twist and the wrench axes, the x-axis is the twist axis, and the z-axis is perpendicular to both the x and the z axes.

2.4.1 Reciprocal screws

Two screws are said to be reciprocal screws if the virtual work between a twist and a wrench associated with the two screws is zero. No work will be done by a unit twist **t** against a unit wrench **w** if the virtual coefficient is equal to zero. That is,

$$\omega = w_1 t_4 + w_2 t_5 + w_3 t_6 + w_4 t_1 + w_5 t_2 + w_6 t_1 = 0. \tag{2.25}$$

2.4.2 Repelling screws

Two screws are said to be repelling if the virtual work between the screws of a twist and a wrench is positive. Therefore two unit screws of a unit twist and a unit wrench will repelling if the virtual coefficient if positive. That is,

$$\omega = w_1 t_4 + w_2 t_5 + w_3 t_6 + w_4 t_1 + w_5 t_2 + w_6 t_1 > 0. \tag{2.26}$$

Work is defined to be positive if the body moves in the same direction as the wrench.

2.4.3 Contrary screws

Two screws are contrary if the virtual work between the screws of a twist and a wrench is negative. Again two unit screws are contrary if the virtual coefficient if negative,

$$\omega = w_1 t_4 + w_2 t_5 + w_3 t_6 + w_4 t_1 + w_5 t_2 + w_6 t_1 < 0. \tag{2.27}$$

2.4.4 Permissible twist

A permissible twist is an infinitesimal motion that the constrained object is allowed to undergo. Assume an object is constrained by a single contact, as shown in figure 2.6. Also assume that the contact can be characterized by a particular contact type. The constraint imposed by the contact can then be represented by its associated set of unidirectional unit bases wrenches as was described earlier.

For convenience, set of bases wrenches is defined in terms of a *contact frame*. The contact frame is a coordinate frame whose origin is the point of

contact between the object and the manipulator. The z-axis is the surface normal at the contact point, and the x and y axes lie in the tangent plane. In general, the choice of the x and y axes of the contact frame is arbitrary; however, a convention for resolving the ambiguity is given in appendix A. This convention, though useful for the implementation, is unimportant in the current discussion. Figure 2.6 then shows the contact frame $O_{c_i}X_{c_i}Y_{c_i}Z_{c_i}$ defined relative to a reference frame $OXYZ$.

Let $\{\mathbf{w_{c_j}}\}$ be the set of unidirectional unit bases wrenches which replace the contact type defined in terms of the contact frame. Their representation in the reference frame $OXYZ$ is given by

$$\mathbf{w_j} = \mathbf{T_i w_{c_j}}, \tag{2.28}$$

where $\mathbf{T_i}$ is the transformation matrix given in equation 2.14 by

$$\mathbf{T_i} = \begin{bmatrix} \begin{bmatrix} l_x & m_x & n_x \\ l_y & m_y & n_y \\ l_z & m_z & n_z \end{bmatrix} & \mathbf{0} \\ \begin{bmatrix} l_z y - l_y z & m_z y - m_y z & n_z y - n_y z \\ l_x z - l_z x & m_x y - m_z x & n_x z - n_z x \\ l_y x - l_x y & m_y y - m_x y & n_y x - n_x y \end{bmatrix} & \begin{bmatrix} l_x & m_x & n_x \\ l_y & m_y & n_y \\ l_z & m_z & n_z \end{bmatrix} \end{bmatrix}_i, \tag{2.29}$$

where $\mathbf{l} = [l_x, l_y, l_z]$, $\mathbf{m} = [m_x, m_y, m_z]$, and $\mathbf{n} = [n_x, n_y, n_z]$ are the direction vectors of the x,y, and z axes and x, y, and z is the origin of the ith contact frame defined with respect to the reference frame $OXYZ$.

The set of twists which are reciprocal or repelling to a single unit basis wrench $\mathbf{w_j}$ is

$$T_j = \{\mathbf{t} : w_{1_j}t_4 + w_{2_j}t_5 + w_{3_j}t_6 + w_{4_j}t_1 + w_{5_j}t_2 + w_{6_j}t_3 \geq 0\}. \tag{2.30}$$

The set of twist reciprocal or repelling to all the unit basis wrenches in $\mathbf{w_j}$ is the intersection of all the sets T_j

$$T_{inf} = \bigcap\{T_1, T_2, \ldots\}. \tag{2.31}$$

The set T_{inf} represents the set of twists which are either reciprocal or repelling to every unidirectional unit basis wrench of a single contact. The set T_{inf} describes only infinitesimal motions which the object is allowed to undergo while constrained by a single contact. In order to determine the possible motions of an object finite motions must be considered.

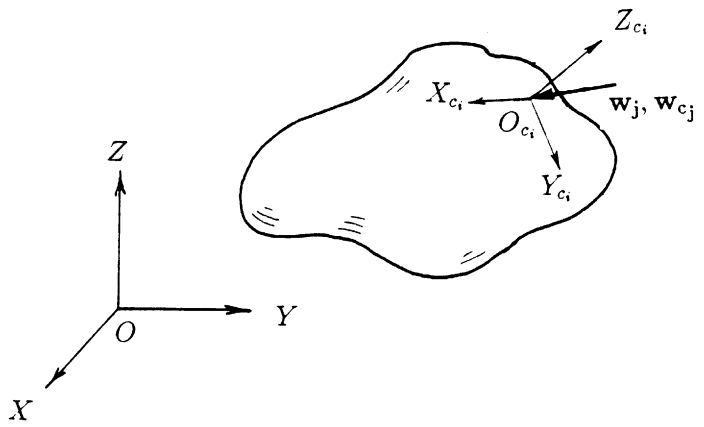


Figure 2.6: The motion of an object is constrained by a single contact. The contact can be characterized by a contact type which can then be represented by a set of unidirectional unit basis wrenches $\{\mathbf{w_j}\}$. The set of all twists which are reciprocal or repelling to every basis wrench, describes the infinitesimal motions the object is allowed to undergo.

2.5 Finite motions

The set of twists T_{inf} given in equation 2.31 describe the infinitesimal motion of an object constrained by a single contact. In order to be useful in determining all the possible ways an actual object can move within a grasp, the definition of permissible motion must be extended to include finite motion. Given an arbitrary unit twist $\mathbf{t}$ defined in screw coordinates $t = [t_1, t_2, t_3, t_4, t_5, t_6]$ relative to some reference frame, let

$$\mathbf{t} = m \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{bmatrix} \tag{2.32}$$

represent a finite motion, where mt_1,mt_2, and mt_3, are finite rotations and mt_4,mt_5, and mt_6, are finite translation along the x, y, and z axes of the reference frame. In general, the twists in the set T_{inf} describe finite as well as

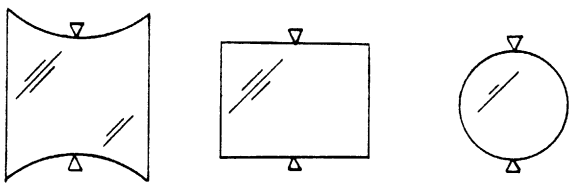


Figure 2.7: Surface geometry in the neighborhood of the contact must be considered to determine the complete set of small finite permissible motions.

infinitesimal motion, except for one important exception. Twists reciprocal to the wrench applied by a point contact without friction may or may not be permissible. Whether motion is permitted depends on the surface geometry in the neighborhood of the contact. Consider the following example of two dimensional objects constrained by two point contacts without friction, as shown in figure 2.7. The location and the orientation of the contacts are identical, thus the set of infinitesimal permissible twists for each object is the same. The finite constraint imposed on the objects, however, is different. The first object, with concave surfaces, is completely constrained in all directions. The second object, the rectangle, can only move horizontally, and the last object, the circle, can both translate or rotate between the contacts. The difference in constraint comes not from the contact types, but from the curvature of the surfaces. In order to determine the complete set of permissible finite motions, the surface geometry in the neighborhood of the contact must be considered.

Therefore, we want to determine the motions which do not violate the geometric constraints imposed by the surface of the object. To solve this problem, assume an object undergoes a hypothetical finite motion describe

by a twist $m\mathbf{t}$, in equation 2.32, then determine whether this particular motion causes the contact point to penetrate the surface of the object, thus violating the geometric constraints. Let the surface of the object be describe by a function $f_{surface}(x, y)$ relative to the contact frame $O_{c_i}, X_{c_i}, Y_{c_i}, Z_{c_i}$. First, determine the motion of the contact point relative to this contact frame as the object undergoes the finite motion describe by $m\mathbf{t}$. Second, map the trajectory of the contact point onto the surface of the object. Finally, compare the trajectory of the contact point with its projection onto the object surface to determine whether this particular motion violates the geometric constraints.

When comparing the motion of the surface of the object relative to the contact points, it is unimportant whether we define the motion as a twist $m\mathbf{t}$ that the object undergoes relative to fixed set of contact points or as a twist $-m\mathbf{t}$ that the contact points undergo relative to a fixed object. For convenience, we will assume the object remains fixed relative to the $OXYZ$ frame and the contact points move with a twist $-m\mathbf{t}$. The trajectory of the contact point is given by

$$\mathbf{x_t} = \mathbf{T_t}\mathbf{x_i} + \mathbf{d}, \tag{2.33}$$

where $\mathbf{T_t}$ is a matrix

$$\mathbf{T_t} = \begin{bmatrix} t_1t_1c_{1m} & t_1t_2c_{1m} - t_3s_m & t_1t_3c_{1m} + t_2s_m \\ t_2t_1c_{1m} + t_3s_m & t_2t_2c_{1m} & t_2t_3c_{1m} - t_1s_m \\ t_3t_1c_{1m} - t_2s_m & t_3t_2c_{1m} + t_1s_m & t_3t_3c_{1m} \end{bmatrix}, \tag{2.34}$$

where $c_{1m} = (1 - \cos(m))$, $c_m = \cos(m)$, and $s_m = \sin(m)$. $\mathbf{T_t}$ multiplied by $\mathbf{x_i}$ yields the displacement due to a rotation, while

$$\mathbf{d} = \begin{bmatrix} mt_4 \\ mt_5 \\ mt_6 \end{bmatrix} \tag{2.35}$$

is a displacement due to translation. Equation 2.33 describes the displacement of the contact point relative to the $OXYZ$ frame, but we want to know the displacement of the contact point relative to the contact frame $O_{c_i}X_{c_i}Y_{c_i}Z_{c_i}$. Therefore, let $\mathbf{x_{traj}}$ be the trajectory the contact point relative to the contact frame. Then $\mathbf{x_{traj}}$ is given by

$$\mathbf{x_{traj}} = \mathbf{T}^{\mathbf{T}}_{\mathbf{r_i}}\mathbf{x_t}, \tag{2.36}$$

where $\mathbf{T_{r_i}}$ is a rotation matrix relating points in the contact frame $O_{c_i}, X_{c_i}, Y_{c_i}, Z_{c_i}$ to points in the reference frame $OXYZ$,

$$\mathbf{T_{r_i}} = \begin{bmatrix} l_x & m_x & n_x \\ l_y & m_y & n_y \\ l_z & m_z & n_z \end{bmatrix}. \tag{2.37}$$

Equation 2.36 describes a locus of points relative to the contact frame. These points represent the trajectory of the contact point would undergo if the object were displaced by a twist $m\mathbf{t}$. For a given twist, the trajectory of the contact point is a function of only one variable, the magnitude m

$$\mathbf{x_{traj}} = \begin{bmatrix} x_{traj} \\ y_{traj} \\ z_{traj} \end{bmatrix} = \begin{bmatrix} f_x(m) \\ f_y(m) \\ f_z(m) \end{bmatrix}. \tag{2.38}$$

By projecting the trajectory of contact onto the surface of the object, it is possible to determine whether a particular twist $\mathbf{t}$ describes a permissible finite motion. Figure 2.8 shows the motion of the contact point and its projection onto the surface. The surface of the object is described by the equation

$$z_{surface} = f_{surface}(x, y). \tag{2.39}$$

and the projection of the contact trajectory onto the surface of the object, is given by

$$\mathbf{x_{proj}} = \begin{bmatrix} x_{proj} \\ y_{proj} \\ z_{proj} \end{bmatrix} = \begin{bmatrix} f_x(m) \\ f_y(m) \\ f_{surface}[f_x(m), f_y(m)]. \end{bmatrix} \tag{2.40}$$

For a motion to be permissible, the contact trajectory must not penetrate the surface. Simply

$$z_{traj} \geq z_{proj}. \tag{2.41}$$

Therefore, the set of finite motions the object may undergo is

$$T_{fin} = \{\mathbf{t} : z_{traj}(\mathbf{t}) \geq z_{proj}(\mathbf{t})\}. \tag{2.42}$$

Finally, the set of all finite motions of an object subject to a single constraint is the intersection of the finite and infinitesimal permissible motions

$$T = T_{fin} \bigcap T_{inf}. \tag{2.43}$$

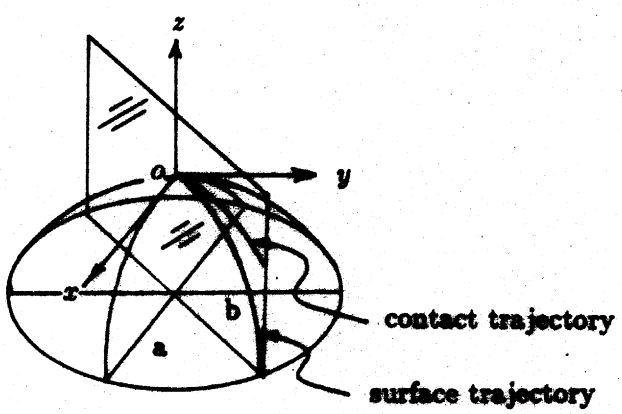


Figure 2.8: The projection of the *contact trajectory* onto the object surface reveals whether a particular motion is possible. A motion which causes the contact to penetrate the surface is not allowed, while motions which cause the object to move away from the surface are permitted.

2.5.1 Value of virtual coefficient

The definitions in the previous section provide mathematical constraints which work well for abstract models. An infinitesimal variation in the direction of the unit twist, however, will change its definition from reciprocal to repelling or contrary. In reality, small changes in constraint should not lead to drastic changes on the object. This problem occurs because a twist is defined to be reciprocal only when the virtual coefficient is exactly zero. In order to compensate for inaccuracies in contact measurement and to allow more generality in object motion, it would be helpful to define a range of values , $-\omega_r$ to ω_r, for which a twist is defined to be reciprocal. That is, a reciprocal unit twist satisfies

$$-\omega_r \leq w_1t_4 + w_2t_5 + w_3t_6 + w_4t_1 + w_5t_2 + w_6t_1 \leq \omega_r. \tag{2.44}$$

This, however, also changes the definition of the repelling and contrary twist. A repelling unit twist will now be defined as

$$w_1t_4 + w_2t_5 + w_3t_6 + w_4t_1 + w_5t_2 + w_6t_1 \geq \omega_r, \tag{2.45}$$

and a contrary unit twist

$$w_1t_4 + w_2t_5 + w_3t_6 + w_4t_1 + w_5t_2 + w_6t_1 \leq -\omega_r. \tag{2.46}$$

For example, consider the case of two fingers grasping a rectangle in two dimensions, as shown in figure 2.9. If tactile sensors are used to resolve the contact point and surface normal, there is to be expected some small error in the reading, figure 2.9. It is obvious that the block can slide between the fingertips in either direction if the grasp is loosened sufficiently. The strict definition of the reciprocal twist, however, indicates only one or the other direction is possible, depending upon error in the sensed surface normal. By expanding the definition of the reciprocal twist, in equation 2.5, we can again obtain the correct result, that the rectangle can move in either horizontal direction. Now that the definition of the reciprocal twist, equation 2.23, has been expanded, the geometric constraints will become more important when determining possible motion.

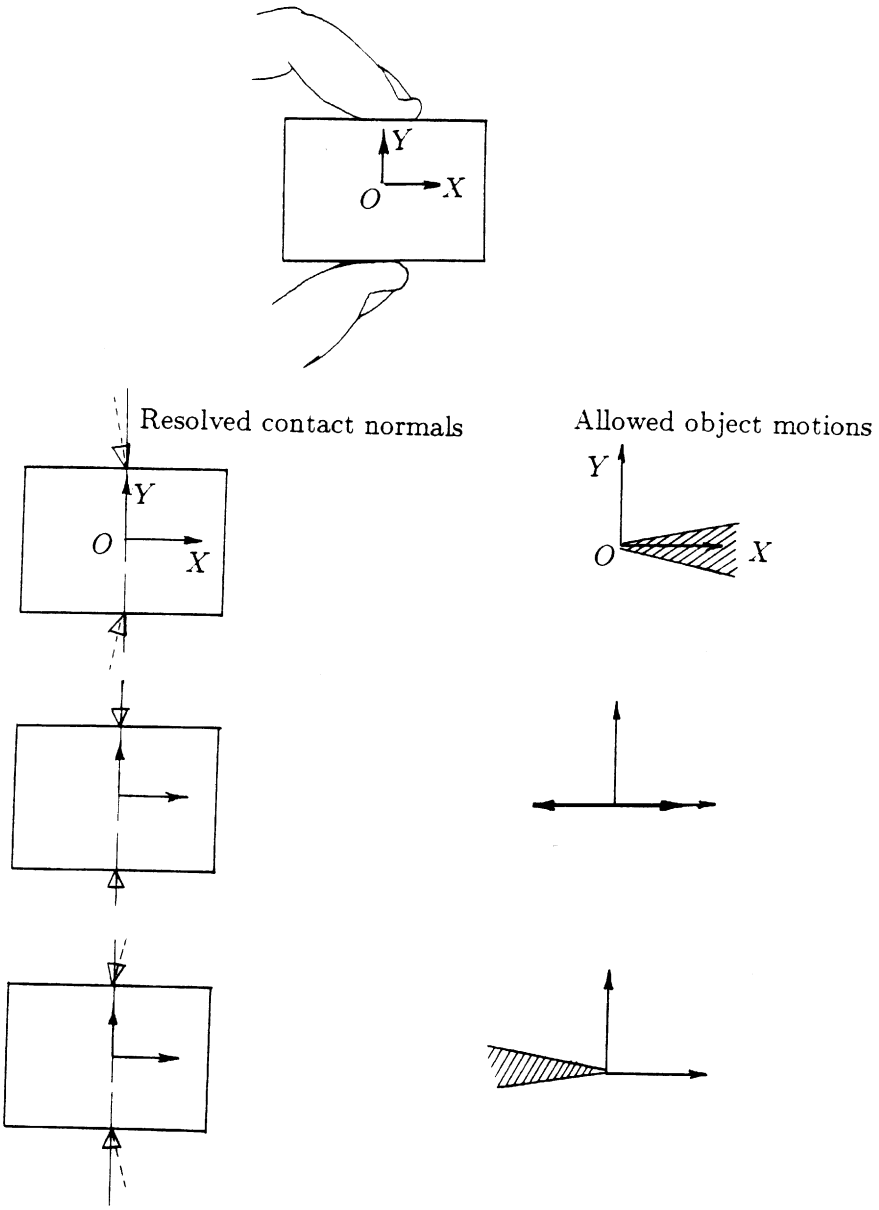


Figure 2.9: Small errors in the definition of the surface normal of the contact on a two dimensional rectangle lead to large variation in the predicted physical system. By expanding the definition of the reciprocal twist, these errors can be accounted for as well as yield more intuitive results in determining the allowed motions of an object within a grasp.

2.6 Multiple contacts

If there are m contacts between the manipulator and the object and n different types of contacts, there will be n^m possible combinations of contacts. Each combination of contacts yield a different set of constraints on the object. An object may be said to be in a particular *constraint state* defined by the number and the types of contacts. For m contacts and n contact types there are n^m possible constraint states. For example, if a three fingered hand grasped an object at the fingertips there are three contacts and, since there are only fingertip contacts, there are four contact types. With three contacts and only four possible contact types there are 4^3 possible combinations, that is 64 constraint states. For grasps made with a human hand at the fingertips, there are now five contacts, and therefore 4^5 or 2048 possible constraint states. A constraint state can be denoted by an ordered list of m elements, each element denoting the contact type. Suppose the three fingered hand has its fingers numbered one, two, and three, and assume the hand grasps an object at the fingertips. If the first and second fingers behave as soft finger contacts, while the third finger slips on the surface, the constraint state will be $[1, 1, 3]$. Similarly, if a human hand grasped a cup, for example, and the thumb, index finger, middle finger, make soft finger contacts on the object, the ring finger slips on the surface, and the small finger is removed. the constraint state would be represented by $[1, 1, 1, 3, 4]$. More generally, the constraint state is denoted by

$$C = [c_1, c_2, \ldots, c_m], \tag{2.47}$$

where c_i is the contact type for the ith contact.

2.7 Permissible Motion

From the previous analysis, the set of permissible motions for an object constrained by a single contact was given by T, equation 2.43. Now assume the object is constrained by multiple contacts and let the set of permitted motions for each contact be given by $T_{[c_i]}$, where c_i represents the contact type for the ith contact.

If there are m constraints on the object, the set of permissible twists for

a grasped object is the intersection of all the $T_{[c_i]}$,

$$T_{[c_1 c_2 \ldots c_m]} = \bigcap \{T_{[c_1]}, T_{[c_2]}, \ldots, T_{[c_m]}\}. \tag{2.48}$$

The goal of this chapter was to describe all the possible ways an object may move within a particular grasp. Equation 2.48 describes, in terms of twists, the set of all permissible finite motions that an object may undergo in a given grasp.

Chapter 3

Strategies for manipulation

3.1 Introduction

In this chapter, we will examine different external forces that can act on an object and how they can be used to move the object around in a grasp. An object may be influenced by a number of other forces besides the grasping fingers of a robot. Gravity, acceleration, electro-magnetic forces, other objects, and free fingers can all exert forces on the object. Although these forces differ, they can all be collected into a single *external wrench* acting on the object. The object will then tend to undergo a twist repelling to this external wrench. The set of twists repelling to the external wrench can be called the *preferred twists*, since they describe the possible motions the object can undergo. The actual motion, however, will depend on the constraints on the object. That is, in order for object to actually move, the set of preferred twists must intersect the set of permissible twists, discussed in chapter 2.

We will begin by examining a general strategy for using the external wrench to move an object in a grasp. Then we will examine different external wrenches including gravity, controlled accelerations, and forces produced by the robot and the environment. For each of these external wrenches, different controlled slip manipulation strategies will be investigated.

3.2 General strategy

Assume a grasped object is subject to an external wrench $\mathbf{w_e}$. The object will then tend to undergo a twist repelling to this external wrench. The set of repelling twist, or preferred twists, is given by

$$T_p = \{\mathbf{t} | w_{e_1} t_4 + w_{e_2} t_5 + w_{e_3} t_6 + w_{e_4} t_1 + w_{e_5} t_2 + w_{e_6} t_3 > 0\}. \quad (3.1)$$

In order for the object to actually move, however, the set of preferred twist must intersect the set of permissible twists. That is,

$$T = T_p \bigcap T_{[c_1,\ldots,c_m]}, \quad (3.2)$$

where $T_{[c_1,\ldots,c_m]}$ is the set of all possible motions the object may undergo while in the $[c_1, \ldots, c_m]$ constraint state.

The set of twists described in equation 3.2 is of particular interest, since this set will describe the way the object will actually move. If $T = \emptyset$, then the object is completely constrained and will not move at all. If T contains a single twist $\mathbf{t}$, then the object will move in a direction described by this twist. This case is especially important in controlled slip manipulation, since the geometry constraints and the external wrench together specify a unique motion of an object within the grasp. However, if T contains more than a single element, the resulting motion cannot be determined from the geometric analysis alone. Dynamics, external forces, internal grasping forces, and local surface friction properties must all be taken into account to determine the exact object motion. Although this type motion is difficult to control even for humans, it should not be ignored in robotic manipulation, since it may be useful in predicting motion resulting from inadvertent slipping or more complex dexterous motions. In fact, geometrically unconstrained motion was analyzed by [Mason], in which he predicted the motion of an object sliding in a plane while subject to a specific velocity at a single point.

The objective of controlled slip manipulation is to control the motion of an object relative to the grasp. There are some motions which are easy to accomplish through controlled slip manipulation. These are motions which are both allowed by the constraints on the object and bound by the geometry contacts. In other words, if the intersection of the permissible and preferred twists, equation 3.2, is a small bounded set of the twists, then these motions which can be easily implemented by a robot. Therefore, when planning

controlled slip motions, consideration should be given to these constrained twists as well as the twists which are particularly desirable. In any case, a *desired twist* is selected $\mathbf{t_d}$, describing the motion the object should undergo in the grasp of the robot. In order to facilitate this desired motion, the external wrench on the object should be optimized so that it will move the object through the desired twist. The optimal external wrench is one in the virtual work of the wrench against the desired twist is a maximum. These external wrenches are

$$\{\mathbf{w_e}|max\{\omega(\mathbf{w_e}, \mathbf{t_d})\}\}. \tag{3.3}$$

In the following sections, we will examine different ways to exert an external wrench on a grasped object and how these external wrenches may be used to control the slipping motion of an object within a grasp.

3.3 Specific strategies

3.3.1 Gravity

People use gravity to their advantage when manipulating objects. We reposition objects in our hands by allowing then to drop, slide, or rotate between our fingers. Consider, for example, when a glass of water is raised from the table. We can allow the glass to rotate between our fingertips so that the glass remains vertical. Relative to our hand, gravity has been used to rotate the glass and in this way maintain the vertical orientation. Gravity can also be used by a robot manipulator to reposition objects within a grasp. The force imposed by gravity can be modeled as a single force at the centroid of the object. Hand orientation can then be used to allow gravity to move the object through a desired twist.

Suppose we wish to use gravity to move an object within a grasp. First, we specify a *desired twist*, $\mathbf{t_d}$, that is, a twist, relative to the grasp, through which we want the object to move. Then, by reorienting the object, the external wrench $\mathbf{w_e}$ relative to the grasp, can be varied. The virtual work created by the external wrench against the desired twist is a function of the object orientation. It is then possible to determine a set of object orientations for which the virtual work is a maximum and at these orientations gravity will be optimally used to move the object.

To illustrate how gravity might be used, suppose a desired twist $\mathbf{t_d}$ is specified in screw coordinates relative to an object frame $O_oX_oY_oZ_o$. The object frame is a coordinate system whose origin is the centroid of the object and whose axes are fixed within the object. Now let the object frame be defined in terms of the hand frame frame $O_hX_hY_hZ_h$, where the z axis of the hand frame is parallel to gravity, figure 3.1 and the x and y axes are defined for a specific hand. The hand coordinate system for the Salisbury robot hand is described in appendix A. In this example, suppose the axes of the object frame are initially aligned with the axes of the hand frame. The wrench caused by gravity acting on the object defined relative to the object frame is

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -mg \\ 0 \\ 0 \\ 0 \end{bmatrix}, \tag{3.4}$$

where g is the gravitational acceleration and the m is the mass of the object. However, if the object frame is rotated, the value of the external wrench defined with respect to the object frame changes. The Salisbury robot hand has the ability to rotate the object through arbitrary angles about the axes of the hand frame. Therefore, assume the object frame is rotated by an angle θ about the x axis and then by an angle ϕ about the y axis of the hand frame. Now the external wrench in terms of the object frame will be

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \end{bmatrix} = \begin{bmatrix} -mg\cos\theta\sin\phi \\ -mg\sin\theta \\ -mg\cos\theta\cos\phi \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{3.5}$$

Only two rotations are used in equation 3.5, since rotations about the z-axis of the reference frame produce no change in the wrench.

For certain values of θ and ϕ the value of the virtual work is a maximum. At these orientations, the axes of the gravitational wrench and the desired twist are the same. Given the virtual work

$$w = -mg(\cos\theta\sin\phi t_{c_4} + \sin\theta t_{c_5} + \cos\theta\cos\phi t_{c_6}), \tag{3.6}$$

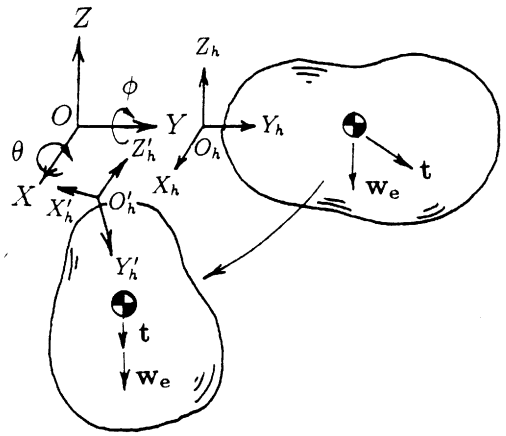


Figure 3.1: The wrist can be used to orient the object in a grasp so that gravity can be used to move the object

the values of θ and ϕ which produce a critical values of virtual work are

$$\begin{aligned} \theta &= \sin^{-1} \frac{t_{c_5}}{\sqrt{t_{c_4}^2 + t_{c_5}^2 + t_{c_6}^2}} \\ \phi &= \cos^{-1} \frac{t_{c_6}}{\cos(\theta)\sqrt{t_{c_4}^2 + t_{c_5}^2 + t_{c_6}^2}}. \end{aligned} \tag{3.7}$$

Substituting the values for θ and ϕ into equation 3.6 yield either positive or negative values for the virtual work. Values of θ and ϕ for which the virtual work is positive, produce the maximum value of virtual work.

Therefore, by using the wrist to rotate the hand about the angles given in equation 3.7, the robot can use the gravitational force to maximum efficiency when moving an object within the grasp. The use of gravity can be an effective means for controlling the motion of an object within the grasp. However, it may also be somewhat difficult, since the external force cannot be controlled; all the control must come from the wrist and the fingertips. The manipulator must also respond quickly to the motion of the object, since once it begins to slide it will continue to slide. Other methods, such as, controlled accelerations and point forces from both non-grasping fingers and external objects, can be used as a more controlled means of producing

an external wrench on the object.

3.3.2 Controlled accelerations

It is possible to accelerate hand to create a force on an object. For example, we loosen our grasp and flip an object in our hands, we have used an acceleration to create a force on the object to produce a desired motion. The strategy is to create an external wrench on the object while adjusting the grasp so that the object is in a constraint state which will allow it to slip.

Using the definitions in the previous section, we will find the direction the hand should be accelerated to move the object through a desired twist. Again, assume a desired motion is specified by a twist $\mathbf{t_d}$ defined in screw coordinates relative to the object frame. To maximize the virtual work, the direciton of the acceleration, in terms of the object frame, should be in the same direction as the twist. Simply,

$$\mathbf{a} = \begin{bmatrix} a_\theta \\ a_\phi \\ a_\psi \\ a_x \\ a_y \\ a_z \end{bmatrix} = \frac{1}{|\,\mathbf{t_d}\,|} \begin{bmatrix} t_{d_1} \\ t_{d_2} \\ t_{d_3} \\ t_{d_4} \\ t_{d_5} \\ t_{d_6} \end{bmatrix}. \tag{3.8}$$

Although, the use of controlled accelerations is one way to move an object within the grasp, by far the most common and effective way to reorient an object within a grasp is the use of free finger, that is, fingers not directly involved with the grasp, and other objects.

3.3.3 Free fingers

Fingers not involved with the actual grasp can be used to move the object. This is very common in human manipulation. We can reorient a pen , for example, by holding it between two of our fingers and spinning it with a third. A multifingered robot can also use free fingers to manipulate objects. While some fingers constrain the object, the others have the freedom to reorient it within the grasp. This may also be an argument for multifingered hands, since the additional fingers allow greater flexibility in producing constraints

on an object as well as allowing the free fingers to manipulate object within the grasp.

Suppose the surface of the object can be represented by a set of points S and for each of the i free fingers there is a subset of S of accessible surface points, $S_i \subset S$.

Figure 3.2 shows an object located with respect to a reference frame $OXYZ$. For each point $\mathbf{x} \in S_i$, a specific set of forces can be exerted through the contact. Assume the forces lie within the friction cone at the contact. That is, the set of wrenches which can be exerted in terms of the contact frame $O_{c_i}X_{c_i}Y_{c_i}Z_{c_i}$, are

$$\{\mathbf{w_j} | \sqrt{w_{c_i}^2 + w_{c_2}^2} \leq \mu \sqrt{w_{c_3}}\}, \tag{3.9}$$

where μ is the coefficient of friction. The set of wrenches defined relative to the reference frame is

$$\{\mathbf{w_e} = \mathbf{T_i w_j}\}. \tag{3.10}$$

The wrenches in equation 3.7 are a combination of both the accessible surface points which the free finger can reach and the forces which it can exert through each contact point. Suppose a twist is defined $\mathbf{t_d}$ defined relative to the reference frame $OXYZ$. The values for the virtual work of the twist $\mathbf{t_d}$ against the wrenches in 3.7 is

$$W = \{\omega | \omega = w_{e_1} t_{d_4} + w_{e_2} t_{d_5} + w_{e_3} t_{d_6} + w_{e_4} t_{d_1} + w_{e_5} t_{d_2} + w_{e_6} t_{d_3}\}. \tag{3.11}$$

The combination of wrenches and accessible surface points which maximizes the virtual work can be expressed as a set of pairs

$$\{(\mathbf{w}, \mathbf{x}) | (\mathbf{w}, \mathbf{x}) \in \max\{W\}\}. \tag{3.12}$$

3.3.4 Other objects

Other objects can also be used effectively to control the motion of an object within a grasp. In the jar example, given in the introduction, we use the edge of the jar to rotate the lid between our fingertips. A robot could also exploit objects in the environment to reposition an object within a grasp. The strategy and the analysis for this type of manipulation are identical to the free finger analysis. Only the accessible surface points S_i are different,

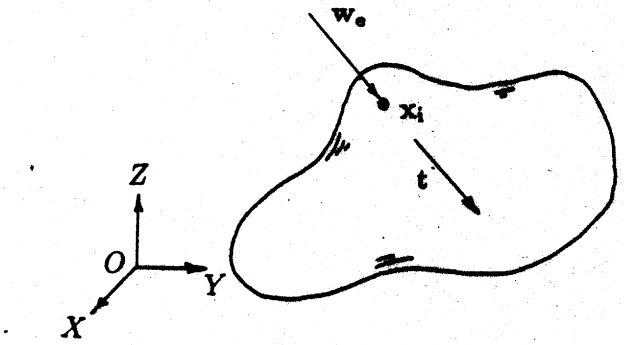


Figure 3.2: A force exerted on a grasped object can move it through a desired motion

since the points on the object open to the environment may be larger than those accessible to the fingers. The external wrench is one variable the robot can control; the next chapter will describe another controllable variable, the grasping force.

Chapter 4

Grasping force

4.1 Introduction

There are many ways for a multifingered hand to squeeze an object. People grasp objects to secure them in their hand or, by carefully controlling the internal forces, allow objects to slide through their fingers. A robot can also use the internal grasping force to create both stable and unstable grasps. By regulating the grasping force, the robot can either hold the object securely or by control the constraint state to allow desired slipping motion.

The variety of ways a hand can squeeze an object depends on the number and type of contacts which exist between the hand and the object. The goal of this chapter then is to determine the space of possible grasping forces on an object and to develop a simple intuitive parameterization of the squeezing force space for two and three fingered grasps.

4.2 Grasp force analysis

Assume the manipulator exerts a wrench on the object

$$\mathbf{w_{g_i}} = \begin{bmatrix} w_{g_1} \\ w_{g_2} \\ w_{g_3} \\ w_{g_4} \\ w_{g_5} \\ w_{g_6} \end{bmatrix}, \tag{4.1}$$

defined in screw coordinates relative to the contact frame $O_{g_i}X_{c_i}Y_{c_i}Z_{c_i}$. Suppose that the contact frame can be defined in terms of a common reference frame $OXYZ$, as shown in figure 4.1. The wrench in terms of the reference frame is

$$\mathbf{w_i} = \mathbf{T_i w_{g_i}}, \tag{4.2}$$

where $\mathbf{T_i}$ is the transformation matrix discussed in chapter 2. The sum of the wrenches from n contacts between the robot and the object is

$$\mathbf{w} = \sum_{i=1}^{n} \mathbf{w_i}. \tag{4.3}$$

Assume the object is not subject to any external wrench and is not accelerating. The sum of the wrenches in equation 4.3 is then zero,

$$\mathbf{w} = \mathbf{0}. \tag{4.4}$$

One additional constraint will be assumed. It is assume that only forces through a point contact contribute to the internal grasping force. The moments at the contact point will be assumed to be zero, so that the wrench, in terms of the contact frame, is given by $\mathbf{w_{g_i}} = [w_{g_1}, w_{g_2}, w_{g_3}, 0, 0, 0]$. With these assumptions, it is now possible to determine the wrench at each contact point relative to the contact frame, as a function of the internal grasping force.

4.3 Two contacts

Assume the robot touches an object at only two points. Therefore the sum of these wrenches, equation 4.4, is

$$\mathbf{T_1 w_{g_1}} + \mathbf{T_2 w_{g_2}} = \mathbf{0}. \tag{4.5}$$

There are six unknowns in equation 4.5. These are the wrenches, $\mathbf{w_1} = [w_{g1_1}, w_{g2_1}, w_{g3_1}, 0, 0, 0]$ and $\mathbf{w_2} = [w_{g1_2}, w_{g2_2}, w_{g3_2}, 0, 0, 0]$. However, equation 4.5 yields only five linear independent equations. Therefore, there is a one space of solutions for which equation 4.5 is true.

The one space of solutions can be thought of as an arbitrary squeezing force, given simply by a grasp force magnitude m_g. Once the grasp force

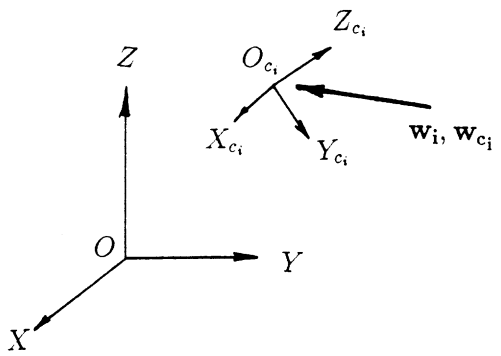


Figure 4.1: Assume each of the grasping fingers can exert a wrench $\mathbf{w_{c_i}}$ defined in terms of the contact frame through the contact point. The wrench defined with respect to the common reference frame is $\mathbf{w_i} = \mathbf{T_i w_{c_i}}$, where $\mathbf{T_i}$ is the transformation matrix discussed in chapter 2. The sum of each of the wrenches is $\mathbf{w} = \sum_{i=1}^{n} \mathbf{w_i}$ and if the object is neither subject to external wrenches nor accelerating, this wrench $\mathbf{w}$ is zero.

magnitude is specified, the values of the wrenches $\mathbf{w_{g_1}}$ and $\mathbf{w_{g_2}}$ can be found.

Let the grasp force magnitude m_g be given by

$$m_g = \sum_{i=1}^{n} \sqrt{w_{g1_i}^2 + w_{2g_i}^2 + w_{g3_i}^2}. \tag{4.6}$$

The screw coordinates of the wrench due to internal grasping force can be found more easily by considering the wrenches in terms of an axis, a pitch, and a magnitude. Assume the two contact forces are described by wrenches $\mathbf{w_1}$ and $\mathbf{w_2}$, defined by wrenches axes $\mathbf{A_1}$ and $\mathbf{A_2}$, zero pitches, and magnitudes m_1 and m_2 respectively. Since the object is neither subject to an external wrench nor accelerating, both the wrench axes must intersect the contact points and have opposite directions, figure 4.3. The wrench axes are given by,

$$\pm \mathbf{A_1} = \mp \mathbf{A_2} = \begin{bmatrix} (x_1 - x_2) \\ (y_1 - y_2) \\ (z_1 - z_2) \\ -z_1(y_1 - y_2) + y_1(z_1 - z_2) \\ z_1(x_1 - x_2) - x_1(z_1 - z_2) \\ -y_1(x_1 - x_2) + x_1(y_1 - y_2) \end{bmatrix}, \tag{4.7}$$

where d is the distance between the contact points

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}. \tag{4.8}$$

Notice the direction of the axes in equation 4.7 are opposite, yet the value of one of them may be either positive or negative. This analysis does not determine the sign in equation 4.7, since this will depend on the physical system. For example, if two fingers grasp a block, the wrench axes would be directed into the block, but if the fingers grasp inside a ring, the wrench axes would be directed outward. Therefore the direction of the wrench axes will depend on the direction of the contact normal. The direction of the axis which yields a negative value of the dot product between the contact normal and the wrench axis will be the correct direction.

The sum of the individual magnitudes is given in equation 4.6

$$m_g = m_1 + m_2. \tag{4.9}$$

The magnitudes of the wrenches, however, must be equal, since if one exceeds the other, there will be a net wrench on the body contradicting the equilibrium assumption; thus,

$$m_1 = m_2. \tag{4.10}$$

Then by equation 4.9 and 4.10

$$m_1 = m_2 = m_g/2. \tag{4.11}$$

The wrenches $\mathbf{w_1}$ and $\mathbf{w_2}$ are now uniquely defined. The wrench axes are given in equation 4.7, the magnitudes in equation 4.11, and the pitch of both wrenches is zero.

It is useful, in the later analysis, to express the wrenches in terms of a reference frame and the contact frame. The wrenches $\mathbf{w_i}$ defined in terms of screw coordinates relative to the reference frame are

$$\mathbf{w_1} = -\mathbf{w_2} = \pm\frac{m_g}{2d}\begin{bmatrix} (x_1 - x_2) \\ (y_1 - y_2) \\ (z_1 - z_2) \\ -z_1(y_1 - y_2) + y_1(z_1 - z_2) \\ z_1(x_1 - x_2) - x_1(z_1 - z_2) \\ -y_1(x_1 - x_2) + x_1(y_1 - y_2) \end{bmatrix}. \tag{4.12}$$

The screw coordinates of the wrenches in terms of the contact frame is a linear transformation of the elements in equation 4.12,

$$\mathbf{w_{g_1}} = \mathbf{T_1^{-1}}\mathbf{w_1} \tag{4.13}$$

$$\mathbf{w_{g_2}} = \mathbf{T_2^{-1}}\mathbf{w_2}, \tag{4.14}$$

where $\mathbf{T_i^{-1}}$ is the inverse of the linear transformation matrix discussed in chapter 2.

4.4 Three contacts

For a three fingered hand, such as the Salisbury or Okada [Okada] robots, three fingertip contacts are possible. Assume the robot touches the object at three points and can only exert forces through the contacts, then

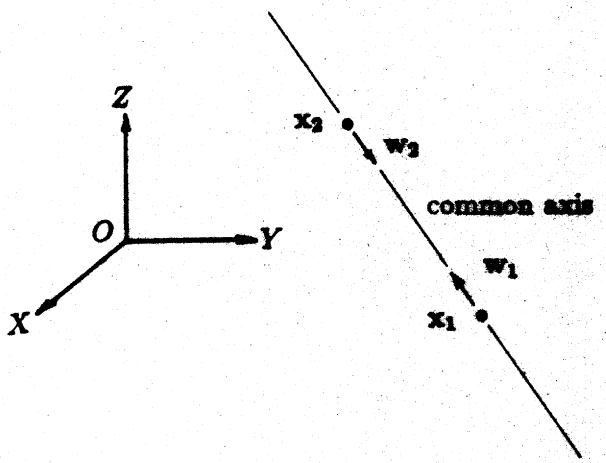


Figure 4.2: If only two fingertip touch an object and we assume only forces are exerted through the contact, the direction of the force vectors lie on the line intersecting the contact points. If the contact forces are described in terms of wrenches, the axes of wrenches intersect the two contact points, the magnitudes are identical, and the pitches are both zero.

the wrench at each contact defined in terms of the contact frame is $\mathbf{w_{g_i}} = [w_{g1_i}, w_{g2_i}, w_{g3_i}, 0, 0, 0]$ and the wrench, in screw coordinates, defined in terms of a reference frame is

$$\mathbf{w_i} = \mathbf{T_i w_{g_i}}. \tag{4.15}$$

The sum of the wrenches, from equation 4.4, is

$$\mathbf{T_1 w_{g_1}} + \mathbf{T_2 w_{g_2}} + \mathbf{T_3 w_{g_3}} = \mathbf{0}. \tag{4.16}$$

There are nine unknowns in equation 4.16, $\mathbf{w_1} = [w_{g1_1}, w_{g2_1}, w_{g3_1}, 0, 0, 0]$, $\mathbf{w_2} = [w_{g1_2}, w_{g2_2}, w_{g3_2}, 0, 0, 0]$, and $\mathbf{w_2} = [w_{g1_2}, w_{g2_2}, w_{g3_2}, 0, 0, 0]$, and equation 4.16 yields only six independent equations. This still leaves three indeterminant variables. The three space of possible solutions can be represented fairly simply. The wrench axes of each of the three wrenches will, in general, intersect at a point, the *grasp force focus*. The grasp force focus must lie on the *grasp plane*, defined to be the plane containing the three contact points. In general, the grasp plane will be well defined; however, if the contact points lie along a single line, the plane collapses to a line and the grasp focus lies on this line. If the grasp plane exists, the grasp force focus, may lie anywhere on the plane. As long as there is no external wrench on the object and the object is not accelerating, the wrenches will intersect at a point, with one exception: the wrench axes may all be parallel and still be in equilibrium. In this case, the grasp force focus parameterization can still be used if we also consider points at infinity on the grasp plane.

There is a third variable, the *grasp force magnitude* m_g. The grasp force magnitude is a scaling factor by which individual wrench values are multiplied. Together the grasp force focus and the grasp force magnitude span the entire three space of internal grasp solutions. By choosing a grasp force focus x_g y_g on the grasp plane and with a grasp force magnitude m_g, we can uniquely define the internal squeezing force created by a three fingered grasp.

To find the wrenches resulting from this parameterization of the grasp force, we first find the wrenches in terms of the *grasp frame*. The grasp frame is the coordinate system whose x and y axes lie on the grasp plane and origin is fixed at the centroid of the three contact points. Let $\mathbf{w_i'}$ represent the wrench defined relative to the grasp frame $\mathbf{x_i'} = [x_i', y_i', 0]$ represent a contact point. The wrench axes $\mathbf{A_i'}$ of each of the wrenches will be the direction

vectors from the contact points $\mathbf{x}_i'$ to the grasp force focus $\mathbf{x_g}$.

$$\mathbf{A}_i' = \pm\frac{1}{|d_i|}\begin{bmatrix} x_g - x_i' \\ y_g - y_i' \\ 0 \\ 0 \\ 0 \\ -y_i'(x_g - x_i') + x_i'(y_g - y_i') \end{bmatrix}. \tag{4.17}$$

where $|d_i| = \sqrt{(x_g - x_i')^2 + (y_g - y_i')^2}$. The sign used in equation 4.17 depends on geometry. If the fingers were grasping a sphere and the grasp force focus was in the center of the sphere, the sign of $\mathbf{A}_i'$ would be positive. Conversely, if the fingers were grasping a ring from the inside the sign of $\mathbf{A}_i'$ would be negative. In general, the sign depends on the direction of the normal to the contact. The sign should be chosen so that the wrench axis and the contact normal are in the same direction. Let the grasp force magnitude be the sum of the individual wrench magnitudes, m_i,

$$m_g = m_1 + m_2 + m_3. \tag{4.18}$$

Since the wrench axes are given by $\mathbf{A}_i'$, in equation 4.17, and the magnitudes are given by m_i, the first two elements of the wrench in screw coordinates relative to the grasp frame are

$$w_{1_i}' = m_i(x_g - x_i')/|d_i| \tag{4.19}$$
$$w_{2_i}' = m_i(y_g - y_i')/|d_i|. \tag{4.20}$$
$$\tag{4.21}$$

By equation 4.4, the sum of the elements in equation 4.21 must sum to zero. Together with equation 4.18 this yields three linearly independent equations and three unknowns,

$$\begin{bmatrix} (x_g - x_1')/|d_1| & (x_g - x_2')/|d_2| & (x_g - x_3')/|d_3| \\ (y_g - y_1')/|d_1| & (y_g - y_2')/|d_2| & (y_g - y_3')/|d_3| \\ 1 & 1 & 1 \end{bmatrix}\begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ m_g \end{bmatrix}. \tag{4.22}$$

Solve for the magnitudes m_i,

$$\begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} = \begin{bmatrix} (x_g - x_1')/|d_1| & (x_g - x_2')/|d_2| & (x_g - x_3')/|d_3| \\ (y_g - y_1')/|d_1| & (y_g - y_2')/|d_2| & (y_g - y_3')/|d_3| \\ 1 & 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ m_g \end{bmatrix}. \tag{4.23}$$

The matrix in equations 4.22 and 4.23 will be invertible except when the three contact points lie on a line and the grasp force focus does not. In this case, choose a grasp force focus on the line and then the magnitude will be proportional to the distance from the contact points to the focus,

$$m_i = m_g|d_i|. \tag{4.24}$$

In any case equation 4.17 defines the wrench axes, equations 4.23 and 4.24 defines the wrench magnitudes, and pitch are all zero. The wrench resulting from the internal force at each contact is now defined by a grasp force focus x_g and y_g and a grasp force magnitude m_g.

The wrenches in screw coordinates relative to the grasp frame are

$$\mathbf{w_i'} = \pm\frac{m_i}{|d_i|}\begin{bmatrix} x_g - x_i' \\ y_g - y_i' \\ z_g - z_i' \\ -z_i'(y_g - y_i') + y_i'(z_g - z_i') \\ z_i'(x_g - x_i') - x_i'(x_g - x_i') \\ -y_i'(x_g - x_i') + x_i'(y_g - y_i') \end{bmatrix}, \tag{4.25}$$

and in terms of the contact frames

$$\mathbf{w_{g_i}} = \mathbf{T_i'}\mathbf{w_i'}. \tag{4.26}$$

where $\mathbf{T_i'}$ is the linear transformation relating the wrench in the contact frame to the grasp frame.

4.5 Four or more fingered grasps

For four fingered hand like the MIT/Utah hand, [Jacobsen], there are twelve unknowns and only six equations, yielding a six space of internal grasping force solutions. For the human hand, for manipulators with six or more

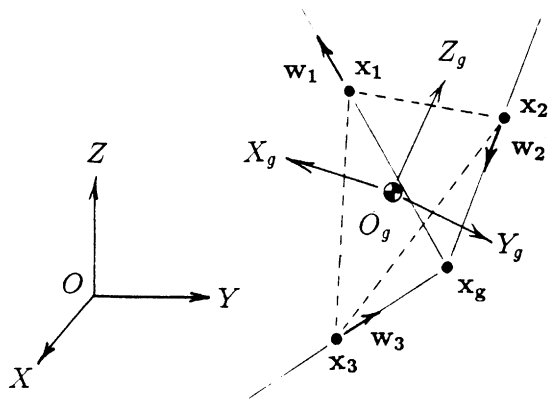


Figure 4.3: There is a three space of solutions for the internal grasping force for a three fingered hand. The axes of the force vectors from the individual contacts must intersect at a point, the *grasp force focus* $\mathbf{x_g} = [x_g, y_g, 0]$ in the *grasp plane.* The *grasp force magnitude* m_g scales the individual fingertip forces. Together, the grasp force centroid and the grasp force magnitude span the three space of internal grasp solutions.

fingers, and for grasps involve more than five contacts, the dimension of the internal force space grows very large. The dimension of the solution space for internal grasp force is

$$Dim\{G\} = 3(N-2) + U, \tag{4.27}$$

where G is the space of internal grasp solutions, N is the number of contacts, and U is the number of freedoms of the grasped object.

Chapter 5

Contact wrenches

5.1 Introduction

In order to determine the different ways an object can move in a grasp, the constraint state of the object must be known. The constraint state of the object can be found knowing the location, orientation, and the types of contacts that exist between the object and the manipulator. A tactile sensor might supply contact location and orientation information, but a sensor which determines the contact type does not exist. In order to find the contact type at the fingertips of the robot, the *contact wrench*, that is the set of forces and moments which exist at the contact point relative to the contact frame, must be found, as well as, a relationship between the contact wrench to the contact type. The purpose of this chapter is to determine the contact wrench. The purpose of the next chapter will be to determine a simple relationship between the contact wrench and the contact type. In this way we can find the constraint state of the object as a function of the grasping force and the external forces.

5.2 Stiffness

Assume a hemispherical compliant fingertip touches a flat plate, as shown in figure 5.2. To begin the analysis, assume the interface between the fingertip and the object can be modeled as a soft finger contact. That is, both forces normal and tangent to the surface as well as moments about the surface nor-

mal can be transmitted through the contact point. A system of translational and rotational springs attached to the contact point will be used to model the fingertip, as shown in figure 5.2. The stiffnesses of the ith fingertip can be represented as a matrix relating the twist to the wrench

$$\mathbf{k_{c_i}} = \begin{bmatrix} & & & kc_1 & 0 & 0 \\ & \mathbf{0} & & 0 & kc_2 & 0 \\ & & & 0 & 0 & kc_3 \\ kc_4 & 0 & 0 & & & \\ 0 & kc_5 & 0 & & \mathbf{0} & \\ 0 & 0 & kc_6 & & & \end{bmatrix}_i . \tag{5.1}$$

For a soft finger contact, the elements kc_4 and kc_5 are zero, that is, the fingertips cannot generate moments about any axis which lies in the tangent plane. However, for generality kc_4 and kc_5 are included, since linear and planar contacts have these stiffness and these contact types may be used in future analyses.

The stiffness given in equation 5.1 is defined in terms of the contact frame $O_{c_i}X_{c_i}Y_{c_i}Z_{c_i}$. The fingertip stiffness defined relative to a common reference frame $OXYZ$ is

$$\mathbf{k_i} = \mathbf{T_i k_{c_i} T_i^{-1}}. \tag{5.2}$$

where $\mathbf{T_i}$ is a transformation matrix of the ith contact discussed chapter 2 equation 2.14. The stiffness of the entire object will be the sum of all the individual stiffness matrices

$$\mathbf{K} = \sum_{i=1}^{n} \mathbf{k_i}. \tag{5.3}$$

Assume the object is displaced by a twist $\mathbf{t}$, the wrench on the object is given by

$$\mathbf{w} = \mathbf{K t}. \tag{5.4}$$

If the object is not accelerating, the wrench given in equation 5.4 must equal the external wrench on the object, $\mathbf{w_e}$,

$$\mathbf{w} = \mathbf{w_e}. \tag{5.5}$$

The twist the object undergoes can be determined as a function of the external wrench on the object, by taking the inverse of $\mathbf{K}$ in equation 5.4,

$$\mathbf{t} = \mathbf{K^{-1} w_e}. \tag{5.6}$$

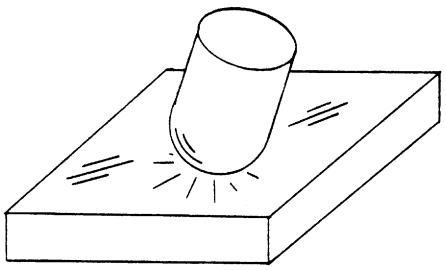


Figure 5.1: The contact between the manipulator and an object is generalized as a hemispherical compliant solid touching a nondeformable flat plate.

The twist in equation 5.6 is defined in screw coordinates relative to the $OXYZ$ reference frame. The twist defined relative to the contact frame can be found by multiplying by the inverse of the transform matrix $\mathbf{T_i}$,

$$\mathbf{t_i} = \mathbf{T_i K^{-1} w_e}, \tag{5.7}$$

and the wrench at the contact, in terms of the contact frame, is

$$\mathbf{w_i} = \mathbf{k_{c_i} T_i K^{-1} w_e}. \tag{5.8}$$

Add to this the wrench due to the grasping force

$$\mathbf{w_i} = \mathbf{k_{c_i} T_i K^{-1} w_e} + \mathbf{w_{g_i}}. \tag{5.9}$$

Thus, equation 5.9 describes the wrench at the contact point defined relative to the contact frame as a function of the stiffness of the fingertips, the forces on the body, and the force the grasp. Using this result and the results in the next chapter, it will be possible to determine the contact type as a function of the forces on the object and the squeezing force. Once the contact types are found, the constraint state of the object will be known.

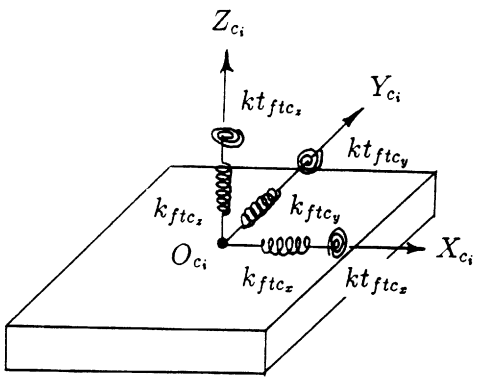


Figure 5.2: The fingertip is modeled as a set of translational and rotational springs connected to the contact point. The stiffnesses are defined in terms of the contact frame. Since a soft finger contact is assumed, there is a stiffness in the normal and tangent directions of the contact plane, as well as, a torsional stiffness about the contact normal.

Chapter 6

Contact wrench/contact type relation

6.1 Introduction

The purpose of this chapter is to determine a relationship between the contact type and the contact wrench. In general, this is a difficult problem. In fact, determining slipping and twisting of one compliant body in contact with another is a current research topic in finite element analysis. In this chapter, therefore, simplifying assumptions will be made to find a simple relation between contact type and contact wrench, so that the overall problem of determining the constraint state can be tractable.

6.2 Two dimensions

For the two dimensional problem, the relationship between the contact wrench and the contact type can be simple. Assuming the fingertips erert only forces though the contact points, there are only three contact types: a point contact with friction, a point contact without friction, and no contact. Assuming coulomb friction, if the tangent force is less than the coefficient of static friction times the normal force, the contact will behave like a point contact with friction. If the tangent force exceeds the coefficient of friction times the normal force, the contact will slide, is modeled as a point contact without

friction. Given a wrench at the contact point defined by

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \tag{6.1}$$

where w_1 force in the tangent direction, w_2 is the force normal to the surface. The possible contact types are

$$\begin{array}{ll} 1 & = \text{Point contact with friction} \\ 2 & = \text{Point contact without friction} \\ 3 & = \text{No contact,} \end{array} \tag{6.2}$$

and the relationship between the contact wrench and the contact types will be

$$\begin{array}{lcl} |w_1| < |\mu w_2| & \Rightarrow & 1 \\ |w_1| \geq |\mu w_2| & \Rightarrow & 2 \\ w_2 > 0 & \Rightarrow & 3. \end{array}$$

For the three dimensional case, an exact relation between the contact type and the contact wrench is difficult; however, a simple relation can be useful.

6.3 Three dimensions

Determining when and where a three dimensional compliant contact breaks or twists in response to an applied wrench is currently at the forefront of finite element analysis. Exact solutions are, in general, unknown. The force distribution over the contact region between two hemi-ellipsoidal objects under an axially applied load can, be found by use of a linearization argument, commonly known as the Hertz contact problem [Lipson]. This simple case does not take into account loads applied tangent to the contact surface or to moments about the contact normal; however, we can use the Hertz solution to generate a simple relation between contact type and contact wrench.

Hertz assumed two bodies in contact could be modeled as solid elliptic disks, each possessing a minimum and maximum radii of R and R'. Furthermore, he assumed that these disks were in contact along their common axes

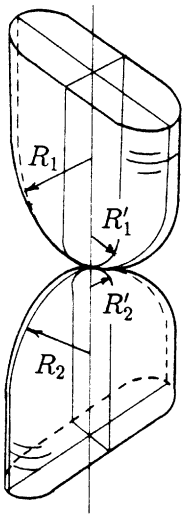


Figure 6.1: Two general solid elliptic disks in contact under an axially applied load. The load is assume small and the disks somewhat rigid, so that a linearization argument may be applied to determine the pressure distribution and the contact region.

under a uniform axially applied load F, as shown in figure 6.3. Hertz deduced that the pressure distribution between the two bodies can be described by a semi-ellipsoid of pressure constructed over the surface of the contact, as shown in figure 6.3.

The pressure distribution is given by

$$P(x,y) = P_o\sqrt{1 - x^2/a^2 - y^2/b^2}. \tag{6.3}$$

The total load, therefore, is equal to the volume of the semi-ellipsoid,

$$F = \frac{2\pi abP_o}{3}. \tag{6.4}$$

Solve for P_o

$$P_o = \frac{3F}{2\pi ab}, \tag{6.5}$$

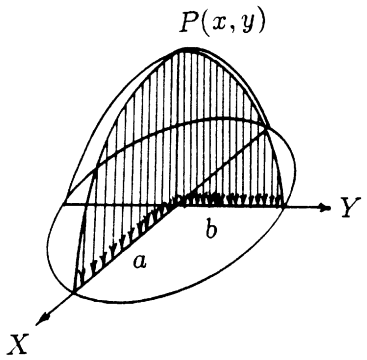


Figure 6.2: Hertz modeled the pressure distribution in the contact area between the two solid elliptic disks as a semi-ellipsoid with a minimum and maximum dimensions a and b.

where a and b are given by Timosheko [Lipson]

$$a = m\sqrt[3]{\frac{3F\Delta}{4(A+B)}}, \tag{6.6}$$

$$b = n\sqrt[3]{\frac{3F\Delta}{4(A+B)}}, \tag{6.7}$$

and

$$\Delta = (1-\mu_1^2)/E_1 + (1-\mu_2^2)/E_2, \tag{6.8}$$

$\mu_1, \mu_2 =$ Poisson's ratio

$E_1, E_2 =$ Modulii of elasticity

$A + B = \frac{1}{2}\left(\frac{1}{R_1} + \frac{1}{R_1'} + \frac{1}{R_2} + \frac{1}{R_2'}\right)$

$$B - A = \frac{1}{2}\sqrt{\left(\frac{1}{R_1} - \frac{1}{R_1'}\right) + \left(\frac{1}{R_2} - \frac{1}{R_2'}\right)^2 + 2\left(\frac{1}{R_1} - \frac{1}{R_1'}\right)^2\left(\frac{1}{R_2} - \frac{1}{R_2'}\right)\cos(2\psi)}$$

$\psi =$ The angle between the planes contacting the curvatures $1/R_1$ and $1/R_2$

$R_1, R_1', R_2, R_2' =$ Mimimum and maximum radii of curvature of the ellisoid disks at the point of contact

$m, n =$ Constants depending on $B - A/B + A$

In the case of the contact between a hemispherical fingertip and a flat rigid plate, equations 6.6 and 6.7 is

$$a = b = \sqrt[3]{3FR_1\Delta/4}, \tag{6.9}$$

and the maximum pressure is given by

$$P_o = 0.578\sqrt[3]{\frac{P}{R_1^2\Delta^2}}. \tag{6.10}$$

If the flat plate is assumed to be rigid (i.e. $E_2 \gg E_1$) then Δ can be approximated by

$$\Delta = (1 - \mu_1^2)/E_1. \tag{6.11}$$

The material on the fingertip of the Salisbury robot hand is a polyureathane with an elastic modulus of approximately 40,000 psi. and a poisson's ratio of about 0.45. The radius of the fingertip is approximately 0.5 in, therefore

$$\begin{aligned} \Delta &= 2.0 \times 10^{-5} \\ P_o &= 1247.9\sqrt[3]{F}\,\mathrm{lb/in^2} \\ a &= 0.020\sqrt[3]{F}\,\mathrm{in.} \end{aligned}$$

The pressure distribution given in equation 6.3 can now be given in polar coordinates, since the pressure region in this case is given by the area of a circle. That is,

$$P(r,\theta) = P_o \frac{\sqrt{a^2 - r^2}}{a}, \tag{6.12}$$

where r is the radial distance from the center of the contact region. Again assuming coulomb friction, the maximum moment the contact can support before rotating is the integral over the contact area of the differential force times the coefficient of static friction times the radial distance from the contact point,

$$M = \mu \int_A r df. \tag{6.13}$$

Substitute these values into equation 6.13

$$M = \mu \int_{r=0}^{r=a} \int_{\theta=0}^{\theta=2\pi} \frac{3F\sqrt{a^2 - r^2}}{2\pi a^3} r^2 dr d\theta, \tag{6.14}$$

and evaluating yields

$$M = \frac{3\pi\mu a F}{16}. \tag{6.15}$$

Substituting the value for a

$$M = \frac{3\pi\mu F}{16} \sqrt[3]{\frac{3FR(1-\mu_1^2)}{4E_1}}. \tag{6.16}$$

Thus, M is proportional to the four-thirds power of the normal force, while the constant of proportionality is a function of constant properties of the fingertip material and the surface,

$$M = \mu_m F^{4/3}, \tag{6.17}$$

where

$$\mu_m = \frac{3\pi\mu}{16} \sqrt[3]{\frac{3R(1-\mu_1^2)}{4E_1}}. \tag{6.18}$$

If the specific values for the Salisbury robot fingertips are substituted along with a value of $\mu = 1.0$ for the coefficient of static friction, the maximum twisting moment will be

$$1.15 \times 10^{-2} F^{4/3}. \tag{6.19}$$

Now ignore the moment and assume a force is exerted tangent to the contact surface. This is simply the case of coulomb friction, and the maximum tangent force will be

$$F_t = \mu F. \tag{6.20}$$

In order to further simplify this analysis, assume the maximum moment which can be exerted about the contact normal is a linear function of the normal force,

$$M = \mu_m F, \tag{6.21}$$

where μ_m is given in equation 6.18.

The purpose of this analysis is to determine the contact type as a function of the contact wrench. Given the contact wrench in the screw coordinates

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \end{bmatrix}, \tag{6.22}$$

where w_4 and w_5 will automatically be zero, since we are only considering fingertip contacts. The relationship between the contact wrench and the contact type, given the linearizing assumptions discussed above, will be

$$\begin{array}{lllll} \sqrt{w_1^2 + w_2^2} < |\mu_f w_3| & \text{and} & \sqrt{w_6^2} < |\mu_m w_3| & \Rightarrow & 1 \\ \sqrt{w_1^2 + w_2^2} < |\mu_f w_3| & \text{and} & \sqrt{w_6^2} \geq |\mu_m w_3| & \Rightarrow & 2 \\ \sqrt{w_1^2 + w_2^2} \geq |\mu_f w_3| & & & \Rightarrow & 3 \\ w_3 \geq 0 & & & \Rightarrow & 4, \end{array} \tag{6.23}$$

where

1 Soft finger contact

2 Point contact with friction

3 Point contact without friction

4 No contact.

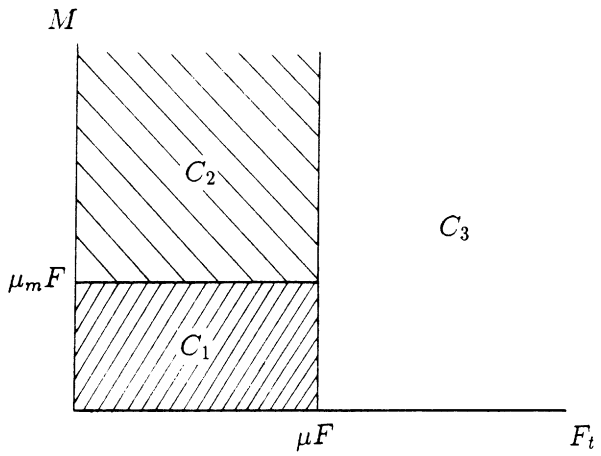


Figure 6.3: When the force tangent to the contact exceeds μ_f times the normal force, assume the contact is sliding on the surface. If the tangent force is less than $\mu_f F$, and the moment about the contact normal exceeds μ_m times the normal force, assume the fingertip is twisting relative to the contact surface. Finally, if both the tangent force and moment are less than $\mu_f F$ and $\mu_m F$ respectively, assume the contact is rigidly fixed to the surface.

We will approximate $\mu_m = 1.83 \times 10^{-3}$ in and $\mu = 1.0$ for the Salisbury robot hand grasping an aluminum can. With the relation given in equation 6.23 and the analysis of the previous chapter, it will now be possible to determine the constraint state of a grasped object as a function of the external forces on the object and the squeezing force of the hand.

Chapter 7

Controlled Slipping

7.1 Introduction

So far only individual modeling issues have been discussed, but we have not yet addressed how these results can be combined and how they can be used to enhance the dexterity of a robot hand. In this chapter, the analyses of the preceding chapters will be integrated together and the result will be used to predict and affect some controlled slip motions. To illustrate these principles, two examples will be considered. The first is a simple two dimensional example of two fingers holding a rectangle in a gravitational field. This example outlines the analyses of the previous chapters in some detail and allows many of the concepts to be graphically illustrated. The second example is of a three fingered robot hand grasping a cylinder. This example demonstrates how controlled slip manipulation might be implemented on an actual robot system. The next chapter describes how these ideas were applied to the multifingered Salisbury robot hand.

7.2 Two dimensional example

Two fingers grasp a rectangle on opposing surfaces, as shown in figure 7.1. In this particular example, the fingers grasp the block at the center of its top and bottom sides. The block has a height of 4 cm and a length of 7 cm and is held horizontally in a gravitational field. There are a number of coordinate systems used in this example, figure 7.2. These are: the reference frame,

hand frame, object frame, and contact frames. These coordinate frames are used throughout the analysis and are described in appendix A.

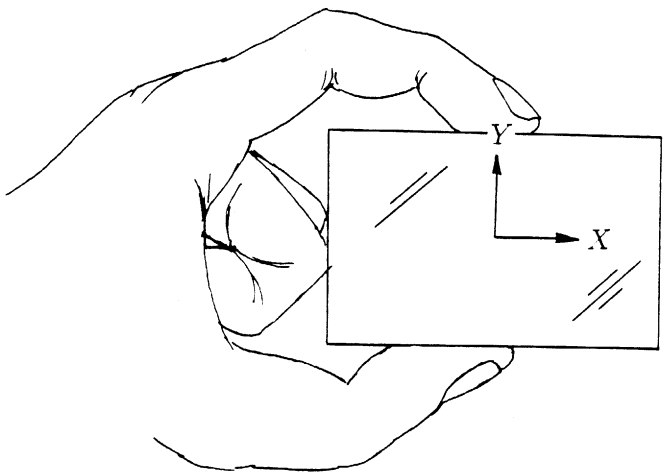


Figure 7.1: Two fingers grasp a two dimensional rectangular block. In this example, we are assuming the block only moves in the xy plane and that gravity exerts a force on the block in the negative y direction.

7.2.1 Constraint

Contact types

There are three different contact types for fingertip contacts in the plane: a point contact with friction, a point contact without friction, and no contact. Each contact type allows only a certain set of wrenches to be transmitted through the interface. As with the three dimensional case, these wrenches may be described by the set of unidirectional unit basis wrenches, which span the space of permissible wrenches. For convenience each contact types is represented by a number as follows:

1 Point contact with friction
2 Point contact without friction
3 No contact

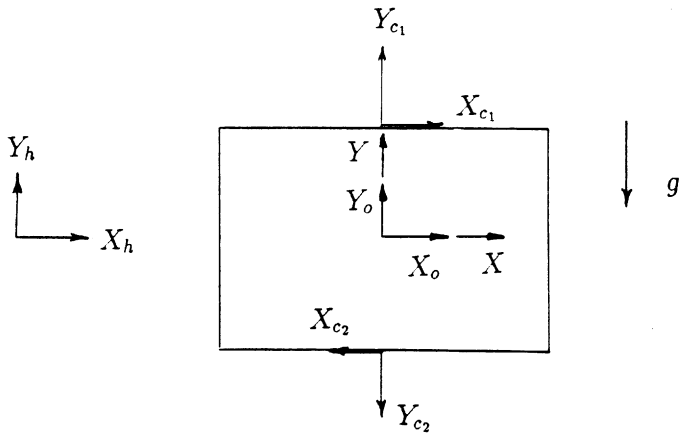


Figure 7.2: There are five coordinate frames shown in this example, the reference frame, the hand frame, the object frame, and the contact frames. The block has dimensions 4x7 cm and a mass of 0.2 kg.

Constraint states

In the example illustrated in figure 7.1, there are only two contacts. With three possible contact types at each contact, yielding a total of 3^2 or nine constraint states. As with the three dimensional case, the constraint state is described as an ordered list with as many elements as there are contacts. Let the contact made by the finger on the top of the rectangle be first element in the list and the finger on the bottom be second. Thus, if the fingertip on the top of the block were beginning to slip, but the finger on the bottom remained fixed, the constraint state of the grasped rectangle would be $[2, 1]$.

Permissible motions

For each constraint state, there are some motions which the object may undergo and some which it may not. As before, the set of allowed motions can be represented by a set of twists. For consistency, the twist will still be represented by the six element vector $\mathbf{t} = [t_1, t_2, t_3, t_4, t_5, t_6]$; however, for the two dimensional case, the elements t_1, t_2, and t_6 are zero, the elements t_4

and t_5 are the translations in the x and y directions, and t_3 is the rotation about the normal to the plane. All the elements of the twist are defined with respect to some reference frame $OXYZ$, which in this case coincides with the object frame, figure 7.1.

In the two dimensional case, the set of permissible twists may be represented graphically. The direction of motion can be described by a vector in three space. coordinate system. Figure 7.3 shows a coordinate system with axes labeled t_4, t_5, and t_3, corresponding to the translations in the x and y directions and the rotation respectively. The direction of motion may be represented by a unit vector radiating from the origin, thus set of twists correspond to a collection of unit vectors, figure 7.4. If the twists all lie in a plane the set is represented as a section of a shaded disk, otherwise sets of twists are shown as sections of a sphere, figures 7.5 and 7.6.

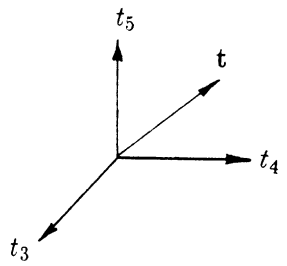


Figure 7.3: The direction of motion in may be represented using a simple graphic technique. Unit twists corresponding to the direction of motion are illustrated as a unit vector radiating from the origin of a coordinate system with axes labeled t_4 and t_5 (translations in the x and y directions), and t_3 (the rotations about the normal to the plane).

In the two dimensional case as in the three, the set of permissible motions is the intersection of all the unit twists repelling to the unit basis wrenches

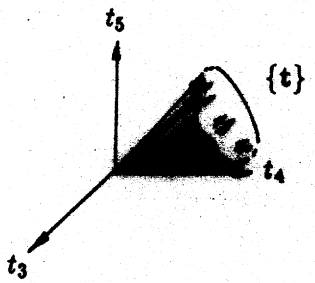


Figure 7.4: Set of unit twist are collections of unit vectors radiating from the origin.

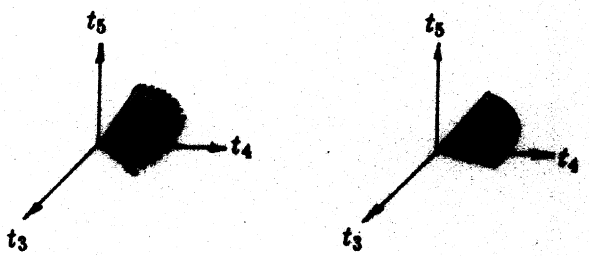


Figure 7.5: Unit twists which lie in a plane are shown as section of a shaded disk for clarity, otherwise they are shown as section of a sphere.

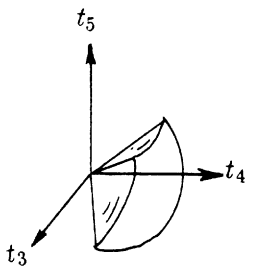


Figure 7.6: Unit twists correspond to subsets of the sphere.

and unit twists reciprocal to the normal basis wrenches which do not violate the geometric constraints of the contact surface. For this example, the set of unit twists repelling or reciprocal to all the unit basis wrenches is shown in figure 7.7. Figure 7.8 shows the additional constraint imposed by the geometry of the contact surface. In the diagrams, permissible motion is represented by either single vectors, shaded disks and sections of spheres.

7.2.2 External wrench

A wrench in two dimensions may be represented by the six element vector $\mathrm{w} = [w_1, w_2, w_3, w_4, w_5, w_6]$, where w_3, w_4, and w_5 are zero. The elements w_1 and w_2 represent force in the x and y directions respectively and w_6 represents the moment about the normal to the plane.

Gravity

The gravitational force on the object can be used to move the object within the grasp. Assume the rectangle is in a gravitational field, with an acceleration g in the negative y direction. Let the rectangle have a mass m. Then

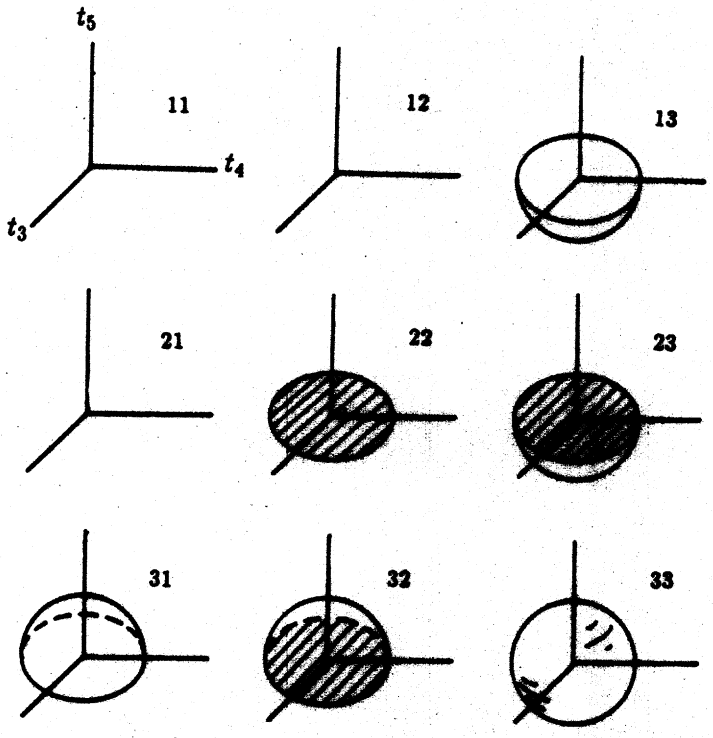


Figure 7.7: Constraint states for the rectangular block ignoring the local surface geometry

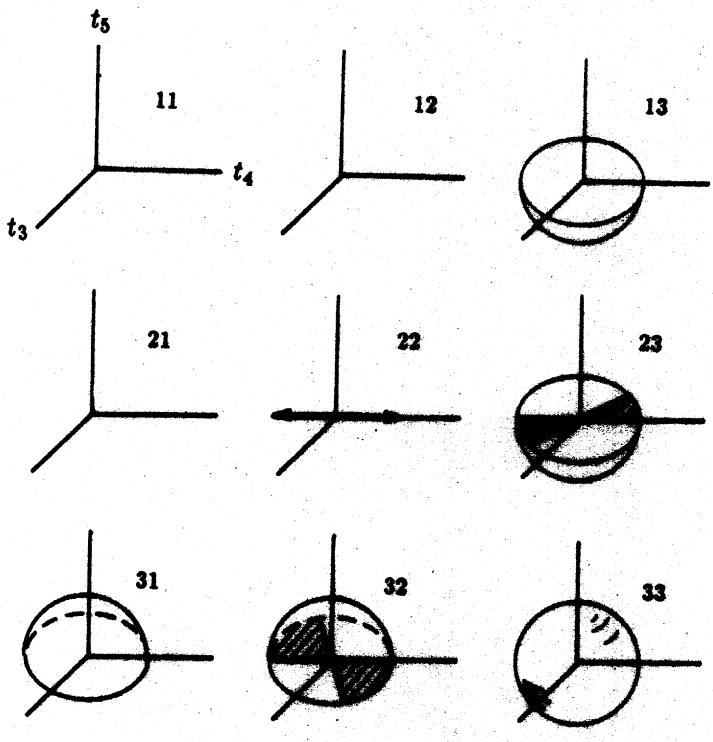


Figure 7.8: Constraint states for the rectangular block including local surface geometry

wrench on the object in screw coordinates defined with respect to the $OXYZ$ reference frame is

$$\mathbf{w} = \begin{bmatrix} 0 \\ -mg \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \tag{7.1}$$

and with respect to the object frame, it is

$$\mathbf{w} = \begin{bmatrix} -mg\sin\theta \\ -mg\cos\theta \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{7.2}$$

Suppose we can reorient the block in the xy plane to any angle θ, figure 7.9. Through the use of gravity, we have means of changing the wrench on the rectangle with respect to the object frame. If the block have a mass of 0.2039 kg, the external wrench (in newtons) is

$$\mathbf{w} = \begin{bmatrix} -2\sin\theta \\ -2\cos\theta \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{7.3}$$

7.2.3 Grasping force

Since there are two point contacts between the hand and the object, the fingers can exert an arbitrary squeezing force. Defined relative to the contact

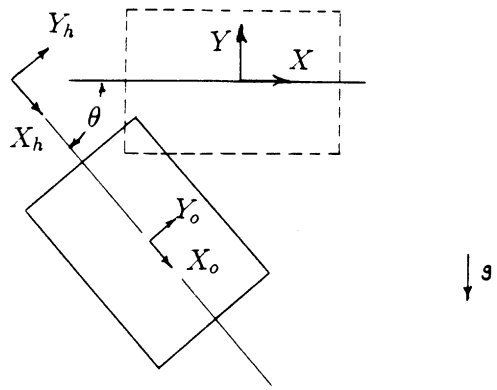


Figure 7.9: The rectangle can be reoriented in a gravitational field to any orientation θ

frames at each of the fingers, this internal force will be

$$\mathbf{w_1} = \mathbf{w_2} = \begin{bmatrix} 0 \\ -F/2 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{7.4}$$

7.2.4 Contact wrenches

Since we are only considering point contacts, in this planar example, only translational stiffness is possible, as shown in figure 7.2.4. The stiffness normal and tangent to the contact surface are given by

$$\begin{aligned} k_{normal} &= 5\text{N/cm} \\ k_{tangent} &= 2\text{N/cm}. \end{aligned} \tag{7.5}$$

The wrench at the ith contact defined with respect to the contact frame is

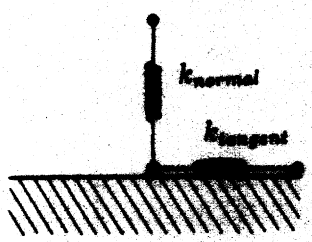


Figure 7.10: Fingertip contacts between a manipulator and an object in two dimensions can be replaced by a set of two springs: a spring normal to the contact with a stiffness of k_{normal} and a spring tangent to the surface $k_{tangent}$.

$$\mathbf{w_i} = \mathbf{k_{c_i}} \mathbf{T_i}^{-1} \mathbf{K}^{-1} \mathbf{w_e} + \mathbf{wg_i}, \tag{7.6}$$

where,

1. k_{c_i} is 6×6 element stiffness matrix representing the the stiffness at the contact point in terms of the contact frame,

$$\mathbf{k_{c_i}} = \begin{bmatrix} 0 & 0 & 0 & -k_{tangent} & 0 & 0 \\ 0 & 0 & 0 & 0 & -k_{normal} & 0 \\ 0 & 0 & 0 & 0 & 0 & -\infty \\ -\infty & 0 & 0 & 0 & 0 & 0 \\ 0 & -\infty & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

   Note the values of the stiffness are negative, since we are interested in the force *on the object* as a result of the displacement *of the fingertips.* Replaced $\mathbf{k_{c_1}}$ and $\mathbf{k_{c_2}}$ with the numerical values

$$\mathbf{k_{c_1}} = \mathbf{k_{c_2}} = \begin{bmatrix} 0 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & -5 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\infty \\ -\infty & 0 & 0 & 0 & 0 & 0 \\ 0 & -\infty & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

2. $\mathbf{T_i}$ is a transformation matrix for the ith contact defined in chapter 5. For two dimensions, this is

$$\mathbf{T_i} = \begin{bmatrix} \begin{bmatrix} l_x & m_x & 0 \\ l_y & m_y & 0 \\ 0 & 0 & 1 \end{bmatrix} & \mathbf{0} \\ \begin{bmatrix} 0 & 0 & y \\ 0 & 0 & -x \\ l_y x - l_x y & m_y x - m_x y & 0 \end{bmatrix} & \begin{bmatrix} l_x & m_x & 0 \\ l_y & m_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{bmatrix}_i,$$

and the inverse

$$\mathbf{T}^{-1} = \begin{bmatrix} \begin{bmatrix} l_x & l_y & 0 \\ m_x & m_y & 0 \\ 0 & 0 & 1 \end{bmatrix} & \mathbf{0} \\ \begin{bmatrix} 0 & 0 & l_y x - l_x y \\ 0 & 0 & m_y x - m_x y \\ y & -x & 0 \end{bmatrix} & \begin{bmatrix} l_x & l_y & 0 \\ m_x & m_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{bmatrix}_i .$$

$\mathbf{m_i} = [m_x, m_y]_i$ and $\mathbf{n_i} = [n_x, n_y]_i$ are the tangent and normal vectors; and, x_i and y_i is the location of the ith contact. Therefore, in this example,

$$\mathbf{T_1} = \begin{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} & \mathbf{0} \\ \mathbf{I} & \mathbf{I} \end{bmatrix},$$

and

$$\mathbf{T_2} = \begin{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} & \mathbf{0} \\ \begin{bmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} & \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{bmatrix},$$

and

$$\mathbf{T_1^{-1}} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \begin{bmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} & \mathbf{I} \end{bmatrix},$$

and

$$\mathbf{T_2^{-1}} = \begin{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} & \mathbf{0} \\ \begin{bmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} & \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{bmatrix}.$$

3. $\mathbf{K}$ is the total stiffness of the grasped object

$$\mathbf{K} = \sum_{i=1}^{n} \mathbf{k_i},$$

where $\mathbf{k_i}$ is the stiffness resulting from the ith contact at the origin of the object frame,

$$\mathbf{k_i} = \mathbf{T_i} \mathbf{k_{c_i}} \mathbf{T_i^{-1}},$$

where $\mathbf{T_i}$ and $\mathbf{T_i^{-1}}$ are given above and

$$\mathbf{k_1} = \begin{bmatrix} 0 & 0 & 2 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & -5 & 0 \\ -\infty & 0 & 0 & 0 & 0 & -\infty \\ -2\infty & 0 & 0 & 0 & 0 & -\infty \\ 0 & -\infty & 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 2 & 0 & 0 \end{bmatrix},$$

and

$$\mathbf{k_2} = \begin{bmatrix} 0 & 0 & 2 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & -5 & 0 \\ -\infty & 0 & 0 & 0 & 0 & -\infty \\ -2\infty & 0 & 0 & 0 & 0 & \infty \\ 0 & -\infty & 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 2 & 0 & 0 \end{bmatrix}.$$

Thus

$$\mathbf{K} = \begin{bmatrix} 0 & 0 & 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & 0 & -10 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2\infty \\ 4\infty & 0 & 0 & 0 & 0 & 0 \\ 0 & 2\infty & 0 & 0 & 0 & 0 \\ 0 & 0 & -4 & 0 & 0 & 0 \end{bmatrix},$$

and

$$\mathbf{K^{-1}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{-4} \\ \frac{1}{-4} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{-10} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

4. $\mathbf{w}_e$ is the external wrench,

$$\mathbf{w}_e = \begin{bmatrix} -2\sin\theta \\ -2\cos\theta \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

5. $\mathbf{w}_{g_i}$ is the internal wrench,

$$\mathbf{w}_{g_i} = \begin{bmatrix} 0 \\ -F/2 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Substituting the values for this particular example, the contact wrenches are

$$\mathbf{w}_1 = \begin{bmatrix} -\sin\theta \\ -\cos\theta - F/2 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \tag{7.7}$$

and

$$\mathbf{w}_2 = \begin{bmatrix} \sin\theta \\ \cos\theta - F/2 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{7.8}$$

Plots of the contact wrenches as a function of the squeeze force, F, and the orientation θ are given in figure 7.11.

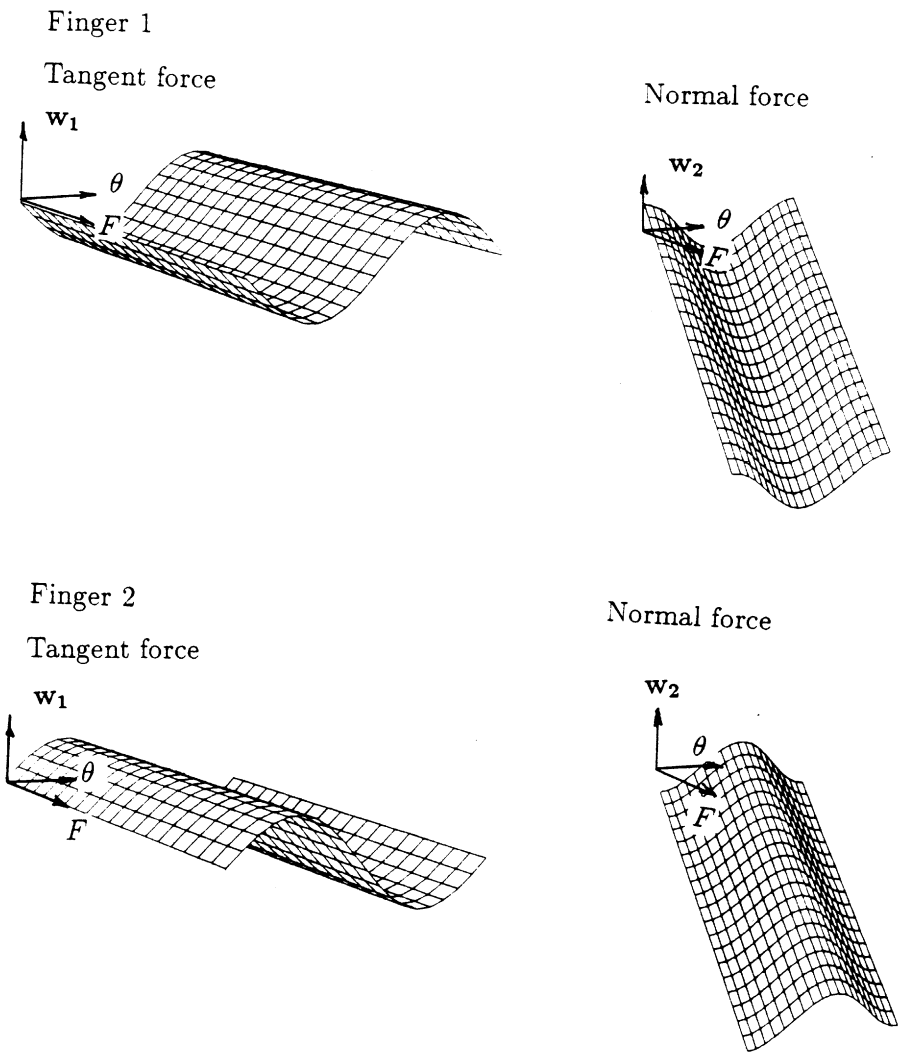


Figure 7.11: The normal force and the tangent force at each contact, relative to the contact frame, as a function of the squeezing force F and the orientation of the block θ.

7.2.5 Contact wrench / Contact type relation

The relation between the contact wrench and the contact type for two dimensions is given in chapter 6,

$|w_1| < |\mu w_2| \Rightarrow$ 1 (point contact with friction)
$|w_1| \geq |\mu w_2| \Rightarrow$ 2 (point contact without friction)
$w_2 > 0 \Rightarrow$ 3 (no contact).

Instead of using the absolute values, it the above relations, it would be more convenient to determine an analytic function. Let S_i be a slipping criteria defined by

$$S_i = \left(\frac{w_{c_1}}{\mu w_{c_2}}\right)^2. \tag{7.9}$$

Therefore the relation between the contact wrench and the contact type is

$S_i \geq 1 \Rightarrow$ 1 (point contact with friction)
$S_i < 1 \Rightarrow$ 2 (point contact without friction)
$w_{2_i} \geq 1 \Rightarrow$ 3 (no contact).

The plots for S_1 and S_2 are shown in figure 7.12, for values of grasping force ranging from $2N$ to $10N$. S_i will approach infinity in some orientations, as the grasping force is reduced to zero. Also, as might be expected, the slipping criteria will tend toward zero as the grasping force is increased to infinity, independent of the orientation of the object. Whenever the rectangle is held in a horizontal position, the slipping criteria S_1 and S_2 are both zero, since this orientation relies on mechanical rather than frictional constraints. The mapping of S_i can therefore be used as a measure of stable grasps. The function S_i will be a minimum on each contact when the ratio of frictional force to normal force is a minimum; that is, S_i will be a minimum when the grasp depends most on geometric rather than frictional constraint. The maximum values of S_1 are at $\pi/4$ and $7\pi/4$ and the maximum values S_2 are at $3\pi/4$ and $5\pi/4$. Intuitively, this is because the block's weight is supported more by the finger on the bottom, unweighting the top finger. Since the tangent force on both fingers is the same, the ratio S_i, of the top finger, is a maximum. The rectangle is still constrained, since the geometry of the surface does not allow rotation. If the surfaces were convex, however, the object might drop out. In the next section regions of constraint will be

defined as a function of object orientation and grasping force.

7.2.6 Constraint state map

A useful plot, in terms of controlled slip analysis, is a constraint state map, which defines the constraint state as a function of the controllable variables, such as squeeze force, orientation, and externally applied forces. In this particular example, the constraint states are a function of the block orientation, θ, and squeeze force m_g, figure 7.13

Suppose the rectangle is held at a 30° angle relative to the horizontal with a squeeze force of 4 N, represented as a point on the constraint state map, as shown in the figure 7.13. By varying the orientation and the grasping force, it is possible to move about the constraint state map, arbitrarily entering and exiting different constraint regions to either avoid unstable grasps or to produce the desired slipping motions. For example, suppose we want to reorient the block by sliding it between the fingers. First, determine which constraint states, if any, allowed this motion. As can be seen in the graphs in figure 7.8, there are four such constraint states: [22], [23], [32], and [33]. Second, from among these constraint states choose the one which produces the maximum constraint on the object, thus minimizing inadvertent motion. In this case, constraint state [22] is the maximally constrained state which allows the desired motion. Third, to produce this constraint, it is necessary to move from the present location in region [11] in the constraint state map to any point in region [22], as shown in figure 7.13. In the region [22] the block may slide through the fingers, but the direction of motion is determined by the orientation of the block. More generally, the direction of motion is the intersection of the permissible and preferred twist, which in this case yields a single vector.

Consider another example. Suppose two fingers grasp adjacent sides of a square, as shown in figure 7.14. Using the assumptions as in the previous case, the graphs of permissible twists are shown in figure 7.15 and a map of constraint state is produced in figure 7.16. Therefore to ensure a stable grasp, it is necessary to remain in the constraint state region [11]. If the grasping force is increased the constraint state changes from [11] to [22], which results in an unstable grasp. The subsequent slipping motion in this region is bounded as shown in figure 7.15, but its exact direction is unpredictable. Constraint regions [12] and [21] yield sets of permissible twists which contain

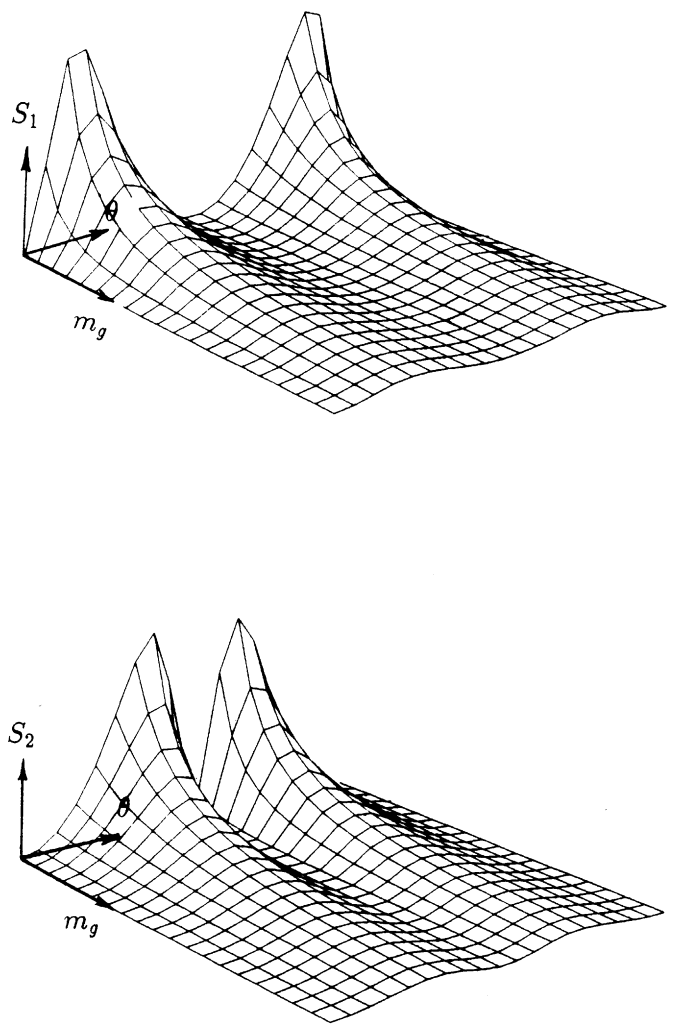


Figure 7.12: This is a plot of the slipping criteria for the fingers on the block. The plot for finger contact on the top of the block is on the top of the page and the plot for the finger on the bottom is on the bottom of the page. When the curve is a minimum, the ratio of the tangent and normal force at the contact is a minimum. In other words, there is a greater dependence on structural constraint rather than frictional constraint at minimums of the S_i plots.

only a single element; however, these sets of permissible twists do not intersect the set of preferred twists when the object is moved into these constraint regions.

Using the constraint state map, along with maps of the permissible twists, it is possible to analyze and actuate controlled slip motion for grasped objects. In the next section, a more general three dimensional example will be considered.

7.3 Three dimensional example

The three dimensional analysis is essentially the same as is two dimensions. The goal is to generate a map of the different constraint states as a function of the controllable variables, such as the external wrench, grasping force, stiffness, etc. Consider the Salisbury three fingered robot hand grasping a can, as shown in figure 7.17. The robot has two fingers on the top of the can and a third on the bottom, equally spaced between the top two. The grasp is somewhat off center from the centroid of the can.

Figure 7.18 shows the coordinate frames which will be used the analysis: the hand frame $O_h X_h Y_h Z_h$, the object frame $O_o X_o Y_o Z_o$, the grasp frame $O_g X_g Y_g Z_g$, and three contact frames $O_i X_i Y_i Z_i$. Descriptions of these coordinate frames are given in appendix A. The centroid of the can is given by a point in the hand frame $\mathbf{x_{cg}} = [-0.8, 2.7, -3.0]$ cm. The can has a diameter of 2.5 in (6.4 cm) and length of 5.0 in (12.2 cm), and mass 0.386kg. In this example, the hand holds the can at a 45° angle in a gravitational field. Therefore, the external wrench on the can, in terms of the hand frame, is

$$\mathbf{w_e} = \begin{bmatrix} 0 \\ 0.273 \\ -0.273 \\ 0.082 \\ -0.218 \\ -0.218 \end{bmatrix} N. \tag{7.10}$$

By changing the orientation of the can or by squeezing it, we can effectively control the constraint state. Suppose we only consider changes in the grasping force $[x_g, y_g, m_g]$. The constraint state map will then be a three dimension map, where the constraint states will be a function of x_g, y_g, and m_g. Figure

7.19 shows two "slices" for constant grasp force magnitude, $m_g = 0.7N$ and $m_g = 2.0N$. The shaded areas represent regions in which the can is fully constrained and will not move.

Suppose the grasp force focus is given by $x_g = 0.0$cm, $y_g = -1.2$cm, and $m_g = 0.7$N, corresponding to a point A within the [2,2,2] region of the $m_g = 0.7$N slice of the constraint state map. A reorientation of the can is possible, in this example, by allowing it to rotate between to of the fingers. Regions [3,2,2] and [4,2,2] allow a single rotation about the second and third fingers. Therefore by moving the grasp force focus from the current location in the [2,2,2] region to any point in the [3,2,2] region, the can will rotation between the fingers as desired.

As in the two dimensional case, we can choose an arbitrary slipping motion and determined the maximally constrained state which allowed the desired motion. This constraint state can then be produced by adjusting the controllable variables such as grasping force and orientation, as indicated by the constraint state map. In the next chapter, we will implement this procedure on a robot hand.

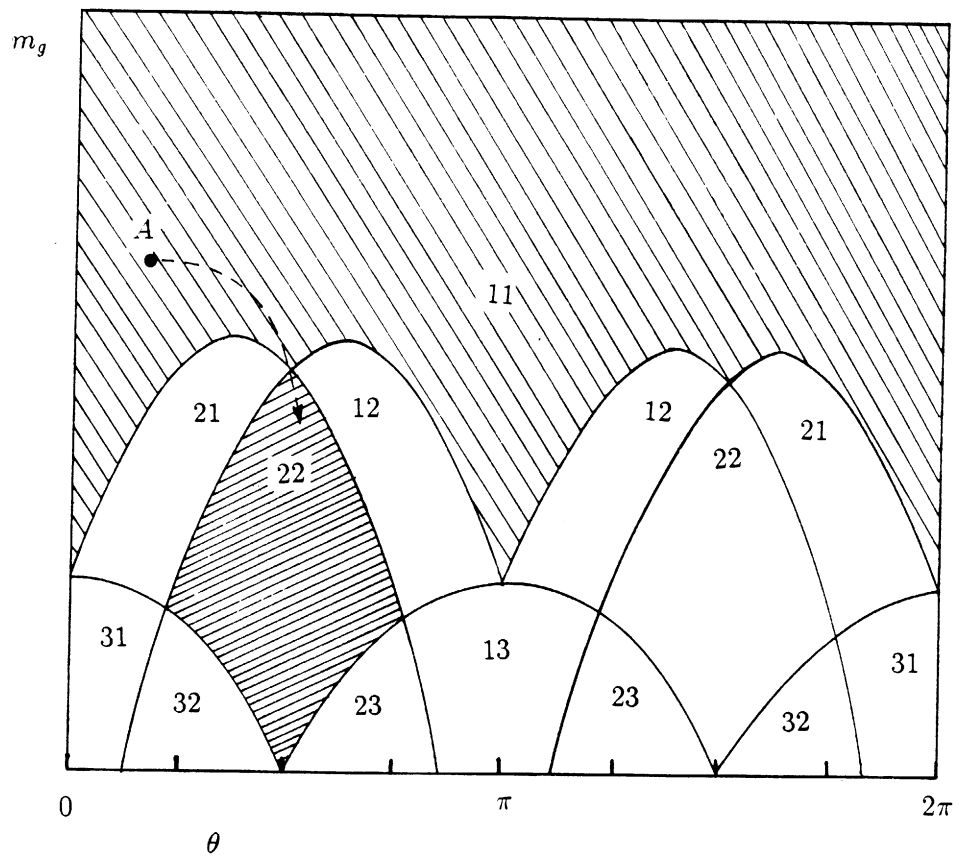


Figure 7.13: The map of the constraint states as a function of the grasping force and the object orientation can be used both to find stable grasps securing the object and unstable grasps allowing the object to slip. In this example, the current state of the grasped rectangle is shown as point A in the figure. By moving along the indicated path, the constraint of the object can be change from [11] to [22]. Once in constraint state [22], the rectangle is to slide between the fingers.

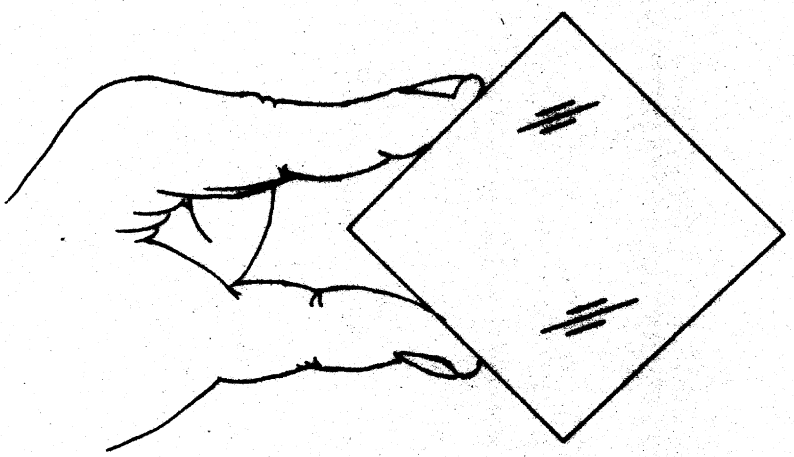


Figure 7.14: Two fingers grasp a square on adjacent sides. Intuitively we know that this type of grasp is more difficult to maintain, as shown in the subsequent figures.

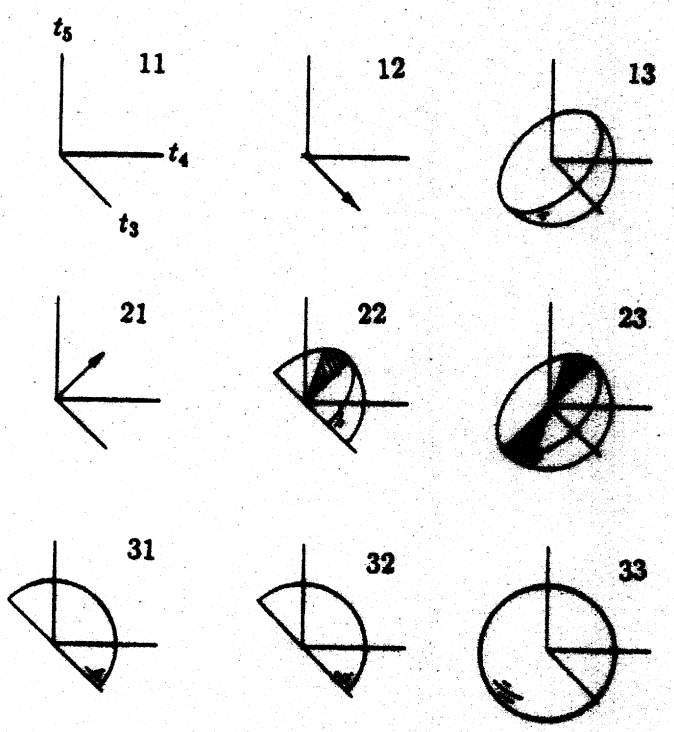


Figure 7.15: Permissible motions for square grasped on adjoining sides.

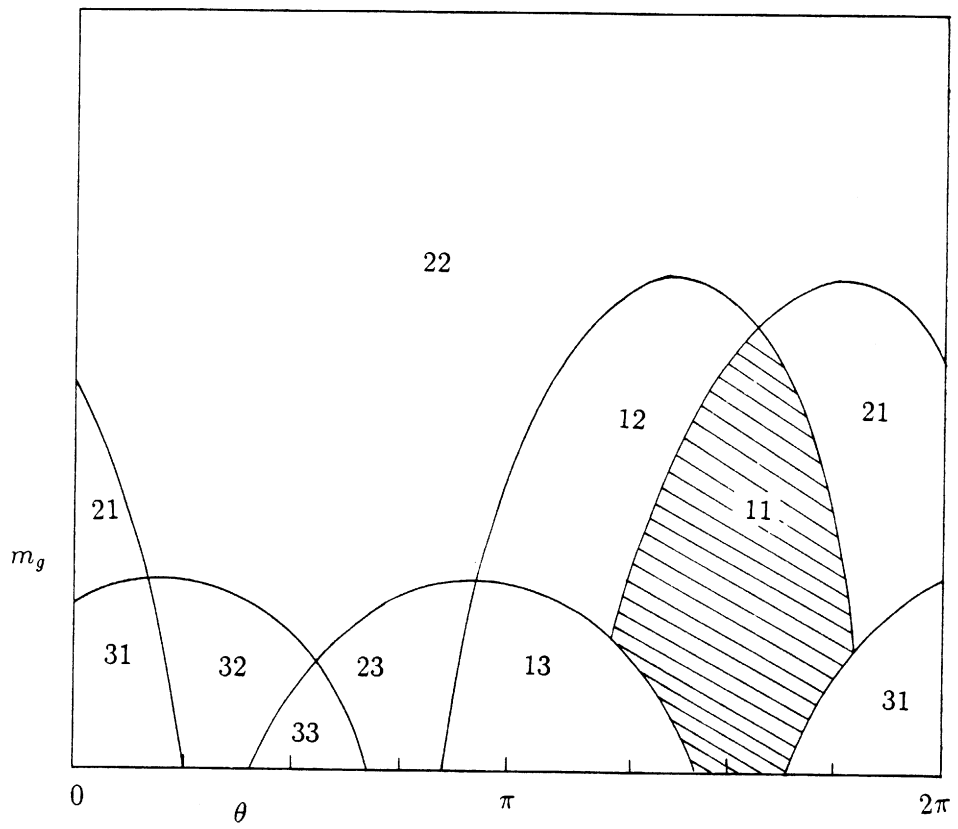


Figure 7.16: This is a map of the constraint states as a function of the grasping force and the orientation of a square held by fingertips on adjacent sides. Notice that simply increasing the grasp force does not guarantee a stable grasp. A stable grasp is only possible in the shaded [11] region. Any other region in the constraint state map will result in an unpredictable slipping motion.

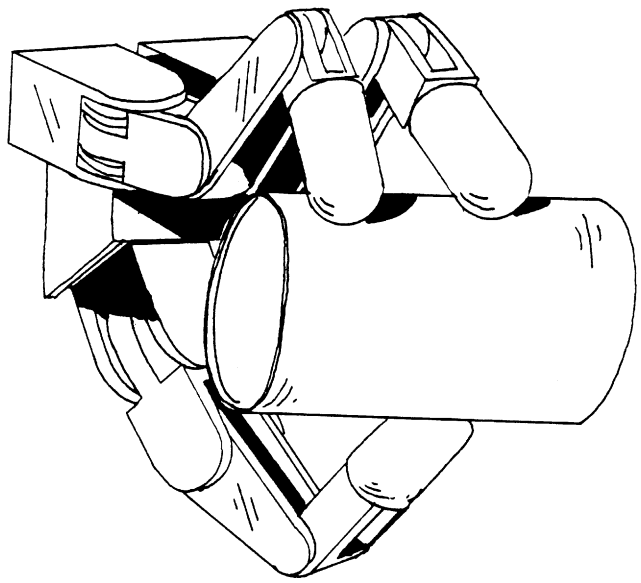


Figure 7.17: By controlling the grasping force, the can be held securely or reoriented by controlled slip.

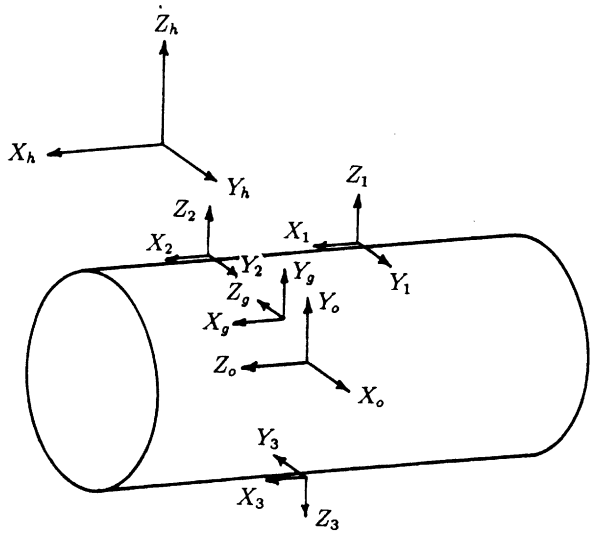


Figure 7.18: There are six coordinate frames used in this example: the hand frame, the object frame, the grasp frame, and the contact frames. The hand is rotated 45° is a gravity field.

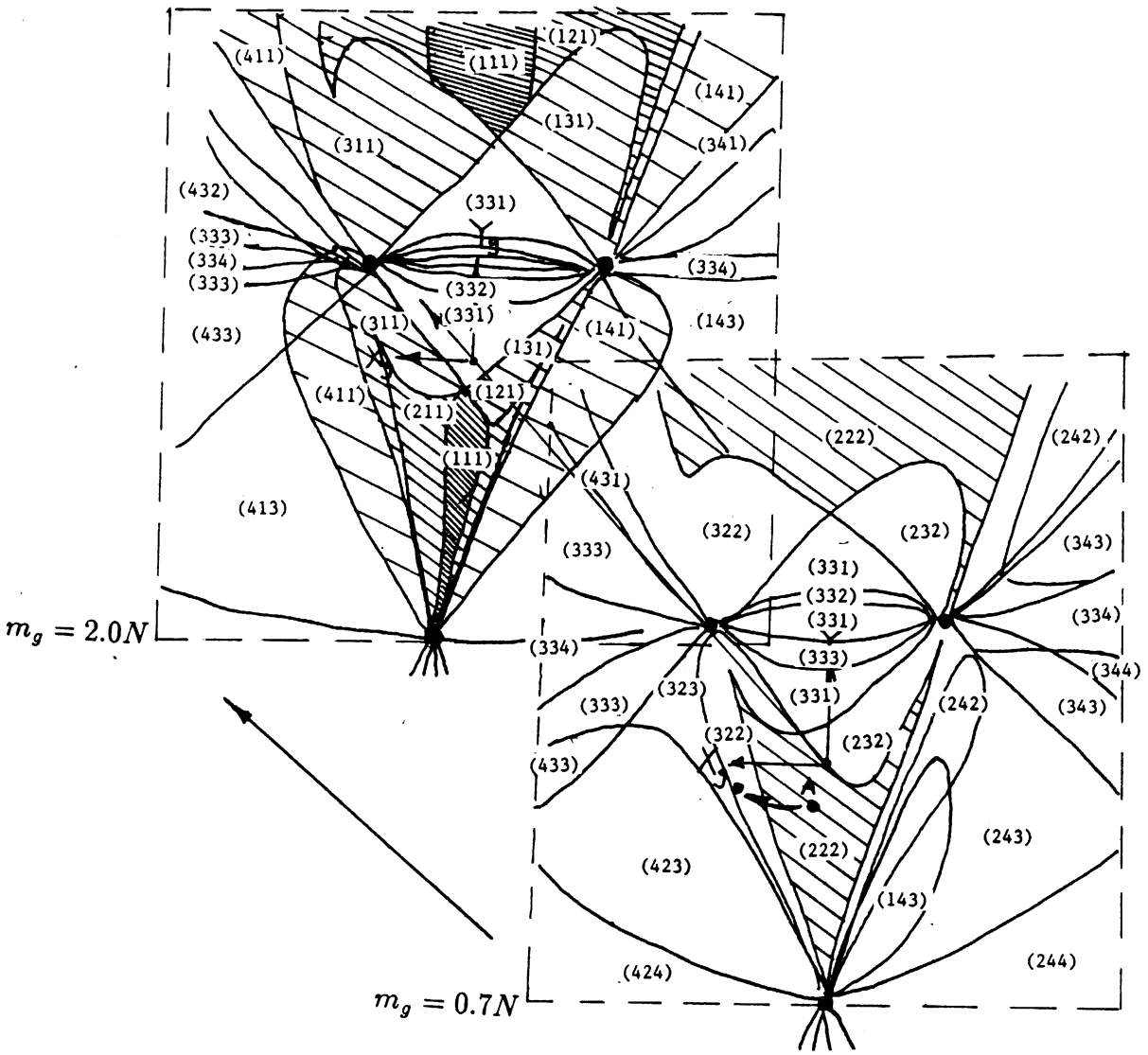


Figure 7.19: The constraint state map, shown here, is a function of the grasping force, x_g, y_g, and m_g. Two "slices" of the three dimensional constraint state map are shown, for $m_g = 0.7N$ and $m_g = 2.0N$. The current position on the constraint state map is shown as a point A in the [2,2,2] region, within the $m_g = 0.7N$ slice. By moving from the present position in the map to any point in the [3,2,2] region the can is free to rotate between the fingertips.

Chapter 8

Implementation

8.1 Introduction

The purpose of controlled slip manipulation is to enhance the dexterity of a multifingered robot manipulator, giving a robot hand a larger repertoire of manipulation strategies. A number of programs were written to automatically analyze the constraints on an object within a particular grasp. One function accepts a desired motion and returns the maximally constrained state which allows that twist. Another determines the constraint state on the object as a function of the grasping force. All these functions were written in LISP and were used in conjunction with the Salisbury robot hand. The following sections describe the hardware of the robot system and the software implementation of the slip analyses.

8.2 Description of Hardware

The Salisbury robot hand is shown in figure 8.2. The hand has three fingers and three joints in each finger, allowing a total of nine degrees of mechanical freedom. Each of the fingers is controlled by four steel tendons running to torque motors in the forearm. Linear encoders on the motors and tendon tension sensors in the fingers determine the joint position and torque allowing the hand to operate in a position, force, or stiffness control mode.

The general control structure is shown in figure 8.2 Two modified Puma robot controllers accomplish the lower level servo control, while higher level

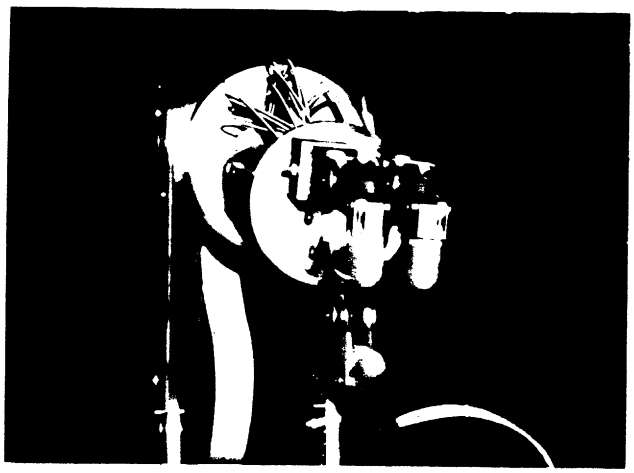


Figure 8.1: The Salisbury robot hand has three fingers and three joints in each finger, yielding a total of nine degrees of mechanical freedom. The hand has linear encoders on the motors to determine joint position and tension sensors on the tendons to resolve the joint torque.

fingertip and joint trajectories are directed from the VAX 11/750. At a still higher level, the Symbolics 3600 controls the coordinated hand functions.

8.3 Description of software

A program, GRASP, was written to analyze the constraints of an object grasped by the three fingered robot hand. The program was designed to be interactive, allowing the user to select specific grasp analysis functions, control the location and orientation of the grasp, or to choose a particular squeeze force. The functions can be selected from a menu, and the results of the analyses are displayed on a graphics window, illustrated in figure 8.3. The screen is divided into three regions. The large pane to the left is a graphics screen showing the hand, finger, object, and contact locations. The smaller screen on the top right is LISP Listener, which accepts and interprets LISP commands. Finally, on the bottom right, a small message monitor screen prints messages which are sent and received from the VAX. The floating

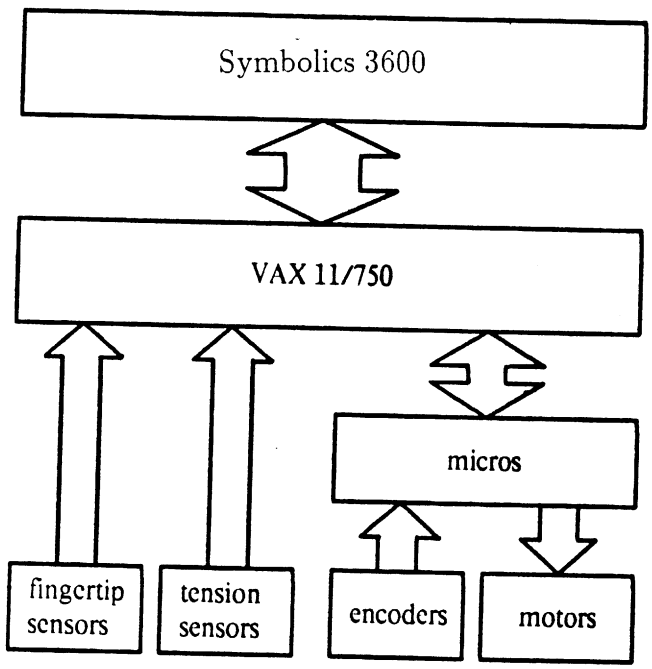


Figure 8.2: The robot hand three basic levels of control. At the lowest level PUMA robot controllers implements the high speed servo control, at the intermediate level a VAX 11/750 controls the joint trajectory and finger stiffness, and Finally, at the highest level a Symbolics 3600 LISP Machine manages the coordinated hand motion.

menu allows the user to select many graphic and actuation options. Each of the options will be described briefly in the following sections. Functions which employed the controlled slip analyses will be described in more detail.

8.3.1 GRASP functions

Standard Graphics Options:

CREATE GRASP SCREEN. Creates the three paneled display screen

CLEARSCREEN. Clears the graphics screen

DRAW COORDINATE SYSTEM. Draws the six coordinate systems on the graphics screen. These are the hand frame, object frame, grasp frame, and the three contact frames. These different coordinate systems are fully described in appendix A.

CHANGE GRAPHICS VARIABLES. Allows the user to change the scaling and the graphics screen origin.

Basic Actuation functions:

GO HOME. Reinitializes the VAX and Lisp Machine trajectory lists and goes to the home position

REINIT VAX TRAJECTORY. Reinitializes the trajectory list on the VAX

REINIT OOLAH TRAJECTORY. Reinitializes the trajectory list on the LISP machine

BASIC MOVES. Allows the users to specify coordinated finger motion. The options allow the user to translate or rotate the grasp frame in any desired cartesian direction.

MOVE FINGER. Moves a selected finger by a specified displacement

Controlled slip analyses:

CHANGE GLOBAL VARIABLES. Allows any of the global variables to be changed.

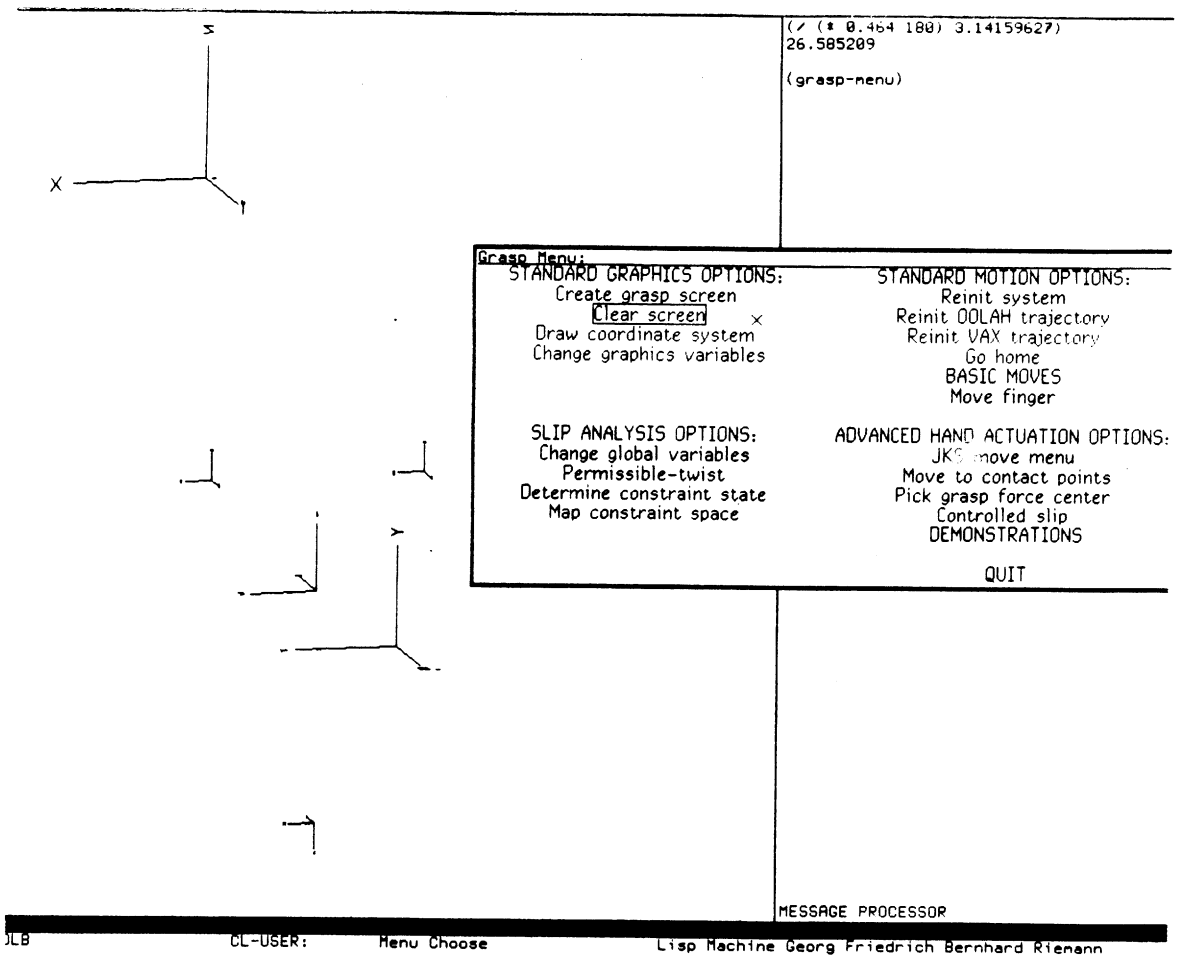


Figure 8.3: The GRASP program was written as an interactive interface between the user and the Salisbury robot hand system. The program allows the user to select a number of functions, including: slip analysis functions, functions to relocate and reorient the grasp, and a function which changes the grasp force on the object.

PERMISSIBLE TWIST. Specify a desired twist and this function will return the maximally constrained state of the object that allows that twist. This is accomplished simply by looking at each contact individually and determining which contact type will allow the specified twist. The contact types are investigated in decreasing order of constraint. That is, a soft finger contact, a point contact with friction, a point contact without friction, and no contact were considered sequentially. When a particular contact type allows the specified motion, this contact type is stored. The contact types are then compiled into a list, thus yielding the constraint state which allows the desired motion. This constraint state is also maximal, since any state of greater constraint would not allow the twist to occur at one or more of the contacts.

DETERMINE CONSTRAINT STATE. This option allows the user to specify the grasping force, $[x_g, y_g, m_g]$, and returns the constraint state of the object for a specific orientation and a given grasp. The program uses the stiffness model described in chapter 5 together with the contact type/contact wrench relation given in chapter 6.

MAP CONSTRAINT SPACE. The user specifies the magnitude of the grasping force, m_g, and the program produces a map of the constraint states, by varying the grasp force focus, $[x_g, y_g]$. The program plots the regions of different constraint and prints the value of the constraint state in each region. For example, figure 8.4 shows a slice of the constraint state map for $m_g = 7.0N$ for the same situation described in chapter 7. Notice that the coordinate systems of the hand, the grasp frame, and contact frames are drawn in the display, since the grasp force focus corresponds to an actual point in space. From this point the user can select the grasping force $[x_g, y_g, m_g]$ by using the option PICK GRASP FORCE CENTER.

Advanced Hand Actuation Options:

MOVE TO CONTACT POINTS. Moves the fingertips to specified contact points on the object. This function takes into account the radius of the fingertip when grasping an object

PICK GRASP FORCE CENTER. The user specifies a grasp force magnitude m_g. Then the program draws the grasp frame coordinate system

on the screen, along with the hand, and contact coordinate systems. The user may then move to any point in the graphics screen and select a grasp force focus, $[x_g, y_g]$, on the grasp plane. The robot will then displace the fingers by an amount and a direction proportional to the specified grasp force, $[x_g, y_g, m_g]$. Given the compliance of the fingers, the resulting force will be proportional to the desired internal grasping force at each of the fingers.

DEMONSTRATIONS. Three demonstration routines were written to illustrated how controlled slipping techniques might be used on an actual robot hand. The three routines show respectively how gravity, free fingers, or other objects may be used to reorient and reposition an object within a grasp.

The first demonstration is the three dimensional example of chapter 7. The robot hand holds a full can of liquid, as shown in figure 8.5. The mass, location, orientation, and surface properties of the can are the same as those given in chapter 7. A constraint state map shown in figure 8.6, for a constant grasp force magnitude of 0.7N. Initially, a grasp force, $[x_g, y_g, m_g] = [0.0\text{cm.}, -1.2\text{cm.}, 0.7N]$ is selected so that the can is in the [2,2,2], a fully constrained region of the constraint state map. The magnitude of the grasp force remains the same, but the focus is moved out of the [2,2,2] region into the adjacent [3,2,2] region. The result, shown in figure 8.7, is that the can spins between the two fingers, under the influence of gravity, into a new orientation. The grasp force focus is then moved back into the [2,2,2] region and the grasp is again secured; however, the can has rotated 90°.

In the second demonstration, the robot hand again holds a can in the same position and orientation. One of the fingers is then removed from the top of the can. The situation is essentially a two fingered grasp or a three fingered grasp with the first finger in a type 4 contact. In any case, the magnitude of the squeezing force between the two grasping fingers can be controlled. In this case, the magnitude of the squeezing force is relaxed, so that when the free finger exerts a force on the front of the can, the constraint moves into the [2,4,2] constraint region and spins between the two fingers. The finger continues to spin the can, until it has been rotated 180° into a new orientation, figure 8.8. The

free finger is then replaced on the can, securing the grasp. The can has been completely reoriented in the grasp.

In the third routine, the hand grasps a box and pushes it against a table, so that the box slides through the fingers. The box is then lifted from the table and the free finger spins it 180°. The hand again grasps the box and again forces it against the table, repeating the same procedure over and over again, as shown in figure 8.9.

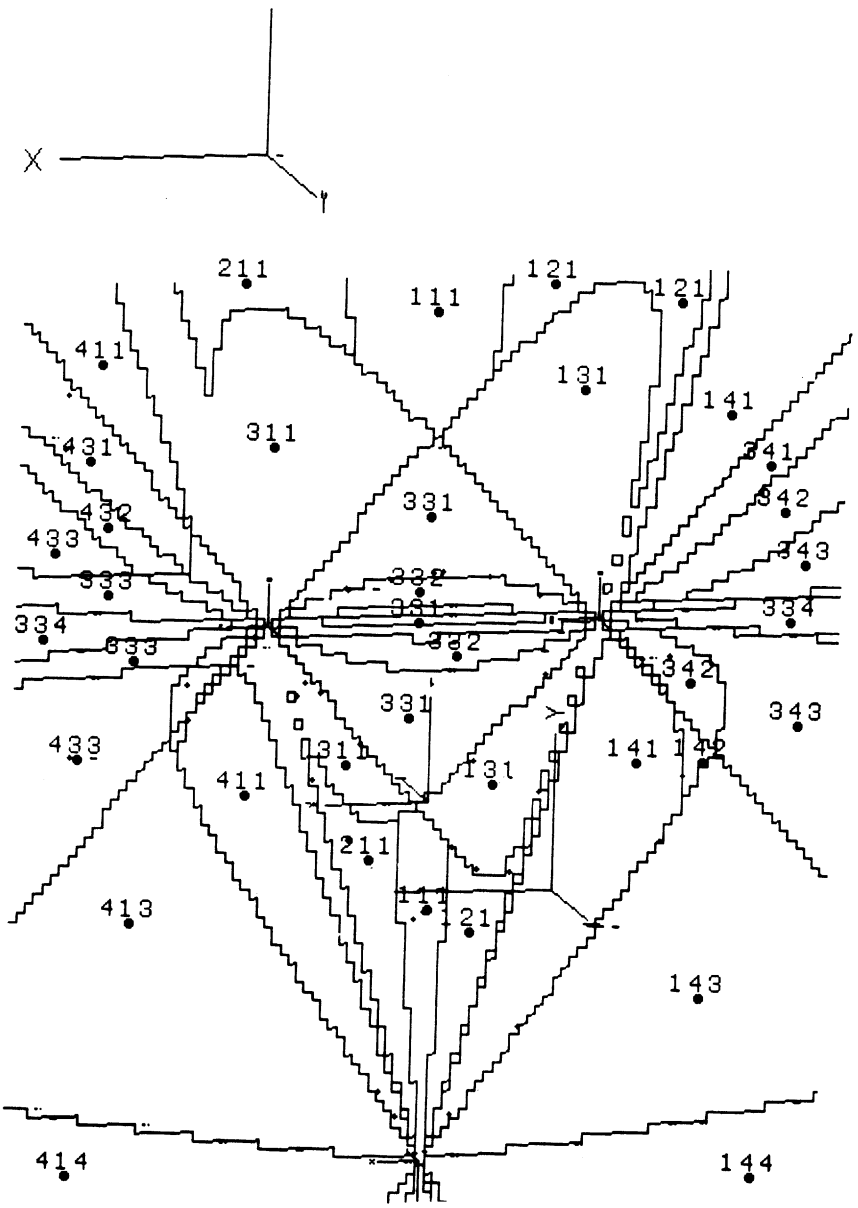


Figure 8.4: Constraint state map produced by the computer plots the constraint state as a function grasp force focus x_g y_g for constant grasp force magnitude, $m_g = 2.0N$. The map show here is for the identical situation outlined in chapter 7.

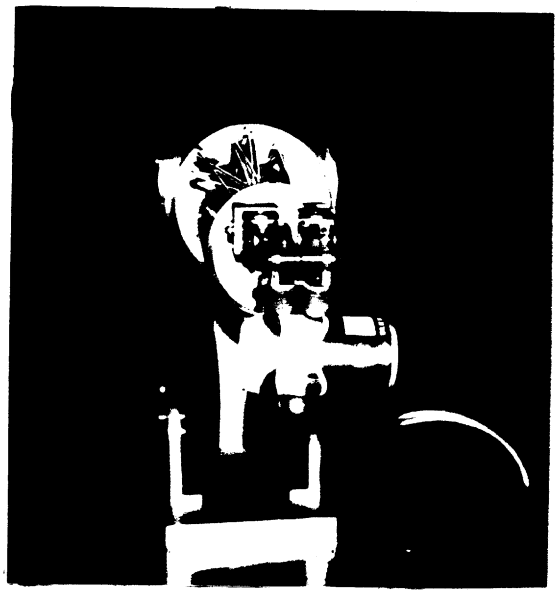


Figure 8.5: The Salisbury robot hand holds a can, as in the example in the previous chapter. By adjusting the squeezing force on the can, the robot can either secure the can in the grasp or change the orientation by allowing it to slip.

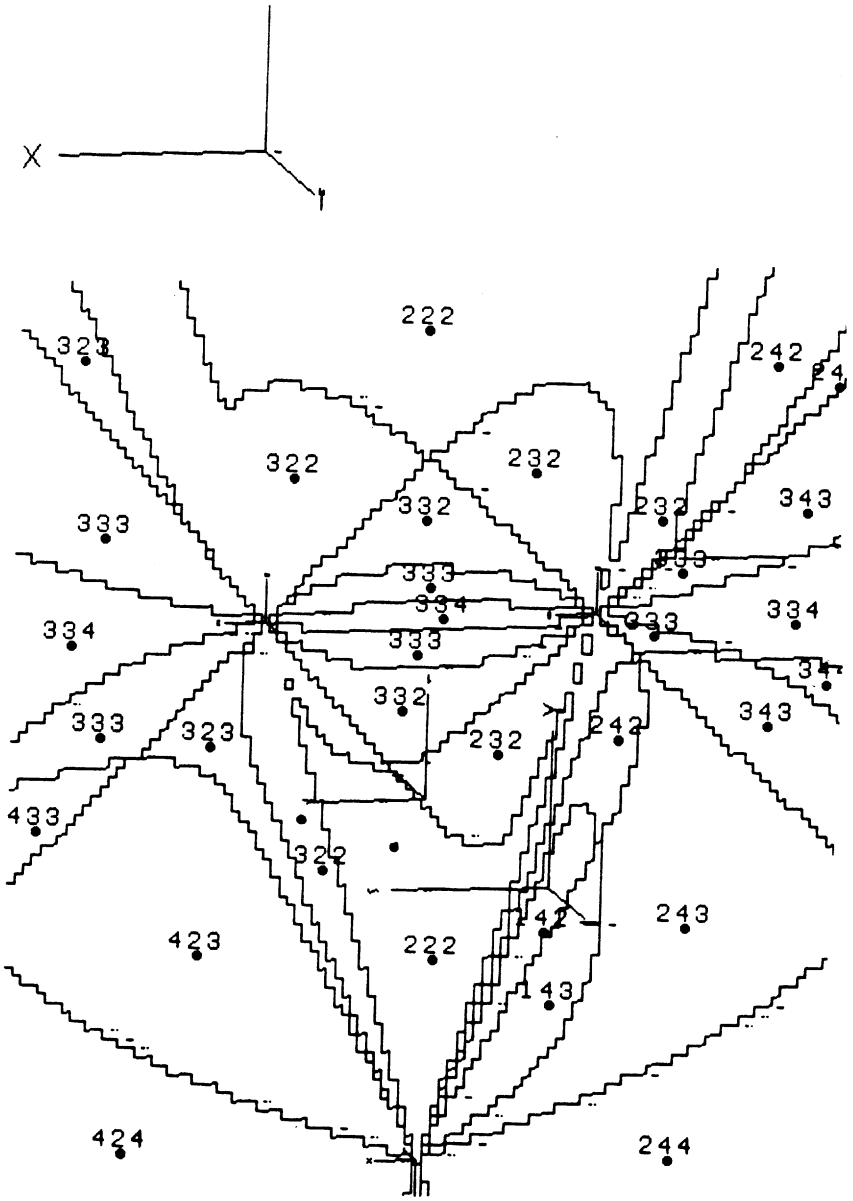


Figure 8.6: This is a map shows the constraint state regions as a function of the grasping force focus x_g y_g for constant grasp force magnitude $m_g = 0.7N$. The current position in the map is shown as a point in the grasp plane. The current constraint state is [2,2,2], a fully constrained region in the map. However, by moving to any point in the adjacent [3,2,2] region, the can will spin between the fingers.

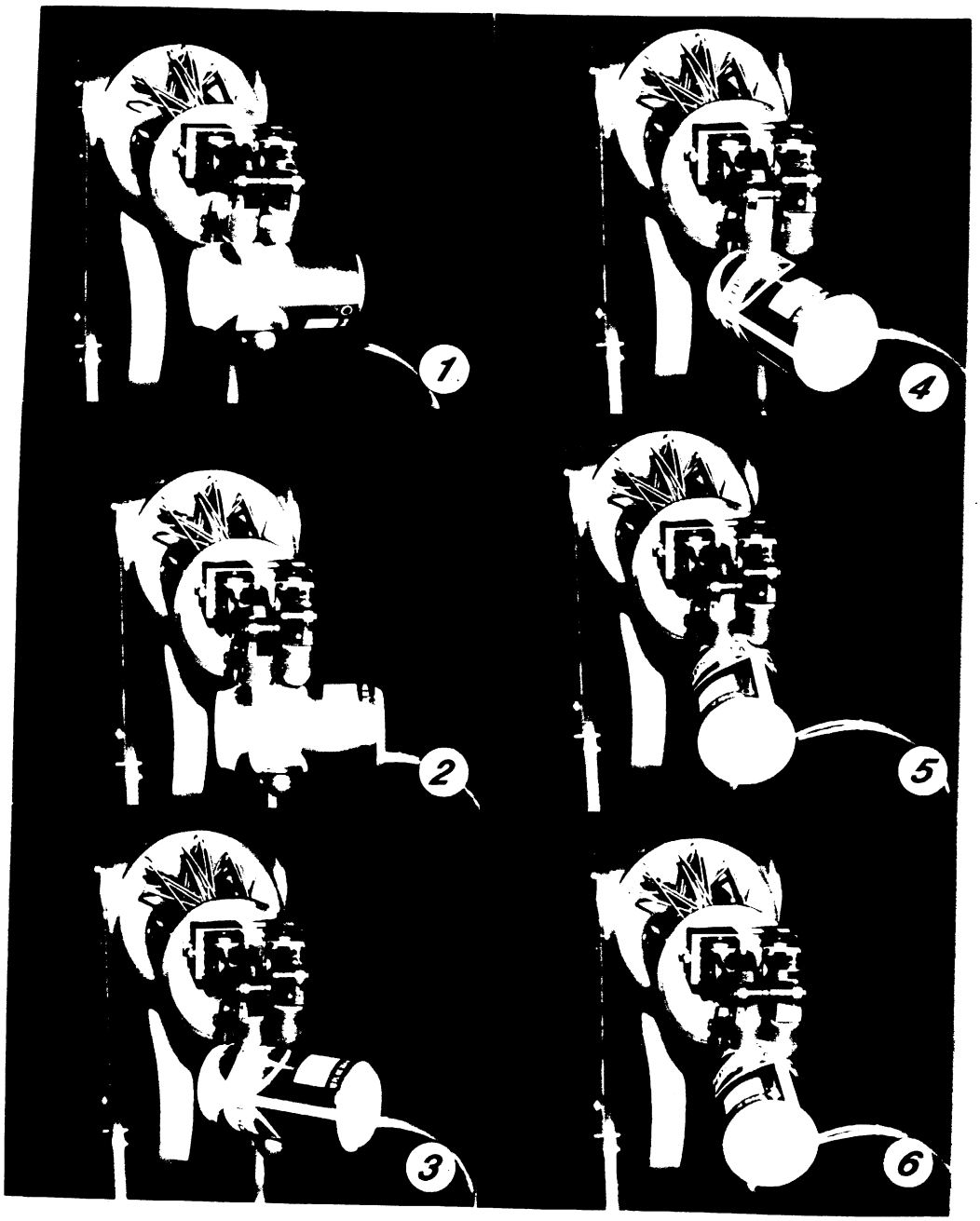


Figure 8.7: The Salisbury robot hand grasps a soda can. The magnitude of the grasping force is 0.7 N and the grasp force focus $x_g = 0.0$cm $y_g = -0.8$cm. so that the can is secure in constraint region [2,2,2]. However, by moving the grasp force focus to $x_g = 0.8$cm $y_g = -0.8$cm, the can will spin between two fingers. The grasp force focus is then moved back into its original position and the grasp is again secured.

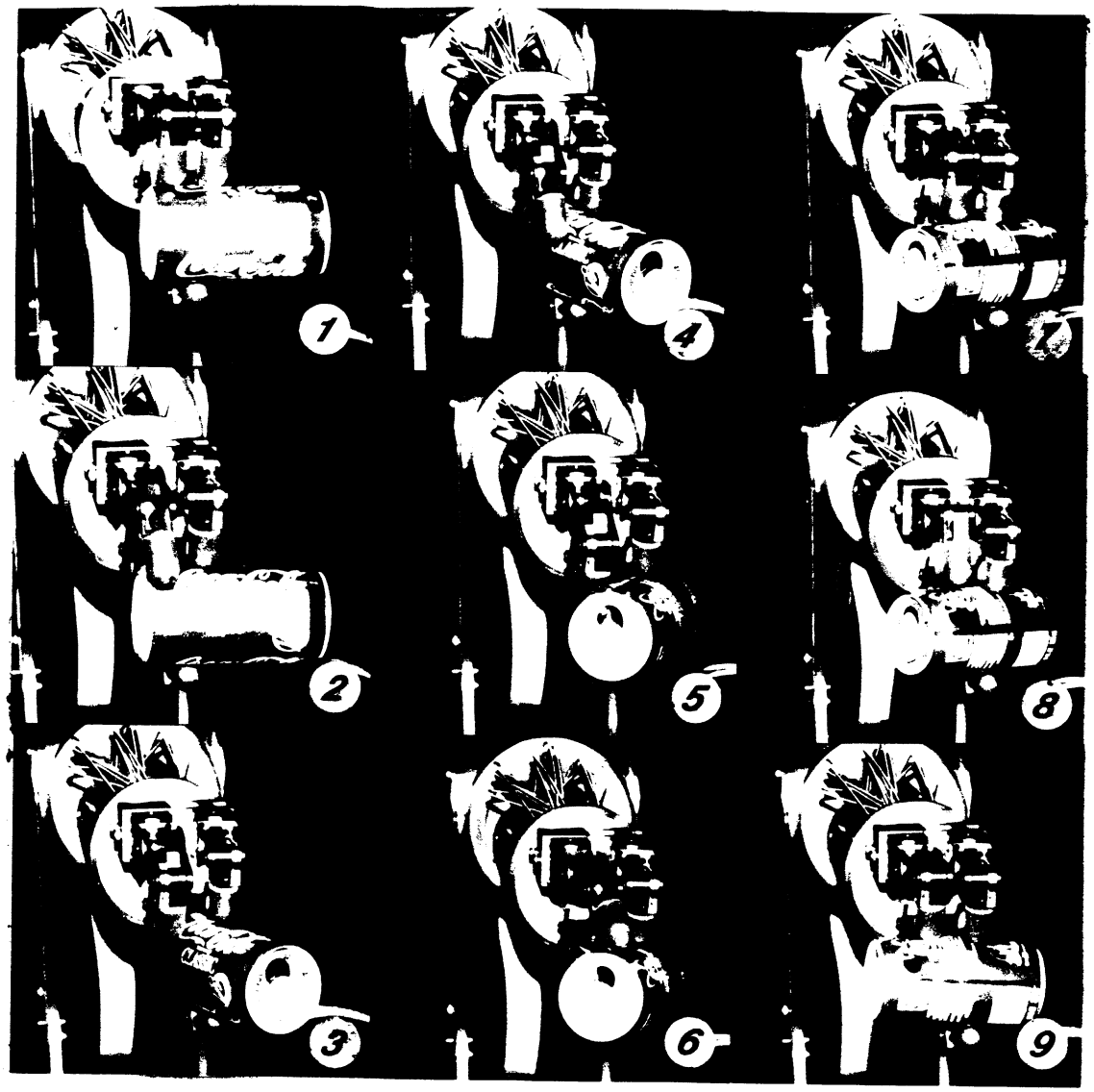


Figure 8.8: A can is initially held securely in the robot hand. The grasp force focus shifts from its current stable position to a point on the line intersecting the contact points made by the first and third fingers. By removing the second finger, a constraint state [1,4,1] is produced and by using the free finger to exert a force, the can rotates in the grasp. In this demonstration the finger continues to spin the can until it has rotated it 180°. Finally, the free finger is returned to its origin position and the grasp force focus is moved to the center of the grasp.

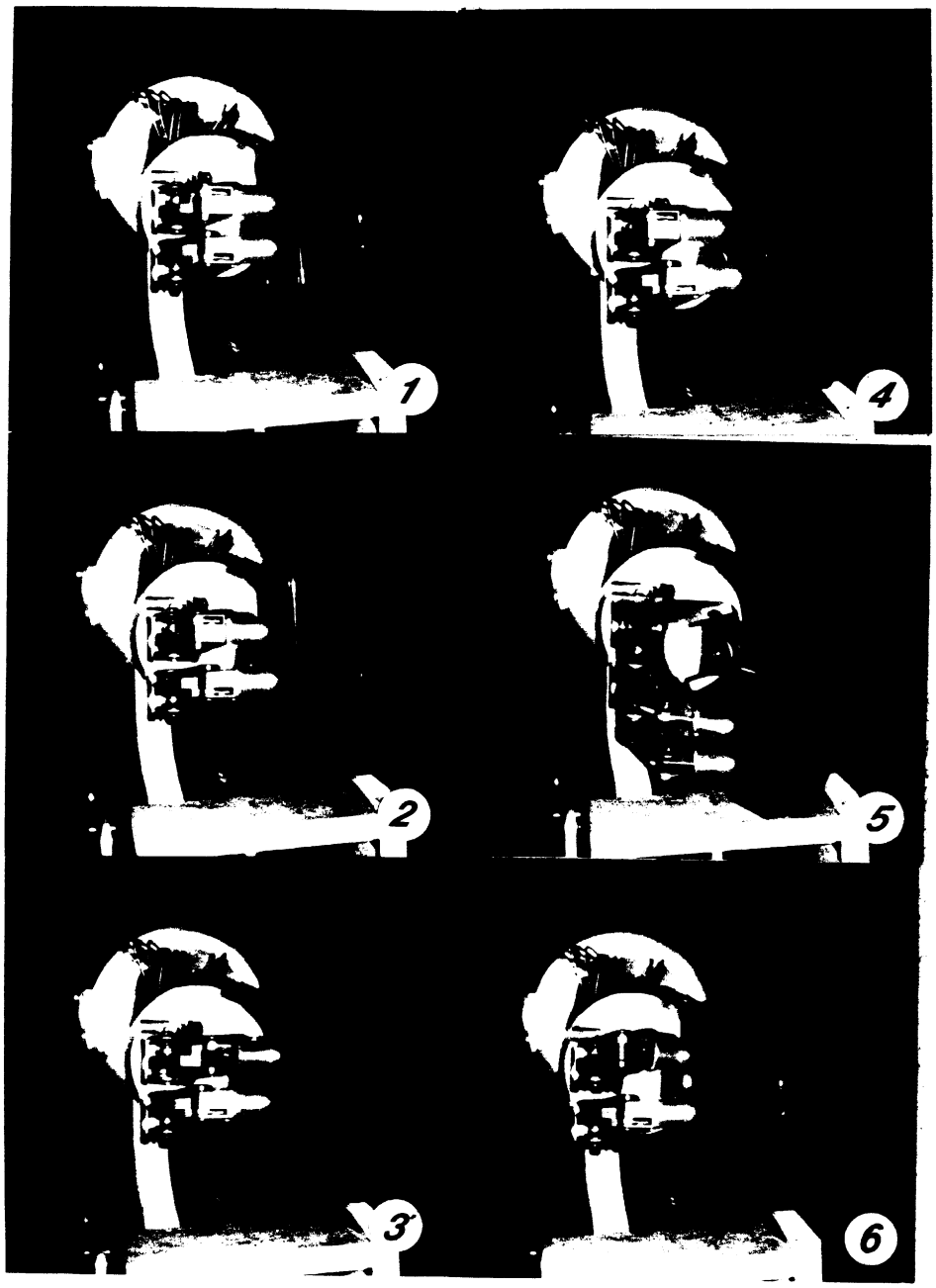


Figure 8.9: Slipping motions can be linked to form a sequence of motions to reorient a part within the hand. In this case, the hand pushes a box against the table, forcing the box through the grasp. The hand then lifts the box, reduces the grasping force magnitude, shifts grasp force focus between two of the fingers, and uses the free finger to spin the box 180°. The hand again grasps the box and forces it against the table and then repeats the entire process.

Chapter 9

Extensions and further research

9.1 Introduction

The goal of this thesis, is to gain a basic understanding of a highly dexterous type of manipulation that we take for granted, namely controlled slip manipulation. An analysis was performed on grasped objects to determine which ways they can move in the grasp. The set of possible motions was found as a function of the constraint state of the object, that is, the number, location, orientation, and the type of each contact. The constraint state is a function of a number of controllable variables, such as the grasp force, orientation, and stiffness of the fingers. By changing the grasp force, for example, the constraint state of the object can be varied to allow slipping within the grasp.

This analysis is only a beginning and many important issues have yet to be addressed. This chapter will outline some extensions to the general problem of controlled slip manipulation. It is hoped that these extensions will allow this type of manipulation to be more useful and predictable, thus enhancing the dexterity of robot hands.

9.2 Determining permissible motion

When a hand grasps an object, there are only so many ways that it can move in a grasp. For some constraint state, the space of permissible motions of the object is quite large, which makes the control of the motion difficult. However, there are some constraint states which yield a very limited range of permissible motions, so that the motion of the object can be determined as a function of the geometric constraints alone. With the current algorithm, a desired twist is specified and the constraint state which allows this twist is determined. Although the algorithm determines the constraint state which provides maximum constraint while still permitting the desired motion, it does not reason as to which motions would be more advantageous for a given grasps or particular task. What would be more useful in terms of controlled slip manipulation, would be to determine, perhaps, using a set of heuristics, a small set of permissible motions, which may be easily achieved through slipping motion. To accomplish this, using the current analysis, would require searching a six dimensional space to determine the extent to which the permissible twist is bounded. This may in fact not be too difficult, but a set of rules may quickly and efficiently produce a set of slipping motions the robot can accomplish. This approach is being investigated and looks promising for practical implementation on a robot hand.

9.3 Determining constraint state

In tandem with the problem of determining set of slipping motions which can be easily implemented in a particular grasp, is the problem of finding the desired constraint. With the present analysis, a constraint state map is generated as a function of controllable variables on the object. To find a desire constraint state, the entire map is generated or the space is searched until the values of grasp force are found which produce the desired constraint state. Again, a set of heuristic rules may allow the robot to quickly find a particular constraint state within a large space, so that the slipping manipulation may be achieved quickly.

These rules may be quite simple, since there are some constraint states which are independent of some of the controllable variables. For example, some grasp foci always produce a particular constraint state.

9.4 Global motion

The current analysis only determines the set of permissible twists for small finite motions. In general, to reorient an object within a hand, large motions need to be considered. In order to study global motions, many other issues must be addressed. For example, the size, shape, location, and orientation of the grasp object, the robot hand, and the surrounding environment must be known. By modeling the robot, the object, and the environment as polyhedral solids the global motion of the object may be determined. This problem is not as difficult as the general path planning problem, since the path is given as the particular twist, only magnitude of the twist needs to be determined. Therefore for a small set of desired twists, their magnitudes can be found and with this information controlled slip trajectories may be planned and implemented on a robot hand.

9.5 Sensory feedback

The discussion so far deals with slipping motion based on a model of the robot, object, and environment. The robot has no feedback on actual object position or contact force. In order to performed controlled slip manipulations accurately, the robot must know the position of the object relative to the grasp, as well as, the location, orientation, and type of contact which exists at each interface between the manipulator and the object. The contact information may be determined using a force and tactile sensing surfaces on the fingertips of the robot hand. A contact resolving sensor was built for the Salisbury robot hand, which determines the location of a contact on the surface, as well as, the normal and tangent force at the interface. Appendix C describes this sensor and issues involved in its design. A sensor like this is important

in controlled slip manipulation, because it can determine the location of the contacts and, using a model of the fingertips, determine the contact type. However, this type of sensor may be used directly to determine contact type. Since slipping usually coincides with high frequency vibrational noise, slipping may be determined by looking at the high frequency components of the force signal. Given a high frequency force signal, together with the mean tangent and normal force readings, it may be possible to determine whether the slipping was translational or rotational. That is, given a high frequency vibration which suggests slip, then

$$\begin{aligned} |w_{c_x}/\mu w_{c_y}| &\approx 1 \quad \Rightarrow \quad C_3 \\ |w_{c_x}/\mu w_{c_y}| &< 1 \quad \Rightarrow \quad C_2 \end{aligned}$$

where C_2 is a point contact with friction, the object is rotating about the contact point, and C_3 is a point contact without friction, the object is translating at the contact.

Determining the location of the object in the grasp is somewhat more difficult. For some class of objects and for some set of contacts, the tactile data alone may be enough to uniquely determine the position of the object. However, tactile sensing alone would not, for example, resolve the location of a cylinder held vertically on its sides by the fingertips of a hand. The contacts and contact forces in this situation are independent of the vertical position of the cylinder. Some other means than tactile sensing would be necessary to determine position. Vision would be a possible solution, except that resolving object location at the speeds necessary for controlling slipping motion is currently an impossible task. However, vision would seem to be the eventual solution, since in order to manipulate objects intelligently, a robot would have to know not only the position of objects within the grasp, but also the position of objects in the environment, since they may interact with the motion of the grasped object. A robot could both avoid external objects and use them to exert controlled forces on the object within the grasp.

9.6 Integrating manipulation techniques

The purpose of a robot is to interact with the environment, to sense its nature and to affect changes. To this end, higher lever plans are needed to determine global actions, as well as, lower level strategies which integrate manipulation techniques, including controlled slip. Methods must be developed to select manipulation strategies to accomplish the lower level goals as set by the higher level plans. For example, to invert a cylinder within a grasp, a robot may put the object down, reposition the hand, and regrasp the object, effectively inverting the can in the grasp. Or, it may, using control slip, spin the cylinder between two of its fingers, and thereby accomplishing the same feat. In spinning the cylinder, the robot could use gravity, acceleration, other objects, or other fingers to reposition the object. To achieve the desired goal state, the robot may have to perform a number of slipping manipulations in a row. This suggests some sort of lower level slip planning based on analysis or experience.

9.7 Conclusion

This thesis presents a basic analysis of controlled slip manipulation and suggests some methods of implementing slip manipulation on robot hands. Given a particular grasp, the set of small finite motions the object can undergo within the grasp was determined. The set of permissible motions within a grasp was found to depend on the constraint state, that is the location and the types of contacts that exist between the robot and the object. The constraint state in turn depends a number of controllable variables, such as grasping force, orientation, finger stiffness, and other variables. By controlling the constraint state and external forces which act on the grasped object, controlled slipping motions can be achieved. Before this type of manipulation can be practically implemented in robot system, more efficient ways must be found to analyze grasps and actuate motions. This thesis does, however, demonstrate that manipulating objects by allowing them to slip and twist at the fingertips is possible for robots as well as for humans.

Bibliography

[Ball] Ball, R.S. "A Treatise on the Theory of Screws" Cambridge University, Press, London, 1900.

[Chiu] Chui, S.L.. "Generating Compliant Motion of Objects with an Articulated Hand" S.M. Thesis, Dept. of Mechanical Engineering, Massachusetts Institute of Technology, June 1985.

[Hunt] Hunt, K.H., *Kinematic Geometry of Mechanisms*, Oxford University Press, 1978.

[Jacobsen] Jacobsen, S.C., et al. "Development of the Utah Artificial Arm," *IEEE Transactions on Biomedical Engineering*, Vol. BME-29, No. 4, Apr. 1982.

[Lipson] Lipson, C., and Juvinal, R.C., *Handbook of Stress and Strength: Design and Material Applications* , New York, 1963, pp. 45-57.

[Lozano-Perez] Lozano-Perez, T., "Automatic Planning of Manipulator Transfer Movements," MIT AI Lab, Memo AIM 606.

[Mason] Mason, M.T., and Salisbury, J.K. *Robot Hands and the Mechanics of Manipulation*, MIT Press, Cambridge, 1985

[Nguyuan] Nguyuan, V., "The Synthesis of Stable Force-Closure Grasps" A.I. Memo 905, MIT Artificial Intelligence Lab., May 1986.

[Ohwovoriole] Ohwovoriole, M.S. "An Extension of Screw Theory and Its Application to the Automation of Industrial

Assemblies" Ph.D. Thesis, Dept. of Mechanical Engineering, Stanford University, April 1980.

[Okada] Okada, T., "Computer Control of Multi-Jointed Finger," 6th International Joint Conference on Artificial Intelligence, Tokyo, Japan, 1979.

[Salisbury] Salisbury, J.K., "Kinematic and Force Analysis of Articulated Hands" Ph.D. Thesis, Dept. of Mechanical Engineering, Stanford University, May 1982.

[Tournassoud] Tournassoud, P., "Regrasping" *IEEE* International Conference on Robotics and Automation, March, 1987.

Appendix A

Coordinate frames

A.1 Introduction

Objects, fingers, fingertips, and contact points can all described relative to different coordinate frames. There are a number of coordinate frames used in this thesis and any point in space may be described with respect to any or all the coordinate frames. There are a total of nineteen reference frames. These are:

$OXYZ$	Arbitrary reference frame
$O_hX_hY_hZ_h$	Hand frame
$O_{f_{ij}}X_{f_{ij}}Y_{f_{ij}}Z_{f_{ij}}$	Finger frames
$O_{t_i}X_{t_i}Y_{t_i}Z_{t_i}$	Fingertip frames
$O_{c_i}X_{c_i}Y_{c_i}Z_{c_i}$	Contact frames
$O_gX_gY_gZ_g$	Grasp frame
$O_oX_oY_oZ_o$	Object frame

Descriptions of the individual reference frames are given in the following sections.

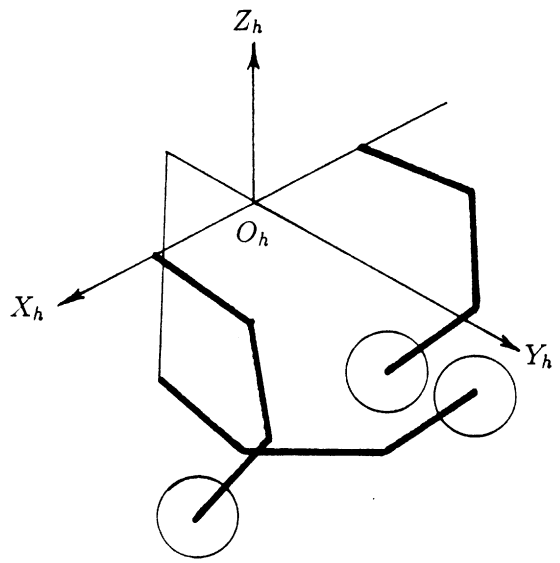


Figure A.1: Hand frame

A.2 Hand frame

The origin of the hand frame is centered between the joints of the first and second fingers of the Salisbury hand. The x-axis is directed out from the origin, through the center of the first joint of the second finger, as shown in figure A.1. The z-axis is vertical, normal to the plane of motion of the first phalange of the first finger. The y-axis is normal to both the x and y axes, directed away from the wrist, lying in the plane of the motion of the first phalange of the first finger.

A.3 Finger frames

The coordinate frames for the finger phalanges are defined the same way each finger. These coordinate frames are designated $O_{f_{i,j}} X_{f_{i,j}} Y_{f_{i,j}} Z_{f_{i,j}}$, where i represents the ith finger and j the jth phalange. The finger frames are shown in figure A.2. The origin of the coordinate frames for the phalanges are all at the center of the joints and the y-axes of every frame is aligned with the central axis of the phalange. The x and z

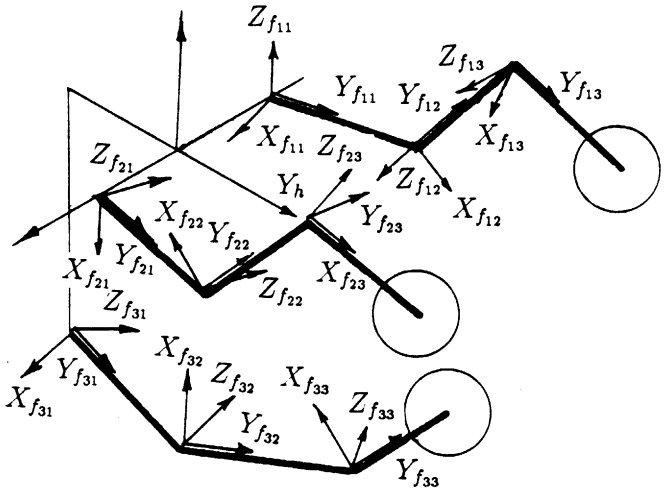


Figure A.2: Finger frames

axes, however, are defined differently. For the first phalange, the z-axis is the axis of joint rotation and the x-axis is normal to the y and z axes in the direction of the x-axis of the hand frame. For the second and third finger phalange frames, the x-axis is the axis of joint rotation and the z-axis is vertical, parallel to the z-axis of the hand frame when the joints are in the zero position.

The length of the phalanges the fingers of the Salisbury robot hand are all the same. Figure A.3 shows a schematic of the hand illustrating the length of the phalanges and the placement of the fingers

A.4 Fingertip frame

The fingertip frames $O_{t_i}X_{t_i}Y_{t_i}Z_{t_i}$ are defined the same way for each fingertip, see figure A.4. The origin of the fingertip frame is the center of the spherical portion of the fingertip. The z-axis is the central axis of the fingertip. The x-axis is parallel to the x-axis of the distal phalange coordinate frame.

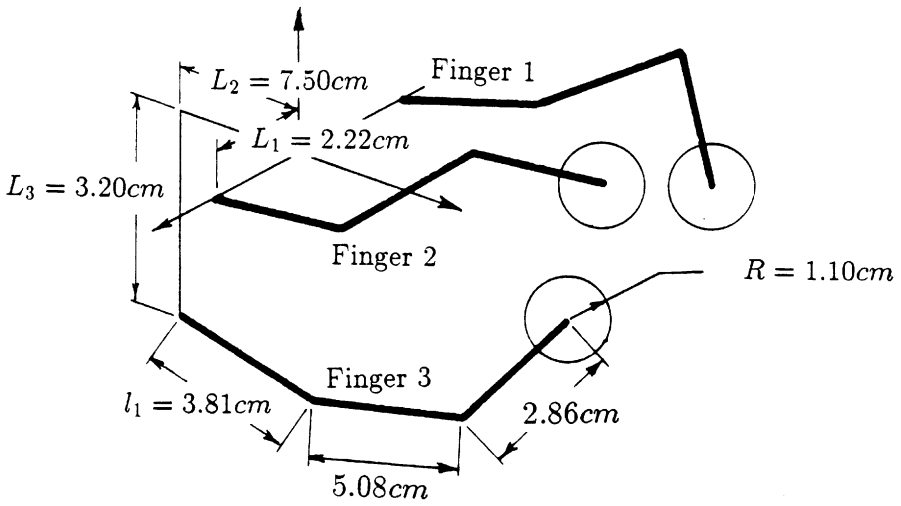


Figure A.3: Phalange length and finger placement

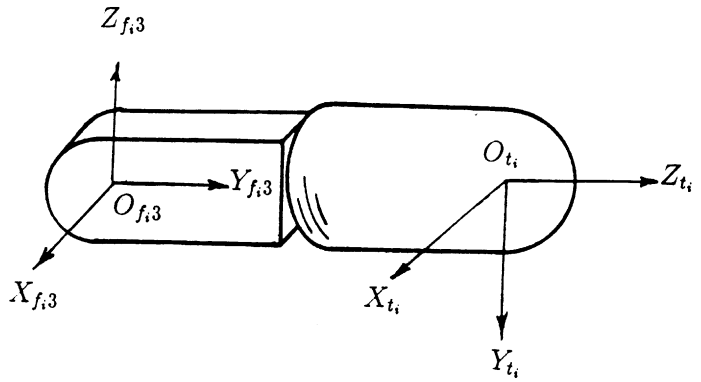


Figure A.4: Fingertip frame

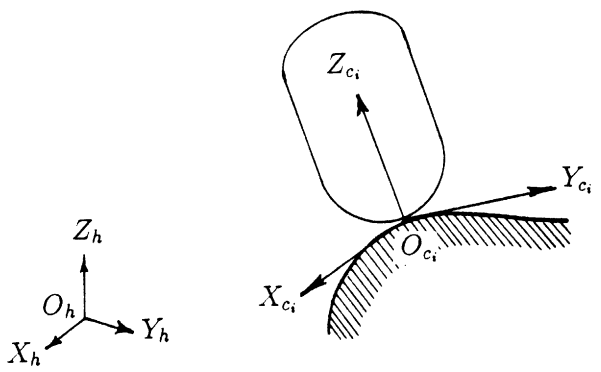


Figure A.5: Contact frame

A.5 Contact frames

The contact frames are defined in terms of the object the hand is grasping. The origin of the contact frame is the contact point between the hand and the object. The z-axis is the surface normal at the contact point. The x and y axes lie in the tangent plane of the surface. For convenience, the x-axis is defined to be parallel to the x-axis of the hand frame or to lie parallel to the plane described by the x and y axes of the hand frame, figure A.5. In this way it is possible to generate an unambiguous definition of the contact frames.

A.6 Grasp frame

The grasp frame is defined in terms of the contact points. The origin of the grasp frame is the centroid between the contact points. The z-axis is normal to the plane described by the three contact points. The x-axis is defined in the same way as the contact frames. That is, the x-axis is either parallel to the x-axis of the hand frame or parallel to

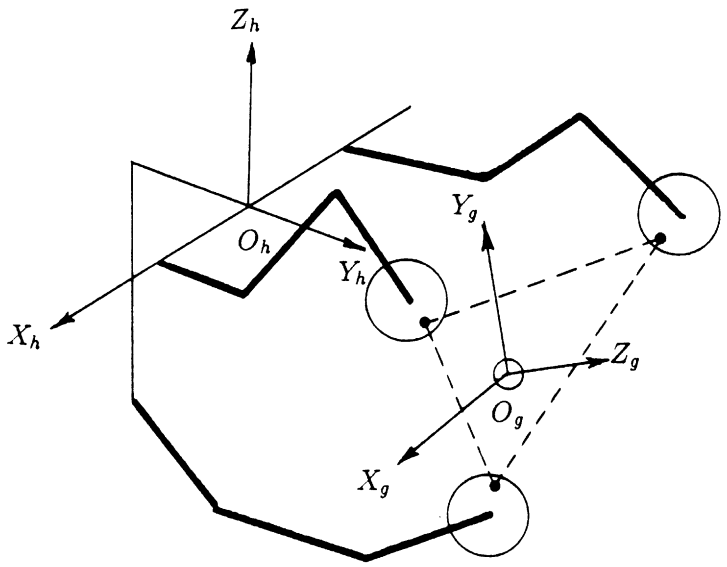


Figure A.6: Grasp frame

the plane described by the x and y axes, figure A.6.

A.7 Object frame

The origin of the object frame is the located at the centroid of the object. The axes of the object frame are arbitrary, but are usually defined along the symmetries of the object for convenience.

Appendix B

Stress state in a fingertip

B.1 Introduction

A recent entry into the field of robotics has been the development of robot hands. These hands typically have at three or more fingers with three or more joints in each finger. The fingertips are usually covered with some kind of compliant material. The Salisbury robot hand had hemispherical polyurethane fingertips, similar to that found on rollerskate wheels. Although most work has be done investigating grasping and manipulation of objects, little has been done in analyzing the complex mechanical interaction between the finger and the grasped object. As a first attempt in addressing this problem, I will examine the stress throughout the body of the fingertip in contact with a flat object.

B.2 Problem definition

The robot fingertip is shown schematically in figure B.1, touching a flat surface. The problem will be to find the stress throughout the body of the fingertip. The fingertip is assumed to a hemisphere made of a homogeneous elastic material. The material on the real robot is a polyurethane with an elastic modulus of approximately 40,000 psi and

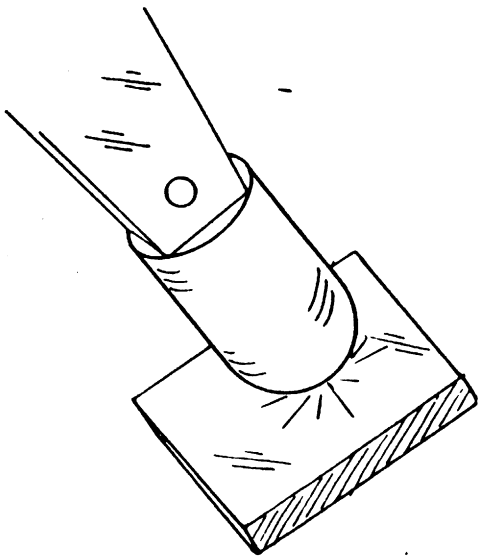


Figure B.1: Robot finger in contact with a flat surface

a Poisson's ratio of about 0.45. For simplicity in the analysis, however, particularly in the finite element procedure, the poisson's ratio will be assumed to be zero. Since the elastic modulus is small, as it is for most elastomers, a nonlinear analysis would seem necessary; however, for this analysis I will assume the contact force is small 1 lb. This force should be small enough so that displacements within the body will not large enough to warrant a complex nonlinear analysis.

B.3 Analytic solution

The analytic solution to the stresses throughout a body in contact will another body has not been worked out. The analytic solution for the stresses on the surface of a body, however, has be developed by Hertz. He assumed the two bodies were solid elliptic disks, each possessing radius of R and R'. Furthermore, these disks were in contact along a common axis under an applied load F, figure B.2. Hertz deduced that the pressure distribution between the two bodies can be described by a semi-ellipsoid of pressure constructed over the surface of the contact,

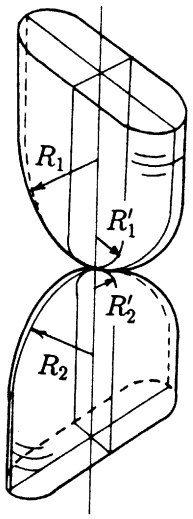


Figure B.2: Two general bodies in contact

as shown in figure B.3

The pressure distribution is given by

$$P(x,y) = P_o\sqrt{1 - x^2/a^2 - y^2/b^2} \tag{B.1}$$

The total load, therefore, is equal to the volume of the semi-ellipsoid,

$$F = \frac{2\pi abP_o}{3} \tag{B.2}$$

Solving for P_o

$$P_o = \frac{3F}{2\pi ab} \tag{B.3}$$

where a and b are given by Timosheko [Lipson]

$$a = m\sqrt[3]{\frac{3F\Delta}{4(A+B)}} \tag{B.4}$$

$$b = n\sqrt[3]{\frac{3F\Delta}{4(A+B)}} \tag{B.5}$$

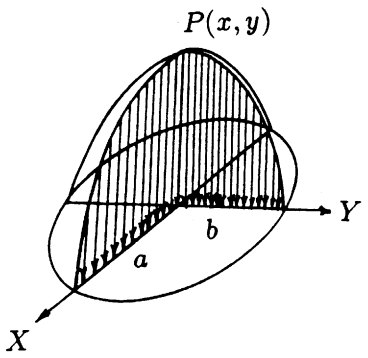


Figure B.3: Pressure distribution in the area of the contact

where

$$\Delta = (1 - \mu_1^2)/E_1 + (1 - \mu_2^2)/E_2 \tag{B.6}$$

where 0.15

$\mu_1\mu_2$ Poisson's ratio

E_1E_2 Modulii of elasticity

$A + B$ $\frac{1}{2}\left(\frac{1}{R_1} + \frac{1}{R_1'} + \frac{1}{R_2} + \frac{1}{R_2'}\right)$

$B - A$ $\frac{1}{2}\sqrt{\left(\frac{1}{R_1} - \frac{1}{R_1'}\right) + \left(\frac{1}{R_2} - \frac{1}{R_2'}\right)^2 + 2\left(\frac{1}{R_1} - \frac{1}{R_1'}\right)^2\left(\frac{1}{R_2} - \frac{1}{R_2'}\right)\cos(2\psi)}$

ψ The angle between the planes contacting the curvatures $1/R_1$ and $1/R_2$

$R_1 R_1' R_2 R_2'$ Minimum and maximum radii of curvature of the ellipsoid disks at the point of contact

m, n Constants depending on $B - A/B + A$

In the case of the contact between a hemispherical fingertip and a flat rigid plate, equations B.4 and B.5 become

$$a = b = \sqrt[3]{3FR_1\Delta/4} \tag{B.7}$$

and the maximum pressure is given by

$$P_o = 0.578\sqrt[3]{\frac{P}{R_1^2\Delta^2}} \tag{B.8}$$

again the definition of Δ is

$$\Delta = (1 - \mu_1^2)/E_1 + (1 - \mu_2^2)/E_2 \tag{B.9}$$

If the flat plate is assumed to be rigid (i.e. $E_2 \gg E_1$) then Δ can be approximated as

$$\Delta = (1 - \mu_1^2)/E_1 \tag{B.10}$$

For this problem,

$$\begin{aligned} F &= 1\text{lb} \\ E_1 &= 40,000\text{psi} \\ R_1 &= 0.5\text{in} \\ \mu &= 0.0 \end{aligned}$$

and therefore

$$\begin{aligned} \Delta &= 2.0 \times 10^{-5} \\ P_o &= 1247.9\sqrt[3]{F}\text{lb/in}^2 \\ a &= 0.020\sqrt[3]{F}\text{in} \end{aligned}$$

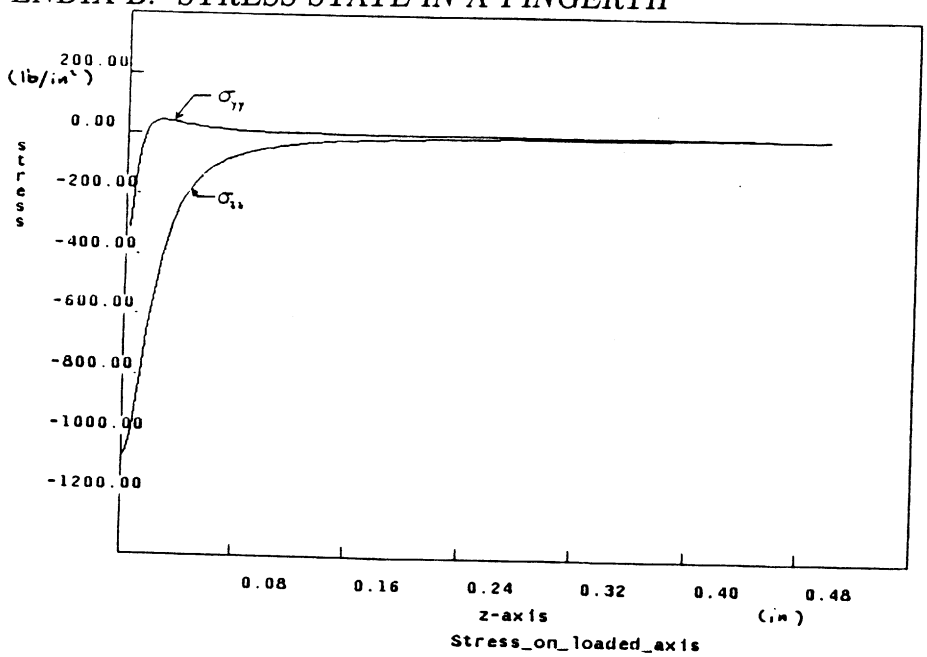


Figure B.4: Stress gradients within a sphere due to contact with a rigid plate

Although Hertz only solved for the stress in the are of the contact, H.R. Thomas and V.A. Hoersch computed the stress within the body along the loaded axis.

$$\sigma_{xx} = \sigma yy = \frac{2a(1/R_1 + 1/R_2)}{\pi\Delta}\left[(1+\mu)((z/a)\cot^{-1}(z/a) - 1) + \frac{a^2}{2(a^2+z^2)}\right] \tag{B.11}$$

$$\sigma_{zz} = \frac{-2a(1/R_1 + 1/R_2)}{\pi\Delta}\left(\frac{a^2}{a^2+z^2}\right) \tag{B.12}$$

Also, because of symmetry, σ_{xx},σyy, and σ_{zz} are the principle stresses, and therefore,

$$\tau_{yz} = \tau_{yx} = 0 \tag{B.13}$$

A plot of the stress gradients due to the contact between a flat rigid plate and a spherical fingertip on the loaded axis is shown in figure B.3

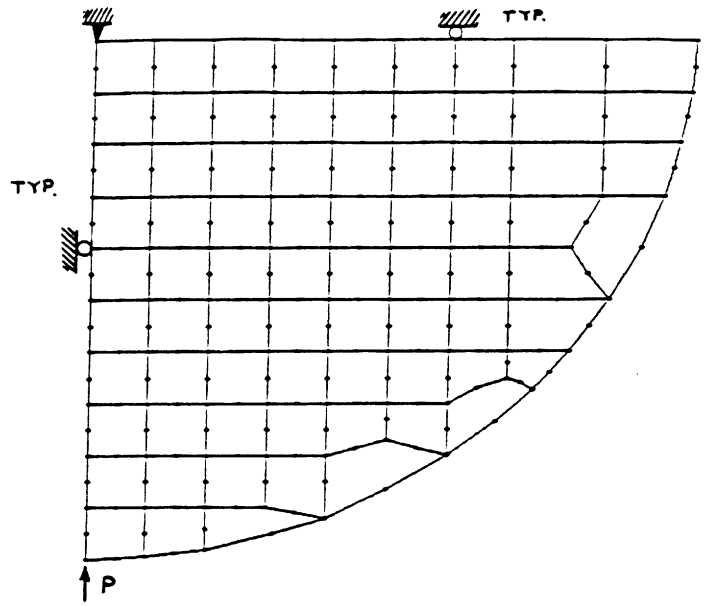


Figure B.5: Initial finite element mesh

B.4 Finite element solution

The finite element mesh that was used to analyze the stress is shown in figure B.4. The figure shows the finite element representation of one radian "slice" of the hemispherical fingertip. Axisymmetric elements are used since the body is symmetric around the axis of the contact. Eight node elements are also used, since they more accurately model the curvature of the sphere. The contact, however, is more difficult to model. A contact element could be used, in which more of the element is subjected to a force as the element deflects. However, since the radius of contact area is small, 0.019 in, compared to the elements which are 0.050 in on a side, a close approximation is a single concentrated load located at the center of the contact area. A test of this assumption and the other approximations will be whether the stresses calculated by the finite element solution match those of the analytic.

To test the validity of the mesh, the stresses between identical nodes on adjacent elements are compared. Figure B.4 shows the finite element mesh. The vertical line drawn on the figure was the line on which

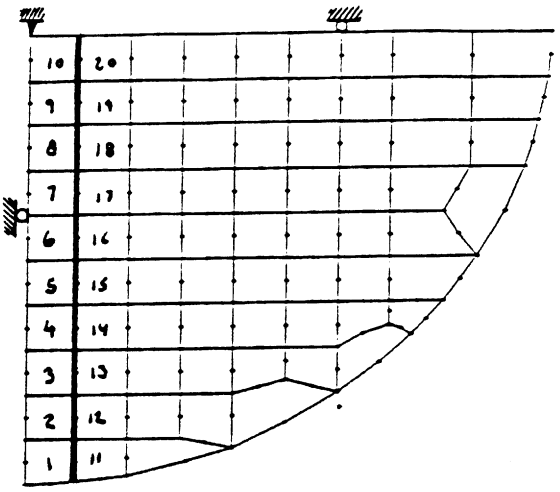


Figure B.6: Line on which nodal stress are compared

stresses were calculated. Figure B.4, B.4, and B.4 shows the stress on the line calculated at nodes on adjacent elements. The two curves in each figure represent the nodal stress on the two columns of elements (i.e. elements 1 through 10 and elements 11 through 20). In every figure, the stresses calculated at the same nodes for adjacent elements are inconsistent for values of z less than 0.25 inches. For example, figure B.4 shows the stress in the y direction, σ_{yy}. At z=0.075 in the stress calculated at the same node vary from -20 psi in element 12 to -100 psi in element 2. On the other hand, for values of z greater than 0.35 in, the variation of stress between elements is negligible, as can be seen in figure B.4. It is also interesting to note that the stress jumps for σ_{yy} occur between elements in adjacent columns, while for σ_{zz} they occur between elements in the same column. For τ_{yz}, the stress jumps between adjacent columns and adjacent rows.

It is clear from this analysis, that for an accurate prediction of stress in the body, this mesh is too coarse. Variations in stresses calculated at the same nodes were greater that 100%, for radial distances of less that 0.25 in from the contact region. Figure B.4 shows a three dimensional

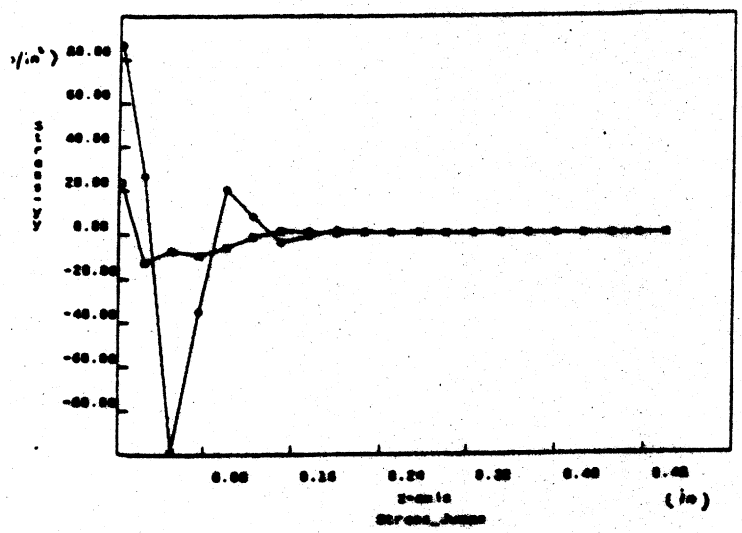


Figure B.7: Inconsistency in calculated nodal stress σ_{yy}

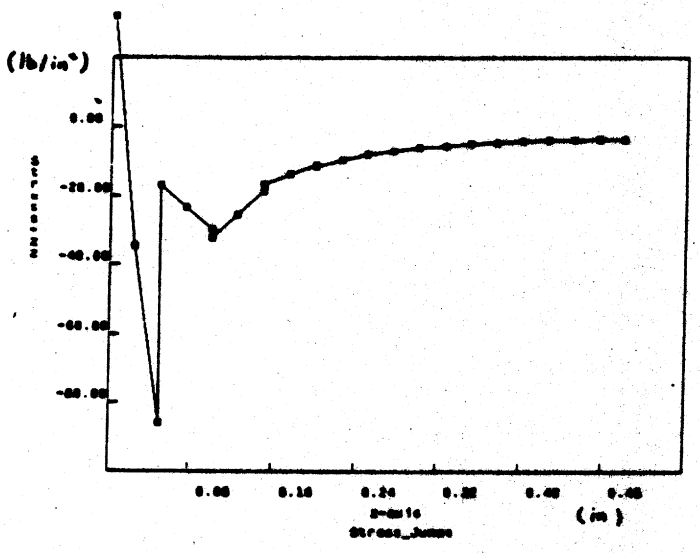


Figure B.8: Inconsistency in calculated nodal stress σ_{zz}

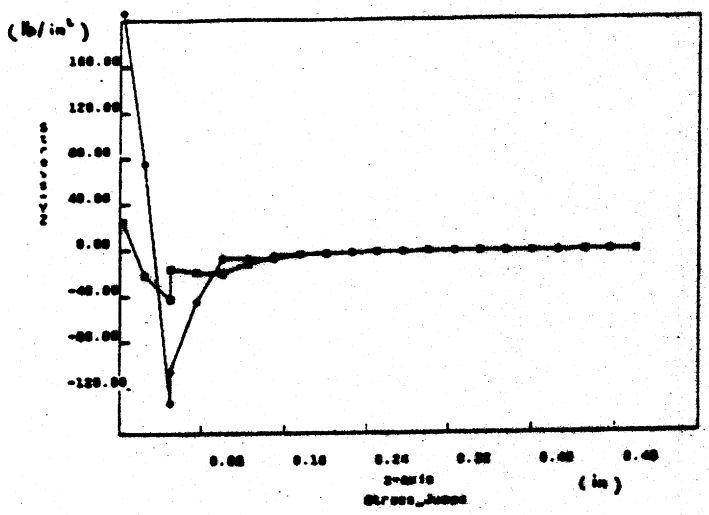


Figure B.9: Inconsistency in calculated nodal stress σ_{ys}

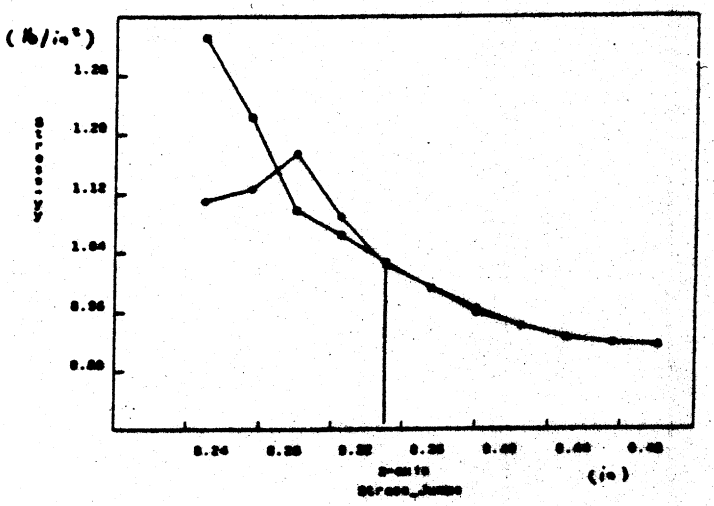


Figure B.10: Location of consistent stress

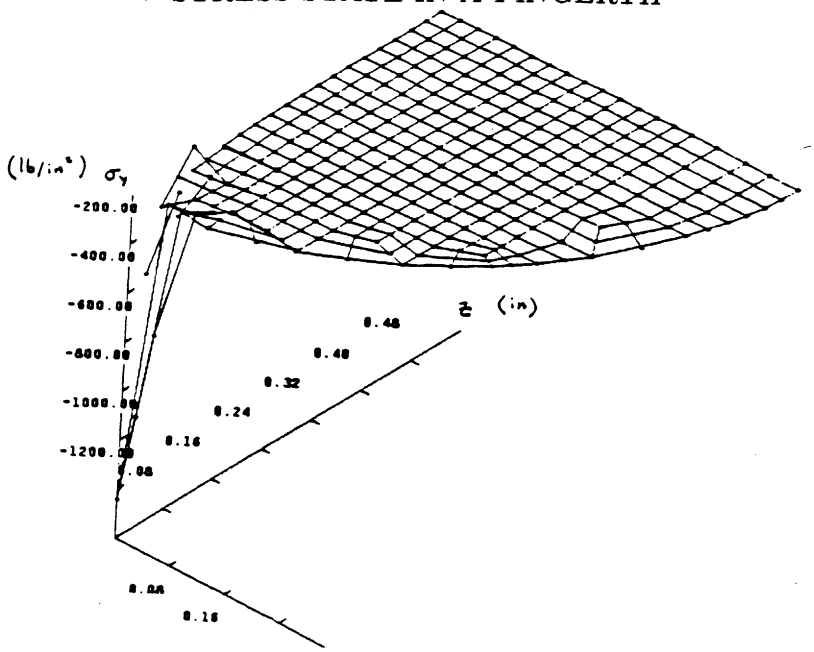


Figure B.11: Stress profile on hemispherical body

plot of stress with the y and z coordinates plotted in the plane and the stress σ_{yy} plotted on the vertical axis. Here it is easy to see the inconsistencies in the stress between elements Therefore, for an accurate calculation of the stress within the body of the fingertip, a new more refined mesh must be constructed.

B.4.1 Refined mesh

To solve the problem of nodal stress inconsistencies, a new refined mesh is constructed. Figure B.4.1 shows the region in which the mesh need to be finer. In figure B.4.1, a new refined mesh is constructed in the specified region and figure B.4.1 shows an expand view of that region. As before, to test the validity of this new mesh, I compared the stress along a single line, as shown in figure B.4.1. The stress calculated at the nodal points between adjacent elements, in this mesh shown almost no inconsistencies for values of z greater than 0.02 in, as shown in figures B.4.1 and B.4.1. This is almost an order of magnitude improvement over the previous case. Figure B.4.1 shows the same three dimensional

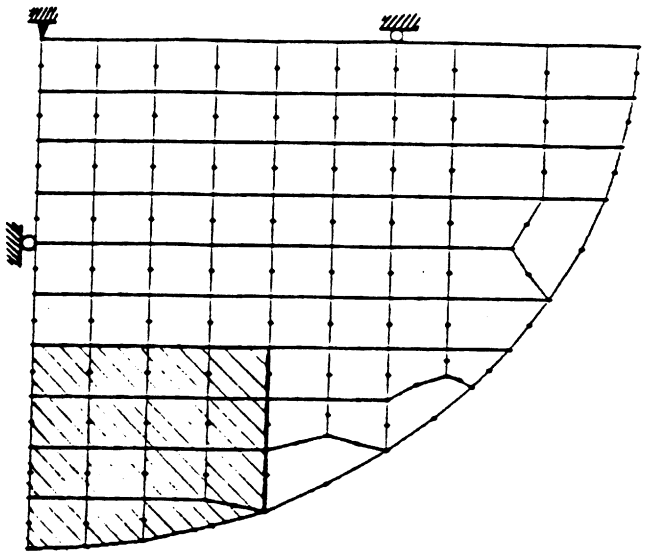


Figure B.12: Region of mesh refinement

stress profile. Again, the y and z coordinates are plotted in the plane, and the σ_{yy} is plotted on the vertical axis. As before, the plot shows stress jumps at small radial distances from the origin of contact. However, if these few values of stress are ignored, the resultant stress field is continuous along the line. Another test for the accuracy of the finite element calculations is to compare the results with an analytic solution.

B.5 Finite element vs. analytic

To compare the finite element solution against the analytic solution, only the stresses on the z axis can be considered. Figure B.5, B.5, and B.5 show the stresses σ_{yy}, σ_{zz}, and σ_{yz} respectively as calculated by the finite element solution and the analytic solution. The figures show an excellent correlation between the analytic and the finite element solution, particularly for values of z greater than 0.04 in. Ignoring these stress calculations near the origin, a accurate representation of the stress within the entire body of the fingertip can be obtained.

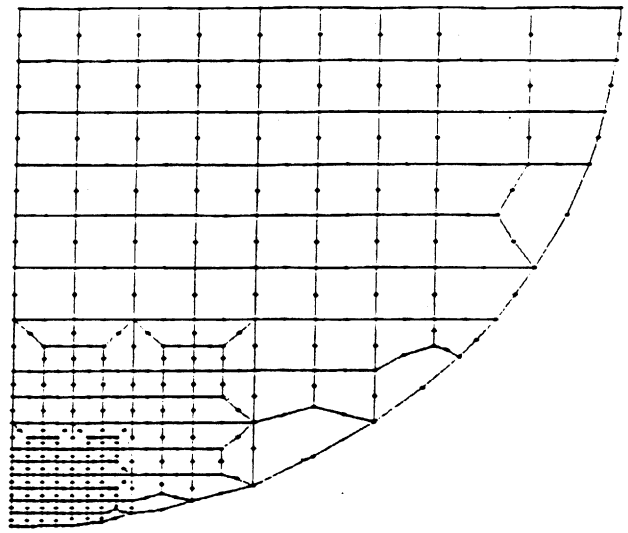


Figure B.13: Refined mesh

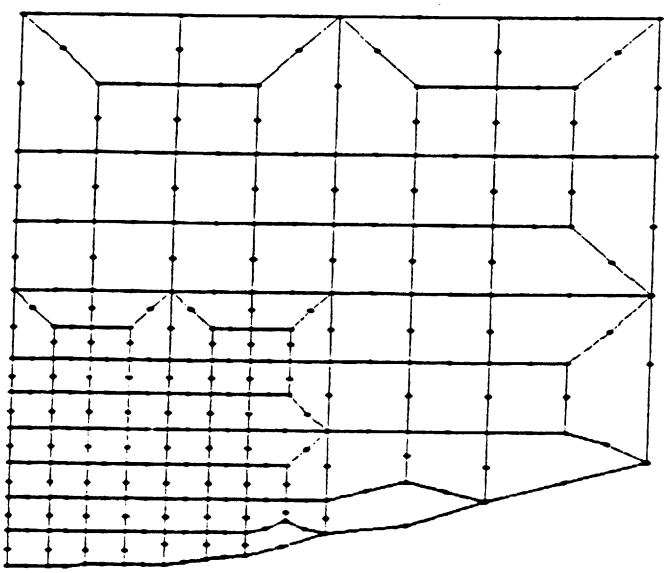


Figure B.14: Expanded view of the new refined region

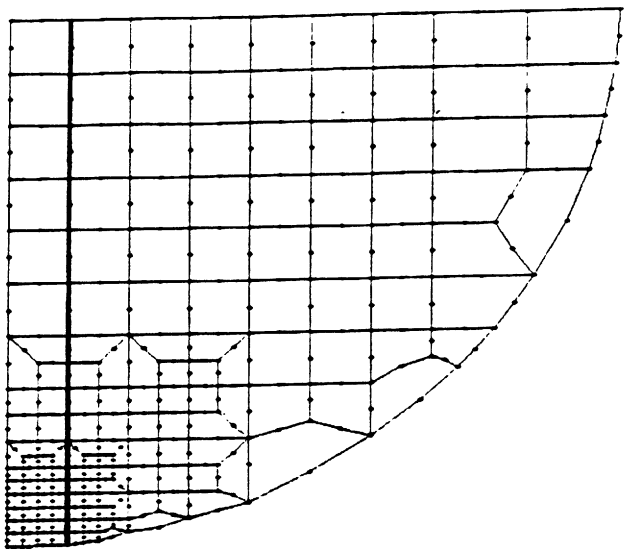


Figure B.15: Line on which nodal stresses are compared

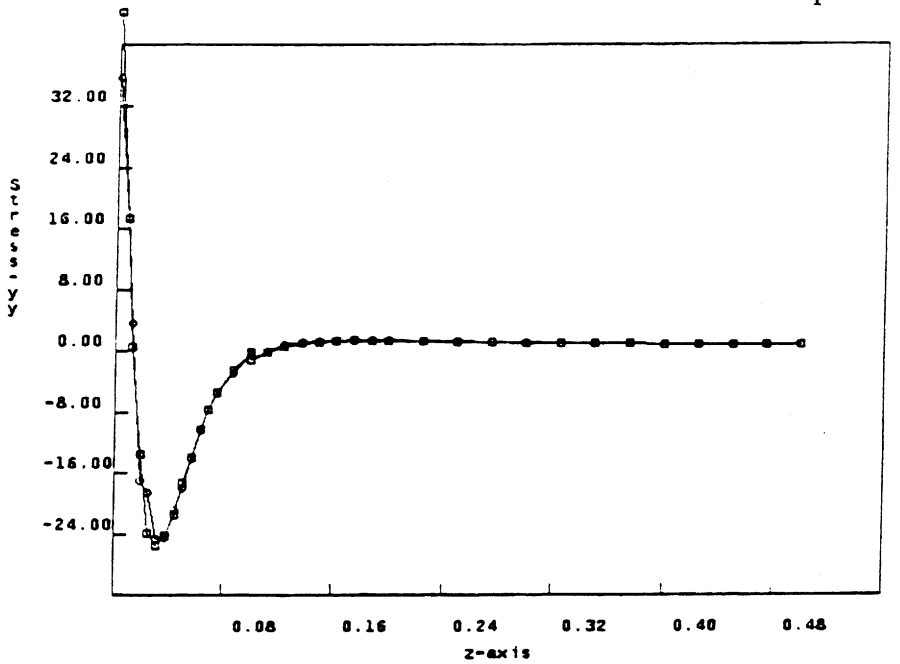


Figure B.16: Consistency in calculated nodal stresses: σ_{yy}

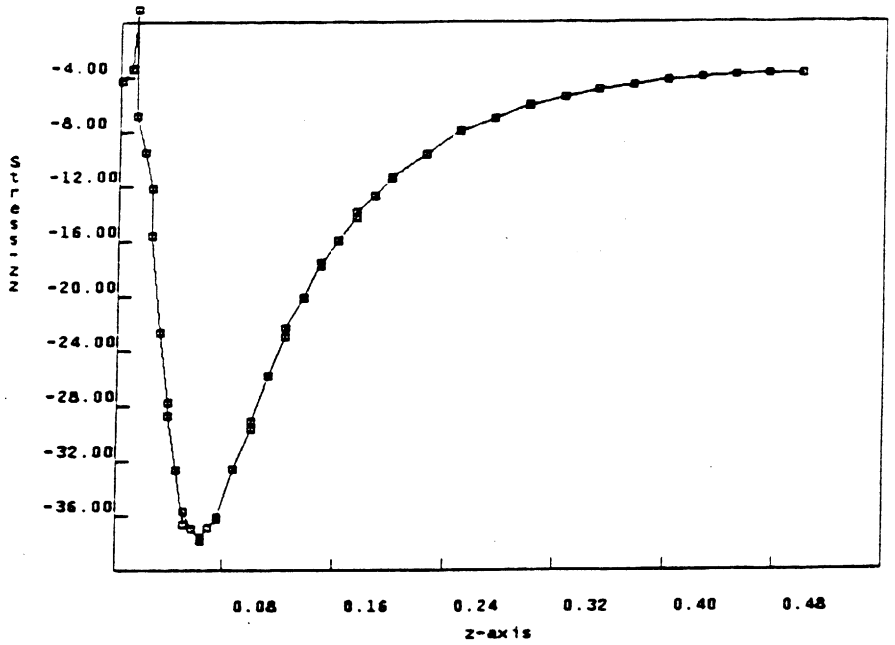


Figure B.17: Consistency in calculated nodal stress: σ_{zz}

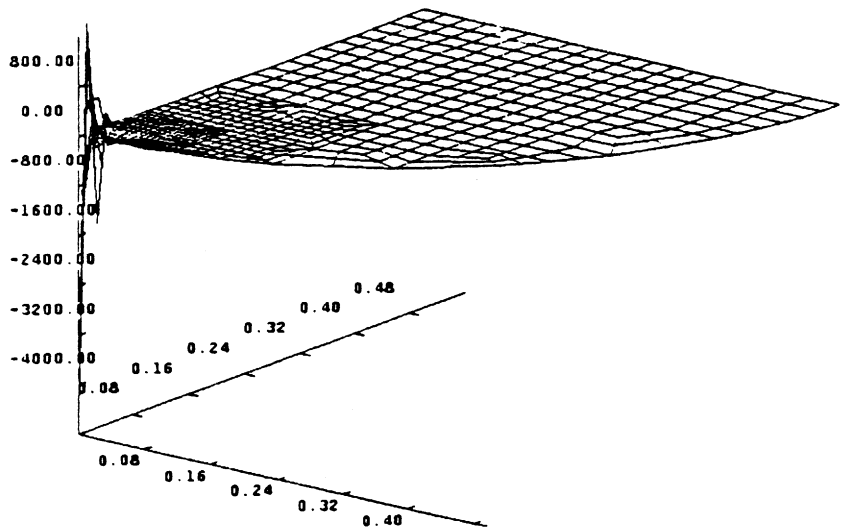


Figure B.18: Larger values of stress at small radial distance from the contact

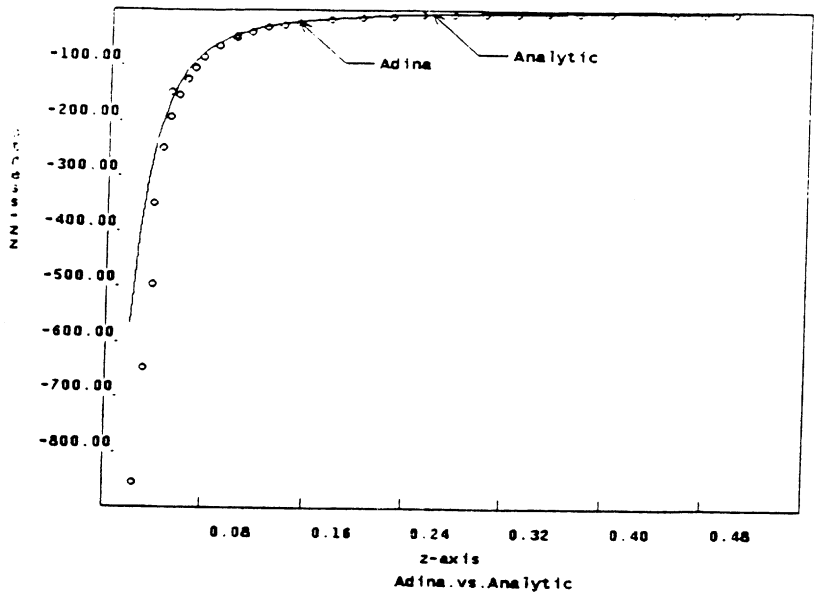


Figure B.19: Agreement between the finite element and analytic solutions: σ_{yy}

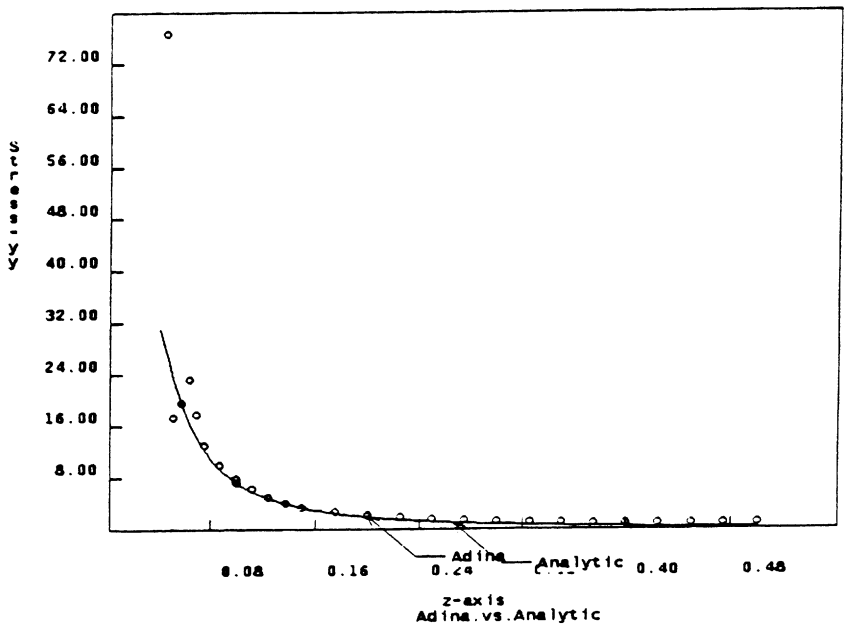


Figure B.20: Agreement between the finite element and analytic solutions: σ_{zz}

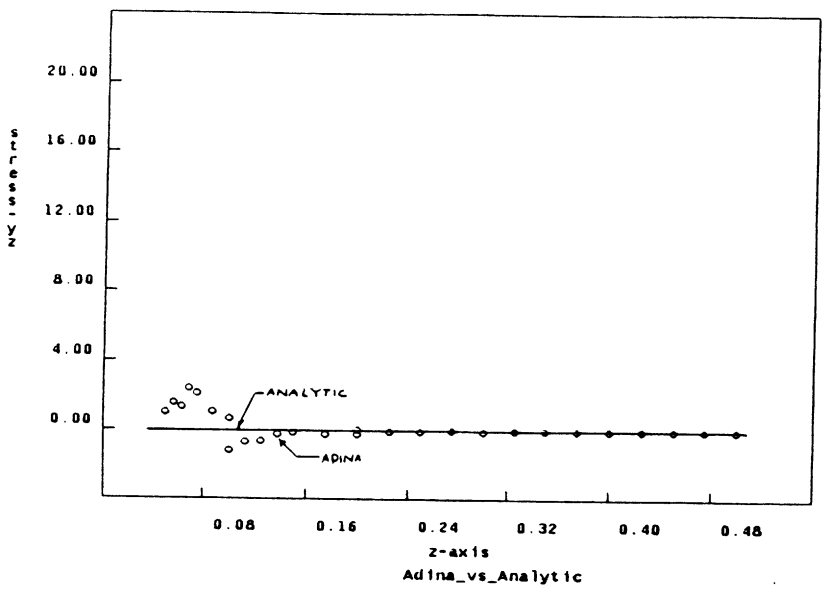


Figure B.21: Agreement between the finite element and analytic solutions: σ_{yz}

B.6 Solution

As a result of the consistency in calculated stress between the element and because of the agreement between the analytic and the finite element solution, the modified mesh is assumed to be sufficient for calculated stresses throughout the body of the hemispherical fingertip. Figures B.7, B.7, and B.7 show the stress profiles through the body of the hemisphere. From the analysis, these stresses can be taken as an accurate representation of the true stress in the fingertip.

B.7 Conclusion

In this analysis, a finite element method was developed to accurately predict the stresses in the body of a hemispherical fingertip in contact with a flat object (for relatively small forces). This information can be useful, for example, in determining the placement of subcutaneous force and tactile sensors.

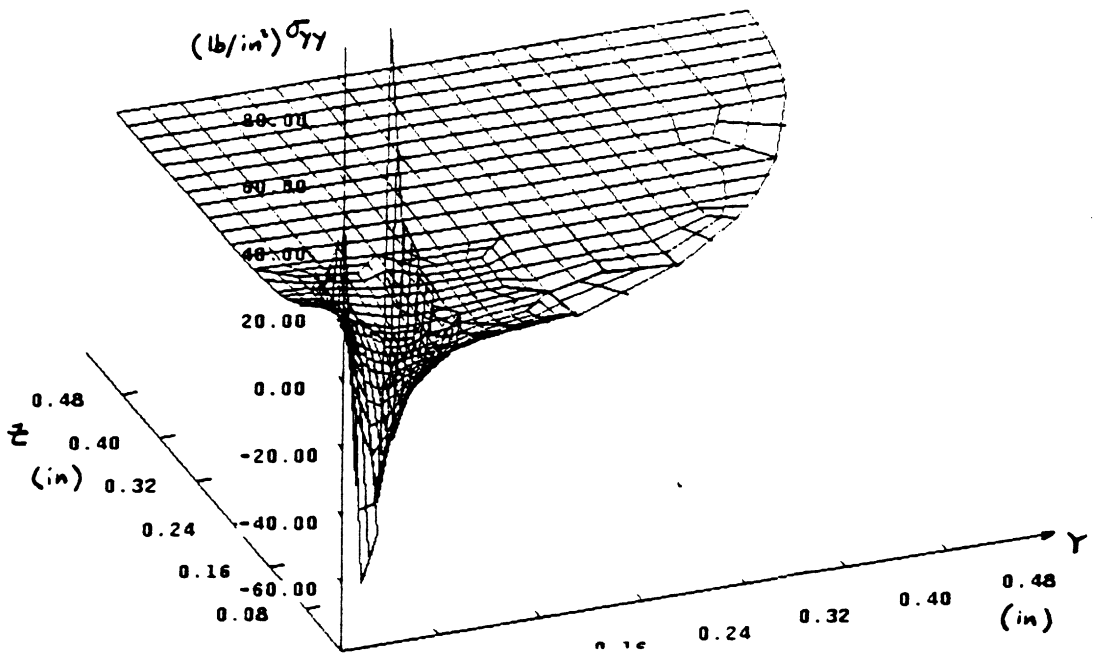


Figure B.22: Stress profile in the fingertip: σ_{yy}

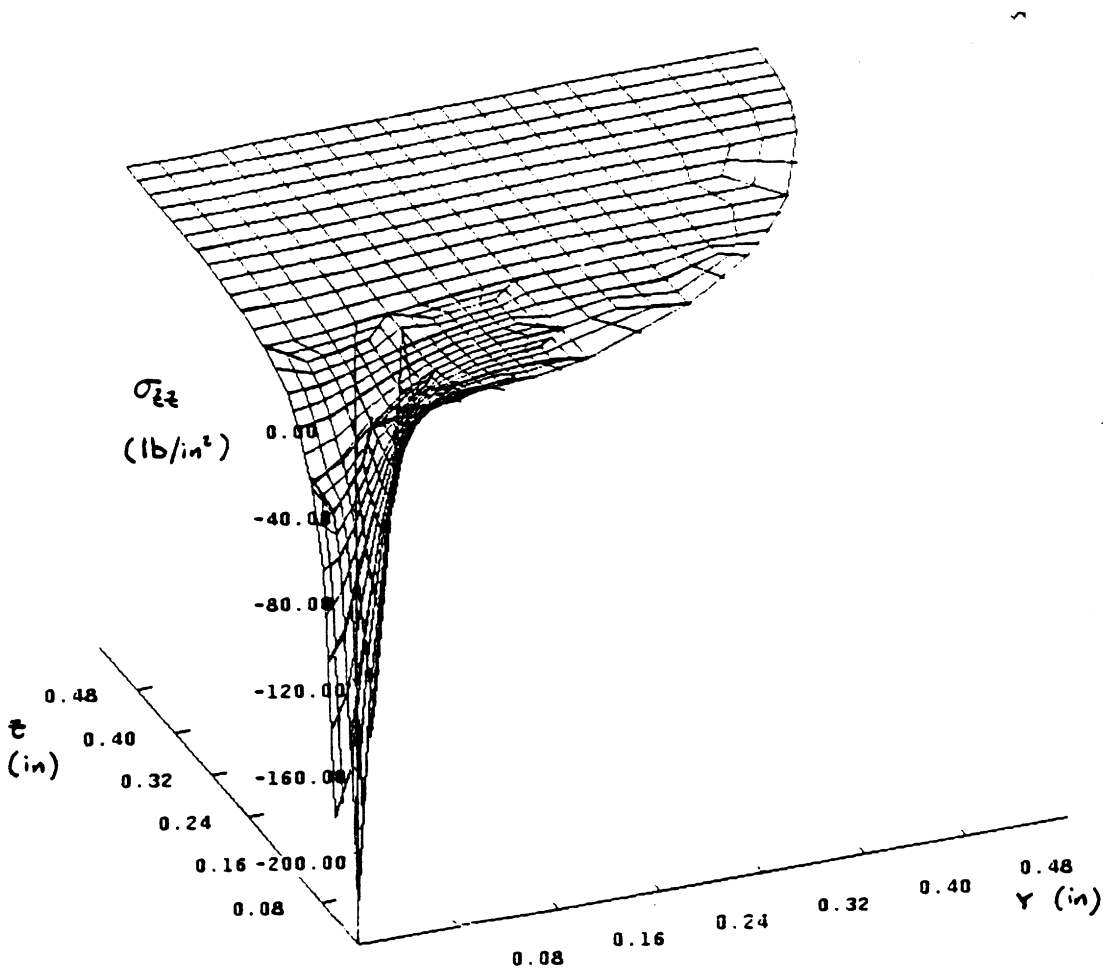


Figure B.23: Stress profile in the fingertip: σ_{zz}

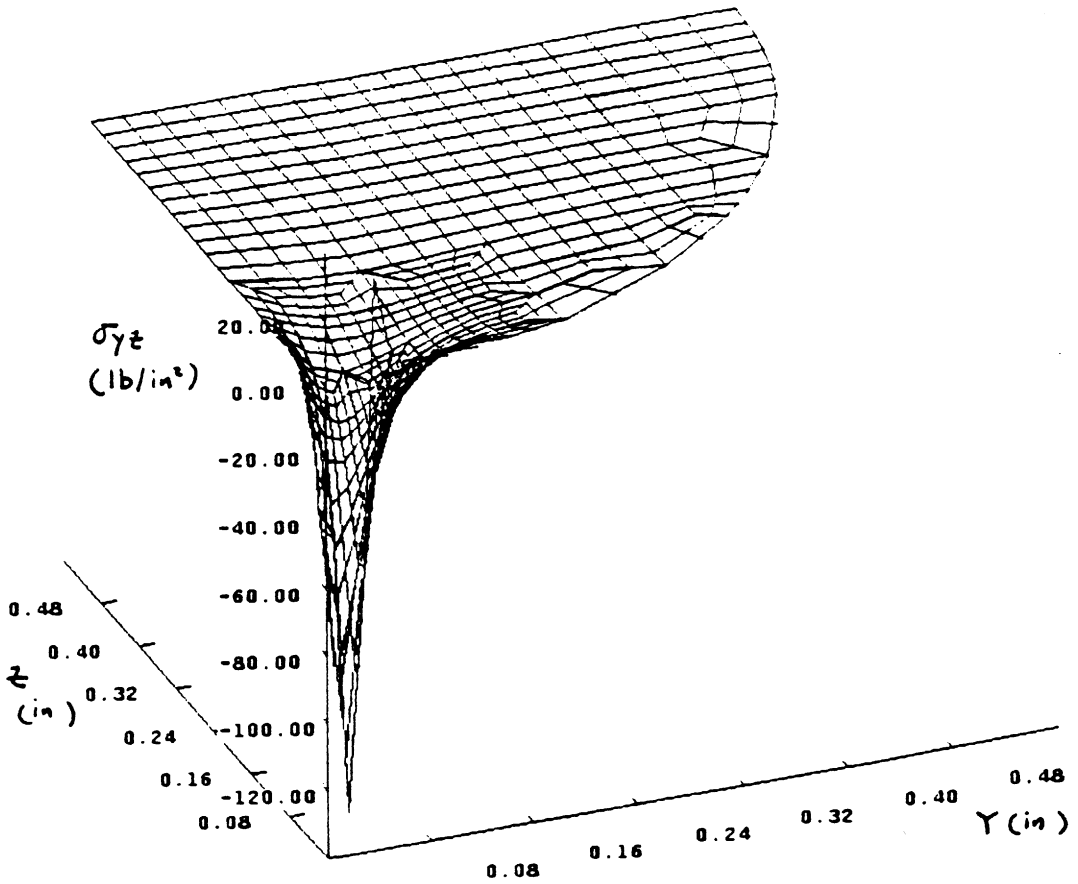


Figure B.24: Stress profile in the fingertip: σ_{yz}

Appendix C

Fingertip sensor

C.1 Introduction

A contact resolving fingertip sensor was developed which determines the location of a contact, as well as the direction and magnitude of the force at the contact point. The sensor is based on silicon strain gauges on steel flexures, which measure the forces and moments at the center of the shell. From these force readings, it is possible to determine the direction of the force vector acting on the shell of the fingertip. Using the moment readings, the location of the force vector can be determined. The intersection of the force vector with the outer surface of the fingertip yields two points. One point corresponds to a pulling on the surface of the fingertip. The other point corresponds to a pushing force on the sensor. Assuming the fingertip only pushes against surfaces, one of the points is immediately eliminated. This leaves a single point, the contact point, through which the force is acting. Since the sensor resolves the components of the force, it can determine the magnitude and direction of the force at the point contact as well.

C.2 Theory

The theories presented in this section were developed by [Salisbury]. Suppose a force $\mathbf{F} = [f_x, f_y, f_z]$ is acting through a point $\mathbf{x_c} = [x_c, y_c, z_c]$ on a convex surface S. The convex surface S is defined as a smooth surface in $\Re^3$, such that for every pair of points p_1 and p_2 in S, the line $mp_1 + (1-m)p_2$ for $m \in \Re$ intersects S in only two points p_1 and p_2. Also for any two points p_1 and p_2 with contact normals n_1 and n_2 respectively, the dot products of the vector $p_1 - p_1$ and the normals n_1 and n_2 have different sign. This condition allows the unambiguous resolution of the location of the force on the surface S.

Let $OXYZ$ be some reference frame so that every point in S can be defined relative to $OXYZ$. The wrench in terms of screw coordinates defined relative to $OXYZ$ due to the force $\mathbf{F}$ acting through the point $\mathbf{x_c}$ is

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \end{bmatrix} = \begin{bmatrix} f_x \\ f_y \\ f_z \\ -z_c f_y + y_c f_z \\ z_c f_x - x_c f_z \\ -y_c f_x + x_c f_y \end{bmatrix}. \tag{C.1}$$

Now we wish to describe the wrench $\mathbf{w}$ in terms of the wrench axis, the pitch, and the magnitude. The direction of the wrench axis $\mathbf{A}$ is

$$\mathbf{d_A} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} \tag{C.2}$$

and the point on the wrench axis nearest the reference frame is

$$\mathbf{x_o} = \begin{bmatrix} x_o \\ y_o \\ z_o \end{bmatrix} = \frac{1}{m} \begin{bmatrix} w_2 w_6 - w_3 w_5 \\ w_3 w_4 - w_1 w_6 \\ w_1 w_5 - w_4 w_2 \end{bmatrix}, \tag{C.3}$$

where m is the magnitude of the wrench and also the magnitude of the force

$$m = \sqrt{w_1^2 + w_2^2 + w_3^3}. \tag{C.4}$$

Any point along the wrench axis can be described by

$$\mathbf{x} = \mathbf{x_o} + \lambda \mathbf{d_A}. \tag{C.5}$$

The intersection of the line in equation C.5 with the surface S yields one or two points. If the intersection yields a single point then the location of the force on the surface is immediately determined. However, if the intersection produces two points, the location of the contact can still be found. Suppose the two points of intersection are x_1 and x_2 with normals n_1 and n_2. Take the dot product of the force F with the normals n_1 and n_2. The point associated with the normal which produces a negative value of the dot product is the contact point.

Therefore it is possible to determine the point of contact $\mathbf{x_c}$ from the wrench $\mathbf{w}$. If it were possible to measure the wrench $\mathbf{w}$, then the location, direction and magnitude of a force exerted through a point contact on a surface S can be found. This is precisely the concept behind the contact resolving fingertip sensor.

C.3 Design

The schematic diagram of the fingertip sensor is shown in figure C.1. The fingertip has a hemispherical top and a cylindrical skirt which bolt onto a loadcell. The loadcell is in the form of a maltese cross and is design to resolve all six components of the wrench, that is, forces and moments in three cartesian directions. Small semiconductor strain gauges are mounted on the legs of the cross. These gauges measure strain which is proportional to the bending moment in the beam which is, in turn, proportional to the forces and moments. The next section describes some critical issues in the design of the loadcell.

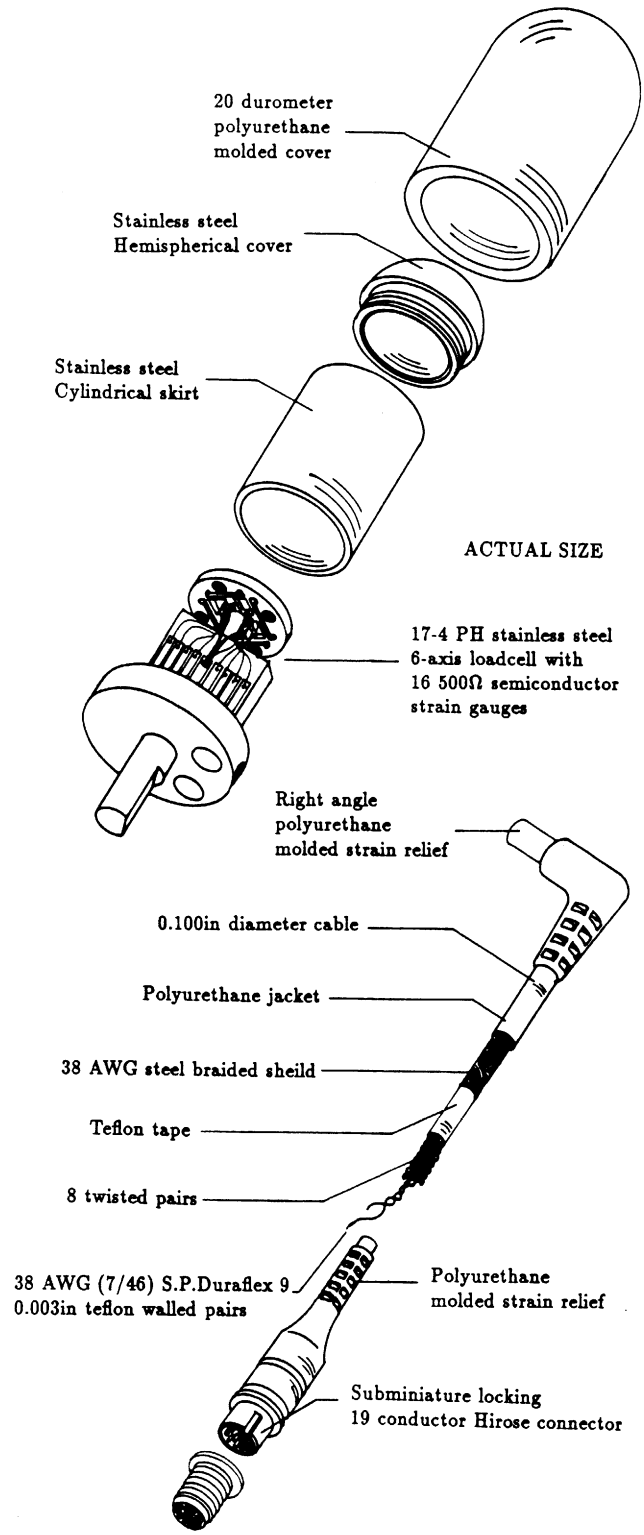


Figure C.1: The fingertip sensor can determine the magnitude, the location, and the direction of a force applied through a point contact on its surface. Small semiconductor strain gauges mounted on small steel flexure are used to resolve the forces and moments on the fingertip shell.

C.4 Design of loadcell

C.4.1 Introduction

The loadcell for the force and tactile sensing fingertip must be able to resolve all six components of forces and moments applied to it. In order to properly design the loadcell, it is necessary to determine the forces and moments under which may act on the loadcell, as well as the state of stress in each of its members. The fingertip may be subject to an infinite variety of forces. However, instead of examining all the possible forces, we will consider only a small set of forces which produce extreme values of force and moment on the loadcell. This set of forces is shown in figure C.2. In the next sections, the forces and moments on the loadcell resulting from the externally applied loads will be determined, along with the associated stress states in the flexures of the loadcell.

The analysis of stress states is only part of the design procedure. The design must also take into account the manufacture, assembly, and gauging procedures, in order to minimize cost and maximize efficiency. From this analysis, the design and dimensions of the loadcell will be determined.

C.4.2 Fingertip sensor dimensions

Figure C.3 illustrates and lists the symbolic values for the dimensions of the fingertip and the loadcell. These values will be used throughout the analysis when calculating stress, strain, and deformation. The constrained values are

$R = 0.406\text{in}$
$H = 0.813\text{in}$
$L_l = 0.070\text{in}$
$L_p = 0.100\text{in}$
$H_1 = 0.633\text{in}$

and the values yet to be determined are

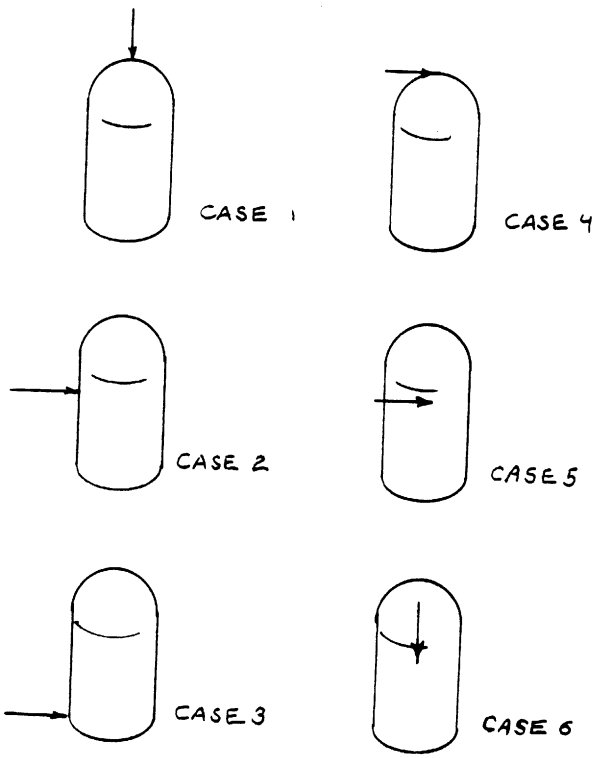


Figure C.2: Forces and moments are generated at the center of the sensor as a function of the externally applied load. The set of external loads which produce maximum internal forces and moments are listed above. Each of these cases will be considered individually and the resulting forces and moments will be calculated, as well as the stresses they create on the load bearing members.

h_l
w_l
h_p
w_p

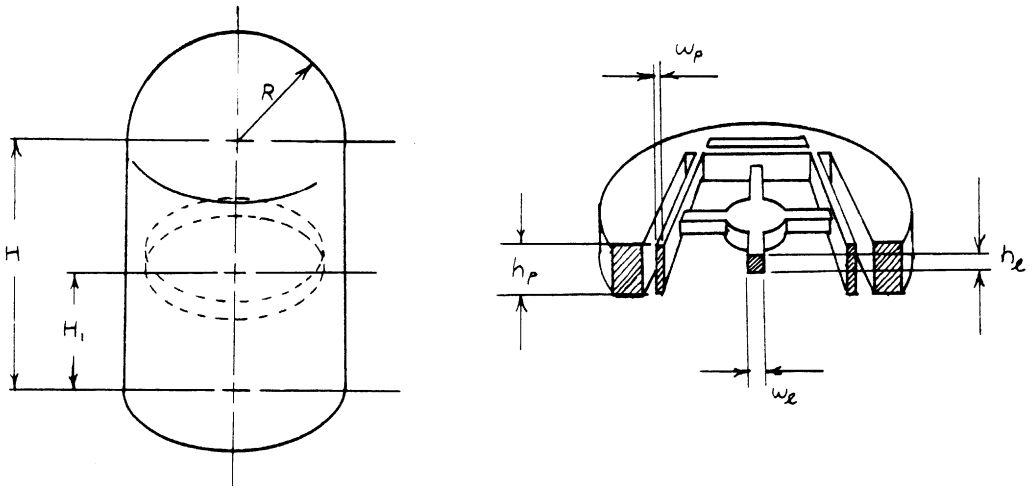


Figure C.3: The dimensions for the loadcell and the fingertip illustrated above will be used through the analysis on the fingertip. From the analysis, optimal values for the dimensions listed will be found to maximize fingertip performance and durability.

C.4.3 Mechanical properties

The loadcell is to be machined from 17-4PH stainless steel. This particular type of steel was chosen because its coefficient of thermal expansion matches that of the semiconductor strain gauges. A similar value of thermal expansion is desired since both the gauges and sensor are heated during the bonding process and cooled, the same contraction rates will not introduce an offset strains in the gauges.

The mechanical properties of the 17-4PH stainless steel are:

$$E = 28.5 \times 10^6 \text{lb/in}^2$$

$$G = 10.6 \times 10^6 \text{lb/in}^2$$

$$\sigma_y = 80,000 - 100,000 \text{lb/in}^2$$

$$\epsilon_y = \sigma_y / E = 3157 \ \mu \text{ strain}$$

The mechanical properties of the silicon strain gauges are:

Maximum strain $= 3000 \ \mu$ strain

C.4.4 General cantilever beam problem

The analysis of the cantilever beam will be used throughout the analysis of the loadcell, so the general problem is stated here along with the standard equations of load, shear force, bending moment, and deformation.

A cantilever beam is shown in figure C.4. It is subject to a force F and a moment M at its end point. The deflection of the end point is given by δ and the angle from the horizontal at the end point is denoted by ϕ.

Bending moment

The load, the shear force, and the bending moment along the beam are given by

Load:

$$q\langle x\rangle = (-FL - M)\langle x\rangle_{-2} + F\langle x\rangle_{-1} + M\langle x - L\rangle_{-2} - F\langle x - L\rangle_{-1} \quad \text{(C.6)}$$

Shear force:

$$v\langle x\rangle = (FL + M)\langle x\rangle_{-1} - F\langle x\rangle^0 - M\langle x - L\rangle_{-1} + F\langle x - L\rangle^0 \quad \text{(C.7)}$$

Bending moment:

$$M_b\langle x\rangle = (-FL - M)\langle x\rangle^0 + F\langle x\rangle^1 + M\langle x - L\rangle^0 - F\langle x - L\rangle^1 \quad \text{(C.8)}$$

Deformation

The displacement of the end of the beam is given by

$$\delta = \frac{FL^3}{3EI} + \frac{ML^2}{2EI} \quad \text{(C.9)}$$

where I is the area moment of inertia given by

$$I = \frac{h^3 w}{12}$$

and the angle ϕ is

$$\phi = \frac{FL^3}{2EI} + \frac{ML}{EI} \quad \text{(C.10)}$$

Stress and strain

In general, the stress due to a bending moment M_b is

$$\sigma = -\frac{M_b y}{I} \quad \text{(C.11)}$$

where I is the area moment of inertia and y is the distance from the point of zero stress to point at which the stress is measured. In the case of the rectangular cantilever beam, the point of zero stress coincides with the center of the beam. Therefore, the maximum stress due to bending is located on the outer surface of the beam. That is, when $y = h/2$, the stress is

$$\sigma = -\frac{6M_b}{h^2 w} \quad \text{(C.12)}$$

C.4.5 Case 1

Assumptions

Assume a force is applied to the fingertip on the top of the hemisphere, figure C.5. The force is applied vertically along the major axis of the sensor. By symmetry, fingertip shell will be displaced vertically, no horizontal motion or rotation will occur. Also by symmetry, the forces will by applied equally to each leg of the cross.

Analysis

A single leg of the cross is shown in figure C.6. By of symmetry, each leg of the cross receives an equal force, $F/4$, and undergoes an equal displacement δ, and angle $\phi = 0$. That is, given

$$\phi = \frac{(F/4)L_l^3}{2EI} + \frac{ML_l}{EI} = 0 \tag{C.13}$$

where

$$I = \frac{w_l^3 h_l}{12} \tag{C.14}$$

then the moment will be

$$M = -\frac{FL_l}{8} \tag{C.15}$$

Therefore the loading is

$$q\langle x\rangle = -FL_l/8\langle x\rangle_{-2} + F/4\langle x\rangle_{-1} \tag{C.16}$$
$$v\langle x\rangle = FL_l/8\langle x\rangle_{-1} - F/4\langle x\rangle^0 \tag{C.17}$$
$$M_b\langle x\rangle = -FL_l/8\langle x\rangle^0 + F/4\langle x\rangle^1 \tag{C.18}$$
$$\tag{C.19}$$

The maximum bending moment occurs at $x = 0$ and $M_b = -FL/8$. Therefore, the maximum stress in the beam under this particular loading is

$$\sigma_{max} = \frac{3FL}{4w_l h_l^2} \tag{C.20}$$

C.4.6 Case 2

Assumptions

A horizontal force is applied to the fingertip, in line with the loadcell, as shown in figure C.7. The force is evenly distributed on either side of the loadcell, as a result no rotation occurs. Since, by symmetry, both sides of the loadcell will behave the same way, one half of the loadcell will be analyzed and is depicted in figure C.8. It is assumed the compressive displacements are negligible, therefore, both the legs parallel to the applied force and the phalanges are assumed to be rigid. In addition, a cantilever beam model is assumed for both the legs and the phalanges.

Analysis

The two phalanges and the leg of the cross on each side of the loadcell are displaced by the same amount δ, figure C.8. The ends of the phalanges and the legs do not rotate, therefore $\phi = 0$ and by equation C.10

$$M_i = -F_i L_i/2 \tag{C.21}$$

Substitute into equation C.9, the displacement is

$$\delta = \frac{F_i L_i}{12 E_i I_i} \tag{C.22}$$

and the force

$$F_i = \left(\frac{E_i w_i^3 h_i}{L_i^3}\right)\delta \tag{C.23}$$

Since the force F is evenly distributed on each side of the loadcell, the sum of the force on the each flexure is $F/2$, that is

$$F/2 = F_p + F_l + F_p \tag{C.24}$$

Substitute C.22 into C.24 yields

$$F/2 = \left[\frac{E w_p^3 h_p}{L_p^3} + \frac{E w_l^3 h_l}{L_l^3} + \frac{E w_p^3 h_p}{L_p^3}\right] \tag{C.25}$$

Solve for δ

$$\delta = \frac{FL_l^3 L_p^3}{2E(w_l^3 h_l L_p^3 + 2w_p^3 h_p L_l^3)} \tag{C.26}$$

The force and the moment on the leg of the cross is

$$F_l = \left(\frac{w_l^3 h_l L_p^3}{w_l^3 h_l L_p^3 + 2w_p^3 h_p L_l^3}\right)\frac{F}{2} \tag{C.27}$$

$$M_l = -\left(\frac{w_l^3 h_l L_p^3}{w_l^3 h_l L_p^3 + 2w_p^3 h_p L_l^3}\right)\frac{FL_l}{4} \tag{C.28}$$

$$\tag{C.29}$$

and the force and moment on the phalanges are

$$F_p = \left(\frac{w_p^3 h_p L_l^3}{w_p^3 h_p L_l^3 + 2w_l^3 h_l L_p^3}\right)\frac{F}{2} \tag{C.30}$$

$$M_p = -\left(\frac{w_p^3 h_p L_l^3}{w_p^3 h_p L_l^3 + 2w_l^3 h_l L_p^3}\right)\frac{FL_p}{4} \tag{C.31}$$

$$\tag{C.32}$$

Finally, the maximum stress on the leg of the cross is

$$\sigma_l = \frac{3w_l L_p^3 L_l F}{2(w_l^3 h_l L_p^3 + 2w_p^3 h_p L_l^3)} \tag{C.33}$$

and the maximum stress on the phalange is

$$\sigma_p = \frac{3w_p L_l^3 L_p F}{2(w_p^3 h_p L_l^3 + 2w_l^3 h_l L_p^3)} \tag{C.34}$$

C.4.7 Case 3

Assumptions

A force is applied at the base of the fingertip shell in the horizontal direction as shown in figure C.9. This particular case produces the largest stress on the members of the loadcell. The force applied in this

orientation and position will therefore be limiting force in the design of the loadcell. It is assumed the loadcell undergoes a rotation and deformation as illustrated in figure C.10. The phalanges parallel to the applied force are assumed to undergo rigid rotation. The phalanges perpendicular to the applied force are subject to a moment, but for this analysis these phalanges are also assumed rigid. Therefore the entire outer structure of the cross undergoes a rigid rotation ϕ. As in the previous case, the legs perpendicular to the external force undergo a displacement, however, since the major stress components are on the legs parallel to the external force, these displacements are assumed to be negligible.

Analysis

A free body diagram of the cross is shown in figure C.11. The deformation from the force F is assumed small in comparison with that from the moment FH_1. The two legs perpendicular to the applied force undergo a rotational twist subject to a moment M_2 and the two legs parallel to the applied force bend under a moment M_1 and a force F_1. By equilibrium,

$$FH_1 = 2F_1L_l - 2M_1 - 2M_2 \tag{C.35}$$

A diagram of a single leg undergoing bending is shown in figure C.12. By geometric compatibility, the displacement equals the length of the beam times the angle, that is, assuming angles are small.

$$\delta = L_l\phi \tag{C.36}$$

Substitute into the equations for displacement and angle and solve for the moment M_1 yields

$$M_1 = -F_1L_l \tag{C.37}$$

and the angle ϕ

$$\phi = \frac{18F_1L_l^2}{Ew_lh_l^3} \tag{C.38}$$

Figure C.13 shows a diagram of a beam under the torsional load, and the equation for angular rotation as a function of moment is

$$\phi = \frac{-M_2 L_l}{c_2 G h_1 w_1^3} \tag{C.39}$$

for $w_1 \leq h_1$ and $c_2 = 0.170$ approx. Equate equations C.38 and C.39 and solve for the moment M_2

$$M_2 = \left(\frac{18 L_l c_2 G w_l^2}{E h_l^2}\right) F_1 \tag{C.40}$$

Now substitute equations C.37 and C.40 into C.35

$$FH_1 = 2F_1 L_l - 2(-F_1 L_l) - 2\left(\frac{18 L_l c_2 G w_l^2}{E h_l^2}\right) F_1 \tag{C.41}$$

and solve for the force F_1

$$F_1 = \frac{FH_1}{\left[4L_l + \frac{36 L_l c_2 G w_l^2}{E H_l^2}\right]} \tag{C.42}$$

Figure C.11 shows a single leg of the cross under the applied loads. By equilibrium the sum of the moments at the root of the leg are zero. That is,

$$-F_1 L_l + M_1 + M' = 0 \tag{C.43}$$

and using equation C.37

$$M' = 2F_1 L_l \tag{C.44}$$

and is also the maximum bending moment under which the beam is subject. Substituting equation C.42 into the above C.44 yields.

$$M_{max} = \frac{FH_1 L_l}{\left[2L_l + \frac{18 L_l c_2 G w_l^2}{E h_l^2}\right]} \tag{C.45}$$

Therefore the maximum stress is

$$\sigma_{max} = \frac{3FH_1 L_l}{w_l h_l^2 \left[L_l + \frac{9 L_l c_2 G w_l^2}{E h_l^2}\right]} \tag{C.46}$$

C.4.8 Case 4

The fingertip may also be subject to a tangent force as shown in figure C.14. This case, however, is identical to case 3, except for the length of the moment arm. In this case it is $R + H - H_1$ instead of H_1 and the maximum stress is therefore

$$\sigma_{max} = \frac{3F(R + H - H_1)L_l}{w_l h_l^2 \left[L_l + \frac{9L_l c_2 G w_l^2}{E h_l^2}\right]} \tag{C.47}$$

C.4.9 Case 5

Assumptions

Figure C.14 shows a force applied to the fingertip, tangent to the surface. It is assumed this force is on the same plane as the loadcell, therefore stress is only induced on the sides of the flexures. In this case, the stress due to the force and the stress from the induced moment superimpose, so that each case must be considered in order to find the maximum stress on the members. First we will determine the maximum stress due to the moment FR and then add the maximum stress determined from case 2. Figure C.15 shows the loadcell rotated as a result of an applied moment M. It is assumed all the phalanges are rigid and that the entire outer structure of the loadcell undergoes a rotation ϕ.

Analysis

A single leg of the cross is shown in figure C.11. It is subject to a force F_i and a moment M_i applied at its endpoint. By geometric compatibility the displacement

$$\delta = \phi L_l \tag{C.48}$$

and using equation C.9, the moment can be found in terms of the force

$$M_i = F_i L_l \tag{C.49}$$

By symmetry, each leg of cross is under the identical loading as shown in figure C.15. The sum of the moments must be zero, therefore,

$$FR = 4F_iL_l + 4F_iL_l \tag{C.50}$$

and

$$F_i = \frac{FR}{8L_l} \tag{C.51}$$

and

$$M_i = \frac{FR}{8} \tag{C.52}$$

A free body diagram of a single leg is shown in figure C.11, sum the moments at the base of the cantilever

$$M_i' = -\frac{FR}{4} \tag{C.53}$$

which is the maximum bending moment on the beam. Substitute into the equation for the stress, gives,

$$\sigma = \frac{3FR}{2h_lw_l^2} \tag{C.54}$$

Now we must also consider the stress from the tangent force

$$\sigma_l = \frac{3w_lL_p^3L_lF}{2(w_l^3h_lL_p^3 + 2w_ph_pL_l^3)} \tag{C.55}$$

The sum, therefore, will be the maximum stress on the beam

$$\sigma_l = \left[\frac{3R}{2h_lw_l^2} + \frac{3w_lL_p^3L_l}{2(w_l^3h_lL_p^3 + 2w_ph_pL_l^3)}\right]F \tag{C.56}$$

C.4.10 Case 6

Suppose the fingertip is subject to tangent force on the outer radius of the hemisphere. As in the previous case, we must again consider the superposition of two stresses, since the stress due to the induce moment

and the vertical force superimpose on one of the legs of the cross. From the analysis in the first case the stress was

$$\sigma_{max} = \frac{3FL}{4w_l h_l^2} \tag{C.57}$$

and from case 2, substitute R for H_1 yields

$$\sigma_{max} = \frac{3F(R+H-H_1)L_l}{w_l h_l^2 \left[L_l + \frac{9L_l c_2 G w_l^2}{E h_l^2}\right]} \tag{C.58}$$

The maximum stress in this case is therefore

$$\sigma_{max} = \left[\frac{3L}{4w_l h_l^2} + \frac{3RL_l}{w_l h_l^2 \left[L_l + \frac{9L_l c_2 G w_l^2}{E h_l^2}\right]}\right] F \tag{C.59}$$

C.4.11 Maximum stress

The maximum stresses under the different loading modes of all six case are listed below

$$\begin{aligned}
\sigma_{max} &= \frac{3FL}{4w_l h_l^2} \\
\sigma_{max} &= \frac{3w_p L_l^3 L_p F}{2(w_p^3 h_p L_l^3 + 2w_l^3 h_l L_p^3)} \\
\sigma_{max} &= \frac{3FH_1 L_l}{w_l h_l^2 \left[L_l + \frac{9L_l c_2 G w_l^2}{E h_l^2}\right]} \\
\sigma_{max} &= \frac{3F(R+H-H_1)L_l}{w_l h_l^2 \left[L_l + \frac{9L_l c_2 G w_l^2}{E h_l^2}\right]} \\
\sigma_{max} &= \left[\frac{3R}{2h_l w_l^2} + \frac{3w_l L_p^3 L_l}{2(w_l^3 h_l L_p^3 + 2w_p^3 h_p L_l^3)}\right] F \\
\sigma_{max} &= \left[\frac{3L}{4w_l h_l^2} + \frac{3RL_l}{w_l h_l^2 \left[L_l + \frac{9L_l c_2 G w_l^2}{E h_l^2}\right]}\right] F
\end{aligned} \tag{C.60}$$

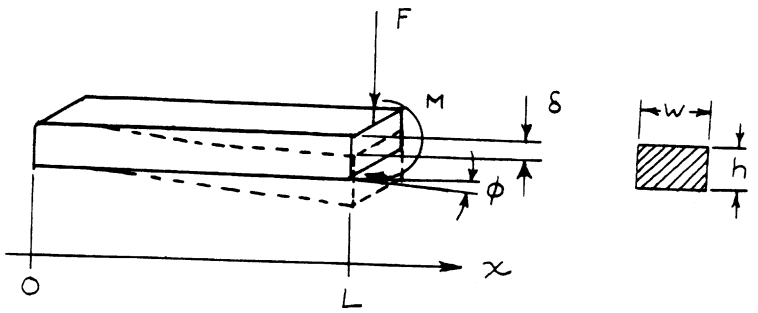


Figure C.4: Cantilever beam subject to a force F and a moment M at its end point.

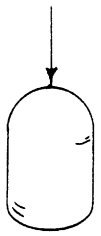


Figure C.5: Force applied on the top of the hemisphere along the axis of the sensor

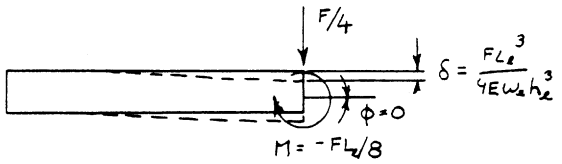


Figure C.6: Leg of the loadcell under a symmetrically applied vertical load

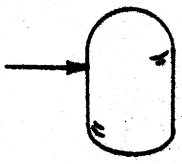


Figure C.7: A horizontal force is applied to the outside of the sensor in line with the inner loadcell

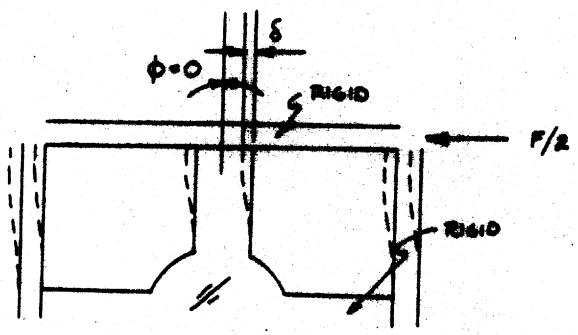


Figure C.8: The members under compressive stress are assumed to be rigid and since the force is applied symmetrically to the loadcell it is assumed to move rigidly and not rotate.

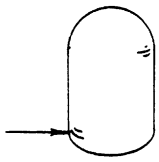


Figure C.9: Horizontal force applied to the lowest portion of the fingertip shell

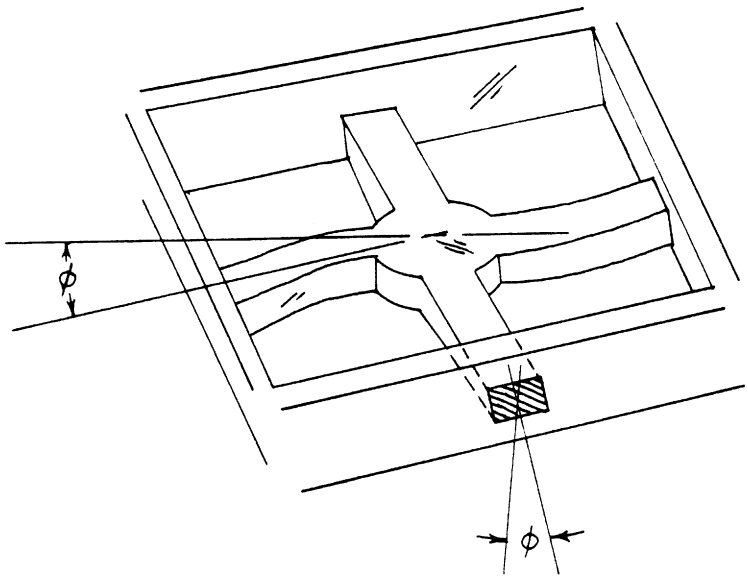


Figure C.10: The outer phalanges are all assumed to be rigid. The major stress components are located on the upper and lower surfaces on the legs parallel to the externally applied force at the cantilevered end.

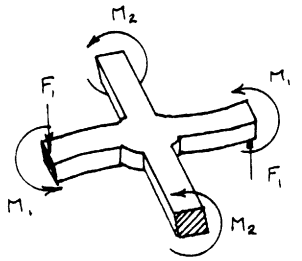


Figure C.11: The legs parallel to the applied force are subject to both a force and a moment at their end points, while the legs perpendicular to the force are only subject to a pure torsional moment.

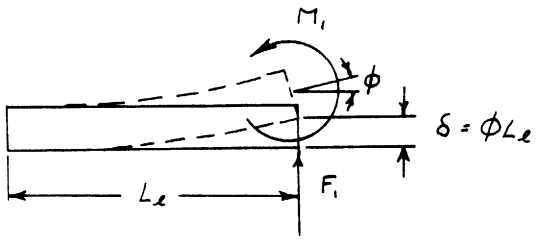


Figure C.12: The legs parallel to the applied force undergo a rotation at the end point of ϕ and a displacement of ϕL_l, under an applied force F_1 and a moment M_1.

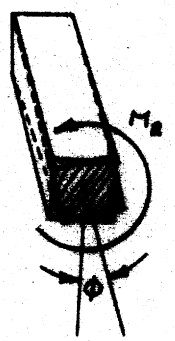


Figure C.13: The leg perpendicular to the applied external force twists to an angle ϕ subject to an applied moment M_z.

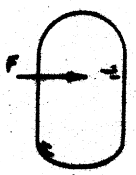


Figure C.14: A horizontal force applied tangent to the surface of the fingertip in the same plane as the loadcell

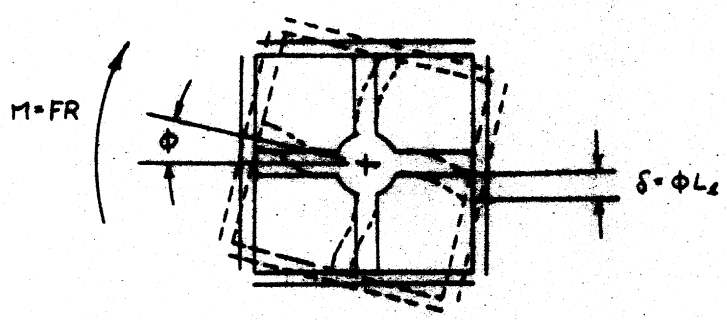


Figure C.15: The outer phalanges are all assumed to be rigid and rotate rigidly to an angle ϕ under and applied moment M. The major stress components are therefore located on sides of the legs of the cross

Appendix D

Slip analysis software

D.1 Introduction

The software was not included just to add bulk to this master's thesis. In fact, the construction of the LISP functions along with the documentation are similar to the development of the chapters in the thesis. The high level functions are listed at the beginning of every section, while the lower level supporting function are listed in a separate section, one for each high level function.

D.2 GRASP

```
;;; -*- Mode:LISP; Base:10; Syntax: Common-lisp; Package: USER -*-
;
;
;                                FILE:   CONSTRAINT.LISP
;
;     This file contains functions which analysis the relationship between a grasped obj!
ect
;  and grasping articulators.  The file is divided into six major sections:
;
;        1.  General functions and variables
;
;               This section contains functions and variables used by all the other
;            sections in the file.  Many of the variables will come from the sensing
;            on the hand. For example *contact-points*, from the fingertip and
;            joint sensors.  Other variables must be specified, but may in the
;            in the future be sensed as well.  For instance, *cg* the center of
;            gravity of the object, and *contact-surfaces* the local surface shape
;            in the area of the contact.
;
;            1.1 Define general global variables
;
;            1.2 Set initial values of the general global variables
;
;            1.3 General utility functions
;                1.3.1 General math functions
;                1.3.2 General matrix manipulation functions
;                      1.3.2.1 Major manipulation functions
;                              a. ADD-ARRAY-LIST
;                              b. MULTIPLY-ARRAY-LIST
;                              c. ROW-REDUCED-ECHELON-MATRIX
;                              d. TRANSFORM-DIRECTION-GLOBAL-FRAME
;                              e. TRANSFORM-DIRECTION-FRAME-GLOBAL
;                              f. TRANSFORM-POINTS-GLOBAL-FRAME
;                              g. TRANSFORM-POINTS-FRAME-GLOBAL
;                              h. TRANSFORM-FRAMES-GLOBAL-FRAME
;                              i. TRANSFORM-FRAMES-FRAME-GLOBAL
;                              j. GENERATE-FRAME
;                              k. GENERATE-FRAME-POINTS
;                      1.3.2.2 Auxiliary functions
;                              a. ADD-ARRAY-LIST
;                              b. MULTIPLY-ARRAY-LIST
;                              c. ROW-REDUCED-ECHELON-MATRIX
;                              d. TRANSFORM-DIRECTION-GLOBAL-FRAME
;                              e. TRANSFORM-DIRECTION-FRAME-GLOBAL
;                              f. TRANSFORM-POINTS-GLOBAL-FRAME
;                              g. TRANSFORM-POINTS-FRAME-GLOBAL
;                              h. TRANSFORM-FRAMES-GLOBAL-FRAME
;                              i. TRANSFORM-FRAMES-FRAME-GLOBAL
;                              j. GENERATE-FRAME
;                              k. GENERATE-FRAME-POINTS
;                              l. General
;
;                1.3.3 General matrix output functions
;                      1.3.3.1 Major matrix output functions
;                              a. PRINT-ARRAY-LIST
;                      1.3.3.2 Auxiliary functions
;                              a. PRINT-ARRAY-LIST
;                              b. General
;
;            1.4 Graphics functions
;                1.4.1 Define graphics variables
;                1.4.2 Defind functions to set graphics variables
;                1.4.3 Set graphics functions
;                1.4.4 Graphics functions
;                      1.4.4.1 Screen creation and initialization functions
;                              a. MAKE-GRASP-SCREEN
```

```
;                              b. KILL-GRASP-SCREEN
;                              c. START-MONITOR-MSG
;                              d. CREATE-GRASP-SCREEN
;                              e. CLEARSCREEN
;                        1.4.4.2 Drawing functions
;                              a. DRAW-2D-GRASP-WINDOW
;                              b. DRAW-3D-GRASP-WINDOW
;                              c. SPHERE-3D-GRASP-WINDOW
;                              d. DRAW-COORDINATE-SYSTEM
;                              e. DRAW-COORDINATES
;                              f. DRAW-3D-LIST
;                   1.4.5 Auxiliary graphics functions

;       2.  Sensed and global variables
;
;             This section defines and initializes sensed and global variables used in
;           the other sections.  The only variables which are truely sensed externally ar!
e:
;
;                     *contact-points*     a list of contact-points in the hand space
;                     *contact-normals*    a list of normals at the contact points in han!
d
;                                          space.
;
;           The force and tactile sensing fingertip need to be working for these variable!
s
;           to be read. Until then  these variables will have to be constructed. So there
;           are a number of functions and variables which are used to construct the varia!
bles
;            *contact-points* and *contact-normals*.  These functions and variables are
;           temporary and are used only for simulation.  When the sensors are connected
;           these variables and functions will have to be removed.
;
;                                Temporary variables used to construct *contact-points*
;
;                     *contact-points-object-space*   a list of contact points in the
;                                                     object space
;                     *contact-normals-object-space*  a list of contact normals in the
;                                                     object space
;                     *cg*                            the center of gravity of the object
;                     *major-axis*                    the normal indicating the direction!
 of
;                                                     major axis of an axisymmetric objec!
t
;

;       3.  Constraint functions
;
;             This section contain functions which analysis the constraint created by
;           the contacts on the grasped object.  An infinitesimal analysis involving
;           virtual work and a finite motion analysis involving the shape of contact
;           surface are both included in determine the overall constraint imposed by
;           the contacts
;
;           3.1 Define constraint variables
;
;           3.2 Set constraint variables
;               3.2.1 Functions for setting constraint variables
;
;           3.3 Constraint analysis functions
;               3.3.1 Major analytic functions
;                     a. CONSTRUNT-VIRTUAL-WORK-LIST
;                     b. DETERMINE-CONTACT-TYPES
;               3.3.1 Auxiliary functions

;       4.  Body wrench
```

```
;
;              In this section the wrench on the object resulting from forces excluding
;            contact forces from the robot is calculated.  Body wrenches may result from
;            gravity, accelerations, electromagnetic forces, and contacts other than
;            fingertips.  The only body wrenches calculated in the present program are
;            those resulting from gravity.
;
;            4.1 Define body wrench variables
;
;            4.2 Set body wrench variables
;
;            4.3 Body wrench functions
;                4.3.1 Major body wrench functions
;                      a. ORIENTATION
;                4.3.2 Auxiliary functions
;                      a. ORIENTATION

;        5.  Contact wrenches
;
;              The wrenches in the contact space for each contact are calculated assumin!
g
;            a certain stiffness at the fingertips, a body wrench (section 3), and an off!
set
;            wrench (section 6).
;
;            5.1 Define contact wrench variables
;
;            5.2 Set contact wrench variables
;                5.2.1 Functions to set contact wrench variables
;
;            5.3 Contact wrench functions
;                5.3.1 Major contact wrench functions
;                      a. CONSTUCT-CONTACT-WRENCH
;                5.3.2 Auxiliary functions
;                      a. CONSTUCT-CONTACT-WRENCH

;        6.  Contact types
;
;              In this section the type of contact is determined given certain friction!
al
;            criteria and the wrench in the contact space.  The different types of contac!
ts
;            are
;                                    (1) Soft finger contact
;                                    (2) Point contact with friction
;                                    (3) Point contact without friction
;                                    (4) No contact
;
;            6.1 Define contact type variables
;
;            6.2 Set contact type variables
;
;            6.3 Contact type functions
;                6.3.1 Major contact type functions
;                      a. CONTACT-TYPE
;                6.3.2 Auxiliary functions
;                      a. CONTACT-TYPE

;        7. Offset wrench
;
;              The internal grasping force exerted by the contacts on an object are not
;            determined by external forces and may be varied arbitarily on an object.
;            The space of possible solutions to the grasping force problem varies with
;            the number of contacts:
;
;                          Number of contacts          Dimension of solution space
```

```
;
;                              1                              0
;                              2                              1
;                              3                              3
;                              4                              6
;                              n                              3*(n-2) n>=3
;
;             For a two fingered grasp, the dimension of the solution space on the
;          internal grasping forces is one.  That is, in general, the squeeze force
;          between the two fingers may be varied.
;             For three fingers there is a three dimensional space of solutions.
;          Conceptually this space can be simplified.  If there are three forces
;          exerted on an object, and at least one force is not parallel to the others,
;          and the object is not accelerating, then the three
;          forces intersect at a point in space.  Furthermore, these intersection point!
s
;          lie on the plane formed by the three contacts points.  The three dimensional
;          solution space of grasping forces is then a point (X,Y) lying on the graspin!
g
;          force plane, and F the grasping force magnitude.
;             For more than three fingers, the solution space grows by 3*(n-2), where n
;          is the number of contacts.
;
;          7.1 CONSTRUCT-OFFSET-WRENCH
;
;          7.2 OFFSET-WRENCH-THREE-CONTACTS

;       8. Controlled slip
;
;             This is an experimental section formed of functions which use the functio!
ns
;          of the previously descibed sections.  The number of controllable variables t!
o
;          execute dexterous control of an object within a grasp of a three fingered ha!
nd
;          is enormous:
;
;                 Controllable variables                  Dimension of space
;
;                    Grasping force                              3
;                    Orientation                                 3
;                    Stiffness                                   9
;
;           (more here)
;
;           8.1  Define controlled slip variables
;
;           8.2  Set controlled slip variables
;
;           8.3  Controlled slip variables
;
;                8.3.1 PERMISSIBLE TWIST
;                            accepts: twist
;                            assumes: orientation
;                                     grasping force center
;                            returns: maximally constrained state which allows the
;                                        allows the specified twist
;                                     orientation
;                                     grasping force magnitude
;
;                8.3.2 DETERMINE-CONSTRAINT-STATE accepts: grasping force center
;                                                          grasp force magnitude
;                                                 assumes: orientation
;                                                 returns: constraint state
;
;                8.3.3 MAP-CONSTRAINT-SPACE
```

```
;                              accepts: force-magnitude
;                              assumes: orientation
;                                       stiffness
;                              returns: two-dimensional map of constrain states
;                                       as a function of (x,y) the grasping force
;                                       center
;


;        9. Hand actuation functions
;
;                This section contains functions to perform actual motions of the hand.
;
;            9.1  Define actuations variables
;
;            9.2  Define functions to set actuation functions
;
;            9.3  Set actuation variables
;
;            9.4  Grasp functions
;                 9.4.1 MOVE-TO-CONTACT-POINTS assumes: *contact-frames*
;                                              returns: moves fingers to points defined !
in *contact-frames*
;                 9.4.2 GRASP accepts: grasping force center
;                                      grasp force magnitude
;                             assumes: stiffness
;                             returns: motion of the fingers
;
;            9.5 Grasp auxiliary functions
;                9.5.1 MOVE-TO-CONTACT-POINTS auxiliary functions
;                9.5.2 GRASP auxiliary funcitons
;
;        10. Menu
;
;            Standard motion options:
;
;            Screen graphics options:
;
;                    CLEARSCREEN                :clears "grasp-window" pane of "grasp-scr!
een"
;                                                screen
;                    DRAW COORDINATE SYSTEM     :draws the three dimensional coordinate s!
ystems
;                                                for the hand, object, and contacts
;                    MAP CONSTRAINT SPACE       :plots the constraint states on the two
;                                                dimensional grasp surface for a specifie!
d
;                                                grasp force magnitude
;
;            Controlled slip options:
;
;                    RESET GLOBAL VARIABLES     :shows the current values of the global
;                                                variables and allows the user to change !
them
;                    DETERMINE CONSTRAINT STATE:accepts the grasping force center and th!
e
;                                                grasping force magnitude
;                                                assumes the stiffness and orientation
;                                                and returns the constraint state
;                    PERMISSIBLE TWIST          :accepts a twist
;                                                assumes an orientation
;                                                and returns the maximally constrained st!
ates
;                                                which allows that twist
;            Hand actuations options:
;
```

```
;
;                    MOVE TO CONTACT POINTS    :moves fingers to the points cooresspondi!
ng to
;                                               those in *contact-frames*
;
;                    PICK GRASP FORCE CENTER   :plots the contacts on the two dimensiona!
l
;                                               grasp surface and allows the user to cho!
ose
;                                               a particular force magnitude and use the!
 joy
;                                               stick to pick a force center and actuate!
s the
;                                               hand accordingly
;
;
;           Advanced hand actuation options
;
;                    CONTROLLED SLIP           :allows the user to enter an object motio!
n
;                                               and the hand tries to actuate it.

;
;           11. Demos
;
;               This section contains demo programs which are based on the analyses of t!
he
;           the previous section.
;

;;;*****************************************************************************!
*****
;
;                                          1. General functions and variables
;
;;;*****************************************************************************!
*****
;                                          1.1 Define variables
;                                          1.2 Set default variable values section
;                                          1.2.1 Functions to set default variable values
;                                          1.2.2 Set default variable values
;                                          1.3 General utility functions

;                                          1.3.1 General math functions
;                                          1.3.1.1 SQR
(defun sqr (x)
  (cond ((not (numberp x)) 0)
        (t (* x x))))

;                                          1.3.2 General matrix manipulation functions
;                                          1.3.2.1 ADD-ARRAY-LIST
(defun add-array-list (array-list)
  (cond ((null array-list) nil)
         ((atom array-list) array-list)
         (t (add-two-arrays (add-array-list (car array-list))
                            (add-array-list (cdr array-list))))))

;                                          1.3.2.1.A ADD-ARRAY-LIST Auxiliary
(defun add-two-arrays (array-1 array-2)
  (cond ((and (not array-1) (not array-2)) nil)
        ((not array-1) array-2)
        ((not array-2) array-1)
        ((vectorp array-1)
         (let* ((elements (array-dimension array-1 0))
                (sum-array (make-array elements)))
           (do ((i 0 (+ i 1)))
```

```
              ((= i elements) sum-array)
            (setf (aref sum-array i) (+ (aref array-1 i) (aref array-2 i))))))
        (t (let* ((rows (array-dimension array-1 0))
                  (columns (array-dimension array-2 1))
                  (sum-array (make-array (list rows columns))))
             (do (( i 0 (+ i 1)))
                 ((= i rows) sum-array)
               (do ((j 0 (+ j 1)))
                   ((= j columns) nil)
                 (setf (aref sum-array i j) (+ (aref array-1 i j) (aref array-2 i j))))))!
))))

;                                          1.3.2.2 MULTIPLY-ARRAY-LIST
(defun multiply-array-list (array-list)
  (cond ((null array-list) nil)
        ((atom array-list) array-list)
        (t (multiply-two-arrays (multiply-array-list (car array-list))
                                (multiply-array-list (cdr array-list))))))

;                                          1.3.2.2.A MULTIPLY-ARRAY-LIST Auxiliary
(defun multiply-two-vectors (array-1 array-2)
  (let* ((elements-1 (array-dimension array-1 0))
         (elements-2 (array-dimension array-2 0))
         (product-array (initialize-array elements-1 elements-2)))
    (cond ((not (= elements-1 elements-2)) nil)
          (t
           (do ((j 0 (+ j 1)))
               ((= j elements-1) product-array)
             (do ((k 0 (+ k 1)))
                 ((= k elements-1) nil)
               (setf (aref product-array j k)
                     (* (aref array-1 j) (aref array-2 k)))))))))

(defun multiply-array-vector (array vector)
  (let* ((rows-1 (array-dimension array 0))
         (elements (array-dimension vector 0))
         (product-array (initialize-vector elements)))
    (do ((j 0 (+ j 1)))
        ((= j rows-1) product-array)
      (do ((k 0 (+ k 1)))
          ((= k elements) nil)
        (setf (aref product-array j)
              (+ (aref product-array j)
                 (* (aref array j k) (aref vector k))))))))

(defun multiply-two-arrays (array-1 array-2)
  (cond ((and (not array-1) (not array-2))  nil)
        ((not array-1) array-2)
        ((not array-2) array-1)
        ((and (not (arrayp array-1)) (not (arrayp array-2))) (* array-1 array-2))
        ((not (arrayp array-1)) (multiply-constant-array array-1 array-2))
        ((not (arrayp array-2)) (multiply-constant-array array-2 array-1))
        ((vectorp array-1)
         (cond ((not (= (array-rank array-2) 1)) nil)
               (t (multiply-two-vectors array-1 array-2))))
        ((not (equalp (array-dimension array-1 1)
                      (array-dimension array-2 0))) nil)
        ((vectorp array-2) (multiply-array-vector array-1 array-2))
        (t  (let* ((rows-1 (array-dimension array-1 0))
                   (rows-2 (array-dimension array-2 0))
                   (columns-2 (array-dimension array-2 1))
                   (product-array (initialize-array rows-1 columns-2)))
              (do (( i 0 (+ i 1)))
                  ((= i columns-2) product-array)
                (do ((j 0 (+ j 1)))
                    ((= j rows-1) nil)p
```

```
                (do ((k 0 (+ k 1)))
                    ((= k rows-2) nil)
                  (setf (aref product-array j i)
                        (+ (aref product-array j i)
                           (* (aref array-1 j k) (aref array-2 k i))))))))))))

(defun multiply-constant-array (constant array)
  (cond ((vectorp array)
         (let* ((elements (array-dimension array 0))
                (product-array (make-array elements)))
           (do ((i 0 (+ i 1)))
               ((= i elements) product-array)
             (setf (aref product-array i) (* constant (aref array i))))))
        (t
         (let* ((rows (array-dimension array 0))
                (columns (array-dimension array 1))
                (product-array (make-array (list rows columns))))
           (do ((i 0 (+ i 1)))
               ((= i rows) product-array)
             (do ((j 0 (+ j 1)))
                 ((= j columns) nil)
               (setf (aref product-array i j) (* constant (aref array i j)))))))))

;                                          1.3.2.3 ROW-REDUCED-ECHELON
(defun row-reduced-echelon (array)
  (let ((rows (array-dimension array 0))
        (columns (array-dimension array 1)))
    (row-reduced-echelon-aux (diagonalize array) ( - rows 1) 0 rows columns)))

;                                          1.3.2.3.A ROW-REDUCED-ECHELON Auxiliary
(defun zero-array (rows columns)
  (cond ((or  (< rows 0) (< columns 0)) nil)
        ((and (= rows 0) (= columns 0)) nil)
        ((= rows 0) (zero-vector columns))
        ((= columns 0) (zero-vector rows))
        (t
         (let ((array (make-array (list rows columns))))
           (do ((i 0 (+ i 1)))
               ((= i rows) array )
             (do ((j 0 (+ j 1)))
                 ((= j columns))
               (setf (aref array i j) 0)))))))

(defun zero-vector (elements)
  (cond ((<= elements 0) nil)
        (t
         (let ((vector (make-array elements)))
           (do ((i 0 (+ i 1)))
               ((= i elements) vector)
             (setf (aref vector i) 0))))))

(defun row-reduced-echelon-aux (array current-row current-column rows columns)
  (cond ((= current-row 0) array)
        ((zero-row array current-column current-row)
         (row-reduced-echelon-aux array (- current-row 1) 0 rows columns))
        ((= current-column columns)
         (row-reduced-echelon-aux array (- current-row 1) 0 rows columns))
        ((< (sqr (aref array current-row current-column)) 0.00001)
         (row-reduced-echelon-aux array current-row (+ 1 current-column) rows columns))
        (t (row-reduced-echelon-aux
             (row-reduced-echelon-aux-1 array current-row current-row current-column row!
s )
           (-  current-row 1)
           0 rows columns))))

(defun row-reduced-echelon-aux-1 (array inc-row current-row current-column rows)
```

```
  (cond ((= inc-row 0) array)
        (t
         (row-reduced-echelon-aux-1
          (add-x-times-r1-row-to-r2-row array current-row current-column (- inc-row 1))
          (- inc-row 1) current-row current-column rows))))

(defun diagonalize (array)
  (let ((rows (array-dimension array 0))
        (columns (array-dimension array 1)))
    (diagonalize-array-aux array 0 0 rows columns)))

(defun zero-column (array current-column current-row)
  (let ((rows (array-dimension array 0))
        (sum 0))
    (do ((i current-row (+ i 1)))
        ((= i rows) (< (sqr sum) 0.00001))
         (setq sum (+ sum (abs (aref array i current-column)))))))

(defun zero-row (array current-column current-row)
  (let ((columns (array-dimension array 1))
        (sum 0))
    (do ((i current-column (+ i 1)))
        ((= i columns) (< (sqr sum) 0.00001))
      (setq sum (+ sum (abs (aref array current-row i)))))))

(defun diagonalize-array-aux (array current-column current-row rows columns)
  (cond ((= current-column (min rows columns)) array)
        ((zero-column array current-column current-row)
         (diagonalize-array-aux array (+ 1 current-column) current-row rows columns))
        (t
         (diagonalize-array-aux
             (initialize-column-aux
               (divide-r-row-by-element-rc array current-row current-column)
               current-column current-row (+ 1 current-row) rows)
             (+ 1 current-column)
             (+ 1 current-row)
             rows
             columns))))

(defun initialize-column-aux (array initial-column initial-row current-row rows)
  (cond ((= current-row rows) array)
        (t
         (initialize-column-aux
           (add-x-times-r1-row-to-r2-row array
               initial-row initial-column current-row)
             initial-column
             initial-row
             (+ 1 current-row)
             rows))))

(defun divide-r-row-by-element-rc (array r c)
  (let ((columns (array-dimension array 1))
        (first-element (aref array r c))
        (new-array (initialize array)))
    (cond ((<  (sqr first-element) 0.00001)
           (divide-r-row-by-element-rc (interchange-last-row-and-row-r array r) r c))
          (t
           (do ((i c (+ i 1)))
               ((= i columns) new-array)
             (setf (aref new-array r i) (/ (aref array r i) first-element)))))))


(defun add-x-times-r1-row-to-r2-row (array r1 c1 r2)
  (cond ((< (sqr (aref array r1 c1)) 0.00001) array)
        (t
         (let ((columns (array-dimension array 1))
```

```
            (new-array (initialize array))
            (constant (* -1 (aref array r2 c1))))
          (do ((i c1 (+ i 1)))
              ((= i columns) new-array)
            (setf (aref new-array r2 i) (+ (* constant (aref array r1 i))
                                           (aref array r2 i))))))))

(defun interchange-last-row-and-row-r (array r)
  (let ((last-row (- (array-dimension array 0) 1))
        (columns (array-dimension array 1))
        (new-array (initialize array)))
    (do ((i 0 (+ i 1)))
        ((= i columns) new-array)
      (setf (aref new-array r i) (aref array last-row i))
      (setf (aref new-array last-row i) (aref array r i)))))

(defun transpose-contact-frame (contact-frame)
  (let* ((t-contact-frame (initialize contact-frame)))
    (do ((i 0 (+ i 1)))
        ((= i 3) t-contact-frame)
      (do ((j 0 (+ j 1)))
          ((= j 3))
        (setf (aref t-contact-frame i j) (aref contact-frame j i))))))

;                                          1.3.2.4 TRANSFORM-DIRECTION-GLOBAL-FRAME
(defun transform-direction-global-frame (frames objects)
  (cond ((and (listp objects) (listp frames))
         (transform-direction-global-frame-lists frames objects))
        ((listp objects)
         (transform-direction-global-frame-list frames objects))
        (t
         (transform-direction-object-global-frame frames objects))))

;                                          1.3.2.4.A TRANSFORM-DIRECTION-GLOBAL-FRAME Auxi!
liary
(defun transform-direction-global-frame-lists (frames objects)
  (cond ((null frames) nil)
        (t (cons (transform-direction-object-global-frame (car frames) (car objects))
                 (transform-direction-global-frame-lists (cdr frames) (cdr objects))))))

(defun transform-direction-global-frame-list (frame objects)
  (cond ((null objects) nil)
        (t (cons (transform-direction-object-global-frame frame (car objects))
                 (transform-direction-global-frame-list frame (cdr objects))))))
(defun transform-direction-object-global-frame (frame object)
  (cond ((vectorp object) (transform-direction-vector-global-frame frame object))
        ((arrayp object) (transform-direction-array-global-frame frame object))
        (t nil)))

(defun transform-direction-array-global-frame (frame array)
  (let* ((rows (array-dimension array 0))
         (columns (array-dimension array 1))
         (new-array (make-array (list rows columns))))
    (do ((i 0 (+ i 3)))
        ((= i rows) new-array)
      (do ((j 0 (+ j 1)))
          ((= j columns))
        (do ((k 0 (+ k 1)))
            ((= k 3))
          (setf (aref new-array (+ i k) j)
                (+ (* (aref array i j)       (aref frame 0 k))
                   (* (aref array (+ i 1) j) (aref frame 1 k))
                   (* (aref array (+ i 2) j) (aref frame 2 k)))))))))

(defun transform-direction-vector-global-frame (frame vector)
  (let* ((elements (array-dimension vector 0))
```

```
         (new-vector (make-array elements)))
    (do ((i 0 (+ i 3)))
        ((= i elements) new-vector)
      (do ((k 0 (+ k 1)))
          ((= k 3))
        (setf (aref new-vector (+ i k))
              (+ (* (aref vector i)       (aref frame 0 k))
                 (* (aref vector (+ i 1)) (aref frame 1 k))
                 (* (aref vector (+ i 2)) (aref frame 2 k)))))))

;                                          1.3.2.5 TRANSFORM-DIRECTION-FRAME-GLOBAL
(defun transform-direction-frame-global (frames objects)
  (cond ((and (listp objects) (listp frames))
         (transform-direction-frame-global-lists frames objects))
        ((listp objects)
         (transform-direction-frame-global-list frames objects))
        (t
         (transform-direction-object-frame-global frames objects))))

;                                          1.3.2.5.A TRANSFORM-DIRECTION-FRAME-GLOBAL Auxi!
liary
(defun transform-direction-frame-global-lists (frames objects)
  (cond ((null frames) nil)
        (t (cons (transform-direction-object-frame-global (car frames) (car objects))
                 (transform-direction-frame-global-lists (cdr frames) (cdr objects))))))

(defun transform-direction-frame-global-list (frame objects)
  (cond ((null objects) nil)
        (t (cons (transform-direction-object-frame-global frame (car objects))
                 (transform-direction-frame-global-list frame (cdr objects))))))

(defun transform-direction-object-frame-global (frame object)
  (cond ((vectorp object) (transform-direction-vector-frame-global frame object))
        ((arrayp object) (transform-direction-array-frame-global frame object))
        (t nil)))

(defun transform-direction-array-frame-global (frame array)
  (let* ((rows (array-dimension array 0))
         (columns (array-dimension array 1))
         (new-array (make-array (list rows columns))))
    (do ((i 0 (+ i 3)))
        ((= i rows) new-array)
      (do ((j 0 (+ j 1)))
          ((= j columns))
        (do ((k 0 (+ k 1)))
            ((= k 3))
          (setf (aref new-array (+ i k) j)
                (+ (* (aref array i j)       (aref frame k 0))
                   (* (aref array (+ i 1) j) (aref frame k 1))
                   (* (aref array (+ i 2) j) (aref frame k 2)))))))))

(defun transform-direction-vector-frame-global (frame vector)
  (let* ((elements (array-dimension vector 0))
         (new-vector (make-array elements)))
    (do ((i 0 (+ i 3)))
        ((= i elements) new-vector)
      (do ((k 0 (+ k 1)))
          ((= k 3))
        (setf (aref new-vector (+ i k))
              (+ (* (aref vector i)       (aref frame k 0))
                 (* (aref vector (+ i 1)) (aref frame k 1))
                 (* (aref vector (+ i 2)) (aref frame k 2)))))))

;                                          1.3.2.6 TRANSFORM-POINTS-GLOBAL-FRAME
(defun transform-points-global-frame (frames objects)
  (cond ((and (listp objects) (listp frames))
```

```
         (transform-points-global-frame-lists frames objects))
        ((listp objects)
         (transform-points-global-frame-list frames objects))
        (t
         (transform-points-object-global-frame frames objects))))

;                                        1.3.2.6.A TRANSFORMP-POINTS-FRAME-GLOBAL Auxili!
ary
(defun transform-points-global-frame-lists (frames objects)
  (cond ((null frames) nil)
        (t (cons (transform-points-object-global-frame (car frames) (car objects))
                 (transform-points-global-frame-lists (cdr frames) (cdr objects))))))

(defun transform-points-global-frame-list (frame objects)
  (cond ((null objects) nil)
        (t (cons (transform-points-object-global-frame frame (car objects))
                 (transform-points-global-frame-list frame (cdr objects))))))

(defun transform-points-object-global-frame (frame object)
  (cond ((vectorp object) (transform-points-vector-global-frame frame object))
        ((arrayp object) (transform-points-array-global-frame frame object))
        (t nil)))

(defun transform-points-array-global-frame (frame array)
  (let* ((rows (array-dimension array 0))
         (columns (array-dimension array 1))
         (new-array (make-array (list rows columns))))
    (do ((i 0 (+ i 3)))
        ((= i rows) new-array)
      (do ((j 0 (+ j 1)))
          ((= j columns))
        (do ((k 0 (+ k 1)))
            ((= k 3))
          (setf (aref new-array (+ i k) j)
                (+ (* (- (aref array i j) (aref frame 0 3))       (aref frame 0 k))
                   (* (- (aref array (+ i 1) j) (aref frame 1 3)) (aref frame 1 k))
                   (* (- (aref array (+ i 2) j) (aref frame 2 3)) (aref frame 2 k)))))))!
))

(defun transform-points-vector-global-frame (frame vector)
  (let* ((elements (array-dimension vector 0))
         (new-vector (make-array elements)))
    (do ((i 0 (+ i 3)))
        ((= i elements) new-vector)
      (do ((k 0 (+ k 1)))
          ((= k 3))
        (setf (aref new-vector (+ i k))
              (+ (* (- (aref vector i) (aref frame 0 3))       (aref frame 0 k))
                 (* (- (aref vector (+ i 1)) (aref frame 1 3)) (aref frame 1 k))
                 (* (- (aref vector (+ i 2)) (aref frame 2 3)) (aref frame 2 k)))))))

;                                        1.3.2.7 TRANSFORM-POINTS-FRAME-GLOBAL
(defun transform-points-frame-global (frames objects)
  (cond ((and (listp objects) (listp frames))
         (transform-points-frame-global-lists frames objects))
        ((listp objects)
         (transform-points-frame-global-list frames objects))
        (t
         (transform-points-object-frame-global frames objects))))

;                                        1.3.2.7.A TRANSFORM-POINTS-FRAME-GLOBAL Auxilia!
ry
(defun transform-points-frame-global-lists (frames objects)
  (cond ((null frames) nil)
        (t (cons (transform-points-object-frame-global (car frames) (car objects))
                 (transform-points-frame-global-lists (cdr frames) (cdr objects))))))
```

```
(defun transform-points-frame-global-list (frame objects)
  (cond ((null objects) nil)
        (t (cons (transform-points-object-frame-global frame (car objects))
                 (transform-points-frame-global-list frame (cdr objects))))))

(defun transform-points-object-frame-global (frame object)
  (cond ((vectorp object) (transform-points-vector-frame-global frame object))
        ((arrayp object) (transform-points-array-frame-global frame object))
        (t nil)))

(defun transform-points-array-frame-global (frame array)
  (let* ((rows (array-dimension array 0))
         (columns (array-dimension array 1))
         (new-array (make-array (list rows columns))))
    (do ((i 0 (+ i 3)))
        ((= i rows) new-array)
      (do ((j 0 (+ j 1)))
          ((= j columns))
        (do ((k 0 (+ k 1)))
            ((= k 3))
          (setf (aref new-array (+ i k) j)
                (+ (* (aref array i j)       (aref frame k 0))
                   (* (aref array (+ i 1) j) (aref frame k 1))
                   (* (aref array (+ i 2) j) (aref frame k 2))
                   (aref frame k 3)))))))))

(defun transform-points-vector-frame-global (frame vector)
  (let* ((elements (array-dimension vector 0))
         (new-vector (make-array elements)))
    (do ((i 0 (+ i 3)))
        ((= i elements) new-vector)
      (do ((k 0 (+ k 1)))
          ((= k 3))
        (setf (aref new-vector (+ i k))
              (+ (* (aref vector i)       (aref frame k 0))
                 (* (aref vector (+ i 1)) (aref frame k 1))
                 (* (aref vector (+ i 2)) (aref frame k 2))
                 (aref frame k 3)))))))

;                                        1.3.2.8 TRANSFORM-FRAMES-GLOBAL-FRAME
(defun transform-frames-global-frame (frames object-frames)
  (cond ((and (listp object-frames) (listp frames))
         (transform-frames-global-frame-lists frames object-frames))
        ((listp object-frames)
         (transform-frames-global-frame-list frames object-frames))
        (t
         (transform-frame-global-frame frames object-frames))))

;                                        1.3.2.8.A TRANSFORM-FRAMES-GLOBAL-FRAME Auxilia!
ry
(defun transform-frames-global-frame-lists (frames object-frames)
  (cond ((null frames) nil)
        (t (cons (transform-frames-global-frame (car frames) (car object-frames))
                 (transform-frames-global-frame-lists (cdr frames) (cdr object-frames)))!
)))

(defun transform-frames-global-frame-list (frame object-frames)
  (cond ((null object-frames) nil)
        (t (cons (transform-frame-global-frame frame (car object-frames))
                 (transform-frames-global-frame-list frame (cdr object-frames))))))

(defun transform-frame-global-frame (frame object-frame)
  (let* ((new-object-frame (make-array '(3 4))))
    (do ((i 0 (+ i 1)))
        ((= i 3))
```

```
      (do ((j 0 (+ j 1)))
          ((= j 3))
        (setf (aref new-object-frame j i)
              (+ (* (aref object-frame 0 i) (aref frame 0 j))
                 (* (aref object-frame 1 i) (aref frame 1 j))
                 (* (aref object-frame 2 i) (aref frame 2 j))))))
    (do ((i 0 (+ i 1)))
        ((= i 3) new-object-frame)
      (setf (aref new-object-frame i 3)
              (+ (* (- (aref object-frame 0 3) (aref frame 0 3)) (aref frame 0 i))
                 (* (- (aref object-frame 1 3) (aref frame 1 3)) (aref frame 1 i))
                 (* (- (aref object-frame 2 3) (aref frame 2 3)) (aref frame 2 i)))))))

;                                          1.3.2.9 TRANSFORM-FRAMES-FRAME-GLOBAL
(defun transform-frames-frame-global (frames object-frames)
  (cond ((and (listp object-frames) (listp frames))
         (transform-frames-frame-global-lists frames object-frames))
        ((listp object-frames)
         (transform-frames-frame-global-list frames object-frames))
        (t
         (transform-frame-frame-global frames object-frames))))

;                                          1.3.2.9.A TRANSFORM-FRAMES-FRAME-GLOBAL Auxilia!
ry
(defun transform-frames-frame-global-lists (frames object-frames)
  (cond ((null frames) nil)
        (t (cons (transform-frame-frame-global (car frames) (car object-frames))
                 (transform-frames-frame-global-lists (cdr frames) (cdr object-frames)))!
)))

(defun transform-frames-frame-global-list (frame object-frames)
  (cond ((null object-frames) nil)
        (t (cons (transform-frame-frame-global frame (car object-frames))
                 (transform-frames-frame-global-list frame (cdr object-frames))))))

(defun transform-frame-frame-global (frame object-frame)
  (let* ((new-object-frame (make-array '(3 4))))
    (do ((i 0 (+ i 1)))
        ((= i 3))
      (do ((j 0 (+ j 1)))
          ((= j 3))
        (setf (aref new-object-frame j i)
              (+ (* (aref object-frame 0 i) (aref frame j 0))
                 (* (aref object-frame 1 i) (aref frame j 1))
                 (* (aref object-frame 2 i) (aref frame j 2))))))
    (do ((i 0 (+ i 1)))
        ((= i 3) new-object-frame)
      (setf (aref new-object-frame i 3)
              (+ (* (aref object-frame 0 3) (aref frame i 0))
                 (* (aref object-frame 1 3) (aref frame i 1))
                 (* (aref object-frame 2 3) (aref frame i 2))
                 (aref frame i 3))))))

;                                          1.3.2.10 GENERATE-FRAME
(defun generate-frame (points vectors)
  (cond ((null points) nil)
        ((atom points) (generate-frame-aux points vectors))
        (t (cons (generate-frame (car points) (car vectors))
                 (generate-frame (cdr points) (cdr vectors))))))

;                                          1.3.2.10 GENERATE-FRAME Auxiliary
(defun generate-frame-aux (point vector)
  (let* ((frame (make-array '(3 4)))
         (length (sqrt (+ (sqr (aref vector 0))
                          (sqr (aref vector 1))
                          (sqr (aref vector 2))))))
```

```
          (nx (/ (aref vector 0) length))
          (ny (/ (aref vector 1) length))
          (nz (/ (aref vector 2) length))
          (lx) (ly) (lz) (mx) (my) (mz))
      (cond ((= nx 0) (setq lx 1) (setq ly 0) (setq lz 0))
            ((= ny 0) (setq lx 0) (setq ly -1) (setq lz 0))
            (t (setq lx (sqrt (/ (sqr (/ ny nx)) (+ 1 (sqr (/ ny nx))))))
               (setq ly (/ (* -1 nx lx) ny))
               (setq lz 0)))
      (setq mx (- (* ny lz) (* nz ly)))
      (setq my (- (* nz lx) (* nx lz)))
      (setq mz (- (* nx ly) (* ny lx)))
      (setf (aref frame 0 0) lx)
      (setf (aref frame 0 1) mx)
      (setf (aref frame 0 2) nx)
      (setf (aref frame 0 3) (aref point 0))
      (setf (aref frame 1 0) ly)
      (setf (aref frame 1 1) my)
      (setf (aref frame 1 2) ny)
      (setf (aref frame 1 3) (aref point 1))
      (setf (aref frame 2 0) lz)
      (setf (aref frame 2 1) mz)
      (setf (aref frame 2 2) nz)
      (setf (aref frame 2 3) (aref point 2))
      frame))

;                                          1.3.2.11 GENERATE-FRAME-POINTS
(defun generate-frame-points (points-1 points-2)
  (cond ((null points-1) nil)
        ((atom points-1) (generate-frame-points-aux points-1 points-2))
        (t (cons (generate-frame-points (car points-1) (car points-2))
                 (generate-frame-points (cdr points-1) (cdr points-2))))))

;                                          1.3.2.11 GENERATE-FRAME-POINTS Auxiliary
(defun generate-frame-points-aux (point-1 point-2)
  (let* ((length (sqrt (+ (sqr (- (aref point-2 0) (aref point-1 0)))
                          (sqr (- (aref point-2 1) (aref point-1 1)))
                          (sqr (- (aref point-2 2) (aref point-1 2))))))
         (nx (/ (- (aref point-2 0) (aref point-1 0)) length))
         (ny (/ (- (aref point-2 1) (aref point-1 1)) length))
         (nz (/ (- (aref point-2 2) (aref point-1 2)) length))
         (normal (make-array '(3))))
    (setf (aref normal 0) nx)
    (setf (aref normal 1) ny)
    (setf (aref normal 2) nz)
    (generate-frame-aux point-1 normal)))

;                                          1.3.3 General-functions
;                                          1.3.3.1 COUNT-ATOMS
(defun count-atoms (list)
  (cond ((null list) 0)
        ((atom list) 1)
        ((+ (count-atoms (car list))
            (count-atoms (cdr list))))))

;                                          1.3.3.2 INITIALIZE-VECTOR
(defun initialize-vector (elements)
  (let ((vector (make-array elements)))
    (do ((i 0 (+ i 1)))
        ((= i elements) vector)
        (setf (aref vector i) 0))))

;                                          1.3.3.3 INITIALIZE-ARRAY
(defun initialize-array (rows columns)
  (let ((array (make-array (list rows columns))))
    (do ((i 0 (+ i 1)))
```

```
        ((= i rows) array)
      (do ((j 0 (+ j 1)))
          ((= j columns) nil)
        (setf (aref array i j) 0)))))

;                                          1.3.3.4 INITIALIZE
(defun initialize (array)
  (let* ((rows (array-dimension array 0))
         (columns (array-dimension array 1))
         (new-array (make-array (list rows columns))))
    (do ((i 0 (+ i 1)))
        ((= i rows) new-array )
      (do ((j 0 (+ j 1)))
          ((= j columns))
        (setf (aref new-array i j) (aref array i j))))))

;                                          1.3.3.5 TRANSPOSE
(defun transpose (array)
  (let* ((rows (array-dimension array 0))
         (columns (array-dimension array 1))
         (new-array (make-array (list columns rows))))
    (do ((i 0 (+ i 1)))
        ((= i rows) new-array)
      (do ((j 0 (+ j 1)))
          ((= j columns))
        (setf (aref new-array j i) (aref array i j))))))

;                                          1.3.3.6 GET-ARRAY-COLUMN
(defun get-array-column (array column)
  (let* ((rows (array-dimension array 0))
         (vector (make-array rows)))
    (do ((i 0 (+ 1 i)))
        ((= i rows) vector)
      (setf (aref vector i) (aref array i column)))))

;                                          1.3.3.7 ABS-ARRAY

(defun abs-array (array-list)
  (cond ((null array-list) nil)
        ((atom array-list) (abs-one-array array-list))
        (t (cons (abs-array (car array-list))
                 (abs-array (cdr array-list))))))

(defun abs-one-array (array)
  (cond ((vectorp array) (abs-one-vector array))
        (t   (let* ((rows (array-dimension array 0))
                    (columns (array-dimension array 1))
                    (new-array (make-array (list rows columns))))
               (do ((i 0 (+ i 1)))
                   ((= i rows) new-array)
                 (do ((j 0 (+ j 1)))
                     ((= j columns))
                   (setf (aref new-array i j) (abs (aref array i j)))))))))

(defun abs-one-vector (vector)
  (let* ((elements (array-dimension vector 0))
         (new-vector (make-array elements)))
    (do ((i 0 (+ i 1)))
        ((= i elements) new-vector)
      (setf (aref new-vector i) (abs (aref vector i))))))

;                                          1.3.3.7.A ABS-ARRAY Auxiliary
;                                          1.3.4 General matrix output functions
;                                          1.3.4.1 PRINT-ARRAY
(defun print-array (array-list)
  (cond ((listp array-list) (print-array-list array-list))
```

```
        (t (print-one-array array-list))))

;                                          1.3.4.1.A PRINT-ARRAY Auxiliary
(defun print-array-list (array-list)
  (cond ((null array-list) nil)
        (t (print-array (car array-list))
           (print-array-list (cdr array-list)))))

(defun print-one-array (array)
  (cond ((vectorp array) (print-vector array))
        (t (print-2d-array array))))

(defun print-2d-array (2d-array)
  (write-char #\newline)
  (write-char #\newline)
  (let ((rows (array-dimension 2d-array 0))
        (columns (array-dimension 2d-array 1)))
    (do ((i 0 (+ i 1)))
        ((= i rows) nil)
      (do ((j 0 (+ j 1)))
          ((= j columns) nil)
        (prin1 (aref 2d-array i j))
        (write-char #\space ))
      (write-char #\newline))))

(defun print-vector (vector)
  (write-char #\newline)
  (write-char #\newline)
  (let ((elements (array-dimension vector 0)))
    (do ((i 0 (+ i 1)))
        ((= i elements) nil)
        (prin1 (aref vector i))
        (write-char #\space ))))

;                                          1.4 Graphics functions

;                                          1.4.1 Define graphics variables
(defvar grasp-screen)
(defvar grasp-window)
(defvar view-frame)
(defvar scale-2d)
(defvar x-origin-2d)
(defvar y-origin-2d)

(defvar scale-3d)
(defvar x-origin-3d)
(defvar y-origin-3d)
(defvar angle-x)
(defvar angle-z)

;                                         1.4.2 Set default variable values section
;                                         1.4.2.1 Define functions to set graphics variabl!
es
(defun construct-view-frame (angle-x angle-z)
  (let* ((ct (cos angle-x))
         (st (sin angle-x))
         (cp (cos angle-z))
         (sp (sin angle-z))
         (view-frame (zero-array 3 4)))
    (setf (aref view-frame 0 0) cp)
    (setf (aref view-frame 0 1) (* -1 ct sp))
    (setf (aref view-frame 0 2) (* -1 st sp))
    (setf (aref view-frame 1 0) sp)
    (setf (aref view-frame 1 1) (* ct cp))
    (setf (aref view-frame 1 2) (* st cp))
    (setf (aref view-frame 2 0) 0)
```

```
    (setf (aref view-frame 2 1) (* -1 st))
    (setf (aref view-frame 2 2) ct)
    view-frame))


;                                          1.4.2.2 Set graphics variables
(setq scale-2d 100)
(setq x-origin-2d 300)
(setq y-origin-2d 300)

(setq scale-3d 200)
(setq x-origin-3d 300)
(setq y-origin-3d 500)
(setq angle-x 0.400)
(setq angle-z 0.400)

(setq view-frame (construct-view-frame angle-x angle-z))

;                                          1.4.3 Graphics functions
;                                          1.4.3.1 Screen definition and initialization fun!
ctions
;                                          1.4.3.1.1 MAKE-GRASP-SCREEN
(defun make-grasp-screen (&optional (proc-msg t))
  (if (create-grasp-screen)
      (progn (send grasp-screen :activate)
             (send grasp-screen :expose)
             (send grasp-screen :send-pane 'top-pane :select))
      (progn (send grasp-screen :select)
             (send grasp-screen :send-pane 'top-pane :select)))
  (if proc-msg (start-monitor-msg)))

;                                          1.4.3.1.2 KILL-GRASP-SCREEN
(defun kill-grasp-screen ()
   (if (variable-boundp grasp-screen)
       (progn (if (variable-boundp pc) (send pc :kill) )
              (send (send grasp-screen :send-pane 'bottom-pane :process) :reset)
               (send grasp-screen :kill)
               (variable-makunbound grasp-screen)
               T)
        nil))
;
;  this function resets the bottom pane lisp listener and starts a the message monitor
;  use (monitor-msg pc tg 'verbose) if you want verbose message processing...
;

;                                          1.4.3.1.3 FORCE-KBD-INPUT-STRING
(defun force-kbd-input-string (window string)
 (loop for i from 0 below (string-length string)
      do (send window :force-kbd-input (char-int (char string i)))))

;                                          1.4.3.1.4 START-MONITOR-MSG
(defun start-monitor-msg ()
  (if (variable-boundp pc)
    (progn  (send (send grasp-screen :send-pane 'bottom-pane :process) :reset)
            (send grasp-screen :send-pane 'bottom-pane ':clear-screen)
            (force-kbd-input-string (send grasp-screen :get-pane 'bottom-pane)
                                    "(monitor-msg pc tg)")
            )
    (send grasp-screen :send-pane 'bottom-pane :line-out
  #.(ZL:STRING "Parallel connection doesn't exist.  Can't start message processor")))
  T)

;                                          1.4.3.1.5 CREATE-GRASP-SCREENt
(defun create-grasp-screen ()
  (setq grasp-screen (tv:make-window
                       'tv:bordered-constraint-frame
```

```
                        ':panes
                        '((top-pane tv:window-pane
                                    :label "MAIN LISP LISTENER"
                                    :more-p nil)
                          (bottom-pane tv:window-pane
                                       :label "MESSAGE PROCESSOR"
                                       :more-p nil)
                          (grasp-window tv:window
                                        :label "GRASP WINDOW"
                                        :activate-p t))
                        ':configurations
                        '((main-config
                            (:layout
                              (main-config :row grasp-window right-side)
                              (right-side :column top-pane bottom-pane))
                            (:sizes
                              (right-side (top-pane :even) (bottom-pane :even))
                              (main-config (right-side 0.3) :then (grasp-window :even))))!
))))

;                                       1.4.3.2 CLEARSCREEN
(defun clearscreen ()
  (send grasp-screen :send-pane 'grasp-window ':refresh))

;                                       1.4.3.3 Draw functions
;                                       1.4.3.3.1 DRAW-2D-GRASP-WINDOW
(defun draw-2d-grasp-window (starting-point ending-point)
  (let ((x-start) (y-start)
        (x-end) (y-end)
        (window-height))
    (setq window-height (send grasp-screen :send-pane 'grasp-window ':height))
    (setq x-start (round (+ x-origin-2d (* scale-2d (aref starting-point 0)))))
    (setq y-start (round (+ window-height
                            (* -1 (+ y-origin-2d (* scale-2d (aref starting-point 1)))))!
))
    (setq x-end (round (+ x-origin-2d (* scale-2d (aref ending-point 0)))))
    (setq y-end (round (+ window-height (* -1 (+ y-origin-2d (* scale-2d (aref ending-po!
int 1)))))))
    (send grasp-screen :send-pane 'grasp-window ':draw-line x-start y-start x-end y-end)!
))

;                                       1.4.3.3.2 DRAW-3D-GRASP-WINDOW
(defun draw-3d-grasp-window (starting-point ending-point)
  (let ((start (zero-vector 3))
        (end (zero-vector 3))
        (x-start nil)
        (y-start nil)
        (x-end nil)
        (y-end nil)
        (window-height nil))
    (setq window-height (send grasp-screen :send-pane 'grasp-window ':height))
    (setq start (transform-points-global-frame view-frame starting-point))
    (setq end (transform-points-global-frame view-frame ending-point))
    (setq x-start (round (+ x-origin-3d (* scale-3d (aref start 0)))))
    (setq y-start (round (+ window-height (* -1 (+ y-origin-3d (* scale-3d (aref start 2!
)))))))
    (setq x-end (round (+ x-origin-3d (* scale-3d (aref end 0)))))
    (setq y-end (round (- window-height (+ y-origin-3d (* scale-3d (aref end 2))))))
    (send grasp-screen :send-pane 'grasp-window ':draw-line x-start y-start x-end y-end)!
))

;                                       1.4.3.3.3 SPHERE-3D-GRASP-WINDOW
(defun sphere-3d-grasp-window (sphere-center radius)
  (let ((center (zero-vector 3))
        (x-center)
        (y-center)
```

```
        (integer-radius)
        (window-height nil))
    (setq window-height (send grasp-screen :send-pane 'grasp-window ':height))
    (setq center (transform-points-global-frame view-frame sphere-center))
    (setq x-center (round (+ x-origin-3d (* scale-3d (aref center 0)))))
    (setq y-center (round (- window-height (+ y-origin-3d (* scale-3d (aref center 2))))))
))
    (setq integer-radius (round (* scale-3d radius)))
    (send grasp-screen :send-pane 'grasp-window ':draw-filled-in-circle x-center y-cente!
r integer-radius tv:alu-ior)))

;                                          1.4.3.3.4 DRAW-COORDINATES
(defun draw-coordinates (frames length)
  (cond ((listp frames) (draw-coordinate-list frames length))
        (t (draw-coordinate frames length))))

;                                          1.4.3.3.4.A DRAW-COORDINATES Auxiliary
(defun draw-coordinate-list (frames length)
  (cond ((null frames) nil)
        (t (draw-coordinate (car frames) length)
           (draw-coordinate-list (cdr frames) length))))

(defun draw-coordinate (frame length)
  (let* ((o-frame (make-array '(3) :initial-contents (list 0 0 0)))
         (x-frame (make-array '(3) :initial-contents (list length 0 0)))
         (y-frame (make-array '(3) :initial-contents (list 0 length 0)))
         (z-frame (make-array '(3) :initial-contents (list 0 0 length)))
         (origin-3d (transform-points-frame-global frame o-frame))
         (x (transform-points-frame-global frame x-frame))
         (y (transform-points-frame-global frame y-frame))
         (z (transform-points-frame-global frame z-frame)))
    (draw-3d-grasp-window origin-3d x)
    (draw-3d-grasp-window origin-3d y)
    (draw-3d-grasp-window origin-3d z))
  (draw-coordinate-labels frame length))

(defun draw-coordinate-labels (frame length)
  (let* ((x-point-1 (make-array '(3) :initial-contents
                                (list (* length 1.1) 0 (* length -0.05))))
         (x-point-2 (make-array '(3) :initial-contents
                                (list (* length 1.17) 0 (* length 0.05))))
         (x-point-3 (make-array '(3) :initial-contents
                                (list (* length 1.1) 0 (* length 0.05))))
         (x-point-4 (make-array '(3) :initial-contents
                                (list (* length 1.17) 0 (* length -0.05))))
         (y-point-1 (make-array '(3) :initial-contents
                                (list 0 (* length 1.1) (* length 0.05))))
         (y-point-2 (make-array '(3) :initial-contents
                                (list 0 (* length 1.135) 0)))
         (y-point-3 (make-array '(3) :initial-contents
                                (list 0 (* length 1.17) (* length 0.05))))
         (y-point-4 (make-array '(3) :initial-contents
                                (list 0 (* length 1.135) (* length -0.05))))
         (z-point-1 (make-array '(3) :initial-contents
                                (list (* length -0.035) 0 (* length 1.15))))
         (z-point-2 (make-array '(3) :initial-contents
                                (list (* length 0.035) 0 (* length 1.15))))
         (z-point-3 (make-array '(3) :initial-contents
                                (list (* length -0.035) 0 (* length 1.1))))
         (z-point-4 (make-array '(3) :initial-contents
                                (list (* length 0.035) 0 (* length 1.1)))))
    (setq x-point-1 (transform-points-frame-global frame x-point-1))
    (setq x-point-2 (transform-points-frame-global frame x-point-2))
    (setq x-point-3 (transform-points-frame-global frame x-point-3))
    (setq x-point-4 (transform-points-frame-global frame x-point-4))
    (setq y-point-1 (transform-points-frame-global frame y-point-1))
```

```
    (setq y-point-2 (transform-points-frame-global frame y-point-2))
    (setq y-point-3 (transform-points-frame-global frame y-point-3))
    (setq y-point-4 (transform-points-frame-global frame y-point-4))
    (setq z-point-1 (transform-points-frame-global frame z-point-1))
    (setq z-point-2 (transform-points-frame-global frame z-point-2))
    (setq z-point-3 (transform-points-frame-global frame z-point-3))
    (setq z-point-4 (transform-points-frame-global frame z-point-4))
    (draw-3d-grasp-window x-point-1 x-point-2)
    (draw-3d-grasp-window x-point-3 x-point-4)
    (draw-3d-grasp-window y-point-1 y-point-2)
    (draw-3d-grasp-window y-point-2 y-point-3)
    (draw-3d-grasp-window y-point-2 y-point-4)
    (draw-3d-grasp-window z-point-1 z-point-2)
    (draw-3d-grasp-window z-point-2 z-point-3)
    (draw-3d-grasp-window z-point-3 z-point-4)))

;                                          1.4.3.3.5 DRAW-3D-LIST
(defun draw-3d-list (list-center list)
  (let ((center (zero-vector 3))
        (x-center)
        (y-center)
        (window-height)
        (element)
        (n-elements)
        (start))
    (setq window-height (send grasp-screen :send-pane 'grasp-window ':height))
    (setq center (transform-points-global-frame view-frame list-center))
    (setq x-center (round (+ x-origin-3d (* scale-3d (aref center 0)))))
    (setq y-center (round (- window-height (+ y-origin-3d (* scale-3d (aref center 2))))))!
))
    (setq n-elements (count-atoms list))
    (do ((i 0 (+ i 1)))
        ((= i n-elements))
      (setq start (round (- x-center (* 10 (/ n-elements 2)))))
      (setq element (car list))
      (draw-character (+ (* 10 i) start) y-center element)
      (setq list (cdr list)))))

;                                          1.4.3.3.6 DRAW-CONTACTS-GRASP-SPACE
(defun draw-contacts-grasp-space ()
  (setq view-frame (construct-view-frame 0.4 0.4))
    (let*   ((contact-frames-grasp-space
               (transform-frames-global-frame *grasp-frame* *contact-frames*))
             (grasp-frame-grasp-space (make-array '(3 4) :initial-contents '((1 0 0 0)
                                                                             (0 1 0 0)
                                                                             (0 0 1 0))))!
))
    (draw-coordinates (first contact-frames-grasp-space) 0.5)
    (draw-coordinates (second contact-frames-grasp-space) 0.5)
    (draw-coordinates (third contact-frames-grasp-space) 0.5)
    (draw-coordinates grasp-frame-grasp-space 1.0)))

;                                          1.4.3.3.7 DRAW-CHARACTER
(defun draw-character (x y char)
  (cond ((numberp char) (send grasp-screen :send-pane 'grasp-window
                              ':draw-string (+ 48 char) x y))
        (t (send grasp-screen :send-pane 'grasp-window ':draw-string char x y))))


;;;*****************************************************************************!
*****
;
;                                          2. Sensed and global variables
;
;;;*****************************************************************************!
*****
```

```
;                                          2.0 Temporary variables used to construct globa!
l
;                                              sensed variables for simulation
;                                          2.0.1 Define temporary variables
(defvar *original-contact-points-object-space*)
(defvar *original-contact-normals-object-space*)
(defvar *original-object-frame*)
(defvar *contact-points-object-space*)
(defvar *contact-normals-object-space*)
(defvar *object-frame*)

(defvar *hand-frame*)
(defvar *grasp-frame*)

;                                          2.0.2 Set default temporary variables
;                                          2.0.2.1 Functions to set default values
(defun set-original-values ()
  (setq *original-contact-points-object-space*
        (list (make-array '(3) :initial-contents '(0 1.3 -0.8))
              (make-array '(3) :initial-contents '(0 1.3 0.8))
              (make-array '(3) :initial-contents '(0 -1.3 0))))
  (setq *original-contact-normals-object-space*
        (list (make-array '(3) :initial-contents '(0 1 0))
              (make-array '(3) :initial-contents '(0 1 0))
              (make-array '(3) :initial-contents '(0 -1 0)))))

(set-original-values)
;                                          2.0.2.2 Set temporary variables
(setq *hand-frame* (make-array '(3 4) :initial-contents '((1 0 0 0)
                                                          (0 1 0 0)
                                                          (0 0 1 0))))
(setq *original-object-frame* (make-array '(3 4) :initial-contents '((0 0 1 -0.8)
                                                                     (1 0 0 2.7)
                                                                     (0 1 0 -3.0))))
(setq *contact-points-object-space* *original-contact-points-object-space*)
(setq *contact-normals-object-space* *original-contact-normals-object-space*)
(setq *object-frame* *original-object-frame*)

;                                          2.1 Define sensed global variables
(defvar *contact-points*)
(defvar *contact-normals*)
(defvar *contact-frames*)

;                                          2.2 Set defualt initial global variables
;                                          2.2.1 Functions to set initial global variables
(defun generate-grasp-frame ()
  (let* ((x1 (aref (first *contact-frames*) 0 3))
         (y1 (aref (first *contact-frames*) 1 3))
         (z1 (aref (first *contact-frames*) 2 3))
         (x2 (aref (second *contact-frames*) 0 3))
         (y2 (aref (second *contact-frames*) 1 3))
         (z2 (aref (second *contact-frames*) 2 3))
         (x3 (aref (third *contact-frames*) 0 3))
         (y3 (aref (third *contact-frames*) 1 3))
         (z3 (aref (third *contact-frames*) 2 3))
         (centroid (make-array '(3)))
         (normal (make-array '(3))))
    (setf (aref centroid 0) (/ (+ x1 x2 x3) 3))
    (setf (aref centroid 1) (/ (+ y1 y2 y3) 3))
    (setf (aref centroid 2) (/ (+ z1 z2 z3) 3))
    (setf (aref normal 0) (- (* (- y3 y1) (- z2 z1)) (* (- z3 z1) (- y2 y1))))
    (setf (aref normal 1) (- (* (- z3 z1) (- x2 x1)) (* (- x3 x1) (- z2 z1))))
    (setf (aref normal 2) (- (* (- x3 x1) (- y2 y1)) (* (- y3 y1) (- x2 x1))))
    (generate-frame centroid normal)))
```

```
(defun normalize-contact-normals (contact-normals)
  (cond ((null contact-normals) nil)
        (t (cons (normalize-contact-normal (car contact-normals))
                 (normalize-contact-normals (cdr contact-normals))))))

(defun normalize-contact-normal (contact-normal)
  (let ((magnitude (sqrt (+ (sqr (aref contact-normal 0))
                            (sqr (aref contact-normal 1))
                            (sqr (aref contact-normal 2))))))
    (setf (aref contact-normal 0) (/ (aref contact-normal 0) magnitude))
    (setf (aref contact-normal 1) (/ (aref contact-normal 1) magnitude))
    (setf (aref contact-normal 2) (/ (aref contact-normal 2) magnitude))
    contact-normal))

(defun normalize-object-frame (object-frame)
  (let ((magnitude (zero-vector 3)))
    (do ((i 0 (+ i 1)))
        ((= i 3))
      (setf (aref magnitude i) (sqrt (+ (sqr (aref object-frame 0 i))
                                        (sqr (aref object-frame 1 i))
                                        (sqr (aref object-frame 2 i)))))
      (do ((j 0 (+ j 1)))
          ((= j 3))
        (setf (aref object-frame j i) (/ (aref object-frame j i) (aref magnitude i))))))
  object-frame)

;                                       2.2.2 Set global-variables
(defun initialize-global-variables ()
  (setq *contact-normals-object-space*
        (normalize-contact-normals *contact-normals-object-space*))
  (setq *object-frame* (normalize-object-frame *object-frame*))
  (setq *contact-points* (transform-points-frame-global *object-frame*
                                                        *contact-points-object-space*))
  (setq *contact-normals* (transform-direction-frame-global *object-frame*
                                                            *contact-normals-object-space*))
  (setq *contact-frames* (generate-frame *contact-points* *contact-normals*))
  (setq *grasp-frame* (generate-grasp-frame)))

(initialize-global-variables)

;;;*****************************************************************************!
*****
;
;                                       3. Constraint
;
;;;*****************************************************************************!
*****

;                                       3.1 Define constraint variables

(defvar *basis-wrenches*)
(defvar *basis-wrench*)
(defvar *virtual-work*)
(defvar *contact-types*)

;                                       3.2.1 Set defualt variable value section
;                                       3.2.1.1 Functions for setting global variables

(defun construct-basis-wrench (contact-frames)
  (cond ((null contact-frames) nil)
        (t (cons (transform-basis-wrench *basis-wrench* (car contact-frames))
                 (construct-basis-wrench (cdr contact-frames))))))

(defun transform-basis-wrench (basis-wrench contact-frame)
  (let* ((transformed-basis-wrench
```

```
          (transform-direction-frame-global contact-frame basis-wrench))
        (x (aref contact-frame 0 3))
        (y (aref contact-frame 1 3))
        (z (aref contact-frame 2 3)))
    (do ((i 0 (+ 1 i)))
        ((= i 11) transformed-basis-wrench)
      (setf (aref transformed-basis-wrench 3 i)
            (+ (* (aref transformed-basis-wrench 1 i) z -1)
               (* (aref transformed-basis-wrench 2 i) y)
               (aref transformed-basis-wrench 3 i)))
      (setf (aref transformed-basis-wrench 4 i)
            (+ (* (aref transformed-basis-wrench 2 i) x -1)
               (* (aref transformed-basis-wrench 0 i) z)
               (aref transformed-basis-wrench 4 i)))
      (setf (aref transformed-basis-wrench 5 i)
            (+ (* (aref transformed-basis-wrench 0 i) y -1)
               (* (aref transformed-basis-wrench 1 i) x)
               (aref transformed-basis-wrench 5 i))))))

;                                        3.2.1.2 Set constraint variables

(setq *contact-types* (list (make-array '(12) :initial-contents
                                        '(1 1 1 0 1 1 1 1 0 0 0 1))
                            (make-array '(12) :initial-contents
                                        '(1 1 1 0 0 0 1 1 0 0 0 0))
                            (make-array '(12) :initial-contents
                                        '(0 0 1 0 0 0 0 0 0 0 0 0))
                            (make-array '(12) :initial-contents
                                        '(0 0 0 0 0 0 0 0 0 0 0 0))))

(setq *basis-wrench* (make-array '(6 12) :initial-contents '((1 0 0 0 0 0 -1 0 0 0 0 0)
                                                             (0 1 0 0 0 0 0 -1 0 0 0 0)
                                                             (0 0 1 0 0 0 0 0 -1 0 0 0)
                                                             (0 0 0 1 0 0 0 0 0 -1 0 0)
                                                             (0 0 0 0 1 0 0 0 0 0 -1 0)
                                                             (0 0 0 0 0 1 0 0 0 0 0 -1))))!
)

(setq *basis-wrenches* (construct-basis-wrench *contact-frames*))


;                                        3.3 Constraint functions
;                                        3.3.1 CONSTRUCT-VIRTUAL-WORK-LIST
(defun construct-virtual-work-list (basis-wrenches twist)
  (cond ((null basis-wrenches) nil)
        (t (cons (construct-virtual-work (car basis-wrenches) twist)
                 (construct-virtual-work-list (cdr basis-wrenches) twist)))))

;                                        3.3.1.A CONSTRUCT-VIRTUAL-WORK-LIST Auxiliary

(defun construct-virtual-work (basis-wrench twist)
  (let ((virtual-work-vector (make-array '(12))))
    (do ((i 0 (+ 1 i)))
        ((= i 12) virtual-work-vector)
      (setf (aref virtual-work-vector i)
            (virtual-work (get-array-column basis-wrench i) twist)))))

(defun virtual-work(wrench twist)
  (+ (* (aref wrench 0) (aref twist 3))
     (* (aref wrench 1) (aref twist 4))
     (* (aref wrench 2) (aref twist 5))
     (* (aref wrench 3) (aref twist 0))
     (* (aref wrench 4) (aref twist 1))
     (* (aref wrench 5) (aref twist 2))))

;                                        3.3.2 DETERMINE-CONTACT-TYPES
```

```
(defun determine-contact-types (virtual-work-pattern-list)
  (cond ((null virtual-work-pattern-list) nil)
        (t (cons (determine-contact-type (car virtual-work-pattern-list) *contact-types*!
)
                 (determine-contact-types (cdr virtual-work-pattern-list))))))

;                                        3.3.2.A DETERMINE-CONTACT-TYPES Auxiliary


(defun determine-contact-type (virtual-work-pattern contact-types)
  (cond ((null contact-types) nil)
        (t (cons (test-slip virtual-work-pattern (car contact-types))
                 (determine-contact-type virtual-work-pattern (cdr contact-types))))))

(defun test-slip (virtual-work-pattern contact-type-pattern)
  (let ((total 0))
    (do ((i 0 (+ 1 i)))
        ((= i 12) (= total 0))
         (setq total (+ total (* (aref virtual-work-pattern i)
                                 (aref contact-type-pattern i)))))))

(defun construct-virtual-work-pattern-list (virtual-work-list)
  (cond ((null virtual-work-list) nil)
        (t (cons (construct-virtual-work-pattern (car virtual-work-list))
                 (construct-virtual-work-pattern-list (cdr virtual-work-list))))))

(defun construct-virtual-work-pattern (virtual-work)
  (let ((virtual-work-pattern-vector (make-array '(12))))
    (do ((i 0 (+ 1 i)))
        ((= i 12) virtual-work-pattern-vector)
      (cond ((>= (aref virtual-work i) 0) (setf (aref virtual-work-pattern-vector i) 0))
            (t (setf (aref virtual-work-pattern-vector i) 1))))))


(defun constraint-state (twist twist-ref)
  (constraint-state-aux (constraint twist twist-ref)))

(defun constraint-state-aux (states)
  (cond ((null states) nil)
        (t (cons (constraint-state-aux-1 (car states))
                 (constraint-state-aux (cdr states))))))

(defun constraint-state-aux-1 (state)
  (cond ((car state) 1)
        ((+ 1 (constraint-state-aux-1 (cdr state))))))

(defun constraint(twist twist-ref)
  (let ((twist-origin (twist-org twist twist-ref)))
    (determine-contact-types
      (construct-virtual-work-pattern-list
        (construct-virtual-work-list *basis-wrenches* twist-origin)))))

(defun twist-org (twist twist-ref)
  (let ((twist-org (make-array '(6)))
        (rx (* -1 (aref twist-ref 0)))
        (ry (* -1 (aref twist-ref 1)))
        (rz (* -1 (aref twist-ref 2))))
    (setf (aref twist-org 0) (aref twist 0))
    (setf (aref twist-org 1) (aref twist 1))
    (setf (aref twist-org 2) (aref twist 2))
    (setf (aref twist-org 3) (+ (aref twist 2)
                                (* (aref twist 1) rz)
                                (* (aref twist 2) ry -1)))
    (setf (aref twist-org 4) (+ (aref twist 4)
                                (* (aref twist 2) rx)
                                (* (aref twist 0) rz -1)))
```

```
    (setf (aref twist-org 5) (+ (aref twist 5)
                                (* (aref twist 0) ry)
                                (* (aref twist 1) rx -1)))
  twist-org))

;                                          3.4 Geometric Constraint
;                                          3.4.1 DRAW-CONTACT-TRAJECTORY
(defun draw-contact-trajectory (contact-point twist twist-ref twist-magnitude steps)
  (let ((new-contact-point (zero-vector 3)))
    (do ((i 1 (+ i 1)))
        ((= i steps))
      (setq new-contact-point
        (determine-contact-point contact-point twist twist-ref
                                 (/ (* i twist-magnitude) steps)))
    (draw-3d-grasp-window contact-point new-contact-point)
    (setq contact-point new-contact-point))))

;                                          3.4.1.A DRAW-CONTACT-TRAJECTORY Auxiliary
(defun determine-contact-point (contact-point twist twist-ref twist-magnitude)
  (let* ((rotation-matrix (zero-array 3 3))
         (twist-radius (zero-vector 3))
         (translation (zero-vector 3))
         (new-contact-point (zero-vector 3))
         (t1 (aref twist 0))
         (t2 (aref twist 1))
         (t3 (aref twist 2))
         (t4 (aref twist 3))
         (t5 (aref twist 4))
         (t6 (aref twist 5))
         (cn (cos twist-magnitude))
         (sn (sin twist-magnitude)))
    (setf (aref rotation-matrix 0 0) (+ (* t1 t1 (- 1 cn)) cn))
    (setf (aref rotation-matrix 0 1) (- (* t2 t1 (- 1 cn)) (* t3 sn)))
    (setf (aref rotation-matrix 0 2) (+ (* t3 t1 (- 1 cn)) (* t2 sn)))
    (setf (aref rotation-matrix 1 0) (+ (* t1 t2 (- 1 cn)) (* t3 sn)))
    (setf (aref rotation-matrix 1 1) (+ (* t2 t2 (- 1 cn)) cn))
    (setf (aref rotation-matrix 1 2) (- (* t3 t2 (- 1 cn)) (* t1 sn)))
    (setf (aref rotation-matrix 2 0) (- (* t1 t3 (- 1 cn)) (* t2 sn)))
    (setf (aref rotation-matrix 2 1) (+ (* t2 t3 (- 1 cn)) (* t1 sn)))
    (setf (aref rotation-matrix 2 2) (+ (* t3 t3 (- 1 cn)) cn))
    (setf (aref twist-radius 0) (- (aref contact-point 0) (aref twist-ref 0)))
    (setf (aref twist-radius 1) (- (aref contact-point 1) (aref twist-ref 1)))
    (setf (aref twist-radius 2) (- (aref contact-point 2) (aref twist-ref 2)))
    (setf (aref translation 0) (* twist-magnitude t4))
    (setf (aref translation 1) (* twist-magnitude t5))
    (setf (aref translation 2) (* twist-magnitude t6))
    (setq new-contact-point (add-array-list (list
                            (multiply-array-list (list rotation-matrix twist-radius))
                            twist-ref translation)))))

;;;*****************************************************************************!
*****
;
;                                          4. Body wrench
;
;;;*****************************************************************************!
*****

;                                          4.1 Define body wrench variables
(defvar *body-wrench*)
(defvar *object-mass*)
(defvar *gravitational-acceleration*)

;                                          4.2 Set body wrench variables section
;                                          4.2.1 Auxiliary body wrench functions
(defun construct-body-wrench ()
```

```
   (let ((body-wrench (zero-vector 6))
         (weight (* *object-mass* *gravitational-acceleration*)))
     (setf (aref body-wrench 2) weight)
     (setf (aref body-wrench 3) (* (aref *object-frame* 1 3) weight))
     (setf (aref body-wrench 4) (* -1 (aref *object-frame* 0 3) weight))
     body-wrench))

;                                          4.2.2 Set body wrench variables
(setq *object-mass* 0)
(setq *gravitational-acceleration* 32.2)
(setq *body-wrench* (construct-body-wrench))

;                                          4.3 Body wrench functions
;                                          4.3.1 ORIENTATION
(defun orientation (twist-cg)
  (let* ((t3 (aref twist-cg 3))
         (t4 (aref twist-cg 4))
         (t5 (aref twist-cg 5))
         (twist-magnitude (sqrt (abs (+ (* t3 t3) (* t4 t4) (* t5 t5)))))
         (theta (asin (/ t3 twist-magnitude)))
         (phi (asin (/ (* -1 t4) twist-magnitude))))
    (list theta phi)))

;                                          4.3.2 TWIST-CG
(defun twist-cg (twist twist-ref cg)
  (let ((twist-cg (make-array '(6)))
        (rx (- (aref cg 0) (aref twist-ref 0)))
        (ry (- (aref cg 1) (aref twist-ref 1)))
        (rz (- (aref cg 2) (aref twist-ref 2))))
    (setf (aref twist-cg 0) (aref twist 0))
    (setf (aref twist-cg 1) (aref twist 1))
    (setf (aref twist-cg 2) (aref twist 2))
    (setf (aref twist-cg 3) (+ (aref twist 2)
                               (* (aref twist 1) rz)
                               (* (aref twist 2) ry -1)))
    (setf (aref twist-cg 4) (+ (aref twist 4)
                               (* (aref twist 2) rx)
                               (* (aref twist 0) rz -1)))
    (setf (aref twist-cg 5) (+ (aref twist 5)
                               (* (aref twist 0) ry)
                               (* (aref twist 1) rx -1)))
  twist-cg))

;;;*****************************************************************************!
*****
;
;                                          5. Contact Wrenches
;
;;;*****************************************************************************!
*****

;                                          5.1 Define contact wrench variables
(defvar *fingertip-stiffness*)
(defvar *finger-stiffness*)

(defvar *contact-stiffness*)
(defvar *contact-stiffness-contact-frame*)
(defvar *contact-stiffness-hand-frame*)

(defvar *grasp-stiffness*)
(defvar *grasp-compliance*)

(defvar *contact-wrench*)
(defvar *contact-wrench-contact-frame*)

;                                          5.2 Set contact wrench variables section
```

```
;                                          5.2.1 Functions to set contact wrench variables
(defun construct-contact-wrench-variables ()
  (setq *contact-stiffness* (construct-contact-stiffness *fingertip-stiffness*
                                                         *finger-stiffness*
                                                         *contact-frames*))
  (setq *contact-stiffness-contact-frame* (construct-contact-stiffness-contact-frame
                                            *contact-stiffness*
                                            *contact-frames*))
  (setq *contact-stiffness-hand-frame* (construct-contact-stiffness-hand-frame
                                         *contact-stiffness*
                                         *contact-frames*))
  (setq *grasp-stiffness* (add-array-list *contact-stiffness-hand-frame*))
  (setq *grasp-compliance* (math:invert-matrix *grasp-stiffness*))
  (setq *contact-wrench* (construct-contact-wrenches *contact-stiffness-contact-frame*
                                                     *offset-wrench*))
  (setq *contact-wrench-contact-frame*
        (construct-contact-wrenches-contact-frame *contact-wrench* *contact-frames*)))

;                                          5.2.1.1 Contact stiffness
(defun construct-contact-stiffness (fingertip-stiffness finger-stiffness contact-frames)
  (cond ((null contact-frames) nil)
        (t (cons (construct-one-contact-stiffness (car fingertip-stiffness)
                                                  (car finger-stiffness)
                                                  (car contact-frames))
                 (construct-contact-stiffness (cdr fingertip-stiffness)
                                              (cdr finger-stiffness)
                                              (cdr contact-frames))))))

(defun construct-one-contact-stiffness (fingertip-stiffness finger-stiffness contact-fra!
me)
  (cond ((null finger-stiffness) (abs-array (transform-direction-frame-global
                                              contact-frame fingertip-stiffness)))
        (t (add-stiffness finger-stiffness
                          (abs-array (transform-direction-frame-global
                                       contact-frame fingertip-stiffness))))))

(defun add-stiffness (stiffness-1 stiffness-2)
  (let* ((stiffness-dimension (array-dimension stiffness-1 0))
         (new-stiffness (zero-vector stiffness-dimension)))
    (do ((i 0 (+ i 1)))
        ((= i stiffness-dimension) new-stiffness)
      (setf (aref new-stiffness i) (/ (* (aref stiffness-1 i) (aref stiffness-2 i))
                                      (+ (aref stiffness-1 i) (aref stiffness-2 i)))))))



;                                          5.2.1.2 Contact stiffness contact frame
(defun construct-contact-stiffness-contact-frame (contact-stiffness contact-frames)
  (cond ((null contact-frames) nil)
        (t (cons (construct-one-contact-stiffness-contact-frame (car contact-stiffness)
                                                                (car contact-frames))
                 (construct-contact-stiffness-contact-frame (cdr contact-stiffness)
                                                            (cdr contact-frames))))))

(defun construct-one-contact-stiffness-contact-frame (contact-stiffness contact-frame)
  (let ((rx (aref contact-frame 0 3))
        (ry (aref contact-frame 1 3))
        (rz (aref contact-frame 2 3))
        (kx (aref contact-stiffness 0))
        (ky (aref contact-stiffness 1))
        (kz (aref contact-stiffness 2))
        (ktx (aref contact-stiffness 3))
        (kty (aref contact-stiffness 4))
        (ktz (aref contact-stiffness 5))
        (contact-stiffness-contact-frame (zero-array 6 6)))
    (setf (aref contact-stiffness-contact-frame 0 1) (* kx rz))
```

```
    (setf (aref contact-stiffness-contact-frame 0 2) (* -1 kx ry))
    (setf (aref contact-stiffness-contact-frame 0 3) kx)
    (setf (aref contact-stiffness-contact-frame 1 0) (* -1 ky rz))
    (setf (aref contact-stiffness-contact-frame 1 2) (* ky rx))
    (setf (aref contact-stiffness-contact-frame 1 4) ky)
    (setf (aref contact-stiffness-contact-frame 2 0) (* kz ry))
    (setf (aref contact-stiffness-contact-frame 2 1) (* -1 kz rx))
    (setf (aref contact-stiffness-contact-frame 2 5) kz)
    (setf (aref contact-stiffness-contact-frame 3 0) ktx)
    (setf (aref contact-stiffness-contact-frame 4 1) kty)
    (setf (aref contact-stiffness-contact-frame 5 2) ktz)
    contact-stiffness-contact-frame))
;                                        5.2.1.3 Contact stiffness hand frame
(defun construct-contact-stiffness-hand-frame (contact-stiffness contact-frames)
  (cond ((null contact-frames) nil)
        (t (cons (construct-one-contact-stiffness-hand-frame (car contact-stiffness)
                                                             (car contact-frames))
                 (construct-contact-stiffness-hand-frame (cdr contact-stiffness)
                                                         (cdr contact-frames))))))

(defun construct-one-contact-stiffness-hand-frame (contact-stiffness contact-frame)
  (let ((rx (aref contact-frame 0 3))
        (ry (aref contact-frame 1 3))
        (rz (aref contact-frame 2 3))
        (kx (aref contact-stiffness 0))
        (ky (aref contact-stiffness 1))
        (kz (aref contact-stiffness 2))
        (ktx (aref contact-stiffness 3))
        (kty (aref contact-stiffness 4))
        (ktz (aref contact-stiffness 5))
        (contact-stiffness-hand-frame (zero-array 6 6)))
    (setf (aref contact-stiffness-hand-frame 0 1) (* kx rz))
    (setf (aref contact-stiffness-hand-frame 0 2) (* -1 kx ry))
    (setf (aref contact-stiffness-hand-frame 0 3) kx)
    (setf (aref contact-stiffness-hand-frame 1 0) (* -1 ky rz))
    (setf (aref contact-stiffness-hand-frame 1 2) (* ky rx))
    (setf (aref contact-stiffness-hand-frame 1 4) ky)
    (setf (aref contact-stiffness-hand-frame 2 0) (* kz ry))
    (setf (aref contact-stiffness-hand-frame 2 1) (* -1 kz rx))
    (setf (aref contact-stiffness-hand-frame 2 5) kz)
    (setf (aref contact-stiffness-hand-frame 3 0) (+ (* ky rz rz) (* kz ry ry) ktx))
    (setf (aref contact-stiffness-hand-frame 3 1) (* -1 kz rx ry))
    (setf (aref contact-stiffness-hand-frame 3 2) (* -1 ky rx rz))
    (setf (aref contact-stiffness-hand-frame 3 4) (* -1 ky rz))
    (setf (aref contact-stiffness-hand-frame 3 5) (* kz ry))
    (setf (aref contact-stiffness-hand-frame 4 0) (* -1 kz rx ry))
    (setf (aref contact-stiffness-hand-frame 4 1) (+ (* kx rz rz) (* kz rx rx) kty))
    (setf (aref contact-stiffness-hand-frame 4 2) (* -1 kx ry rz))
    (setf (aref contact-stiffness-hand-frame 4 3) (* kx rz))
    (setf (aref contact-stiffness-hand-frame 4 5) (* -1 kz rx))
    (setf (aref contact-stiffness-hand-frame 5 0) (* -1 ky rx rz))
    (setf (aref contact-stiffness-hand-frame 5 1) (* -1 kx ry rz))
    (setf (aref contact-stiffness-hand-frame 5 2) (+ (* kx ry ry) (* ky rx rx) ktz))
    (setf (aref contact-stiffness-hand-frame 5 3) (* -1 kx ry))
    (setf (aref contact-stiffness-hand-frame 5 4) (* ky rx))
    contact-stiffness-hand-frame))

;                                        5.2.1.3 Grasp stiffness
;                                        5.2.1.4 Grasp compliance
;                                        5.2.1.5 Contact wrench
(defun construct-contact-wrenches (contact-stiffness-contact-frame offset-wrench)
  (cond ((null contact-stiffness-contact-frame) nil)
        (t (cons (construct-contact-wrench (car contact-stiffness-contact-frame)
                                           (car offset-wrench))
                 (construct-contact-wrenches (cdr contact-stiffness-contact-frame)
                                             (cdr offset-wrench))))))
```

```
(defun construct-contact-wrench (contact-stiffness-contact-frame offset-wrench)
  (add-array-list (list (multiply-array-list
                          (list
                            (multiply-array-list
                              (list contact-stiffness-contact-frame *grasp-compliance*))
                            *body-wrench*))
                        offset-wrench)))

;                                       5.2.1.5 Contact wrench contact frame
(defun construct-contact-wrenches-contact-frame (contact-wrenches contact-frames)
  (cond ((null contact-wrenches) nil)
        (t (cons (transform-direction-global-frame (car contact-frames)
                                                   (car contact-wrenches))
                 (construct-contact-wrenches-contact-frame (cdr contact-wrenches)
                                                           (cdr contact-frames))))))

;                                       5.2.2 Set contact wrench variables
(setq *fingertip-stiffness* (list (make-array '(6) :initial-contents '(10 10 25 0 0 6))
                                  (make-array '(6) :initial-contents '(10 10 25 0 0 6))
                                  (make-array '(6) :initial-contents '(10 10 25 0 0 6)))!
)

(setq *finger-stiffness* (list nil nil nil))

(construct-contact-wrench-variables)

;;;*****************************************************************************!
*****
;
;                                       6. Contact-types
;
;;;*****************************************************************************!
*****

;                                       6.1 Define contact type variables
(defvar *fric-coeff*)
(defvar *mom-coeff*)

;                                       6.2 Set contact type variables section
(setq *fric-coeff* 0.6)
(setq *mom-coeff* 0.1)

;                                       6.3 Contact type functions
;                                       6.3.1 CONTACT-TYPE
(defun contact-type (contact-wrenches)
  (cond ((null contact-wrenches) nil)
        (t (cons (contact-type-aux (car contact-wrenches))
                 (contact-type (cdr contact-wrenches))))))

;                                       6.3.1.A Auxiliary contact type functions
(defun contact-type-aux (contact-wrench)
  (let* ((normal-force (aref contact-wrench 2))
         (tangent-force (sqrt (+ (* (aref contact-wrench 0) (aref contact-wrench 0))
                                 (* (aref contact-wrench 1) (aref contact-wrench 1)))))
         (moment (abs (aref contact-wrench 5)))
         (max-tangent-force (* *fric-coeff* (abs normal-force)))
         (max-moment (* *mom-coeff* (abs normal-force))))
    (cond ((>= normal-force 0) 4)
          ((>= tangent-force max-tangent-force) 3)
          ((>= moment max-moment) 2)
          (t 1))))

;;;*****************************************************************************!
****
;
```

```
;                              7. Offset wrench
;
;;;*****************************************************************************!
*****

;                                       7.1 Define offset wrench variables
(defvar *offset-wrench*)
(defvar *offset-wrench-contact-frame*)
(defvar *scale-force*)

;                                       7.2 Set offset wrench defaults
(setq *offset-wrench* (list (zero-vector 6)(zero-vector 6)(zero-vector 6)))
(setq *offset-wrench-contact-frame* (zero-vector 6))
(setq *scale-force* 1.0)

;                                       7.3 Offset wrench functions
;                                       7.3.1 CONSTRUCT-OFFSET-WRENCHES
(defun construct-offset-wrenches (force-center force-magnitude)
  (setq *offset-wrench* (offset-wrench-three-contacts force-center force-magnitude))
  (setq *offset-wrench-contact-frame*
        (transform-direction-global-frame *contact-frames* *offset-wrench*)))

;                                       7.3.1,A OFFSET-WRENCH-THREE-CONTACTS
;
; There are two solutions for the forces. One solution in which most of the force vector!
s are
; directed into the force-centroid and other in which two or more force vectors are
; directed away from the centroid.
(defun offset-wrench-three-contacts (force-center force-magnitude)
  (let* ((force-array (zero-array 3 3))
         (reduced-force-array (make-array '(3 3)))
         (offset1 (zero-vector 6))
         (offset2 (zero-vector 6))
         (offset3 (zero-vector 6))
         (length-1) (length-2) (length-3)
         (force-1) (force-2) (force-3)
         (x (aref force-center 0))
         (y (aref force-center 1))
         (z (aref force-center 2))
         (a) (b))

    (setq length-1 (sqrt (+ (sqr (- x (aref (first *contact-frames*) 0 3)))
                            (sqr (- y (aref (first *contact-frames*) 1 3)))
                            (sqr (- z (aref (first *contact-frames*) 2 3))))))
    (setq length-2 (sqrt (+ (sqr (- x (aref (second *contact-frames*) 0 3)))
                            (sqr (- y (aref (second *contact-frames*) 1 3)))
                            (sqr (- z (aref (second *contact-frames*) 2 3))))))
    (setq length-3 (sqrt (+ (sqr (- x (aref (third *contact-frames*) 0 3)))
                            (sqr (- y (aref (third *contact-frames*) 1 3)))
                            (sqr (- z (aref (third *contact-frames*) 2 3))))))

    (setf (aref force-array 0 0) (/ (- x (aref (first *contact-frames*) 0 3)) length-1))
    (setf (aref force-array 0 1) (/ (- x (aref (second *contact-frames*) 0 3)) length-2)!
)
    (setf (aref force-array 0 2) (/ (- x (aref (third *contact-frames*) 0 3)) length-3))
    (setf (aref force-array 1 0) (/ (- y (aref (first *contact-frames*) 1 3)) length-1))
    (setf (aref force-array 1 1) (/ (- y (aref (second *contact-frames*) 1 3)) length-2)!
)
    (setf (aref force-array 1 2) (/ (- y (aref (third *contact-frames*) 1 3)) length-3))!

    (setf (aref force-array 2 0) (/ (- z (aref (first *contact-frames*) 2 3)) length-1))
    (setf (aref force-array 2 1) (/ (- z (aref (second *contact-frames*) 2 3)) length-2)!
)
    (setf (aref force-array 2 2) (/ (- z (aref (third *contact-frames*) 2 3)) length-3))
    (if (< force-magnitude 0) (setq force-array (multiply-array-list (list -1 force-arra!
y))))
```

```
    (setq reduced-force-array (row-reduced-echelon force-array))
    (setq a (aref reduced-force-array 0 2))
    (setq b (aref reduced-force-array 1 2))
    (setq force-3 (sqrt (/ (sqr force-magnitude) (+ (sqr a) (sqr b) 1))))
    (setq force-2 (* -1 b force-3))
    (setq force-1 (* -1 a force-3))

    (setf (aref offset1 0) (* *scale-force* force-1 (aref force-array 0 0)))
    (setf (aref offset1 1) (* *scale-force* force-1 (aref force-array 1 0)))
    (setf (aref offset1 2) (* *scale-force* force-1 (aref force-array 2 0)))
    (setf (aref offset2 0) (* *scale-force* force-2 (aref force-array 0 1)))
    (setf (aref offset2 1) (* *scale-force* force-2 (aref force-array 1 1)))
    (setf (aref offset2 2) (* *scale-force* force-2 (aref force-array 2 1)))
    (setf (aref offset3 0) (* *scale-force* force-3 (aref force-array 0 2)))
    (setf (aref offset3 1) (* *scale-force* force-3 (aref force-array 1 2)))
    (setf (aref offset3 2) (* *scale-force* force-3 (aref force-array 2 2)))
    (list offset1 offset2 offset3)))


;;;*****************************************************************************!
*****
;
;                                        8. Controlled Slip
;
;;;*****************************************************************************!
*****

;                                        8.1 DETERMINE-CONSTRAINT-STATE
(defun determine-constraint-state  (force-center force-magnitude)
  (let ((grasp-vector-list)
        (constraint-state)
        (label-point
          (transform-points-global-frame *grasp-frame* force-center)))
    (construct-offset-wrenches force-center force-magnitude)
    (setq grasp-vector-list (construct-grasp-points force-center force-magnitude))
    (construct-contact-wrench-variables)
    (setq constraint-state (contact-type *contact-wrench-contact-frame*))
    (draw-coordinate-system)
    (sphere-3d-grasp-window (first grasp-vector-list) 0.01)
    (sphere-3d-grasp-window (second grasp-vector-list) 0.01)
    (sphere-3d-grasp-window (third grasp-vector-list) 0.01)
    (sphere-3d-grasp-window force-center 0.02)
    (setf (aref label-point 1) (+ (aref label-point 1) 0.025))
    (setq label-point (transform-points-frame-global *grasp-frame* label-point))
    (draw-3d-list label-point constraint-state)))

;                                        8.2 MAP-CONSTRAINT-SPACE


(defun map-constraint-state
       (x-start x-end x-inc y-start y-end y-inc force-magnitude)
  (let ((force-center-grasp-frame (zero-vector 3))
        (force-center-hand-frame (zero-vector 3))
        (label-point)
        (constraint-state))
    (do ((x x-start (+ x x-inc)))
        ((>= x x-end))
      (do ((y y-start (+ y y-inc)))
          ((>= y y-end))
        (setf (aref force-center-grasp-frame 0) x)
        (setf (aref force-center-grasp-frame 1) y)
        (setq label-point force-center-grasp-frame)
        (setq force-center-hand-frame
              (transform-points-frame-global *grasp-frame* force-center-grasp-frame))
        (construct-offset-wrenches force-center-hand-frame force-magnitude)
        (construct-contact-wrench-variables)
```

```
        (setq constraint-state (contact-type *contact-wrench-contact-frame*))
        (sphere-3d-grasp-window force-center-hand-frame 0.02)
        (setf (aref label-point 1) (+ (aref label-point 1) 0.025))
        (setq label-point (transform-points-frame-global *grasp-frame* label-point))
        (draw-3d-list label-point constraint-state)))))



(defun map-constraint-space (force-magnitude)
 (let* ((point-grasp-frame (make-array '(3) :initial-contents '(0 0 0)))
        (point-hand-frame  (make-array '(3) :initial-contents '(0 0 0)))
        (constraint-state))
   (do ((i -1 (+ i 0.2)))
       ((>= i 1))
     (do ((j -1 (+ j 0.2)))
         ((>= j 1))
       (setf (aref point-grasp-frame 0) i)
       (setf (aref point-grasp-frame 1) j)
       (setq point-hand-frame (transform-points-frame-global *grasp-frame* point-grasp-f!
rame))
       (setq constraint-state (determine-constraint-state point-hand-frame force-magnitu!
de))
       (sphere-3d-grasp-window point-hand-frame 0.01)
       (setf (aref point-grasp-frame 1) (+ j 0.01))
       (setq point-hand-frame (transform-points-frame-global *grasp-frame* point-grasp-f!
rame))
       (draw-3d-list point-hand-frame constraint-state)))))

;;;*****************************************************************************!
*****
;
;                                        9. Hand actuation routines
;
;;;*****************************************************************************!
****

;                                        9.1 Define actuation constants and variables

(defvar *default-trajectory-gen* (if (boundp 'tg) tg))
(defvar *default-parallel-conn* (if (boundp 'pc) pc))
(defvar *fingertip-radius*)
(defvar *test-point*)
(defvar *scale-force*)

;                                        9.3 Set actuation variables

(setq *fingertip-radius* 0.408)
(setq *test-point* (make-array '(3) :initial-contents '(0.0 2.5 0.0)))
(setq *scale-force* 1.0)
(setq *offset-wrench* (list (zero-vector 6) (zero-vector 6) (zero-vector 6)))

;                                        9.4 Grasp functions

;                                        9.4.1 MOVE-TO-CONTACT-POINTS

(defun move-to-contact-points ()
  (let* ((fingertip-vector (construct-fingertip-vector
                            (determine-fingertip-centers *contact-points* *contact-fram!
es*)))
         (dur 0.4))
    (if (send tg :move-fingers-to fingertip-vector :duration dur)
        (send tg :send-traj pc))))

;                                        9.4.2 MOVE-TO-GRASP-CENTER
```

```
(defun move-to-grasp-center (force-center force-magnitude)
  (let* ((force-center-hand-frame (transform-points-frame-global *grasp-frame* force-cen!
ter))
         (magnitude (* *scale-force* force-magnitude))
         (grasp-vector-list (construct-grasp-points force-center-hand-frame magnitude))
         (fingertip-center-list
           (determine-fingertip-centers grasp-vector-list *contact-frames*))
         (fingertip-vector (construct-fingertip-vector fingertip-center-list))
         (dur 0.4))
    (construct-offset-wrenches force-center force-magnitude)
    (clearscreen)
    (draw-coordinate-system)
    (sphere-3d-grasp-window (first grasp-vector-list) 0.01)
    (sphere-3d-grasp-window (second grasp-vector-list) 0.01)
    (sphere-3d-grasp-window (third grasp-vector-list) 0.01)
    (sphere-3d-grasp-window force-center-hand-frame 0.02)
   (if (send tg :move-fingers-to fingertip-vector :duration dur)
       (send tg :send-traj pc))))



;                                          9.5 Auxiliary functions

;                                          9.6.1 MOVE-TO-CONTACT-POINTS auxiliary function!
s


(defun determine-fingertip-centers (contact-points contact-frames)
  (cond ((null contact-frames) nil)
        (t (cons (determine-fingertip-center (car contact-points) (car contact-frames))
                 (determine-fingertip-centers (cdr contact-points) (cdr contact-frames)))!
)))

(defun determine-fingertip-center (contact-point contact-frame)
  (let* ((fingertip-center (zero-vector 3)))
    (setf (aref fingertip-center 0)
          (+ (* (aref contact-frame 0 2) *fingertip-radius*) (aref contact-point 0)))
    (setf (aref fingertip-center 1)
          (+ (* (aref contact-frame 1 2) *fingertip-radius*) (aref contact-point 1)))
    (setf (aref fingertip-center 2)
          (+ (* (aref contact-frame 2 2) *fingertip-radius*) (aref contact-point  2)))
    fingertip-center))

(defun construct-fingertip-vector (fingertip-centers)
  (let* ((n-fingertips (count-atoms fingertip-centers))
         (fingertip-vector (zero-vector (* 3 n-fingertips))))
    (do ((i 0 (+ i 1)))
        ((= i n-fingertips) fingertip-vector)
      (do ((j 0 (+ j 1)))
          ((= j 3))
        (setf (aref fingertip-vector (+ (* i 3) j)) (* 2.54 (aref (car fingertip-centers!
) j))))
      (setq fingertip-centers (cdr fingertip-centers)))))

;                                          9.5.2 MOVE-TO-GRASP-CENTER auxiliary functions

(defun construct-grasp-points (force-center force-magnitude)
  (let* ((grasp-points (list (zero-vector 3) (zero-vector 3) (zero-vector 3)))
         (offset-points (offset-wrench-three-contacts force-center force-magnitude)))
    (setf (aref (first grasp-points) 0) (+ (aref (first *contact-frames*) 0 3) (aref (fi!
rst offset-points) 0)))
    (setf (aref (first grasp-points) 1) (+ (aref (first *contact-frames*) 1 3) (aref (fi!
rst offset-points) 1)))
    (setf (aref (first grasp-points) 2) (+ (aref (first *contact-frames*) 2 3) (aref (fi!
rst offset-points) 2)))
```

```
    (setf (aref (second grasp-points) 0) (+ (aref (second *contact-frames*) 0 3) (aref (!
second offset-points) 0)))
    (setf (aref (second grasp-points) 1) (+ (aref (second *contact-frames*) 1 3) (aref (!
second offset-points) 1)))
    (setf (aref (second grasp-points) 2) (+ (aref (second *contact-frames*) 2 3) (aref (!
second offset-points) 2)))

    (setf (aref (third grasp-points) 0) (+ (aref (third *contact-frames*) 0 3) (aref (th!
ird offset-points) 0)))
    (setf (aref (third grasp-points) 1) (+ (aref (third *contact-frames*) 1 3) (aref (th!
ird offset-points) 1)))
    (setf (aref (third grasp-points) 2) (+ (aref (third *contact-frames*) 2 3) (aref (th!
ird offset-points) 2)))
    grasp-points))


;;All the move functions will return either NIL or an integer.  NIL is returned if the
;;trajectory is not feasible.  NIL is also returned when there's transmission
;;problems in the parallel connection.


;                                        9.7 Move load functions
;                                        i.e. generates a trajectory but does not send i!
t

;                                        9.7.1 Advanced load moves

(defun move-to-contact-points-load ()
  (let* ((fingertip-vector (construct-fingertip-vector
                            (determine-fingertip-centers *contact-points* *contact-fram!
es*)))
         (dur 0.4))
   (send tg :move-fingers-to fingertip-vector :duration dur)))

(defun move-to-grasp-center-load (force-center force-magnitude)
  (let* ((force-center-hand-frame (transform-points-frame-global *grasp-frame* force-cen!
ter))
         (magnitude (* *scale-force* force-magnitude))
         (grasp-vector-list (construct-grasp-points force-center-hand-frame magnitude))
         (fingertip-center-list
           (determine-fingertip-centers grasp-vector-list *contact-frames*))
         (fingertip-vector (construct-fingertip-vector fingertip-center-list))
         (dur 0.8))
    (print-array-list grasp-vector-list)
    (setq *offset-wrench* (list (multiply-array-list
                                 (list *scale-force*
                                 (add-array-list (list (first grasp-vector-list)
                                                       (multiply-array-list
                                                       (list -1 (first *contact-points*)))!
))))
                                 (multiply-array-list
                                 (list *scale-force*
                                 (add-array-list (list (second grasp-vector-list)
                                                       (multiply-array-list
                                                       (list -1 (second *contact-points*))!
)))))
                                 (multiply-array-list
                                 (list *scale-force*
                                 (add-array-list (list (third grasp-vector-list)
                                                       (multiply-array-list
                                                       (list -1 (third *contact-points*)))!
))))))
    (clearscreen)
    (draw-coordinate-system)
    (sphere-3d-grasp-window (first grasp-vector-list) 0.01)
```

```
    (sphere-3d-grasp-window (second grasp-vector-list) 0.01)
    (sphere-3d-grasp-window (third grasp-vector-list) 0.01)
    (sphere-3d-grasp-window force-center-hand-frame 0.02)
    (send tg :move-fingers-to fingertip-vector :duration dur)))

;                                          9.7.2 Basic load moves

(defun move-up-load (&optional (dist 1.0) &key number-of-segs duration
                                              (traj-gen *default-trajectory-gen*))
  (let ((nsegs (if number-of-segs number-of-segs (send traj-gen :default-segs-per-move))!
)
        (dur (if duration duration (send traj-gen :default-duration))))
    (setf (aref work-transl-vect 0) 0.0
          (aref work-transl-vect 1) 0.0
          (aref work-transl-vect 2) dist)
    (if (send traj-gen :generate-traj (list (list nil nil work-transl-vect))
              :number-of-segs nsegs
              :duration dur)
        (progn (setf (aref *object-frame* 2 3) (+ (aref *object-frame* 2 3) (/ dist 2.54!
)))
               (initialize-global-variables)))))

 (defun move-down-load (&optional (dist 1.0) &key number-of-segs duration
                                               (traj-gen *default-trajectory-gen*))
  (let ((nsegs (if number-of-segs number-of-segs (send traj-gen :default-segs-per-move))!
)
        (dur (if duration duration (send traj-gen :default-duration))))
    (setf (aref work-transl-vect 0) 0.0
          (aref work-transl-vect 1) 0.0
          (aref work-transl-vect 2) (- dist))
    (if (send traj-gen :generate-traj (list (list nil nil work-transl-vect))
                       :number-of-segs nsegs
                       :duration dur)
        (progn (setf (aref *object-frame* 2 3) (- (aref *object-frame* 2 3) (/ dist 2.54!
)))
               (initialize-global-variables)))))

(defun move-left-load (&optional (dist 1.0) &key number-of-segs duration
                                                (traj-gen *default-trajectory-gen*))
  (let ((nsegs (if number-of-segs number-of-segs (send traj-gen :default-segs-per-move))!
)
        (dur (if duration duration (send traj-gen :default-duration))))
    (setf (aref work-transl-vect 0) (- dist)
          (aref work-transl-vect 1) 0.0
          (aref work-transl-vect 2) 0.0)
    (if (send traj-gen :generate-traj (list (list nil nil work-transl-vect))
                       :number-of-segs nsegs
                       :duration dur)
        (progn (setf (aref *object-frame* 0 3) (- (aref *object-frame* 0 3) (/ dist 2.54!
)))
               (initialize-global-variables)))))

(defun move-right-load (&optional (dist 1.0) &key number-of-segs duration
                                                 (traj-gen *default-trajectory-gen*))
  (let ((nsegs (if number-of-segs number-of-segs (send traj-gen :default-segs-per-move))!
)
        (dur (if duration duration (send traj-gen :default-duration))))
    (setf (aref work-transl-vect 0) dist
          (aref work-transl-vect 1) 0.0
          (aref work-transl-vect 2) 0.0)
    (if (send traj-gen :generate-traj (list (list nil nil work-transl-vect))
                       :number-of-segs nsegs
                       :duration dur)
        (progn (setf (aref *object-frame* 0 3) (+ (aref *object-frame* 0 3) (/ dist 2.54!
)))
               (initialize-global-variables)))))
```

```
(defun move-in-load (&optional (dist 1.0) &key number-of-segs duration
                                               (traj-gen *default-trajectory-gen*))
  (let ((nsegs (if number-of-segs number-of-segs (send traj-gen :default-segs-per-move))!
)
        (dur (if duration duration (send traj-gen :default-duration))))
    (setf (aref work-transl-vect 0) 0.0
          (aref work-transl-vect 1) (- dist)
          (aref work-transl-vect 2) 0.0)
    (if (send traj-gen :generate-traj (list (list nil nil work-transl-vect))
                       :number-of-segs nsegs
                       :duration dur)
        (progn (setf (aref *object-frame* 1 3) (- (aref *object-frame* 1 3) (/ dist 2.54!
)))
               (initialize-global-variables)))))

(defun move-out-load (&optional (dist 1.0) &key number-of-segs duration
                                                (traj-gen *default-trajectory-gen*))
  (let ((nsegs (if number-of-segs number-of-segs (send traj-gen :default-segs-per-move))!
)
        (dur (if duration duration (send traj-gen :default-duration))))
    (setf (aref work-transl-vect 0) 0.0
          (aref work-transl-vect 1) dist
          (aref work-transl-vect 2) 0.0)
    (if (send traj-gen :generate-traj (list (list nil nil work-transl-vect))
                       :number-of-segs nsegs
                       :duration dur)
        (progn (setf (aref *object-frame* 1 3) (+ (aref *object-frame* 1 3) (/ dist 2.54!
)))
               (initialize-global-variables)))))

(defun rotate-x-load (&optional (ang 10.0) &key number-of-segs
                                               duration
                                               (traj-gen *default-trajectory-gen*))
  ;;convert angles from degrees to radians
  (setq ang (* ang 0.01745329))
  (let ((nsegs (if number-of-segs number-of-segs (send traj-gen :default-segs-per-move))!
)
        (dur (if duration duration (send traj-gen :default-duration)))
        (temp-frame (make-array '(3 4) :initial-contents '((1 0 0 0)
                                                           (0 1 0 0)
                                                           (0 0 1 0)))))
    (setf (aref temp-frame 0 3) (aref *grasp-frame* 0 3))
    (setf (aref temp-frame 1 3) (aref *grasp-frame* 1 3))
    (setf (aref temp-frame 2 3) (aref *grasp-frame* 2 3))
    (setq *object-frame* (transform-frames-global-frame temp-frame *object-frame*))
    (setf (aref temp-frame 0 1) (cos ang))
    (setf (aref temp-frame 0 2) (* -1 (sin ang)))
    (setf (aref temp-frame 1 1) (sin ang))
    (setf (aref temp-frame 1 2) (cos ang))
    (setq *object-frame* (transform-frames-frame-global temp-frame *object-frame*))
    (draw-coordinates *object-frame* 0.5)
    (if (send traj-gen :generate-traj (list (list 'xhat ang nil))
              :number-of-segs nsegs :duration dur)
        (progn (setq *object-frame* (transform-frames-frame-global temp-frame *object-fr!
ame*))
               (initialize-global-variables)))))

(defun rotate-y-load (&optional (ang 10.0) &key number-of-segs
                                               duration
                                               (traj-gen *default-trajectory-gen*))
  (setq ang (* ang 0.01745329))
  (let ((nsegs (if number-of-segs number-of-segs (send traj-gen :default-segs-per-move))!
)
        (dur (if duration duration (send traj-gen :default-duration)))
        (temp-frame (zero-array 3 4)))
```

```
    (setf (aref temp-frame 1 1) 1)
    (setf (aref temp-frame 0 0) (cos ang))
    (setf (aref temp-frame 0 2) (* -1 (sin ang)))
    (setf (aref temp-frame 2 0) (sin ang))
    (setf (aref temp-frame 2 2) (cos ang))
    (setf (aref temp-frame 0 3) (- (aref *grasp-frame* 0 3) (aref *object-frame* 0 3)))
    (setf (aref temp-frame 1 3) (- (aref *grasp-frame* 1 3) (aref *object-frame* 1 3)))
    (setf (aref temp-frame 2 3) (- (aref *grasp-frame* 2 3) (aref *object-frame* 2 3)))
    (if (send traj-gen :generate-traj (list (list 'yhat ang nil))
                       :number-of-segs nsegs :duration dur)
        (progn (setq *object-frame* (transform-frames-frame-global temp-frame *object-fr!
ame*))
               (initialize-global-variables)))))

(defun rotate-z-load (&optional (ang 10.0) &key number-of-segs
                                           duration
                                           (traj-gen *default-trajectory-gen*))
  (setq ang (* ang 0.01745329))
  (let ((nsegs (if number-of-segs number-of-segs (send traj-gen :default-segs-per-move))!
)
        (dur (if duration duration (send traj-gen :default-duration)))
        (temp-frame (zero-array 3 4)))
    (setf (aref temp-frame 2 2) 1)
    (setf (aref temp-frame 0 0) (cos ang))
    (setf (aref temp-frame 0 1) (* -1 (sin ang)))
    (setf (aref temp-frame 1 0) (sin ang))
    (setf (aref temp-frame 1 1) (cos ang))
    (setf (aref temp-frame 0 3) (- (aref *grasp-frame* 0 3) (aref *object-frame* 0 3)))
    (setf (aref temp-frame 1 3) (- (aref *grasp-frame* 1 3) (aref *object-frame* 1 3)))
    (setf (aref temp-frame 2 3) (- (aref *grasp-frame* 2 3) (aref *object-frame* 2 3)))
    (if (send traj-gen :generate-traj (list (list 'zhat ang nil))
                       :number-of-segs nsegs :duration dur)
        (progn (setq *object-frame* (transform-frames-frame-global temp-frame *object-fr!
ame*))
               (initialize-global-variables)))))

(defun move-finger-load (finger-number displacement-vector)
  (let ((traj-gen *default-trajectory-gen*)
        (dur 0.3)
        (displacement (zero-vector 9)))
    (setf (aref displacement (+ (* 3 (- finger-number 1)) 0)) (aref displacement-vector !
0))
    (setf (aref displacement (+ (* 3 (- finger-number 1)) 1)) (aref displacement-vector !
1))
    (setf (aref displacement (+ (* 3 (- finger-number 1)) 2)) (aref displacement-vector !
2))
    (send traj-gen :move-fingers-by displacement :duration dur)))

(defun go-hand ()
  (send tg :send-traj pc))

;                                          9.8 Basic moves

(defun up (&optional (distance 2.0))
  (if (move-up-load distance)
      (progn (clearscreen)
             (draw-coordinate-system)
             (go-hand))))
(defun down (&optional (distance 2.0))
  (if (move-down-load distance)
      (progn (clearscreen)
             (draw-coordinate-system)
             (go-hand))))
(defun right (&optional (distance 2.0))
  (if (move-right-load distance)
      (progn (clearscreen)
```

```
              (draw-coordinate-system)
              (go-hand))))
(defun left (&optional (distance 2.0))
  (if (move-left-load distance)
      (progn (clearscreen)
             (draw-coordinate-system)
             (go-hand))))
(defun in (&optional (distance 2.0))
  (if (move-in-load distance)
      (progn (clearscreen)
             (draw-coordinate-system)
             (go-hand))))
(defun out (&optional (distance 2.0))
  (if (move-out-load distance)
      (progn (clearscreen)
             (draw-coordinate-system)
             (go-hand))))
(defun rot+x (&optional (angle 10.0))
  (if (rotate-x-load angle)
      (progn (clearscreen)
             (draw-coordinate-system)
             (go-hand))))
(defun rot+y (&optional (angle 10.0))
  (if (rotate-y-load angle)
      (progn (clearscreen)
             (draw-coordinate-system)
             (go-hand))))
(defun rot+z (&optional (angle 10.0))
  (if (rotate-z-load angle)
      (progn (clearscreen)
             (draw-coordinate-system)
             (go-hand))))
(defun rot-x (&optional (angle -10.0))
  (if (rotate-x-load angle)
      (progn (clearscreen)
             (draw-coordinate-system)
             (go-hand))))
(defun rot-y (&optional (angle -10.0))
  (if (rotate-y-load angle)
      (progn (clearscreen)
             (draw-coordinate-system)
             (go-hand))))
(defun rot-z (&optional (angle -10.0))
  (if (rotate-z-load angle)
      (progn (clearscreen)
             (draw-coordinate-system)
             (go-hand))))

;;;*****************************************************************************!
*****
;
;                                          10. Menu
;
;;;*****************************************************************************!
****

;                                          10.1 Define menu variables

(defvar menu)
(defvar moves-menu)
(defvar dem-menu)
;                                          10.2 Define menu

(setq menu (tv:make-window 'tv:momentary-menu
        ':label '(:string #.(zl:string "Grasp Menu:  "))
        ':geometry (list 2)
```

```
        ':borders 3
        ':item-list '(("STANDARD GRAPHICS OPTIONS: " :no-select)
                      ("STANDARD MOTION OPTIONS: " :no-select)
                      ("Create grasp screen " :funcall make-grasp-screen)
                      ("Reinit system " :funcall reinit)
                      ("Clear screen" :funcall clearscreen)
                      ("Reinit OOLAH trajectory " :funcall init-oolah)
                      ("Draw coordinate system " :funcall draw-coordinate-system)
                      ("Reinit VAX trajectory " :funcall init-vax)
                      ("Change graphics variables" :funcall change-graphics-variables-me!
nu)
                      ("Go home " :funcall go-home-and-init)
                      ("" :no-select)
                      ("BASIC MOVES " :funcall basic-moves-menu)
                      ("" :no-select)
                      ("Move finger " :funcall move-finger-menu)
                      ("" :no-select)
                      ("" :no-select)
                      ("SLIP ANALYSIS OPTIONS: " :no-select)
                      ("ADVANCED HAND ACTUATION OPTIONS: " :no-select)
                      ("Change global variables " :funcall change-global-variables-menu)
                      ("JKS move menu " :funcall move-menu)
                      ("Permissible-twist " :funcall permissible-twist-menu)
                      ("Move to contact points" :funcall move-to-contact-points)
                      ("Determine constraint state " :funcall determine-constraint-state!
-menu)
                      ("Pick grasp force center " :funcall grasp-force-center-menu)
                      ("Map constraint space " :funcall map-constraint-state-menu)
                      ("Controlled slip " :funcall controlled-slip-menu)
                      ("" :no-select)
                      ("DEMONSTRATIONS" :funcall demo-menu)
                      ("" :no-select)
                      ("" :no-select)
                      ("" :no-select)
                      ("QUIT" :eval 999))))

;                                       10.3 Standard graphics options

;                                       10.3.1 MAKE-GRASP-SCREEN
;                                       10.3.2 CLEARSCREEN
;                                       10.3.3 DRAW-COORDINATE-SYSTEM

(defun draw-coordinate-system ()
  (setq view-frame (construct-view-frame angle-x angle-z))
  (draw-coordinates *hand-frame* 1)
  (draw-coordinates *object-frame* 0.75)
  (draw-coordinates *grasp-frame* 0.5)
  (draw-coordinates *contact-frames* 0.2))

;                                       10.3.4 CHANGE-GRAPHICS-VARIABLES-MENU

(defun change-graphics-variables-menu ()
  (let ((zl-user:scale-3d scale-3d)
        (zl-user:x-origin-3d x-origin-3d)
        (zl-user:y-origin-3d y-origin-3d)
        (zl-user:angle-x angle-x)
        (zl-user:angle-z angle-z))
    (zl-user:choose-user-options zl-user:graphics-variables-menu)
    (setq scale-3d zl-user:scale-3d)
    (setq x-origin-3d zl-user:x-origin-3d)
    (setq y-origin-3d zl-user:y-origin-3d)
    (setq angle-x zl-user:angle-x)
    (setq angle-z zl-user:angle-z)))

;                                       10.4 Standard motion options
;                                       10.4.0 REINIT
```

```
;                                                    10.4.1 INIT-OOLAH
(defun init-oolah ()
  (send tg :init))
;                                                    10.4.2 INIT-VAX
(defun init-vax ()
  (send tg :init-traj pc))
;                                                    10.4.3 GO-HOME-AND-INIT
(defun go-home-and-init ()
  (set-original-values)
  (setq *object-frame* *original-object-frame*)
  (setq *contact-points-object-space* *original-contact-points-object-space*)
  (setq *contact-normals-object-space* *original-contact-normals-object-space*)
  (initialize-global-variables)
  (init-oolah)
  (init-vax)
  (go-home)
  (init-vax)
  (go-home)
  (clearscreen)
  (draw-coordinate-system))

;                                                    10.4.4 BASIC-MOVES

(defun basic-moves-menu ()
  (do ((i 0 (+ i 1)))
    ((equalp (send moves-menu ':choose) 999) 'Done)))


(setq moves-menu (tv:make-window 'tv:momentary-menu
        ':label '(:string #.(zl:string "Basic moves menu:  "))
        ':geometry (list 3)
        ':borders 2
        ':item-list '(("OUT" :funcall out)
                      ("UP" :funcall up)
                      ("" :no-select)
                      ("LEFT" :funcall left)
                      ("QUIT" :eval 999)
                      ("RIGHT" :funcall right)
                      ("" :no-select)
                      ("DOWN" :funcall down)
                      ("IN" :funcall in)
                      ("" :no-select)
                      ("" :no-select)
                      ("" :no-select)
                      ("ROT-X " :no-select)
                      ("ROT-Y " :no-select)
                      ("ROT-Z " :no-select)
                      (" + " :funcall rot+x)
                      (" + " :funcall rot+y)
                      (" + " :funcall rot+z)
                      (" - " :funcall rot-x)
                      (" - " :funcall rot-y)
                      (" - " :funcall rot-z))))

;                                                    10.4.5  MOVE-FINGER-MENU

(defun move-finger-menu ()
  (let ((displacement (make-array '(3))))
    (zl-user:choose-user-options zl-user:finger-menu)
    (setf (aref displacement 0) zl-user:x-displacement)
    (setf (aref displacement 1) zl-user:y-displacement)
    (setf (aref displacement 2) zl-user:z-displacement)
    (move-finger-load zl-user:finger-number displacement)
    (go-hand)))

;                                                    10.5 SLIP-ANALYSIS
```

```
;                                          10.5.1 CHANGE-GLOBAL-VARIABLES-MENU
(defvar variables-menu)
(defun change-global-variables-menu ()
  (do ((i 0 (+ i 1)))
    ((equalp (send variables-menu ':choose) 999) 'Done)))

(setq variables-menu (tv:make-window 'tv:momentary-menu
        ':label '(:string #.(zl:string "Variables moves menu:  "))
        ':geometry (list 1)
        ':borders 2
        ':item-list '(("Object variables" :funcall object-variable-menu)
                      ("Contact variables" :funcall contact-variable-menu)
                      ("Body wrench" :funcall body-wrench-menu)
                      ("Finger variables" :no-select)
                      ("" :no-select)
                      ("QUIT" :eval 999))))


(defun object-variable-menu ()
  (let ((zl-user:*object-frame* *object-frame*))
    (zl-user:object-variable-menu)
    (setq *object-frame* zl-user:*object-frame*)
    (initialize-global-variables)))

(defun contact-variable-menu ()
  (let ((zl-user:*contact-points-object-space* *contact-points-object-space*)
        (zl-user:*contact-normals-object-space* *contact-normals-object-space*))
    (zl-user:contact-variable-menu)
    (setq *contact-points-object-space* zl-user:*contact-points-object-space*)
    (setq *contact-normals-object-space* zl-user:*contact-normals-object-space*)
    (initialize-global-variables)))


(defun body-wrench-menu()
  (setq zl-user:*body-wrench* *body-wrench*)
  (zl-user:body-wrench-menu)
  (setq *body-wrench* zl-user:*body-wrench*))

;                                          10.5.2 PERMISSIBLE-TWIST-MENU

(defun permissible-twist-menu ()
  (let ((twist (make-array '(6))))
    (zl-user:permissible-twist-menu)
    (setq twist zl-user:*twist*)
    (permissible-twist twist)))

;                                          10.5.3 DETERMINE-CONSTRAINT-STATE-MENU


(defun determine-constraint-state-menu ()
  (draw-coordinate-system)
  (let ((magnitude 0)
        (force-center (zero-vector 3))
        (force-center-hand-frame (zero-vector 3)))
    (zl-user:choose-user-options zl-user:constraint-state-menu)
    (setq magnitude zl-user:magnitude)
    (setq force-center (get-mouse-coordinates-grasp-frame))
    (setq force-center-hand-frame (transform-points-frame-global *grasp-frame* force-cen!
ter))
    (determine-constraint-state force-center-hand-frame magnitude)))

;                                          10.5.4 MAP-CONSTRAINT-STATE-MENU


(defun map-constraint-state-menu ()
```

```
  (draw-coordinate-system)
  (zl-user:choose-user-options zl-user:map-state-menu)
  (let ((x-inc (/ (- zl-user:x-end zl-user:x-start) zl-user:x-steps))
        (y-inc (/ (- zl-user:y-end zl-user:y-start) zl-user:y-steps)))
    (map-constraint-state zl-user:x-start zl-user:x-end x-inc
                          zl-user:y-start zl-user:y-end y-inc
                          zl-user:magnitude)))


;                                        10.6 ADVANCED-HAND-ACTUATION-OPTIONS
;                                        10.6.1 JKS MOVE-MENU
;                                        10.6.2 MOVE-TO-CONTACT-POINTS
;                                        10.6.3 GRASP-FORCE-CENTER-MENU

(defun grasp-force-center-menu ()
  (let ((magnitude 0)
        (force-center (zero-vector 3)))
    (zl-user:choose-user-options zl-user:constraint-state-menu)
    (setq magnitude zl-user:magnitude)
    (setq force-center (get-mouse-coordinates-grasp-frame))
    (move-to-grasp-center force-center magnitude)))

;                                        10.6.3.1 GRASP-FORCE-CENTER-MENU Auxiliary func!
tions

(defun get-mouse-coordinates-grasp-frame ()
  (let ((mouse-coordinates-view-frame (get-mouse-coordinates-view-frame)))
    (transform-normals-view-grasp-frame mouse-coordinates-view-frame)))


(defun get-mouse-coordinates-view-frame ()
  (let ((mouse-coordinates (get-mouse-coordinates))
        (mc-view-frame (zero-vector 3))
        (window-height (send grasp-screen :send-pane 'grasp-window ':height)))
    (setf (aref mc-view-frame 0) (/ (- (first mouse-coordinates) x-origin-3d) scale-3d))
    (setf (aref mc-view-frame 2) (/ (- window-height (+ (second mouse-coordinates) y-ori!
gin-3d))
                                    scale-3d))
    mc-view-frame))

(defun get-mouse-coordinates ()
  (let ((mouse-coordinates (cdr (multiple-value-list (tv:with-mouse-and-buttons-grabbed
                                                       (tv:wait-for-mouse-button-down)))!
)))
    mouse-coordinates))

(defun transform-normals-view-grasp-frame (point-view-frame)
  (let* ((grasp-to-view-frame (construct-grasp-to-view-frame))
         (a11 (aref grasp-to-view-frame 0 0))
         (a12 (aref grasp-to-view-frame 0 1))
         (a31 (aref grasp-to-view-frame 2 0))
         (a32 (aref grasp-to-view-frame 2 1))
         (a14 (aref grasp-to-view-frame 0 3))
         (a34 (aref grasp-to-view-frame 2 3))
         (determinant (- (* a11 a32) (* a12 a31)))
         (xv (aref point-view-frame 0))
         (zv (aref point-view-frame 2))
         (point-grasp-frame (zero-vector 3)))
    (setf (aref point-grasp-frame 0) (/ (+ (* a32 (- xv a14))
                                           (* -1 a12 (- zv a34))) determinant))
    (setf (aref point-grasp-frame 1) (/ (+ (* -1 a31 (- xv a14))
                                           (* a11 (- zv a34))) determinant))
    point-grasp-frame))

(defun construct-grasp-to-view-frame ()
  (let ((grasp-to-view-frame (zero-array 3 4)))
```

```
    (do ((i 0 (+ i 1)))
        ((= i 3))
      (do ((j 0 (+ j 1)))
          ((= j 3))
        (do ((k 0 (+ k 1)))
            ((= k 3))
          (setf (aref grasp-to-view-frame i j) (+ (* (aref *grasp-frame* k j)
                                                     (aref view-frame k i))
                                                  (aref grasp-to-view-frame i j))))))
    (do ((i 0 (+ i 1)))
        ((= i 3))
      (do ((j 0 (+ j 1)))
          ((= j 3))
        (setf (aref grasp-to-view-frame i 3) (+ (* (aref view-frame j i)
                                                   (- (aref *grasp-frame* j 3)
                                                      (aref view-frame j 3)))
                                                (aref grasp-to-view-frame i 3)))))
        (setf (aref grasp-to-view-frame 0 2) 0)
        (setf (aref grasp-to-view-frame 1 0) 0)
        (setf (aref grasp-to-view-frame 1 1) 0)
        (setf (aref grasp-to-view-frame 1 2) 0)
        (setf (aref grasp-to-view-frame 1 3) 0)
        (setf (aref grasp-to-view-frame 2 2) 0)
    grasp-to-view-frame))

;                                       10.6.4 DEMO-MENU

(defun demo-menu ()
  (do ((i 0 (+ i 1)))
    ((equalp (send dem-menu ':choose) 999) 'Done)))

;                                       10.7  GRASP-MENU

(defun grasp-menu ()
  (do ((i 0 (+ i 1)))
    ((equalp (send menu ':choose) 999) 'Done)))


;;;*****************************************************************************!
*****
;
;                                       11. Demos
;
;;;*****************************************************************************!
*****


;                                       11.1 DEMO-MENU
(defvar dem-menu)

(setq dem-menu (tv:make-window 'tv:momentary-menu
        ':label '(:string #.(zl:string "Advanced dexterity demonstrations:  "))
        ':geometry (list 1)
        ':borders 2
        ':item-list '(("Can demo" :funcall dlb-can-demo)
                      ("Can twirl demo" :funcall can-twirl)
                      ("Can spin demo" :funcall can-spin)
                      ("Box demo" :funcall box-spin)
                      ("QUIT"  :eval 999))))

;                                       11.2 Demonstration programs

;                                       11.2.0 DLB-CAN-DEMO

(defun dlb-can-demo ()
  (print "You have 3 seconds to place the can")
```

```
  (go-home-and-init)
  (sleep 3)
  (grab 4.2)
  (sleep 2)
  (move-in-load 5 :duration 0.8)
  (move-out-load 3 :duration 0.8)

  (move-left-load 5 :duration 0.6)
  (move-right-load 10 :duration 0.8)
  (move-left-load 5 :duration 0.6)

  (move-up-load 4 :duration 0.6)
  (move-down-load 6 :duration 1.0)
  (move-up-load 2 :duration 0.6)

  (rotate-x-load 10 :duration 0.8)
  (rotate-x-load -10 :duration 0.8)

  (rotate-y-load 30 :duration 0.8)
  (rotate-y-load -60 :duration 1.6)
  (rotate-y-load 30 :duration 0.8)

  (rotate-z-load 40 :duration 0.8)
  (rotate-z-load -80 :duration 1.6)
  (rotate-z-load 40 :duration 0.8)
  (go-hand)
  (sleep 15)
  (print "Finished absolute moves")
  (set-cube-frame)

  (move-left-load 5 :duration 0.6)
  (move-right-load 10 :duration 0.8)
  (move-left-load 5 :duration 0.6)
  (move-up-load 4 :duration 0.6)
  (move-down-load 6 :duration 1.0)
  (move-up-load 2 :duration 0.6)

  (rotate-x-load 20 :duration 0.8)
  (rotate-x-load -40 :duration 0.8)
  (rotate-x-load 20 :duration 0.8)
  (rotate-y-load 40 :duration 0.8)
  (rotate-y-load -80 :duration 1.6)
  (rotate-y-load 40 :duration 0.8)
  (rotate-z-load 40 :duration 0.8)
  (rotate-z-load -80 :duration 1.6)
  (rotate-z-load 40 :duration 0.8)

  (sleep 15)
  (go-hand)

  (move-up-load 2 :duration 0.6)
  (move-left-load 2 :duration 0.6)
  (move-down-load 4 :duration 1.2)
  (move-right-load 4 :duration 1.2)
  (move-up-load 4 :duration 1.2)
  (move-left-load 2 :duration 0.6)
  (move-down-load 2 :duration 0.6)
  (sleep 6)
  (go-hand)
  (send tg :back-to-basic))

;                                          11.2.1 TWIRL-CAN

(defun can-twirl ()
  (print "You have 3 seconds to place the can")
  (go-home)
```

```
  (sleep 3)
  (setq *object-frame* (make-array '(3 4) :initial-contents '((1 0 0 0)
                                                              (0 0 -1 5.5)
                                                              (0 1 0 -2.7))))
  (setq *contact-points-object-space*
      (list (make-array '(3) :initial-contents '(-0.8 1.3 2.5))
            (make-array '(3) :initial-contents '(0.8 1.3 2.5))
            (make-array '(3) :initial-contents '(0 -1.3 2.5))))
  (initialize-global-variables)
  (move-to-contact-points))
  (sleep 1)
  (let* ((force-magnitude 1.0)
         (force-center (zero-vector 3)))
    (move-to-grasp-center-load force-center force-magnitude)
    (move-right-load 4 :duration 0.6)
    (move-left-load 8 :duration 1.0)
    (move-right-load 4 :duration 0.6)
    (move-down-load 2 :duration 0.6)
    (move-up-load 4 :duration 1.0)
    (move-down-load 2 :duration 0.6)
    (move-to-grasp-center-load force-center force-magnitude)

    (setq force-magnitude 0.2)
    (setf (aref force-center 0) 0.9)
    (setf (aref force-center 1) 0)
    (move-to-grasp-center-load force-center force-magnitude)

    (setq force-magnitude 1.2)
    (setf (aref force-center 0) 0)
    (move-to-grasp-center-load force-center force-magnitude)
    (move-right-load 4 :duration 0.6)
    (move-left-load 8 :duration 1.0)
    (move-right-load 4 :duration 0.6)
    (move-down-load 2 :duration 0.6)
    (move-up-load 4 :duration 0.8)
    (move-down-load 2 :duration 1.0)
    (move-to-grasp-center-load force-center force-magnitude)
    (go-hand)))

;                                          11.2.2 CAN-CRAWL

;                                          11.2.2 CAN-SPIN
(defun can-spin ()
  (print "You have 2 seconds to place the can")
  (go-home)
  (sleep 2)
  (setq *object-frame* (make-array '(3 4) :initial-contents '((1 0 0 0)
                                                              (0 0 -1 5.5)
                                                              (0 1 0 -2.5))))
  (setq *contact-points-object-space*
      (list (make-array '(3) :initial-contents '(-0.8 1.3 2.5))
            (make-array '(3) :initial-contents '(0.8 1.3 2.5))
            (make-array '(3) :initial-contents '(0 -1.3 2.5))))
  (initialize-global-variables)
  (move-to-contact-points)
  (sleep 2)
  (let* ((force-magnitude 0.3)
         (force-center (zero-vector 3))
         (displacement (zero-vector 3)))
    (move-to-grasp-center-load force-center force-magnitude)

    (setq force-magnitude 0.4)
    (setf (aref force-center 0) -0.6)
    (move-to-grasp-center-load force-center force-magnitude)

    (setf (aref displacement 0) 0)
```

```
(setf (aref displacement 1) 1)
(setf (aref displacement 2) 1)
(move-finger-load 2 displacement)

(setf (aref displacement 0) 0)
(setf (aref displacement 1) 2)
(setf (aref displacement 2) -2)
(move-finger-load 2 displacement)

(setf (aref displacement 0) 0)
(setf (aref displacement 1) -4)
(setf (aref displacement 2) -1)
(move-finger-load 2 displacement)

(setf (aref displacement 0) 1)
(setf (aref displacement 1) 1)
(setf (aref displacement 2) 2)
(move-finger-load 2 displacement)

(setf (aref displacement 0) -0.5)
(setf (aref displacement 1) 3)
(setf (aref displacement 2) -2)
(move-finger-load 2 displacement)

(setf (aref displacement 0) -0.5)
(setf (aref displacement 1) -4.5)
(setf (aref displacement 2) -1)
(move-finger-load 2 displacement)

(setf (aref displacement 0) 0.5)
(setf (aref displacement 1) 1.5)
(setf (aref displacement 2) 3)
(move-finger-load 2 displacement)

(setf (aref displacement 0) 0)
(setf (aref displacement 1) 3)
(setf (aref displacement 2) 1)
(move-finger-load 2 displacement)

(setf (aref displacement 0) -0.5)
(setf (aref displacement 1) 1.5)
(setf (aref displacement 2) -0.5)
(move-finger-load 2 displacement)

(setf (aref displacement 0) 0)
(setf (aref displacement 1) -4)
(setf (aref displacement 2) -0.5)
(move-finger-load 2 displacement)

(setf (aref displacement 0) 1)
(setf (aref displacement 1) 1)
(setf (aref displacement 2) 1)
(move-finger-load 2 displacement)

(setf (aref displacement 0) 0)
(setf (aref displacement 1) 2)
(setf (aref displacement 2) 1)
(move-finger-load 2 displacement)

(setf (aref displacement 0) 0)
(setf (aref displacement 1) 0.5)
(setf (aref displacement 2) -2)
(move-finger-load 2 displacement)

(setf (aref displacement 0) 0)
(setf (aref displacement 1) -2.2)
```

```
    (setf (aref displacement 2) 0)
    (move-finger-load 2 displacement)

    (setf (aref displacement 0) 0)
    (setf (aref displacement 1) 0)
    (setf (aref displacement 2) 1)
    (move-finger-load 2 displacement)

    (setq force-magnitude 0.5)
    (setf (aref force-center 0) -0.6)
    (move-to-grasp-center-load force-center force-magnitude)

    (setq force-magnitude 0.5)
    (setf (aref force-center 0) 0)
    (move-to-grasp-center-load force-center force-magnitude)

    (go-hand)))

;                                        11.2.3 BOX-SPIN

(defun box-spin ()
  (print "You have 3 seconds to place the box")
  (go-home)
  (sleep 3)
  (setq *object-frame* (make-array '(3 4) :initial-contents '((1 0 0 1.5)
                                                              (0 1 0 3.0)
                                                              (0 0 1 -2.21))))
  (setq *contact-points-object-space*
      (list (make-array '(3) :initial-contents '(-0.8 0.0 0.8))
            (make-array '(3) :initial-contents '(0.8 0.0 0.8))
            (make-array '(3) :initial-contents '(0.8 0.0 -0.8))))
  (initialize-global-variables)
  (move-to-contact-points-load)
  (move-left-load 7.6)
  (go-hand)
  (clearscreen)
  (draw-coordinate-system)

  (let* ((force-magnitude 0.05)
         (force-center (zero-vector 3))
         (displacement (zero-vector 3)))
    (initialize-global-variables)
    (setf (aref force-center 0) 0.8)

    (do ((i 0 (+ i 1)))
        ((= i 3))
      (setq force-magnitude 0.2)
      (setf (aref force-center 0) 0.8)
      (move-to-grasp-center-load force-center force-magnitude)

;     Bring the finger back
      (setf (aref displacement 0) 1)
      (setf (aref displacement 1) 0)
      (setf (aref displacement 2) 2)
      (move-finger-load 1 displacement)
      (setf (aref displacement 0) 1.13)
      (setf (aref displacement 1) -1.13)
      (setf (aref displacement 2) 0)

;     Push in and rotate
      (move-finger-load 1 displacement)
      (setf (aref displacement 0) 0)
      (setf (aref displacement 1) 0)
      (setf (aref displacement 2) -2.5)
      (move-finger-load 1 displacement)
      (setf (aref displacement 0) -1.13)
```

```
        (setf (aref displacement 1) 1.13)
        (setf (aref displacement 2) 0)
        (move-finger-load 1 displacement)
        (setf (aref displacement 0) 1.13)
        (setf (aref displacement 1) 1.13)
        (setf (aref displacement 2) 0)

;       Bring the finger back
        (move-finger-load 1 displacement)
        (setf (aref displacement 0) 0)
        (setf (aref displacement 1) 0)
        (setf (aref displacement 2) 2.5)
        (move-finger-load 1 displacement)
        (setf (aref displacement 0) -1.13)
        (setf (aref displacement 1) -1.13)
        (setf (aref displacement 2) 0)
        (move-finger-load 1 displacement)
        (setf (aref displacement 0) 1.13)
        (setf (aref displacement 1) -1.13)
        (setf (aref displacement 2) 0)

;       Push in and rotate
        (move-finger-load 1 displacement)
        (setf (aref displacement 0) 0)
        (setf (aref displacement 1) 0)
        (setf (aref displacement 2) -2.5)
        (move-finger-load 1 displacement)
        (setf (aref displacement 0) -1.13)
        (setf (aref displacement 1) 1.13)
        (setf (aref displacement 2) 0)
        (move-finger-load 1 displacement)
        (setf (aref displacement 0) 1.13)
        (setf (aref displacement 1) 1.13)
        (setf (aref displacement 2) 0)
        (move-finger-load 1 displacement)

;       Bring the finger back
        (setf (aref displacement 0) 0)
        (setf (aref displacement 1) 0)
        (setf (aref displacement 2) 2.5)
        (move-finger-load 1 displacement)
        (setf (aref displacement 0) -1.13)
        (setf (aref displacement 1) -1.13)
        (setf (aref displacement 2) 0)
        (move-finger-load 1 displacement)
        (setf (aref displacement 0) 1.13)
        (setf (aref displacement 1) -1.13)
        (setf (aref displacement 2) 0)

;       Push in and rotate
        (move-finger-load 1 displacement)
        (setf (aref displacement 0) 0)
        (setf (aref displacement 1) 0)
        (setf (aref displacement 2) -2.5)
        (move-finger-load 1 displacement)
        (setf (aref displacement 0) -1.13)
        (setf (aref displacement 1) 1.13)
        (setf (aref displacement 2) 0)
        (move-finger-load 1 displacement)
        (setf (aref displacement 0) 1.13)
        (setf (aref displacement 1) 1.13)
        (setf (aref displacement 2) 0)

;       Bring the finger back
        (move-finger-load 1 displacement)
        (setf (aref displacement 0) 0)
```

```
      (setf (aref displacement 1) 1)
      (setf (aref displacement 2) 3)
      (move-finger-load 1 displacement)
      (setf (aref displacement 0) -1.13)
      (setf (aref displacement 1) -1.13)
      (setf (aref displacement 2) 0)
      (move-finger-load 1 displacement)

      (setq force-magnitude 0.0)
      (setf (aref force-center 0) 0)
      (move-to-grasp-center-load force-center force-magnitude)
      (move-right-load 7.6)
      (setq force-magnitude 0.3)
      (setf (aref force-center 0) -0.5)
      (move-to-grasp-center-load force-center force-magnitude)
      (move-left-load 7.6))
    (go-hand)))



(defun back-rub()
  (print "You have 2 seconds to place the back")
  (go-home)
  (sleep 2)
  (setq *object-frame* (make-array '(3 4) :initial-contents '((1 0 0 0)
                                                              (0 0 -1 5.5)
                                                              (0 1 0 -2.5))))
  (setq *contact-points-object-space*
      (list (make-array '(3) :initial-contents '(-0.8 1.3 2.5))
            (make-array '(3) :initial-contents '(0.8 1.3 2.5))
            (make-array '(3) :initial-contents '(0 -1.3 2.5))))
  (initialize-global-variables)
  (move-to-contact-points)
  (go-hand))


  (sleep 2)
  (let* ((force-magnitude 0.3)
         (force-center (zero-vector 3))
         (displacement (zero-vector 3)))
    (move-to-grasp-center-load force-center force-magnitude)

    (setq force-magnitude 0.4)
    (setf (aref force-center 0) -0.6)
    (move-to-grasp-center-load force-center force-magnitude)

    (setf (aref displacement 0) 0)
    (setf (aref displacement 1) 1)
    (setf (aref displacement 2) 1)
    (move-finger-load 2 displacement)

    (setf (aref displacement 0) 0)
    (setf (aref displacement 1) 2)
    (setf (aref displacement 2) -2)
    (move-finger-load 2 displacement)

    (setf (aref displacement 0) 0)
    (setf (aref displacement 1) -4)
    (setf (aref displacement 2) -1)
    (move-finger-load 2 displacement)

    (setf (aref displacement 0) 1)
    (setf (aref displacement 1) 1)
    (setf (aref displacement 2) 2)
    (move-finger-load 2 displacement)
```

```
    (setf (aref displacement 0) -1)
    (setf (aref displacement 1) 3)
    (setf (aref displacement 2) -2)
    (move-finger-load 2 displacement)

    (setf (aref displacement 0) -0.5)
    (setf (aref displacement 1) -4.5)
    (setf (aref displacement 2) -1)
    (move-finger-load 2 displacement)

    (setf (aref displacement 0) 0.5)
    (setf (aref displacement 1) 1.5)
    (setf (aref displacement 2) 3)
    (move-finger-load 2 displacement)

    (setf (aref displacement 0) 0)
    (setf (aref displacement 1) 3)
    (setf (aref displacement 2) 1)
    (move-finger-load 2 displacement)

    (setf (aref displacement 0) -1)
    (setf (aref displacement 1) 1.5)
    (setf (aref displacement 2) -0.5)
    (move-finger-load 2 displacement)

    (setf (aref displacement 0) 0)
    (setf (aref displacement 1) -5)
    (setf (aref displacement 2) -1)
    (move-finger-load 2 displacement)

    (setf (aref displacement 0) 1)
    (setf (aref displacement 1) 1)
    (setf (aref displacement 2) 1)
    (move-finger-load 2 displacement)

    (setf (aref displacement 0) 0)
    (setf (aref displacement 1) 2)
    (setf (aref displacement 2) 1)
    (move-finger-load 2 displacement)

    (setf (aref displacement 0) 0)
    (setf (aref displacement 1) 0.5)
    (setf (aref displacement 2) -2)
    (move-finger-load 2 displacement)

    (setf (aref displacement 0) 0)
    (setf (aref displacement 1) -2.5)
    (setf (aref displacement 2) 0)
    (move-finger-load 2 displacement)

    (setf (aref displacement 0) 0)
    (setf (aref displacement 1) 0)
    (setf (aref displacement 2) 1)
    (move-finger-load 2 displacement)

    (setq force-magnitude 0.5)
    (setf (aref force-center 0) -0.6)
    (move-to-grasp-center-load force-center force-magnitude)

    (setq force-magnitude 0.5)
    (setf (aref force-center 0) 0)
    (move-to-grasp-center-load force-center force-magnitude)

    (go-hand)))

(defun pic ()
```

```
(sleep 10)
(let ((displacement (make-array '(3))))
  (setf (aref displacement 0) 0.0)
  (setf (aref displacement 1) 0.0)
  (setf (aref displacement 2) -2.5 )
  (move-finger-load sl-user:finger-number displacement)
  (go-hand)
  (setf (aref displacement 0) 0.0)
  (setf (aref displacement 1) 0.0)
  (setf (aref displacement 2) 2.5 )
  (move-finger-load sl-user:finger-number displacement)
  (go-hand)))
```

CS-TR Scanning Project
Document Control Form

Date : 9 / 28 / 95

Report # AI-TR-992

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 220 (226 - IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:
☒ DOD Form (2) ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): Follow ii, iii

Photographs/Tonal Material (by page number): 85, 93, 95-97,

Other (note description/page number):

Description :	Page Number:
IMAGE MAP: (1-12) PAGES #'ED ii, UN# BLANK, iii, UN# BLK, IV-XI	
(13-168) PAGES #'ED 1-156	APPENDIX D
(169-220) 52 UN#'ED PAGES (FILE: CONSTRAINTS.LISP)	
(221-226) SCANCONTROL, DOD (2), TRGT'S (3)	

Scanning Agent Signoff:
Date Received: 9 / 28 / 95 Date Scanned: 10 / 11 / 95 Date Returned: 10 / 12 / 95

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR 992	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Enhancing the Dexterity of a Robot Hand Using Controlled Slip		5. TYPE OF REPORT & PERIOD COVERED technical report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) David Lawrence Brock		8. CONTRACT OR GRANT NUMBER(s) N00014-86-K-0685 N00014-85-K-0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE May 1987 13. NUMBER OF PAGES 218
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

robot hand
dexterity
slip
controlled slip

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Abstract. Humans can effortlessly manipulate objects in their hands, dexterously sliding and twisting them within their grasp. Robots, however, have none of these capabilities, they simply grasp objects rigidly in their end effectors. To investigate this common form of human manipulation, an analysis of controlled slipping of a grasped object within a robot hand was performed. The Salisbury robot hand demonstrated many of these controlled slipping techniques, illustrating many results of this analysis.

over →

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block 20 cont.

First, the possible slipping motions were found as a function of the location, orientation, and types of contact between the hand and the object. Second, for a given grasp, the contact types were determined as a function of the grasping force and the external forces on the object. Finally, by changing the grasping force, the robot modified the constraints on the object and affect controlled slipping motions.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 10/11/1995

M.I.T. Libraries
Document Services