Dexterous Robotic Hands: Kinematics and Control

by

Sundar Narasimhan
B. Tech. Mechanical Engineering
Indian Institute of Technology, Madras, India
(1983)

Submitted in Partial Fulfillment
of the Requirements of the
Degree of

Master of Science
in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology
January 1988

Signature of Author _____

Department of Electrical Engineering and Computer Science
January, 1988

Certified by _____

Prof. John M. Hollerbach
Thesis Supervisor

Accepted by _____

Prof. Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Dexterous Robotic Hands: Kinematics and Control

by

Sundar Narasimhan

Submitted to the Department of Electrical Engineering and Computer Science on October, 1987 in partial fulfillment of the requirements for the Degree of Master of Science in Electrical Engineering and Computer Science

Abstract. This thesis presents issues relating to the kinematics and control of dexterous robotic hands using the Utah-MIT hand as an illustrative example. The emphasis throughout is on the actual implementation and testing of the theoretical concepts presented. The kinematics of such hands is interesting and complicated owing to the large number of degrees of freedom involved.

The implementation of position and force control algorithms on such tendon driven hands has previously suffered from inefficient formulations and a lack of sophisticated computer hardware. Both these problems are addressed in this thesis. A multiprocessor architecture has been built with high performance microcomputers on which real-time algorithms can be efficiently implemented. A large software library has also been built to facilitate flexible software development on this architecture. The position and force control algorithms described herein have been implemented and tested on this hardware.

Thesis Supervisor: Dr. John M. Hollerbach
Associate Professor of Brain and Cognitive Sciences

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 the System Development Foundation, in part by the Office of Naval Research under contract N00014-81-K-0494, and in part by the Advanced Research Project Agency under Office of Naval Research contracts N00014-80-C-050 and N00014-82-K-0334. Support for the author is currently provided by the University Research Initiative program conducted by the Office of Naval Research under contract number N00014-86-K-0685.

Acknowledgements

There are a number of people who've inspired, taught, and worked with me during these years.

First and foremost, I thank David Siegel, who has been responsible for the hardware design of both the Version I and II systems on which most of the work was implemented. Without him, the hardware would simply not exist. He has been fun to work with, and has taught me many tricks. We have had to fix bad blocks on disks, fix gateways, bring up operating systems on bare machines, run innumerable cables, and do other such fun things. He has always been around to share the work, and share cold pizza at 3am.

This work would not have been possible without the commitment, faith and support from my advisor, Prof. John Hollerbach. His dedication toward experimental validation of theoretical results has been inspiring. I thank him for his encouragement, and for his understanding of the trials of being a graduate student.

I must also acknowledge the tremendous contribution made by a number of people to the CONDOR development environment described in this thesis. It has been their work as much as it has been mine. Firstly, I would like to thank the Real Time Systems Group working under Prof. Robert Halstead. Our first implementation was based largely on their *Concert* system. Secondly, I would like to thank David Kriegman, and George Gerpheide for sharing a summer working on the first implementation of the CONDOR. Next, I would like to thank David Taylor for designing and implementing the ptrace emulator that forms the basis for our debugger and modifying the Sun kernel to receive vectored interrupts from the Ironics processors. I also thank Steve Drucker for sharing the initial grunge work on the user interface code.

Chris Atkeson and Chae An have been good friends – their complaining about bugs in my software on Version I was partly the reason for the cleanliness of Version II. Steve Buckley, Bruce Donald, Vanduc Nguyen, Michael Erdmann, John Canny, and the rest of the Motion Planning group have been fun to get around with. Through them, I have gained the knowledge to appreciate that this thesis could be considered in some circles as 'mere implementation detail'.

I'd also like to thank the official and unofficial support staff at the Artificial Intelligence Laboratory. People like Chris Lindblad, Gerry Roylance, Laurel Simmons, Penny Berman, Ron Wiken, Inaki Garabieta and Peggy Fong keep the laboratory running. Their jobs are often trying and unrewarding, and I thank them for coming through for the rest of us.

Inaki Garabieta, Noble Larson, and John Purbrick shared their technical expertise with me on more than one occasion. I'd also like to thank Priscilla Cobb, Marilyn Melithoniotes and the rest of the fiscal office for tracking down all those purchase orders for me.

Most of all, I'd like to thank everyone at the Artificial Intelligence Laboratory for making it the place it is.

And last but not the least, I thank the people at Utah, notably Prof. Steve Jacobsen and Prof. John Wood, who led the project there, Ed Iverson, Jim Olson, Don Knutti and Tony Jacobs, who built the device that still puzzles me at times, and Klaus Biggers for yelling at me at the right times to unwedge me.

Contents

1 Introduction 1

1.1 Dexterous Robot Hands 2

1.2 A Framework for Hand Control 5

1.2.1 Forward Kinematics 6

1.2.2 Inverse Kinematics 7

1.2.3 Sensing 8

1.2.4 Programming 9

1.2.5 Modeling and Planning 11

1.2.6 Error Detection and Recovery 13

1.2.7 Engineering and Design Issues 13

1.3 Outline of this Thesis 14

2 Dexterous Hands - Past and Future 16

2.1 Design Issues 16

2.1.1 Early Studies 16

2.1.2 Industrial Grippers 17

2.1.3 Dexterous Robot Hands 18

2.2 Workspace and Kinematics 19

2.3 Work in Sensing 20

2.4 Work in Control 20

2.5 Planning and Programming 21

3 Kinematic Issues 23

3.1 Description of the Utah-MIT Hand/Arm System 23

3.2 Forward Kinematics 25

3.2.1 Forward Kinematics of the Utah-MIT Hand 27
3.3 Inverse Kinematics 29
3.3.1 Inverse Kinematics of the Utah-MIT Hand 30
3.4 Other Issues 38

4 Control 40
4.1 Tendon Management 41
4.2 Joint Level Control 42
4.3 Implementation Issues 52
4.4 Higher Level Control 53
4.4.1 Hand Primitive Motions 53
4.4.2 Cartesian Space Control 57
4.4.3 Motions Specified using Homogenous Transforms 59
4.4.3.1 Motion Specified Relative to an Absolute Reference 59
4.4.3.2 Motion Specified Relative to an Alternate Frame 60
4.4.4 Motions Specified using Quaternions 62

5 Force Control 64
5.1 Introduction and Previous Work 64
5.2 The Force Control Algorithm 67
5.2.1 Lines, Screws, Wrenches, and Twists 67
5.2.2 Internal Forces 68
5.3 Algorithm 71
5.4 Computing the Object Displacement 72
5.5 Computing the Fingertip Forces 75
5.5.1 Three Point Contact with Friction 76
5.5.2 Four Point Contact with Friction 78
5.6 Computing the Joint Torques 83
5.7 Implementation 84
5.8 Future Work 85

6 Computational Architecture 86
6.1 Design Motivation 87
6.1.1 Comparison between Version I and Version II 92
6.2 Software 94

6.2.1 Devices 96
6.2.2 Interrupts 100
6.2.3 Message Passing 101
6.2.3.1 Introduction 101
6.2.3.2 Messages 101
6.2.3.3 Support for Message Passing on the Sun End 105
6.2.3.4 Message Passing and its Implication for Control 107
6.2.4 Virtual Terminals, The File Server, Debugging 109
6.2.4.1 The Pseudo Terminal Emulator 109
6.2.4.2 File Server 110
6.2.4.3 The Debugger 111
6.2.5 The CONDOR User Interface 113
6.2.6 Controller Implementation 113

7 Conclusions and Future Work **115**
7.1 Control 115
7.2 Planning 116
7.3 Sensing 117
7.4 Conclusion 117

References **118**

A Kinematics of the Utah-MIT hand **132**
A.1 Co-ordinate transforms based on D-H matrices 132
A.2 The thumb frames 138

B Function Index for the Condor system **141**
B.1 Programs 141
B.2 Math Library 142
B.2.1 Vectors and Matrices 143
B.2.2 Vectors 148
B.2.3 Homogenous Transforms 149
B.3 Storage Management 150
B.3.1 Examining and modifying memory locations 150
B.4 The I/O Package 151

B.4.1 Strings Library 152
B.5 Data Transfer routines 153
B.6 Simple Real Time Tasking 154
B.7 Dealing with multiple processors 155
B.8 Command Parser Library - Input routines 156
B.8.1 Miscellaneous input routines 157
B.8.2 Window system functions 157
B.9 Hash Tables 157
B.10 Buffer routines 158
B.11 Tree library 159
B.12 Small set package 160
B.13 Message Passing routines 161
B.14 Support for Real Time tasks 162
B.15 Debugging Support 163
B.16 Conclusion 164

List of Figures

1.1 The Stanford-JPL Hand 2
1.2 Picture of a conventional 2-jaw gripper 3
1.3 Block diagram of controller 6

3.1 Picture of the Utah-MIT hand 23
3.2 Picture of the Utah-MIT hand arm system 25
3.3 Vertical trajectory using fixed orientation constraint 34
3.4 Horizontal trajectory using fixed orientation constraint 34
3.5 Vertical trajectory using equal angles constraint constraint 36
3.6 Horizontal trajectory using equal angles constraint constraint 36
3.7 Linear trajectory using equal angles constraint constraint 37

4.1 The simple rectification function used 42
4.2 Step response from second order joint model 43
4.3 Actual step response 44
4.4 Single joint example 44
4.5 Tendon tensions - summing before rectifying 46
4.6 Tendon tensions - rectification before summing 47
4.7 Actual tendon tensions 47
4.8 Simple 2-joint controlled by four actuators 48
4.9 Coupling action between joints 48
4.10 Step response after decoupling 50
4.11 Actual torques to show effect of decoupling 51
4.12 Joint controller black box 51

5.1 Three point contact with friction 76
5.2 Four point contact with friction 80

6.1 Block Diagram of Version I of the hardware 91
6.2 Block Diagram of Version II of the hardware 92
6.3 The CONDOR running under the X Window System 113

A.1 Numbering of the joints on the Utah-MIT hand 132
A.2 Picture of a single finger 133
A.3 The relation between the palm and the zeroeth joint frame 134
A.4 The first two frames for the thumb 139

List of Tables

3.1 D-H Parameters for Non-Thumb Fingers 28
3.2 D-H Parameters for the Thumb 28

4.1 Performance of the finger level controller 52
4.2 Performance of the Version II finger level controller 53
4.3 Hand Primitives 55

5.1 Summary of external force components 70
5.2 Summary of computational requirements: Three point contact 79
5.3 Summary of computational requirements: Four point contact 82
5.4 Summary of computations for the force control algorithm 84

6.1 Comparisons of processing power available from alternative hardware configurations 89
6.2 Comparisons of different types of interconnect 89
6.3 Performance of the Message Passing System 108
6.4 Opcodes for messages implementing ptrace 112

A.1 D-H Parameters for Non-Thumb Fingers 136
A.2 D-H Parameters for the Thumb 140

Within the artificial intelligence community, robotics is often characterized as having to do with "relating perception to action". Perception deals with the acquisition of information through a wide variety of mechanisms, and the process of using such information to affect the state of the world intelligently is the task of present day robotics.

While there have been great strides in our understanding of some of the problems associated with "relating perception to action", there are others which still defy the concentrated attack of a number of research efforts.

One of the important capabilities required of robots is the ability to grasp and manipulate objects. In order that we discover the fundamental principles that guide manipulation, it is extremely important that theoretical analyses proceed hand in hand with practical implementations. The increasing availability of dexterous hands and other mechanically sophisticated devices, provides ample scope for constructing testbeds for experimentation.

It must also be mentioned that while the technology for building better robots has advanced tremendously, our understanding of how to best use these robots has scarcely widened. Partly this has been due to the lack of sophisticated computer architectures to control robots. Often, (even in research laboratories) one finds a costly robot hooked up to an old computer on its last legs, programmed by a few wizards in an arcane assembler language. Such situations result in ever-widening gaps between theory and practice, with most theories of manipulation being relegated to doctoral theses reports.

In this thesis, I hope to present a view of robotics as it applies to dexterous robotic hands. These robotic hands tend to be extremely complex devices with a large number of joints compared with the traditional six degree of freedom robot. Consequently, as we will see in subsequent chapters, problems that have been considered solved for all theoretical purposes, reappear owing to purely practical considerations. Besides the problems that beset conventional robots, dexterous hands give rise to quite a few problems of their own.

This thesis addresses a number of issues that arise when one tries to implement control algorithms on such robot hands, using the Utah-MIT hand as an illustrative example. The problems that had to be solved involved kinematics, control and computational architecture issues, and consequently involved different areas of theoretical and experimental inquiry. It

is my hope that after reading this thesis the reader will get a sense of the issues involved and get some insight into a few of the problems that I have tried to address.

1.1 Dexterous Robot Hands

One of the glaring deficiencies in today's robots is in their flexibility. The earliest arguments for the use of robots indeed stressed their versatility when compared to conventional machine tools. And yet, today one finds 85 percent of most robots used in industry being relegated to tasks like spot welding or spray painting – tasks that do not require a great deal of dexterity or co-ordination.

The problem of building mechanically better robots capable of higher performance is being addressed (see Salisbury [1982], Jacobsen et al. [1983]). In this thesis, I deal exclusively with such advanced robots which are mechanically more complex than conventional robots. Their actuation and transmission systems are rather sophisticated. These *dexterous* hands as they are called, are devices with a number of joints (see Fig. 1.1 for a picture of the Stanford-JPL hand which is an example of such a device).

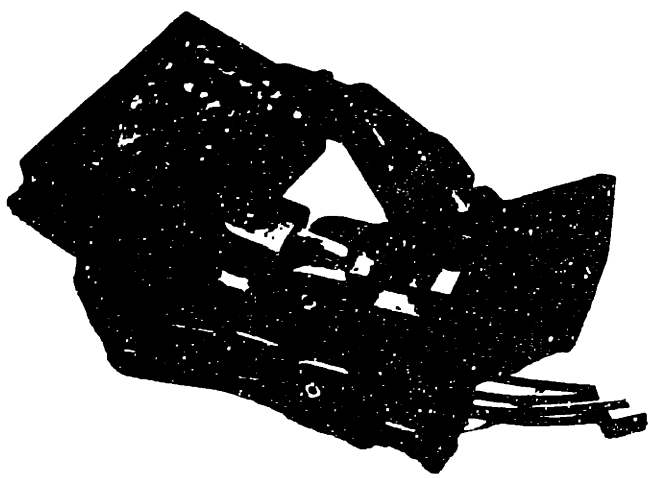


Figure 1.1: The Stanford-JPL Hand.

Since robots were invented, there has been a wide variation in their designs, but most of them are equipped with what has become famous as the *two-jaw* gripper. This gripper is the robot's end-effector, its primary way of interacting with objects in its environment

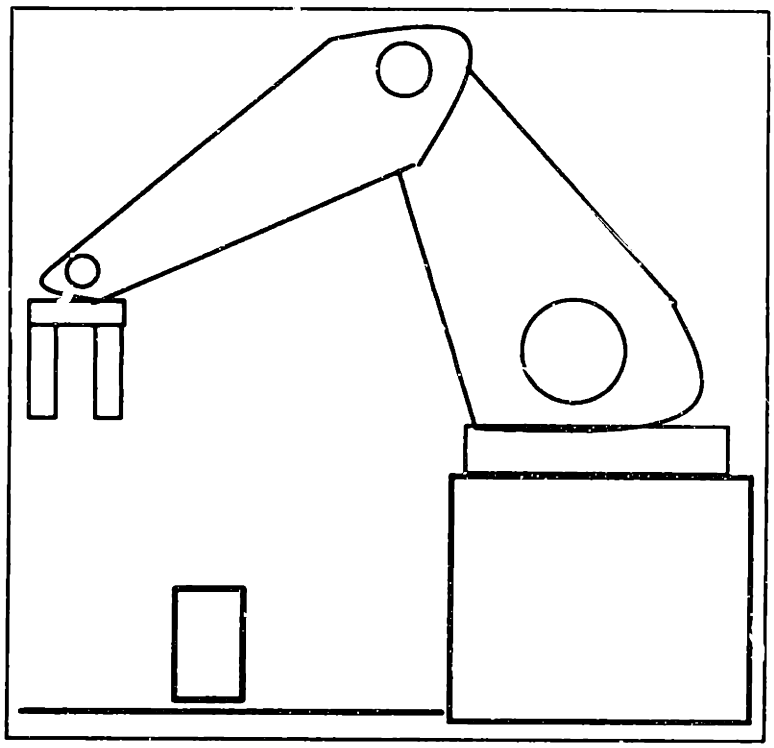


Figure 1.2: Picture of a conventional 2-jaw gripper.

(see Figure 1.2). But as is rather obvious, the range of motions that a robot can impart to a grasped object and the range of forces it can exert on it with such a gripper, is rather limited. Dexterous hands with a number of joints afford a more flexible alternative to this primitive form of grasping and manipulating objects.

There are a number of reasons why studying such hands could be immensely useful.

- *Flexibility argument:*

  Conventional robot grippers are often just a pair of vice-like jaws. Articulated mechanical hands with a number of fingers are much more versatile. Such hands, with their large number of joints, can adapt to objects of different shapes. They can grasp objects more stably because of a larger number of points of contact or gripping surfaces. They can perform fine motions without moving the entire arm around, and can reorient the grasped object without regrasping.

  Augmented with sensory capabilities, these hands will become useful tools for haptic exploration. Touch sensors can help in the identification and recognition of objects

grasped by such hands. They can also help in determining properties of objects like surface texture and temperature, and provide information as to what kind of contact is being made between the hand and a grasped object. Such information is extremely important to understand the physics of manipulation tasks, but is not usually accessible to non-contact sensing modalities like vision.

- *The Artificial Intelligence argument:*
  Our current understanding of machine dexterity is fairly low. One of the important capabilities required of intelligent, autonomous machines, is that they be able to manipulate the objects in their immediate environment. This is necessary to acquire active information about objects that passive sensory mechanisms like vision cannot provide, to interact with these objects in a meaningful fashion as required by the task, and to use and manipulate tools that vary in size and shape. With hands that have multiple fingers and multiple degrees of freedom, such a capability becomes possible.

  Recently, there have been efforts to move away from the "blocks-world" paradigm in A.I. and build real systems that must deal with the nature of the real world (see for example Brooks [1986]). This thesis represents an effort to deal with issues involved in building and programming a robot to manipulate objects.

- *Empirical argument:*
  Robot hands are complex mechanical devices, far more complex than present day robots. From a kinematic viewpoint, this makes them interesting objects worth studying for their own sake. The study of such complex kinematic chains can lead to a better understanding of some of the basic principles involved in machine design and pave the way to the synthesis of better robots and hands.

  Besides this, studying such hands can also provide insight into the study of how human hands function. Such an understanding could have substantial impact on human hand prostheses, and our understanding of biological motor control.

Although some of the earliest mechanical hands were aimed at prosthetic applications, the focus of present-day robot hands has shifted toward researchers in robotics laboratories. These later attempts are aimed at providing a basic understanding of what it means to achieve ***dexterous manipulation.*** A number of such hands have been built, and hand control research has become an active field of inquiry in the past few years.

1.2 A Framework for Hand Control

In this section an overall framework for robot hand control and planning is presented. Different problems that arise in the control and programming of robot hands are introduced, and I hope that such a discussion will provide the motivation for the work presented in this thesis. Such a discussion should also be helpful in evaluating and understanding the contributions made by previous and future research efforts. A brief summary of previous work pertaining to hand control research is presented in the next chapter.

Hand control involves a number of different issues. To begin with, it is illustrative to consider the block diagram shown in Figure 1.3. As can be seen from this relatively high level specification, there are basically two kinds of hand control which follow from their counterparts in conventional robotics. We will first briefly describe what these types of control involve, and then enumerate the problems to be solved before they can be implemented to run efficiently.

Pure position control of a conventional robot involves controlling the robot's position at all times to achieve a specified task. For controlling a revolute manipulator, for example, these positions could be a stream of joint angles over time, or the cartesian positions and orientations of the tip of the manipulator. The criteria by which a position controller's performance can be quantitatively measured involve both *accuracy* and *repeatability*. Grasping and manipulating an object can be specified purely in terms of the motions that the fingers go through, if each of the fingers is treated as an independent robot. It is easier however to think of the motions that a grasped object is making, and specify these motions in terms of this object. Such a specification would be modular and would be independent of the particular kind of robot hand used.

Force control, on the other hand, involves specifying and controlling the forces of interaction between a robot and its environment. Since a robot hand is almost always in contact with an object in its environment while it is performing useful work, this mode of control is uniquely important for robot hands. Such a force controller specification usually takes the form of a stiffness matrix that the grasped object together with the hand control system present to the environment.

There are many problems that arise in actually implementing these types of control algorithms.

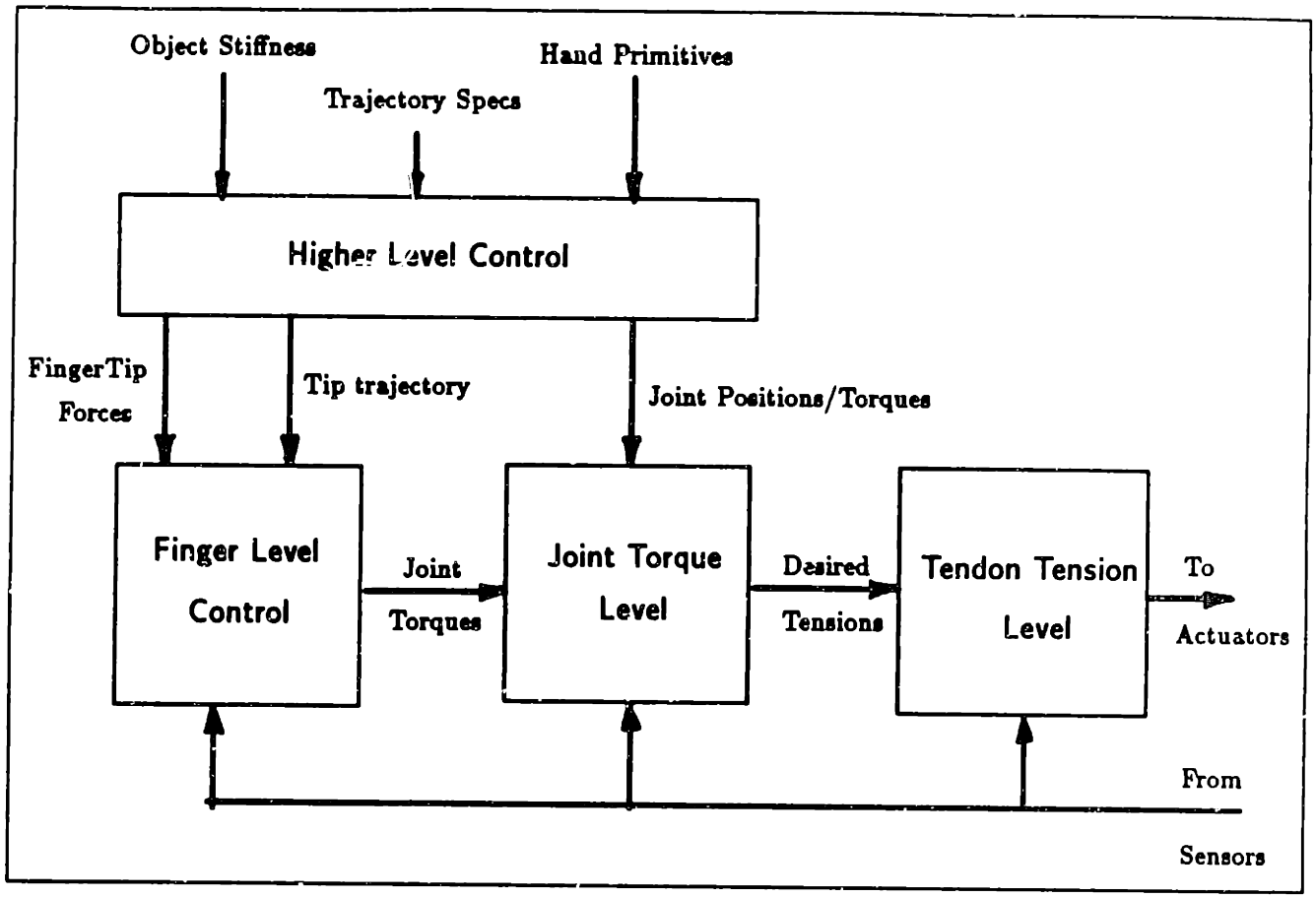


Figure 1.3: Block diagram of the controller.

1.2.1 Forward Kinematics

This refers to the problem of computing the cartesian co-ordinates of the finger tips of an articulated hand from its various joint angles. Fortunately, most of the articulated hands' fingers are individually less complex than conventional six degree of freedom robots. Hence the forward kinematic problem is usually simpler for articulated hands at the finger level. Such a treatment can be useful, for example, in the initial phases of a manipulation operation, while the hand's fingers are moving in free space, and are not in contact with the environment.

At the object level, once the fingers of a robot hand come into contact with a grasped object, the fingers can no longer be approximated as serial link kinematic chains. What

we really have is a closed loop kinematic chain, whose degrees of freedom at the contacting surfaces depend on the type of contact made. The forward kinematic problem for such a closed loop chain mechanism involves computing the position and orientation of the grasped object (which is the output variable that one is interested in), given the various joint angles of the robot hand. This is a harder problem for robot hands than it is for conventional robots.

Another factor that compounds the problem in the case of robot hands is that multi-link contact occurs often. While conventional robots typically rely on making contact with a grasped object with only the tips of their end effector, this is not the case with dexterous hands. We only need to look at any typical human manipulation task to realize that we almost always make contact with a grasped object at more surfaces of our fingers than just the finger tips. With multi-link contact, one is no longer interested in just the positions and orientations of the end-point or end-effector, but in the positions and orientations of intermediate links as well. Not only does such a contact help secure a grasped object more stably, but it also restricts the mobility of the hand's joints in a complex fashion.

To summarize, the problems are:

1. Computing the configuration of the finger tips given the joint angles of each finger.
2. Computing the configuration of a grasped object given the joint angles, assuming a finger-tip grasp.
3. Computing the configuration of a grasped object given the joint angles and locations of contact on the various finger joints.

1.2.2 Inverse Kinematics

The problem of computing the joint angles given the cartesian co-ordinates of the finger tips or end-effector is more complicated, even for conventional robots. In the case of articulated hands, the problem is exacerbated by the enormous amount of redundancy involved. From our statement of the forward kinematic problem for robot hands, it can be seen that the inverse kinematic problem is one of finding out the joint angles corresponding to a configuration of the grasped object.

If purely finger tip[1] grasps are assumed, this problem can be broken into two subproblems that involve

[1] A *finger tip* grasp is one wherein an object is grasped purely by the tips of the serial kinematic chains that form the hand's various fingers. Such a grasp leads to every closed loop in the system containing all

1. computing the positions of the finger tips given the position and orientation of the grasped object, and geometric information pertaining to the grasp, and then

2. computing the joint angles of each finger given its finger tip position and orientation.

Only the second portion of such a computation would then depend on the kinematics of the particular hand involved. A similar formulation can also be obtained for the forward kinematic problem given this rather restrictive assumption.

For a purely positioning and orienting task, the number of degrees of freedom required is only six, and yet the human hand-arm system comprises twenty-nine degrees of freedom. This would seem to be far more than what most tasks typically require. The success of the human hand-arm system in manipulation tasks, however, serves as an existence proof for the desirability of such redundancy. Even with very simple robot hands, multiple inverse kinematic solutions can be expected to exist. This points to the need for quantitative criteria that can help optimization algorithms choose the best configurations of the hand's joints for particular tasks. As is the case with most optimizing problems, the problems involved are

1. defining computable indices of performance, and

2. developing algorithms that can use these indices to compute optimal solutions.

1.2.3 Sensing

It is possible that the success of the human hand in manipulation tasks is purely related to the sophistication and rich variety of the sensor information that the motor system can make use of when needed. Current robots and robot hands are not very sophisticated devices when it comes to sensing their environment. Most robots come equipped with position and maybe velocity sensors but very little else. In manipulation tasks however, it is important to know where contact is being made with a grasped object along the finger tips and how the nature of this contact is changing.

Designing sensors for dexterous hands is complicated for two reasons:

1. Since there are a large number of joints and actuators, each sensor has to be replicated a number of times. The enormous number of sensors necessitates sophisticated multiplexing and layout schemes to minimize wiring and maximize reliability.

---

the links of two fingers and the grasped object.

2. The space constraints in a dexterous hand are rather severe. Often sensors must be designed to fit into the only space available for them, which may make it impossible to use certain transduction technologies.

There are many ways in which sensors interact with the controller that may be structured as shown in Figure 1.3.

Conventional position, velocity and force sensors can be used to implement stable and repeatable position and force control feedback loops. Touch sensors, on the other hand, can be used to sense when contact is made and precisely what kind of contact is being made. The *object localization problem*, refers to the problem of computing the configuration of a grasped object based on information obtained from such sensors.

Touch sensors can be used at the higher levels to implement *guarded moves*. For example, most intermediate level robot programming languages have a construct similar to the one shown below:

```
move y until force y > 50
```

Such conditional action statements can form the basis for iteration, execution or logic branching and even recursion.

Besides such statements that trigger when force or position values exceed certain thresholds, other uses for sensor information are also possible. One possibility that has been suggested in the force control literature is the switching of control laws once contact has been made with a surface in the environment.

Sensor information could also be used to trigger further planning steps. Such a flow of information from the lowest to the highest levels in the control hierarchy has begun to be explored only recently. Information about errors occurring during the execution of a task can be used to *learn* over successive trials. Tactile sensors have been constructed and even mounted on finger tips of a dexterous hand, but information provided by such sensors has until now not been used for control or planning.

1.2.4 Programming

Robot programming is still a tedious and unrewarding task. Starting with teach pendants of yester years, we have progressed today to what one may call *intermediate level* programming languages. A number of research papers have addressed the issue of *automatic* and *task level* programming but none have yet resulted in practical systems. This

may partly be due to the difficulty of the problem but also due to considerable ambiguity in what researchers typically consider a task to be. Even now, no useful taxonomy of *tasks* exists that a theoretician or a practitioner can agree upon as a set of useful tasks that a robot must be capable of doing. The search for a truly general purpose task level programming language can be considered similar to the problem of searching for a truly general purpose instruction set to program computers with.

Current robot programming languages require the user or programmer to be aware of too much detail with respect to the kinematics and control. If a programming language is to be successful at all, it has to endeavor to hide this complexity below workable abstractions. It is easy to write statements in a fictitious automatic programming language like

```
moveto coke_can
grasp  coke_can
pickup coke_can
```

without realizing the number of problems that need to be solved before even a subset of these primitives can be implemented.

One should also be wary of the temptation to shy away from these high level programming languages. At the lowest level, robots are composed of joints and hence, in principle, can always be programmed in joint level programming languages. But such methods are akin to using machine and assembler level languages to attack computational problems.

One argument against automatic programming has been purely economic. The proponents of this argument claim that if the time taken to develop and debug a robot program is x and the time it spends running (i.e. actually executing) is y, it makes sense to reduce x only if it is considerable when compared to the production time. This does happen to be true for complex robots today. Higher level programming languages not only reduce design and debug time, but they are infinitely more readable and easy to maintain. Moreover, they can be capable of doing tasks that low level robot programs can never do.

It is my view that the conventional practice of extending regular programming languages to perform robot manipulation is perhaps the best we can hope for in the near future. Given this basic assumption, to program complex robots like the Utah-MIT hand there should be available to the programmer a hierarchy of tools. He should be able to drop down to the individual joint level if need be, and be able to use the high level abstractions in cases where such facilities are needed. Such features can be built today as program libraries on top of any existing programming language.

1.2.5 Modeling and Planning

There are many levels at which a task to be performed by a dexterous hand has to be planned. The lowest level is perhaps the trajectory planning problem, wherein an algorithm computes a stream of cartesian positions or joint torques that the underlying controller ought to achieve to be able to perform a given task correctly.

This problem involves computing the trajectories of the grasping surfaces (which may or may not be just the finger tips), given the trajectory of the grasped object. In cases where the hand is actually grasping an object, this problem is quite complicated, since the constraints imposed at the grasp surfaces by the object on the links of the hand need to be satisfied at every instant throughout the trajectory.

While position trajectories are easy to compute under certain assumptions about types of contact, force trajectories are not. As can be seen from the block diagram of the controller, computing a stream of torques and forces to be exerted on a grasped object in order to make it move, may be highly inappropriate and even impossible for some tasks. For other tasks, like turning a screw driver however, such a specification may be easy to come by and may even be the most natural one.

Besides trajectory planning, there are other levels in which planning dexterous hand manipulation tasks can be done. One important problem is the *grasp planning* problem, which is really composed of quite a few hard sub problems. There has been a lot of research in this area in the recent past.

For the purposes of the following discussion, a *grasp* can be considered to be just a matching between *grasp surfaces* (which could be a point, edge, or surface on the fingers of a robot hand), and *component surfaces* on a grasped object. Like most computational geometric problems, a grasp involves *topological* information as well as *geometric* information. The former specifies what the contacting surfaces are, and the latter specifies where exactly they make contact and how.

A number of obstacles remain before such grasp planning algorithms can be used in practice.

1. *Geometric modeling*: Owing to the complexity involved in representing complex objects (despite recent progress in CAD/CAM and computational geometry), many of the algorithms are based on polygonal models. The stability of grasps, however, depends on the local curvature of the objects at the grasp points. This suggests that for grasp planners to be truly effective, surfaces have to be modeled rather accurately, or

existing polygonal modeling systems have to augmented to deal with local curvature information in some fashion.

2. *Feasibility*: Most of the grasp planners plan in an abstract geometric world, and ignore constraints that could be exploited to result in useful plans. The set of *feasible* grasps is the set of those grasps that are possible for a particular kinematic mechanism *and* a particular object.

   There are two interesting questions that can be asked of the feasibility issue.

   (a) Given a grasp between a particular hand and an object, determine if the grasp is feasible. This we will call the **F-A-problem**. This can be seen equivalent to the problem of finding a position and orientation of the palmar plane such that the inverse kinematics of the grasp points expressed relative to this position results in a set of feasible joint angles. In fact, even if one such solution exists, then the grasp must be feasible.

   (b) Given a particular hand and an object, synthesize all feasible grasps. This can be called the **F-S-problem**.

   Looked at one way, these questions involve rather complex calculations involving the workspaces of the different fingers of the robot hand. Looked at another way, such constraints provide powerful heuristics to prune the set of possible grasps.

3. *Reachability:* This is a more complex problem than the **F-S** or **F-A** problems. A *reachable* grasp is one wherein there exists a collision-free path for each of the fingers from their current state to the state wherein the object has been stably grasped by the hand. All *reachable* grasps are *feasible* but not all *feasible* grasps may be *reachable*.

   The reachability problem as described above may be a very hard problem, since it involves as a subproblem, the motion planning problem. It also involves taking into consideration other objects in the environment, and perhaps the nature of the task. Similar to the **F-S** and **F-A** problems one can ask questions as to the reachability of a particular grasp and the set of all reachable grasps.

   One approach to simplifying the problem may be to relax the collision-free requirement so as to not include objects in the environment.

4. *Task Level Synthesis:* How does one use information about the nature of a task when picking a particular grasp? When you pick up an eraser to erase a whiteboard, you

do not hold it by its erasing surface since you know that the nature of the task requires you to use this surface for something else. Similar examples abound when one considers the innumerable tools and other objects that humans use everyday. Such information about task level constraints needs to be mapped into the domain of grasps and may prove to be one more heuristic that helps solve the grasp planning problem.

Besides *grasp planning*, other tasks associated with robot hands that could currently benefit from automation include methods to deal with the complexity of controlling such a robot on a day-to-day basis. An automatically calibrating, self-tuning robot is still quite some distance away, but the existence of humans and other animals provides proof for the hypothesis that such systems can be built.

1.2.6 Error Detection and Recovery

After the planning phase has been completed, the *execution* of the task needs to be carried out. It has been this execution that I have concerned myself with in this thesis. Robots, even conventional ones, however, are prone to failure (especially on days when an important demonstration is scheduled). The current complexity of hands makes them even more fragile than conventional six degree of freedom robots.

There are two approaches to solving the problem. One is to simplify mechanically and in terms of computer control, the software and hardware associated with a dexterous hand. The other approach is to develop strategies that can detect errors and recover from them. Recently, researchers have begun to explore the theoretical issues associated with error detection and recovery strategies (see Donald [1987]). Failure analyses of tasks could indicate the potential sources of problems and areas in which modeling and planners need to be made more accurate. Sensors could play a big role in the detection of errors. Their intelligent use could help recovery.

1.2.7 Engineering and Design Issues

Some of the problems involved in building a robot that has more than ten degrees of freedom, are purely practical in nature. These have to do with dealing with the enormous number of wires, communication bandwidths, computing power, and other such tradeoffs. There are immense problems to be solved while building high performance actuators, flexible transmission systems, and adequate sensors. There has been some principled work done

in the area of mechanical design in order to determine the parameters of a workable hand. However, the design space for dexterous hands has by no means been completely explored or systematically studied.

1.3 Outline of this Thesis

In the sections above, I have tried to outline the various issues and problems associated with dexterous hands, specifically trying to point out those areas in which solutions from conventional robotics research cannot be blindly applied. Later on in this thesis, solutions to some of these problems are presented. The emphasis throughout will be on actual implementation and testing of ideas.

The first section will be devoted to discussing previous work in related areas, to place the research to be described in this thesis in its proper context. I will then look at the problem of controlling a robotic device like the Utah-MIT dexterous hand by looking at its kinematic structure, and solving the forward and inverse kinematic problems mentioned above. Algorithms for doing conventional position control are outlined, and I present a threaded interpretive command language that can drive such a hand using the position control paradigm. A computationally efficient force control algorithm is described in the section following this. Finally, I discuss the computational architecture that we have developed on which most of this work has been implemented and tested.

The main contributions of this thesis are:

1. The solution to the kinematics (both forward and inverse) of the Utah-MIT hand. Two approaches to resolving the redundancy are suggested and compared.

2. A hierarchical controller for the Utah-MIT hand that includes a stable position and force controller, a trajectory generator, and a threaded interpretive primitive command language.

3. Algorithms for solving the trajectory control problem, assuming finger tip grasps with no slip, for finite small motions.

4. Algorithms for computing the differential screw displacement of a grasped object, given the displacements of the finger tips, as part of the force control algorithm.

5. A novel force control algorithm that is computationally very efficient. The efficiency of the algorithm derives from its avoidance of the Grip Jacobian in its computations.

6. A new computational architecture that forms the basis of a new generation of multiprocessor robot controllers in terms of hardware and software. The real-time development environment described herein should form a useful tool to conduct experiments with robots, ranging from the very simple to the most complex.

This chapter contains a brief description of research relevant to hand control and planning. The discussion has been organized into sections suggested by the framework presented in the previous chapter.

2.1 Design Issues

Mechanical hands have been built and studied for a long time. Early attempts to build such hands were motivated because of their use as prosthetic devices. Issues that were considered important in the design of such hands were quite different from those that concern robot hand designers today. Besides, the human hand-arm system is an extremely complicated system to imitate. Consequently, most of the prosthetic hands built to date have concentrated only on providing a very limited subset of mechanical capabilities.

Even though the design goals of such hands were modest, and the issues involved in their design were quite different, the biological studies of these early efforts have influenced recent robot hand designs.

2.1.1 Early Studies

The studies of Schlesinger [1919] and Skinner [1975] guided the design during these early attempts to emulate the so called six types of human grasping patterns. The human hand, however, has twenty-two degrees of freedom with which to accomplish these tasks (Tubiana [1981]). To this date, no artificial hand has approached this level of complexity. Studies carried out by Keller [1947] show that of these six grasping patterns, humans typically tend to use the palmar grasp. The lateral grasp was found to be the second most frequently used grasping pattern. This illustrates an important difference between human and prosthetic hands and present day robot hands; while the former rely on *force* or *power* grasps to accomplish their tasks, the emphasis of present-day robotic hands has shifted toward *dexterous* grasps that are often accomplished with just the finger tips. Finger tip grasps enable dexterous manipulation of the grasped object and precise control of its positioning. Power grasps that rely on structural restraint are more secure in the presence

of disturbance forces since they rely more on the kinematic structure of the grasp to keep the object from slipping rather than on the forces imparted to it through the finger tips. When present day robot hands have to manipulate tools and exert large forces on the environment with the objects/tools they are grasping, the need to plan and execute power grasps may arise.

Most of this early work was descriptive. One could hope that a taxonomy for describing human and robotic grasping patterns in a fashion that would be concise and powerful would arise out of such descriptions, but none has been forthcoming. Recent attempts in a similar vein have included Cutkosky et al. [1986] and Iberall [1987].

It is clear that humans typically tend to use relatively few grasping patterns very successfully to deal with a large number of objects. One could perhaps hope for a database of such grasping patterns indexed by the nature of a task's constraints and object geometry to arise from these taxonomic studies but this seems to be a long-term goal that is not practically achievable in the near future.

2.1.2 Industrial Grippers

Paralleling the development of mechanisms whose primary application was human prostheses, industrial machines on the factory floor have also become more sophisticated. Most robots used in factories today, however, are severely constrained by the kinematic structure of their hands which are often just a pair of parallel vice-like jaws. In fact, their inability to conform to a wide variety of shapes and their inability to make small movements without moving the entire arm, are partly the reasons for robots being used today primarily in spot welding and spray painting tasks: tasks that do not require a great deal of dexterity or co-ordination (Seering [1984]).

To address this problem, many alternative solutions have been proposed. Changeable grippers have been proposed by Luo [1984]. Although these quick-change grippers enable high performance in those tasks for which the grippers are specially designed, they fail miserably to adapt to other tasks. Hollerbach [1982] suggests that the cost involved in designing such specialized grippers for a large number of different operations could be very expensive. When the robot changes from one task to another such grippers would have to be switched. This involves additional lost time due to gripper changing operations.

Other exotic solutions like vacuum-suction grippers, electro-magnetic grippers and chucks, flexible element grippers etc., have been applied to bin picking or parts handling. Although

some of these designs are mechanically ingenious, these devices are often highly task specific too. A reasonable account of such mechanisms used in industry is provided by Chen [1982]. Chelpanov et al. [1983] provide a more detailed account of some of the problems with the mechanics of industrial robot grippers. Kato [1982] provides an illustrated collection of numerous hand designs used both in industry and for prostheses.

2.1.3 Dexterous Robot Hands

Some of the limitations of present day robots could be overcome if they had arms with more degrees of freedom (Yoshikawa [1983], Hollerbach [1984]). The other solution with which we will be concerning ourselves for the rest of this thesis is one which involves multi-fingered robot hands. These robot hands are very different from conventional robot grippers. Some of them are very anthropomorphic. Almost all of them have many fingers, each of which is composed of a number of joints. A number of such hands have been constructed with recent advances in actuation, sensing and control technologies. (Crossley et al. [1977], Okada [1979], Salisbury [1982], Abramowitz [1982], Jacobsen et al. [1984]). Articulated hands such as these are capable of adapting to a wide variety of object shapes and are capable of making extremely fine motions under computer control. Thus they reduce the need for special tooling while being adaptable to a wide variety of operating tasks.

Salisbury [1982] presents an analysis using the condition number of the Jacobian matrix of a mechanism to choose parameters involved in the linkage design of such robot hands. Mobility analysis of different mechanisms may also be used to choose optimal configurations of the various fingers. The hand presented in this work is driven by tendons attached to electric motors. Mechanically, the device comprises three fingers each with three joints. Salisbury et al. [1985] continue this early work and discuss the problem of choosing link lengths for a redundant mechanism.

In an another important effort, Jacobsen et al. [1984] present the design issues pertaining to the Utah-MIT hand which will be described in detail later. There have been a series of papers devoted to discussing the various engineering aspects of this project (see Jacobsen et al. [1985], Jacobsen et al. [1986]). This hand is pneumatically driven and is tendon operated. In contrast to the Salisbury hand, the design is anthropomorphic with four fingers. Each finger has four degrees of freedom.

Besides these two hands, perhaps the only other hand that has actually been used to

perform non-trivial manipulation tasks is the hand built by Okada [1979] (see also Okada [1982]).

If any conclusion can be drawn at all from these various efforts, it is that the engineering problems involved in hand design can be separated into four very broad categories which involve actuation, transmission, mechanism design and sensor design. For actuation electric motors, hydraulic and pneumatic actuators have been used. For the transmission system, tendons made out of a variety of materials have proven to be the most successful. The mechanism and sensor design spaces however are much too large and the choices made by these efforts thus far have to be characterized as careful engineering tradeoffs.

2.2 Workspace and Kinematics

The kinematics of multi-fingered hands is quite different from that of conventional robots. Workspace issues involving such hands can be quite complicated. Each finger of a robot hand is usually simpler than a conventional six degree of freedom robot. Consequently, texts like those of Paul [1982], Craig [1986] or Asada and Slotine [1986], can be expected to provide the machinery needed to tackle the kinematic structure of the individual fingers. The problem of having to deal with multiple fingers, however, is hard, considering the closed-loop nature of the mechanism they give rise to, once the hand has grasped an object.

In solving the kinematics problems for individual fingers, complications could arise owing to the redundancy present relative to a task. For example, a single finger of the Utah-MIT hand has four degrees of freedom. Positioning the tip of such a finger in three dimensional cartesian space, however, involves only three degrees of freedom. For this task, therefore, such a finger is redundant. Resolving the redundancies in such lower degree of freedom kinematic chains presents many interesting problems.

If no such redundancy were present, then solving the kinematics of the individual fingers is a rather trivial problem. Once the fingers of a hand have grasped an object, the resulting closed loop chain's kinematics become complicated. The motion of the grasped object and its configuration at any given moment are dependent on the nature and position of the grasping surfaces and the relative degrees of freedom at these surfaces. Contacts could be made at a point, along a line or along a planar surface. Each of these different types of contact gives rise to a different relative mobility between the hand and the grasped object.

Workspace issues have traditionally been difficult to resolve owing to the complexity of the issues involved. Bajpai and Roth [1986] discuss the issues associated with closed-loop

manipulators. Kerr's [1985] thesis contains a section devoted to workspace issues associated with dexterous robot hands.

2.3 Work in Sensing

In recent years, there has been an explosion of work in sensing and designs of sensors for various tasks. Some of these have been motivated by careful analyses and followed up by careful testing and evaluation. Most, however, have concentrated merely on the exhibition of a new transduction process and have stopped after a proof of concept had been demonstrated.

Siegel [1986] has presented objective criteria for the design and testing of tactile sensors. This work also presented a promising tactile sensor based on a capacitance technology. Other designs have been based on optical (Begej [1984], Crosnier [1986]), capacitance (Boie [1984]), rheological (Brockett [1985]), magnetoelastic (Checinski [1986]), ultrasonic (Grahn [1986]), pneumatic (Hanafusa [1976]), and piezoelectric (Nakamura [1986]) transduction principles.

A series of papers by Harmon provide valuably informative surveys on the state of the art with respect to issues pertaining to sensing (see for example Harmon [1980], Harmon [1984] and Harmon [1985]).

Besides the construction of these sensors, there have been numerous papers on how these sensors can be used in algorithms for object identification (Ellis [1987], Grimson et al. [1984], Okada [1977]), perception of the environment to build models (Brock et al. [1985], Dario et al. [1985]), estimating the configuration of objects (Driels [1986], Palm [1985]), and to actually help in the manipulation of objects (Fearing [1986]). Of these efforts, Fearing [1986] (see also Fearing [1987]) and Brock et al. [1985] have actually mounted their sensors on robot hands. The former also provides a detailed study of the solid mechanics problems involved in contact phenomena. Similar such studies are being pursued by Cutkosky et al. ([1986] and [1987]). A preliminary discussion of theoretical issues associated with tactile sensing can be found in Montana [1986].

2.4 Work in Control

In his early work on the hand that he built, Salisbury [1982] also presented control algorithms for achieving stable force control. His method was based on the so-called Grip Jacobian matrix, and is, as we will see in the chapter on force control, computationally very

expensive. The form of control introduced in this seminal thesis is now known as *stiffness control.* Other forms of force control have also been documented in the literature, but none has been applied to robot hands. Approaches similar to this one have been proposed by Kobayishi et al. [1986], and by Yoshikawa [1987].

Chiu [1983] presents the implementation of a controller and a brief description of a programming language for the Salisbury hand. He introduced the concept of a *grasp-frame* that is computed as a function of the positions of the finger tips. This frame can then be used to compute trajectories of the finger tips once the trajectory of the grasped object is specified. Biggers et al. [1986] provide a summary of the issues involved in low level control of the Utah-MIT hand. Model based control of the Salisbury hand is addressed by Loucks et al. [1987] and multivariable control issues are tackled by Venkataraman et al. [1987]. Other issues associated with identification and estimation are addressed in a thesis by Speeter [1987].

2.5 Planning and Programming

To summarize our survey of previous related work, we look at issues associated with planning that has been the target of numerous research efforts.

Some of the problems associated with robot grasp planning were outlined in an early thesis by Lozano-Pérez [1976]. In this paper that dealt with automated robot programming, the choice of a single grip from amongst the number of grips possible was recognized as a hard problem. The *configuration space*[1] approach was applied to solve a simple version of the problem for a parallel jaw gripper.

Hanafusa et al. [1977] present the solution to a planar version of the grasp selection problem using a potential field approach to choose stable grasps. Further attempts to solve the planar version of the problem with simple and articulated fingers can be found in Abel et al. [1985], and Nguyen [1986].

There have been a number of attempts at analyzing the stability of a given grasp, but relatively few attempts toward synthesizing grasps. The work of Kerr [1984], Jameson [1985] and Cutkosky [1984] represent the state of the art with respect to analysis. Perhaps the best work related to synthesis is offered by Nguyen [1986] and Nguyen [1987]. This work presents algorithms that synthesize grasps that are stable and force closed based on

[1]This approach has yielded important results with respect to robot motion planning problems in the recent past.

purely geometric calculations. The objects to be grasped are modeled as polyhedra and the fingers are treated as points.

It must be mentioned that the problems associated with feasibility and reachability of a grasp, mentioned in our earlier framework remain unsolved, even for hands with a relatively simple kinematic structure.

The previous chapters have provided a brief overview of the various problems associated with dexterous robot hands and a look at previous research efforts that have tried to address some of these problems.

Dexterous hands are complex manipulating devices, and some of the interesting problems that arise in using such devices are purely kinematic in nature. This section presents some of these problems using the Utah-MIT hand as an illustrative example.

3.1 Description of the Utah-MIT Hand/Arm System

The Utah-MIT hand consists of four fingers, powered pneumatically and driven by tendons. The first version of the hand used kevlar with dacron wound around it for tendon material, while the second and present version of the hand uses polyethylene (see Figure 3.1 for a picture of the hand and Figure 3.2 for the hand arm system).

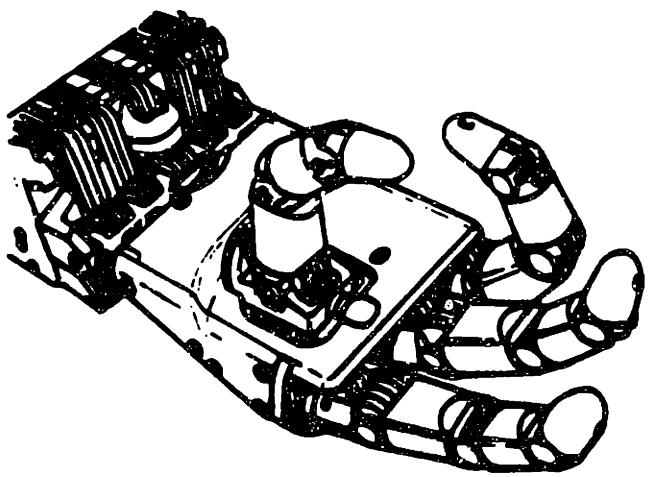


Figure 3.1: Picture of the Utah-MIT hand.

The hand comprises sixteen revolute joints, divided into four fingers each with four joints. Each joint is driven by 2 tendons routed over pulleys. The tendons are routed from

the actuator pack which houses the actuators, to the hand's joints through a *remotizer* that allows the actuators to be located remotely relative to the joints. The hand can carry a payload of about 15 pounds and it weighs about as much.

Current sensing capabilities of the hand include two tendon tension sensors per joint and Hall effect angular rotation sensors, that enable the position of any joint to be measured accurately.

The Utah-MIT hand has a number of novel features that make it one of the best robot hands designed to date. As mentioned earlier, the issues involved in the design of this hand have been presented in a series of papers by Jacobsen et al. [1984], [1985] and [1986]. The innovative features of this hand are summarized below:

1. Each joint is driven by two tendons requiring $2n$ tendons to drive n joints. As a point of contrast, the Salisbury hand requires $n + 1$ tendons to drive n joints. The engineering tradeoff was between the added complexity of routing the extra tendons compared to the simplicity of the decoupling involved and the added power obtained.

2. Each pneumatic actuator is driven by a modular valve design. The actuator essentially drives a graphite piston moving in a glass cylinder to which a tendon is attached.

3. Compact Hall effect position and tendon-tension sensor designs.

4. Polyethylene tendons that are strong and durable.

5. Each finger comprises four degrees of freedom and there are four such fingers. The hand looks anthropomorphic since the design includes an opposing thumb and the axes of all the distal revolute joints are parallel, much like in the human hand. The size and shape of the various joints are also comparable to that of the human hand.

A hand that cannot move about in a workspace is of little use. To facilitate the palmar plane to be moved around, a cartesian table was constructed on which the hand could be mounted. This table is a four degree of freedom robot. All four axes of the table are driven by stepper motors. The stepper motor controller uses electromagnetic sensing to determine the position of a stepper motor very accurately, and hence it is possible to operate the cartesian table without relying on feedback from devices like optical shaft encoders. Out of the four cartesian axes, two are oriented in the same direction, which facilitates tracking operations.

Figure 3.2: Picture of the Utah-MIT hand arm system.

The table was designed and built so that we would be able to position the hand in a workspace that was roughly a cube 18 inches on its side.

All of the implementations and experiments that we are to describe were conducted on this robot. The computer architecture designed for it, however, is a truly general purpose real time development machine, and hence has been applied to control other robots like the MIT Serial Link Direct Drive Arm.

3.2 Forward Kinematics

The problem of forward kinematics, very simply put, is the problem of finding out the coordinates of the finger tips in *cartesian* space given the joint configuration as input. With dexterous hands, this problem is always well defined since most dexterous hands can be represented by tree-like open-loop kinematic chains (see Salisbury [1982], Jacobsen et al. [1983] for examples of such designs). This is so because most robot hands today are manipulating devices that have many fingers (each with a number of joints) attached to some kind of wrist which in turn can be positioned in space by other joints and links.

The forward kinematics problem can therefore be expressed as computing:

$$\mathbf{x} = f(\theta) \tag{3.1}$$

where $\mathbf{x}$ is the vector of finger tip locations expressed in cartesian space, and θ is a vector

of joint angles.

There are two reasons why computing the forward kinematics efficiently, is important.

(a) As we will see later, some control algorithms depend on being able to compute the forward kinematics as a subproblem. In particular, force and position control algorithms that need to compute the position of a grasped object in cartesian space use forward kinematics computations extensively. Most of these algorithms assume that contact is made between a grasped object and the finger tips. Knowing the positions of the finger tips in cartesian space helps in determining the position and orientation of the grasped object.

(b) The forward kinematics computation is an important part of a kinematic simulator, that could be used for a variety of purposes.

There are many ways of solving the forward kinematic problem. Here I present two of the most widely used. The first method relies on what I call the *modified Denavit-Hartenburg* notation.

The Denavit-Hartenburg notation is perhaps the most widely used to represent the kinematic structure of robots. Briefly stated, it involves representing the transformation between two successive links in a kinematic chain (composed of revolute or prismatic joints) as homogenous transformation matrices. These matrices can be derived uniquely from the four so-called D-H parameters of the link, which can be described as follows:

1. θ_i which represents the angle made by the i'th joint,

2. α_i which represents the twist angle between the two axes of movement (these axes would be rotational if the axes were revolute and translational if the axis were prismatic),

3. a_i which represents the distance along the common normal between the two axes, and

4. d_i, the distance along the z_{i-1} axis between the origins of the two coordinate systems.

It must be pointed out that this notation is *not* unique, even for serial kinematic chains. In fact, when applied to tree-structured kinematic chains, this notation is ambiguous (see Khalil [1986] for an example of such a situation). Alternatives have been proposed by many to remedy this situation with respect to tree-structured and closed-loop kinematic chains. Sheth and Uicker [1971] propose a rather complex alternative wherein they separate the

information pertaining to the shape of the link from the variable part that varies with joint motion. This notation is not only overly complex, but it is computationally quite expensive, as noted by Roth [1985].

It is rather easy to see that there are two basic ways of dealing with the ambiguity in the DH notation with respect to tree structured manipulators, and both have to do with the numbering scheme chosen to deal with branch-points in the tree-structured chain. A breadth-first numbering scheme would number the k joints attached to a joint n as $n+1$, $n+2$ ... to $n+k$. A depth-first numbering scheme on the other hand can be described to recursively compute the numbering as follows:

```
joint_number(joint)
{
    number_of(joint) = no + 1
    no = no + 1
    for all sons of joint do
        joint_number(son_joint)
}
```

One can see that this would result in a branch point at joint k being numbered as $n+1$ where n was the last number used in the previous serial chain encountered instead of $k+1$ as in the previous scheme.

Once the ambiguity has been resolved, the same D-H matrices can be used to deal with tree-structured manipulator chains as before.

3.2.1 Forward Kinematics of the Utah-MIT Hand

The Utah-MIT hand is a four-fingered dexterous hand. Each finger of the hand has four joints, making a total of sixteen degrees of freedom. As was mentioned earlier, the hand is mounted on a four degree of freedom x-y table.

The structure of the arm must be factored into the forward kinematic calculations to determine the position and orientation of the finger tips. It was for this reason that a cartesian design was chosen for the x-y table: namely, to simplify the kinematic considerations involved in dealing with the arm. Currently there is no way of measuring the orientation of the palmar plane relative to the z-axis. This deficiency is to be corrected shortly.

While considering the forward kinematics problem, therefore, we will split the problem into two subproblems. First the position and orientation of the palmar plane will be deter-

mined using the arm system. Then the position of the finger tips will be determined relative to the palmar plane. Notice that this procedure will extend to arbitrary tree structured kinematic chains rather trivially. The position and orientation of each node that forms a branching point when computed, allows the computation of the positions of joints further down the tree to be computed rather easily.

The Denavit-Hartenburg parameters of the Utah-MIT hand are given in Table 3.1[1].

Joint+1	d_i	a_i	α
1	$\frac{l_p}{cos(\phi_p)} + l_0\ cos(\phi_0)$	$l_0\ sin(\phi_0)$	$-\pi/2$
2	0	l_1	0
3	0	l_2	0
4	0	l_3	0

Table 3.1: D-H Parameters for non-thumb fingers.

The single thumb of the Utah-MIT hand is different from the rest of the hand's fingers. The Denavit-Hartenburg parameters for the thumb are given by Table 3.2.

Joint+1	d_i	a_i	α
1	0	h_0	$-\pi/2$
2	0	l_1	0
3	0	l_2	0
4	0	l_3	0

Table 3.2: D-H Parameters for the thumb.

Since each finger can be individually considered to be a robot affixed to the palmar plane, one can use the final ${}^0A_5$ frame to describe the position and orientation of the finger tip relative to the palm.

Notice also that in general, we might have a tactile pad mounted at the tip of the finger and hence the co-ordinates of the actual contact location will be expressed relative to the

[1] A more detailed derivation and explanation of the various coordinate systems used can be found in Appendix A, which also explains in detail the various symbols used.

co-ordinate system of the final reference frame. This would mean that to derive the actual contact location in the palmar frame these co-ordinates have to be pre-multiplied by the ${}^0A_5$ matrix.

We choose to describe the cost of these particular computations in terms of elementary operations like multiplications (denoted by M), additions (denoted by A) and transcendental function calculations (denoted by T) rather than the asymptotic analysis that is more common, since we desire to get a very detailed picture of the computations involved. Such an understanding is required to facilitate fast implementations operating in real time. Computationally, it would take 26M+16A+8T operations to compute all the elements of this matrix, which describes completely the position and orientation of a finger tip relative to the palm for a non-thumb finger. To compute the co-ordinates of a contact point relative to the palmar plane, an equal number of operations would be required. For three fingers therefore, a total of 78M+48A+24T would be required.

For some computations, the entire matrix ${}^0A_5$ need not be computed. If only the cartesian positions of the finger tips are needed for example, such information can be obtained with 12M+14A+8T operations.

Since the thumb is kinematically simpler, it takes only 12M+9A+8T operations to compute the elements of the ${}^0A_5$ matrix (also known as the T matrix) for the thumb.

The total number of operations required to solve the forward kinematics problem for the Utah-MIT hand's fingers is therefore given by:

$$C(fkin) = 94M + 63A + 32T \ \ operations \tag{3.2}$$

3.3 Inverse Kinematics

The inverse kinematics problem can be stated as that of computing the inverse of the mapping f mentioned in Equation 3.1. The problem is to find a configuration of joint angles given the cartesian coordinates of the end points of the finger tips as given by:

$$\theta = f^{-1}\mathbf{x} \tag{3.3}$$

This problem turns out to be, surprisingly, quite a hard one. This is because of the fact that while the number of joints in a manipulator is restricted only by physical constraints and perhaps by its designer's imagination, the cartesian space the manipulator operates in is the familiar three dimensional world that its tasks are performed in. This means that

while six coordinates suffice to describe the cartesian or task space, the joint coordinates associated with robots can be considerably more in number.

This is true especially of dexterous robot hands. The Utah-MIT hand has, for example, four joints on each of its four fingers, requiring sixteen joint angles to completely describe just the configuration of the hand (see Jacobsen et al. [1983]). The Stanford-JPL hand has three fingers with three degrees of freedom each (see Salisbury [1982]). This large number of degrees of freedom present in dexterous hands means that the inverse kinematic mapping may not be well defined or there may be multiple solutions to choose from. The mapping f^{-1} is not a function but is a one to many mapping.

The inverse kinematic problem basically involves the solution of the non-linear equation expressed by Eqn. 3.3.

3.3.1 Inverse Kinematics of the Utah-MIT Hand

We will now consider the inverse kinematics of one finger of the Utah-MIT hand. The hand has four fingers, each of which has four degrees of freedom. Each finger is not equivalent to a full-fledged six degree of freedom robot and it will not be possible to specify the end point position and orientation of a finger individually. We do have control of four parameters however, and one can choose these parameters in a number of different ways. If we choose to specify end-point position in 3-space, we will have to provide as input to the trajectory planner the x, y and z coordinates of the finger tip and would still have one degree of freedom left over. Stated another way, a four degree of freedom finger will be *redundant* by one degree of freedom for this 3 degree of freedom positioning task.

The question then arises as to how to use this extra degree of freedom. A number of researchers have investigated redundant arms and there have been many different proposals seeking to utilize this redundancy for the purposes of obstacle avoidance (Maciejewski and Klein [1985]), energy minimization (Liegeois [1977]), and joint torque optimization (Hollerbach and Suh [1985]).

A number of local dexterity measures have been proposed to solve this problem and to aid in obtaining the solution to the inverse kinematics problem. Usually the problem is expressed in the velocity domain and the task is then to find the solution $\dot{\theta}$ from:

$$\dot{\mathbf{x}} = \mathbf{J}\dot{\theta} \tag{3.4}$$

where **J** is the Jacobian matrix.

Usually, this is done by the following equation:

$$\dot{\theta} = \mathbf{J}^{+}\dot{\mathbf{x}} + (\mathbf{I} - \mathbf{J}^{+}\mathbf{J})\,\mathbf{z} \tag{3.5}$$

where $\mathbf{J}^{+}$ is usually some kind of generalized or pseudo inverse, and $\mathbf{z}$ is an arbitrary vector chosen to optimize secondary criteria.

A number of different so-called *dexterity* measures have been proposed in the literature, as the appropriate secondary criterion to minimize or maximize. These include the determinant of the Jacobian, (Paul and Stevenson [1983]), the square root of $\mathbf{J}\,\mathbf{J}^{T}$ (Yoshikawa [1984]), the condition number of the Jacobian (Salisbury and Craig [1982]), the minimum singular value, and the joint-range availability index to minimize the possibility of the joints attaining their joint limits. Comparative studies between these numerous dexterity measures have been rare (see Klein [1987] for an example), and even when performed have been largely inconclusive, since the purpose of each of the dexterity measures is different.

Rather than use one of these studied approaches which treat the problem in the velocity domain, we will look at the problem in the *position* domain. We will call such methods for resolving the redundancy as *direct* methods.

Perhaps the simplest of all approaches is to treat one of the joint angles as fixed. Since the axis of rotation of the three distal joints of the Utah-MIT hand are parallel, fixing joint 0 would be wasting a very important degree of freedom, one that changes the entire plane of operation of the distal links. Fixing the joint angle of any of the three distal joints, however, would enable us to reduce the finger to a 3 degree of freedom finger much like the fingers of the Stanford-JPL hand (the link that is considered fixed can be considered to be just a part of the previous link). This approach, while easy to implement and understand, makes no use of the additional degree of freedom and hence we will not concern ourselves with it any further.

There are two other ways to address this question. Both involve the specification of an extra constraint equation or an extra parameter that can be controlled. To my knowledge, the direct specification of these constraint equations in joint space has not been explored in detail in the past.

We propose two ways of specifying an extra constraint directly in joint space. Both ways are simple and result in compact equations. The need for simplicity at this level is motivated by the need to evaluate these equations in real time for certain operations.

The first way to resolve the redundancy issue is to specify the endpoint orientation relative to the palmar plane in addition to the position information. Although this may

seem equivalent to fixing one of the joint angles, it is different in that this added parameter can be specified and controlled meaningfully. The r.h.s of Equation 3.3 will then be a four by one vector involving three cartesian positions and one angle.

To solve the resulting inverse kinematics equations, we first convert the x, y, z coordinates expressed in the palmar plane coordinate system, to the coordinate system affixed to the first link. This can be done simply by premultiplying the desired cartesian position by the ${}^1A_0$ matrix. For non-thumb fingers this matrix can be found out from ${}^0A_1{}^{-1}$ which is given by:

$$ {}^1A_0 = \begin{bmatrix} 0 & 1 & 0 & -r_p \\ sin(\phi_p) & 0 & cos(\phi_p) & h_p\, cos(\phi_p) - l_p\, sin(\phi_p) \\ cos(\phi_p) & 0 & -sin(\phi_p) & tan(\phi_p)[l_p\, sin(\phi_p) - h_p\, cos(\phi_p)] \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.6} $$

Once the coordinates have been converted to be in this coordinate system, joint angle θ_0 (or the joint angle of the proximal link attached to the palm) of each finger can be determined simply from

$$ \theta_0 = arctan(y/x) \tag{3.7} $$

We still have to determine the three remaining joint angles. Now that we know angle θ_0, we can as before express the coordinates relative to a frame affixed to the next distal joint. Since the axes of the three most proximal joints in each finger are parallel to one another, the problem is now essentially that of deducing three joint angles given the x,y position of the finger tip, in this plane of operation.

The inverse of the second matrix is given by ${}^2A_1$, which is equal to ${}^1A_2{}^{-1}$.

$$ {}^2A_1 = \begin{bmatrix} C_0 & S_0 & 0 & -l_0\, sin(\phi_0) \\ 0 & 0 & -1 & l_0\, cos(\phi_0) + \dfrac{l_p}{cos(\phi_p)} \\ -S_0 & C_0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.8} $$

If we let the orientation of the final link be fixed at some angle θ_f, (which can be computed either relative to the palm or as a function of the path during trajectory execution), we can see that what we have is essentially the familiar two degree of freedom problem, where

$$ \begin{aligned} x_t &= x_d - l_3\, cos(\theta_f) \\ y_t &= y_d - l_3\, sin(\theta_f) \end{aligned} \tag{3.9} $$

and (x_d, y_d) are the co-ordinates of the finger tip after the multiplication expressed by Equation 3.8 has been made.

Given x_t and y_t we can compute the angles θ_1 and θ_2 using the familiar formulae given below. Note that the kinematic structure of the Utah-MIT hand's fingers prevents the attainment of what is normally called the elbow-down (or curling out) configuration, and therefore there is only one solution possible for this reduced two degree of freedom problem.

$$\begin{aligned} \theta_2 &= acos\left(\frac{x_t^2 + y_t^2 - l_1^2 - l_2^2}{2\, l_1\, l_2}\right) \\ \theta_1 &= atan\left(\frac{y_t}{x_t}\right) - atan\left(\frac{l_2\, sin\theta_2}{l_1 + l_2\, cos(\theta_2)}\right) \\ \theta_3 &= \theta_f - (\theta_1 + \theta_2) \end{aligned} \tag{3.10}$$

The condition for reachability for the two degree of freedom problem can be given as

$$\sqrt{x_t^2 + y_t^2} \leq (l_1 + l_2) \tag{3.11}$$

Notice that the angle that the finger tip makes with the palm has been assumed to have been specified in the plane of operation of the three distal joints. This may not be necessarily true. It is more realistic to expect this angle be specified as a vector that is perpendicular to the surface of the finger tip. Going from this latter or any other representation of this angle to its projection on the plane perpendicular to the axes of revolution of the three distal joints is usually quite simple.

The effect of choosing this way to resolve the redundancy is illustrated in Figure 3.4, which simulates a horizontal trajectory directly above the palmar plane. Notice that the orientation of the distal link relative to the palm remains fixed throughout the entire motion. Figure 3.3 illustrates the consequence of the same assumption on a vertical trajectory.

Keeping the orientation of the last link fixed is not the only possibility. One could vary this orientation as a function of the trajectory to be executed. Such a capability could be used for example in manipulating objects with known shapes, for it allows a precise control over the locus of the contact point on the finger tip surface.

Another approach to resolving the redundancy was motivated partly by the observation that in human fingers the angles made by the two most distal joints remain approximately equal throughout the course of a motion. This is essentially the specification of an extra constraint equation of the form

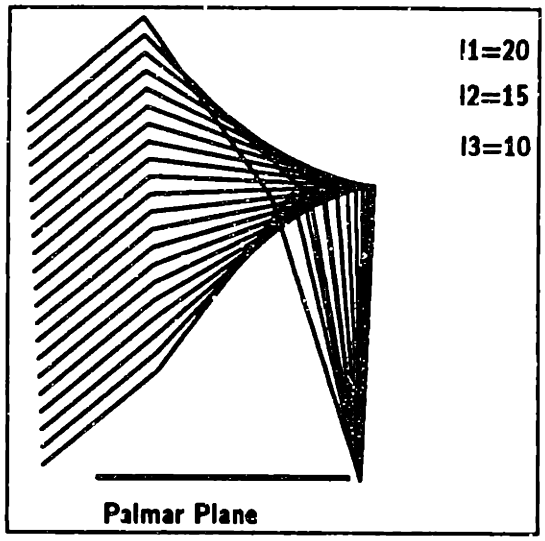


Figure 3.3: Vertical trajectory using fixed orientation constraint.

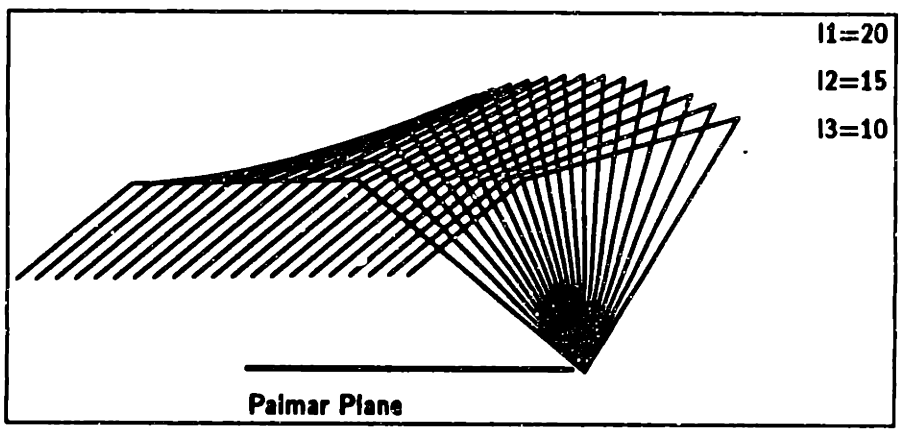


Figure 3.4: Horizontal trajectory using fixed orientation constraint.

$$\theta_3 = K \times \theta_2 \tag{3.12}$$

where K is some proportionality constant. Setting K to 1 specifies that the two angles are equal.

Using a constraint equation of this form (where K=1), one can rewrite the kinematic

equations and solve for the inverse kinematics as follows[2].
We first note that:

$$\begin{aligned} x_d &= l_1\, cos(\theta_1) + l_2\, cos(\theta_1 + \theta_2) + l_3\, cos(\theta_1 + \theta_2 + \theta_3) \\ y_d &= l_1\, sin(\theta_1) + l_2\, sin(\theta_1 + \theta_2) + l_3\, sin(\theta_1 + \theta_2 + \theta_3) \end{aligned} \tag{3.13}$$

Squaring the above equations and adding, we get:

$$l_1^2 + l_2^2 + l_3^2 + 2\, l_1\, l_2\, cos(\theta_2) + 2\, l_2\, l_3\, cos(\theta_2) + 2\, l_1\, l_3\, cos(2\, \theta_2) = x_d^2 + y_d^2 \tag{3.14}$$

Using the double angle formula for the cosine, we can see that the above equation reduces to a quadratic in θ_2.

$$4\, l_1\, l_3 cos^2(\theta_2) + 2\, (l_1\, l_2 + l_2\, l_3) cos(\theta_2) - (x_d^2 + y_d^2 - l_1^2 - l_2^2 - l_3^2 + 2\, l_1\, l_3) = 0 \tag{3.15}$$

This quadratic equation can be solved for the value of $cos(\theta_2)$ rather easily. Again, because of the kinematic structure of the Utah-MIT hand's fingers, only one of the solutions is possible. Since K equals 1 in the above example, θ_3 is equal to the above value of θ_2. The last remaining joint angle can be found from:

$$\theta_1 = atan\left(\frac{y_d}{x_d}\right) - atan\left(\frac{l_2\, sin(\theta_2) + l_3\, sin(2\, \theta_2)}{l_1 + l_2\, cos(\theta_2) + l_3\, cos(2\, \theta_2)}\right) \tag{3.16}$$

This way of resolving the redundancy certainly results in different joint angles for the same cartesian finger tip positions.

The effect of this method is illustrated on the horizontal and vertical cartesian space trajectories as before in Figures 3.5 and 3.6. Besides having a human-like graceful motion, the second method offers the advantage of a larger workspace in certain regions. This can be seen from the fact that the first method essentially reduces the workspace to that of a two-degree-of-freedom revolute manipulator augmented ellipsoidally by the length of the third link. In regions of the work space where the fixed-orientation requested is parallel to the orientations of joints 1 and 2, the trajectory planner is limited by the fact that it cannot use the third link very much.

Figure 3.7 shows a diagonal linear trajectory using the second method, which illustrates the large workspace attainable. The curling motion of the fingers during such a motion could be useful for acquisition operations as we will see later.

[2]I have shown the derivation only for the last three joint angles. The derivation for θ_0 remains the same as before.

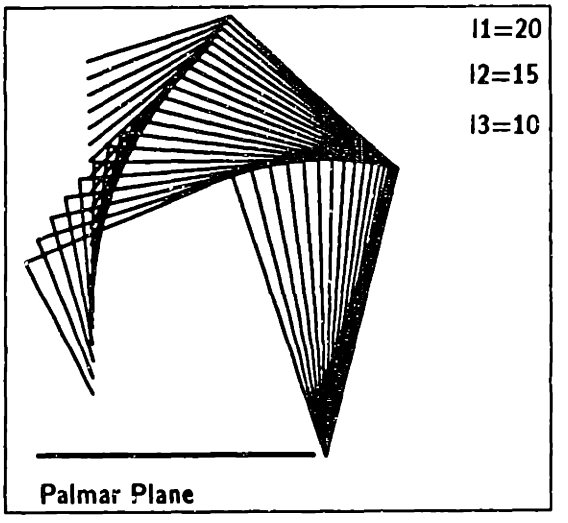


Figure 3.5: Vertical trajectory using equal angles constraint.

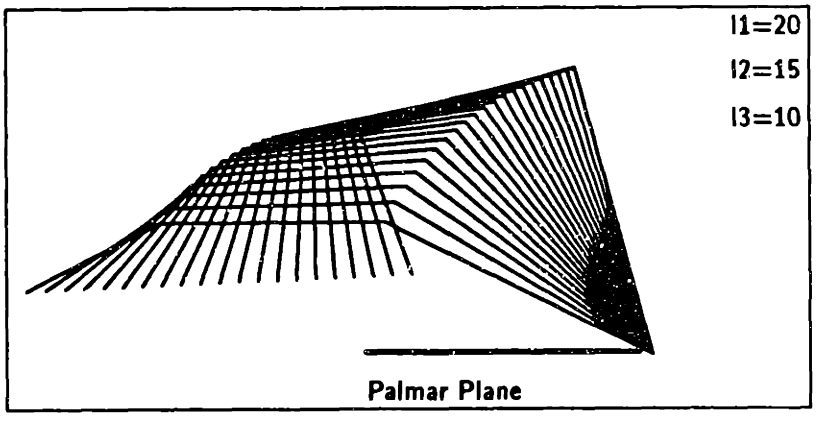


Figure 3.6: Horizontal trajectory using equal angles constraint.

On the other hand, the vertical trajectory illustrates a disadvantage of the second method. If the trajectory had been computed attempting to move a grasped object one can see that as the motion begins at the top, the finger's joints are clear of the object. But as the finger moves progressively down, the distal joint would attempt to actually move into the object. Depending on the stiffness of the grasped object and the finger servos, this could result in large forces of interaction.

One would like to quantitatively measure the difference between these different ways

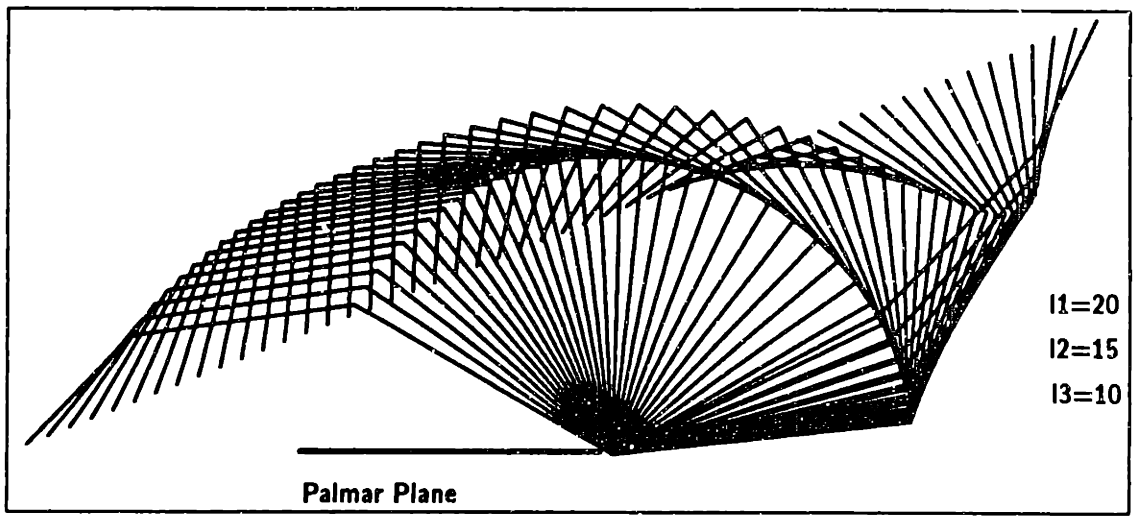


Figure 3.7: Linear trajectory using equal angles constraint.

of choosing the extra constraint equation. The norm of the joint-angle differences and the difference in arc length along the homogenous solution path have both been proposed as possible measures that could be used for this purpose in a recent paper (Klein [1987]). While it is true that such measures do indirectly measure other desirable features of trajectories, intuitively one finds it troublesome that in all recent attempts to compare and measure the effects of these different measures no importance is given to the *task* to be performed. No attempts have been made also to judge the different performance measures over widely different areas in the workspace and through widely different trajectories. Daunting as such a task may be, a principled approach to such evaluation may yield lasting results.

One must also mention that the second method of specifying an extra constraint equation raises other interesting possibilities. In some cases, configuration space planners rely on taking what are known as *slices* of the configuration space (see for example Lozano-Pérez [1986]). This usually involves fixing one joint angle and looking at the range of orientations possible for the other joints. Stated another way, these planners sweep a generalized hyperplane through configuration space, one that is perpendicular to some axis of the configuration space. *Critical slices* are those slices at which the topology of the configuration space changes drastically. It is important for a planner to at least find all such critical slices, since otherwise it could fail to recognize the presence of an obstacle between two slices (see Erdmann and Lozano-Pérez [1986]). The specification of constraint equations as

above illustrates the notion that slices can be taken of the configuration space in other ways besides the usual. While we have used this constraint to determine a slice where in the problem of inverse kinematics becomes solvable, the advantages of such slices with respect to detecting critical slices still remain largely unexplored.

3.4 Other Issues

As we mentioned earlier the Utah-MIT hand is mounted on a cartesian table with a simple kinematic structure. Mounting the hand on such a device however raises other issues, which I can only briefly mention here.

The first of these has to do with how end-points are specified in conventional robotics. The usual way end point configuration of a robot is specified is through a homogenous transform or some other representation that provides information as to the cartesian location (in x, y, z) and orientation of the end point. In the case of a dexterous robot hand, one has not only to provide this information for an arm, but in addition the end-point positions of the various fingers that make up the hand. But providing information as to where each finger tip is located throughout a trajectory may be overly wasteful. What one needs instead is a decoupling between the hand's kinematics and the arm's kinematics where possible. This decoupling however is a function of the task to be performed and hence only heuristic decompositions may be possible. In a pick and place operation, for example, we can obviously exploit the fact that we do not expect to move the hand's joints during certain parts of the motion trajectory.

In the case of anthropomorphic hands like the Utah-MIT hand mounted on a six degree of freedom robot, there is a natural decoupling between the wrist's positioning and the positioning of the finger joints. However, this may not be the case if either the hand gets more complex or if the arm gets simpler.

Another thorny issue is determining when to move the hand's fingers and when to move the arm. We mentioned earlier that one of the advantages in using a dexterous hand is that for small incremental motions, one needs to move only the fingers associated with the hand and not the entire arm. However, it seems intuitively obvious that for larger motions we would prefer to move the arm rather than the hand. In practice, one thumb rule that could be used is for motions larger than the size of the hand itself, the arm should be used. However, if one is to use the higher bandwidth available at the hand's finger level, more intelligent ways of partitioning the motion must be considered.

Workspace questions are usually harder to pose and answer even for conventional robots. They are even more so for dexterous hands. During the motion of a robot hand, one must make sure that none of the fingers collide with one another, or with the grasped object in a fashion that would cause the task to fail. When one holds an object, which is small relative to the size of one's hand, it is easy to see that one can manipulate it far easier over larger regions in the hand's workspace than if one were holding a large object. This rather complex interaction between the shape of a grasped object and the kinematics of the hand's fingers proves to be a rather interesting area where much fruitful research can be expected in the future.

This section will present a hierarchical controller that has been implemented to run under the multiprocessor architecture to be described in a later section. There are a number of ways that a complex device like the Utah-MIT hand can be controlled. Each way of controlling the hand has its own advantages in terms of convenience and performance relative to a particular task.

The controller is hierarchical in that there are many control loops each of which implements a particular input output relationship. One of the important advantages of the computational architecture to be described in a later section is the ease with which the addition of such control loops can be done.

The choice of the hierarchy is partly dictated by the necessity to separate different levels of abstraction. The highest level needs necessarily to operate in object space (or task space). Subsequent levels of the hierarchy move lower toward the actuator space. For example, if the highest level controlled a grasped object's cartesian position, intermediate levels could involve the computation of finger tip cartesian positions and joint angles corresponding to these tip positions. A lower level then could compute tendon tensions from these joint angles and the last level in the hierarchy would actually compute the actuator voltages needed to achieve these tendon tensions. It is interesting to note that a similar hypothesis has been proposed for how human movements are formulated (see Hollerbach [1982]). In fact, one finds that the hierarchy proposed by Hollerbach [1982] comes very close to describing how the hand controller hierarchy operates.

In this and subsequent sections, the different spaces in which the controller operates are outlined along with the algorithms implemented by each level. This section is not intended to cover design aspects involved in synthesizing a digital controller. Such information is more easily found in a number of texts devoted to the subject (see for example Aström and Wittenmark [1984], Ogata [1987]). Rather, this section is intended to provide a flavor of how such low level computations are structured by using the Utah-MIT hand as a concrete example.

All the implementations to be described below are fully digital implementations. There are many reasons for leaning toward a digital implementation, foremost amongst which are:

(a) In most cases, the hardware can be made simpler.

(b) Digital controllers do not exhibit troublesome features such as drift, electrical nonlinearities, etc., that are usually associated with analog components.

(c) They are flexible, and allow fast prototyping and testing of different control algorithms.

(d) They need not necessarily mimic their analog counterparts and can indeed use special algorithms for nonlinear compensations and dead-beat control that have no counterpart in the analog domain.

4.1 Tendon Management

The controller was designed using a layered approach proceeding from the bottom (which is the lowest level of control) to the top. At the lowest level, tendon management issues predominate. At this level the controller servos tendon tensions associated with a particular actuator.

In most robots, the lowest level of the controller has been termed the *actuator* level, and the space in which the controller operates the *actuator* space (see Craig [1986]). In conventional robots driven by electric motors, this would be the level that would, for example, convert joint torques to currents or voltages. Since the Utah-MIT hand is driven by pneumatic actuators and tendons, the ultimate variable controlled by the lowest level of the controller is the tendon tension associated with an actuator. (In practice, there may or may not be one further level of control beneath this level of digital control, usually associated with the amplifiers controlling the actuator). The important variables associated with this level of the controller are the tendon tension loop gain, the co-contraction level, and the rectification function, which will be described shortly.

The tendon space controller has to be extremely simple so that it can be run extremely fast. A simple *P*, *PD* or *PV* controller is usually good enough at this level.

Physically speaking, tendons can only be pulled and not pushed. Hence it becomes necessary to rectify the desired tendon tension, so that it is always positive. There are many ways of doing this rectification. Jacobsen et al. [1983] mention the possibility of using an exponential function to perform the rectification. In practice, the complexity of implementing a non-trivial rectification function is not commensurate with the benefits it

brings in terms of performance. Hence, we decided to use a simple rectification scheme pictured in Figure 4.1.

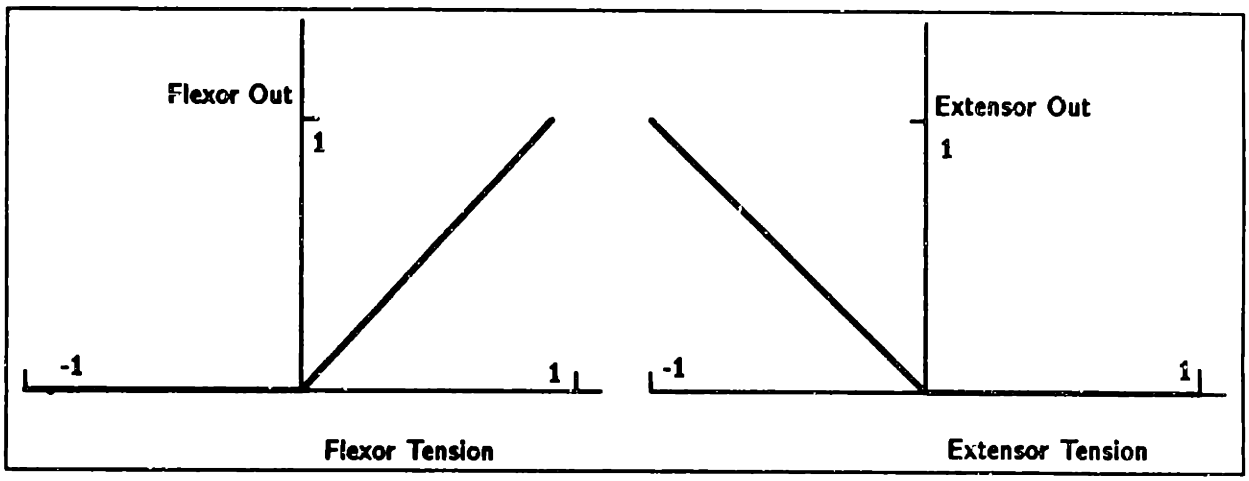


Figure 4.1: Rectification function.

In practice, deciding what levels of co-contraction should be used is non-trivial. For fast operation, the non-linear nature of the system tends to make co-contraction similar to active damping – i.e., the higher the co-contraction, the higher the damping. After experimenting with various ideas on dynamically varying the co-contraction, it was decided to keep it constant at a level which would permit smooth operation over a wide range of speeds.

4.3 Joint Level Control

To develop the next level of the controller, I used a simple second order model of the joint in simulations[1]. With the underlying tendon space controller operating at high sampling rates (see Page 52 for performance information), it becomes possible to treat the lower level as a simple second order system for the purposes of higher level control.

In this section, we will assume that our purpose is to do position control (i.e., the controller will take as input joint angles and output actuator space commands – in the case

[1]Most of the simulations were performed using MatrixX, a control package developed by Integrated Systems Inc.

of the Utah-MIT hand this output will be desired tendon tensions).

Figure 4.2 shows the step response of the underlying joint, which is modeled as a continuous time system. The step response of the actual system after the controller has been tuned appropriately is shown in Figure 4.3.

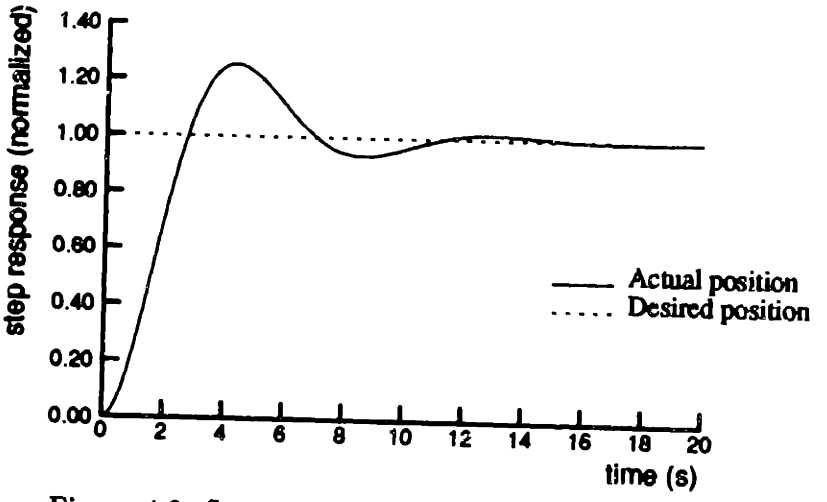


Figure 4.2: Step response of the model.

To understand the final form of the controller, it is illustrative to begin with a simple form of what we would like to achieve in terms of position control:

$$\mathbf{M_p} = \mathbf{K_p} \times \Delta\theta \tag{4.1}$$

where $\mathbf{M_p}$ is the torque to be exerted about a joint and is proportional by some gain $\mathbf{K_p}$ to the position error $\Delta\theta$.

Note that the simple form given by the above equation does not take into account a number of factors, but is extremely easy to understand. It simply produces a restoring torque in order to reduce positional error. The task before the joint angle controller is now to convert this restoring torque (called τ_p to indicate that it is a position torque) into actuator space commands.

To illustrate how this is done, consider the simple single joint shown in Figure 4.4. The

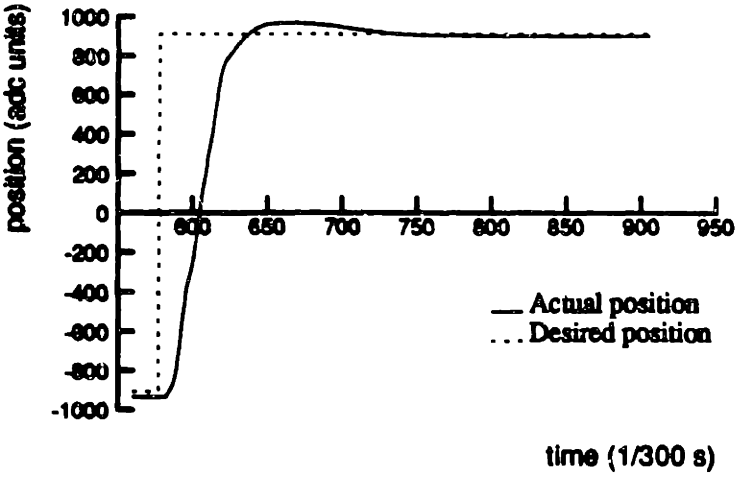


Figure 4.3: Actual step response of final controller.

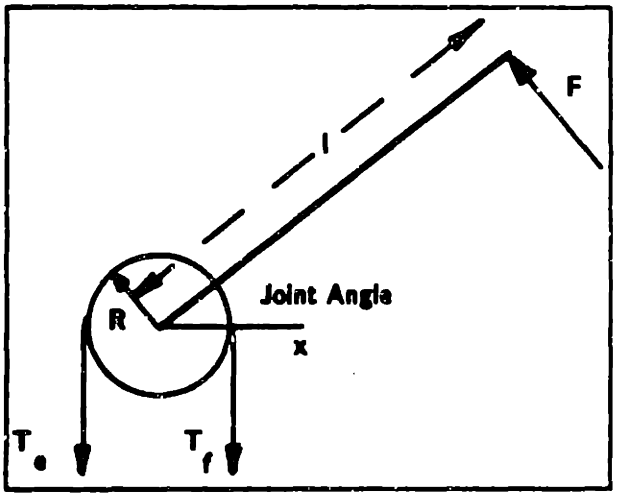


Figure 4.4: Picture of a single joint controlled by 2 tendons.

torque about the joint can be written as

$$\tau = (T_f - T_e) \times R \tag{4.2}$$

where T_f stands for the flexor tension, T_e stands for the extensor tension and R stands for the pulley radius. Notice that this equation implies that if we need to solve for tendon tensions from torques, we have two unknowns but only one equation. However, if we assume that the two tensions will be symmetric about the co-contraction point T_c, we can write

$$\begin{aligned} T_f &= T_c + T_{diff} \\ T_e &= T_c - T_{diff} \end{aligned} \tag{4.3}$$

From these, it is easy to solve for the tension desired as given by:

$$T_{diff} = \frac{\tau}{R \times 2} \tag{4.4}$$

In practice, we would like to augment equation 4.1 with a factor depending on the velocity of the joint as given by

$$\mathbf{M_p} = \mathbf{K_p} \times \Delta\theta + \mathbf{K_v} \times \Delta\dot{\theta} \tag{4.5}$$

A plot of the tendon tensions generated by the controller using the modified equation, for the step response pictured earlier, is shown in Figure 4.5. Since we do not have a way of directly measuring joint angle velocities in the Utah-MIT hand, we rely on using the first difference of joint positions as an estimate of the velocity. More complicated ways of estimating the velocities and using them in the control equation could also be implemented.

The plot does not include co-contractions, which would just shift the entire graph along the y-axis. Notice, however, that the plots indicate a non-synergistic interaction between the actuators. A more synergistic interaction occurs if instead of summing the position and velocity torques and then computing the tendon tensions from this summed value, we were to apply the rectification function on each of the individual torques and then sum the resulting tensions.

The equation for this would be:

$$\mathbf{T} = \mathit{Rect}(\mathbf{K}_p) + \mathit{Rect}(\mathbf{K}_v) + \mathbf{C} \tag{4.6}$$

instead of the earlier

$$\mathbf{T} = \mathit{Rect}(\mathbf{K}_p + \mathbf{K}_v) + \mathbf{C} \tag{4.7}$$

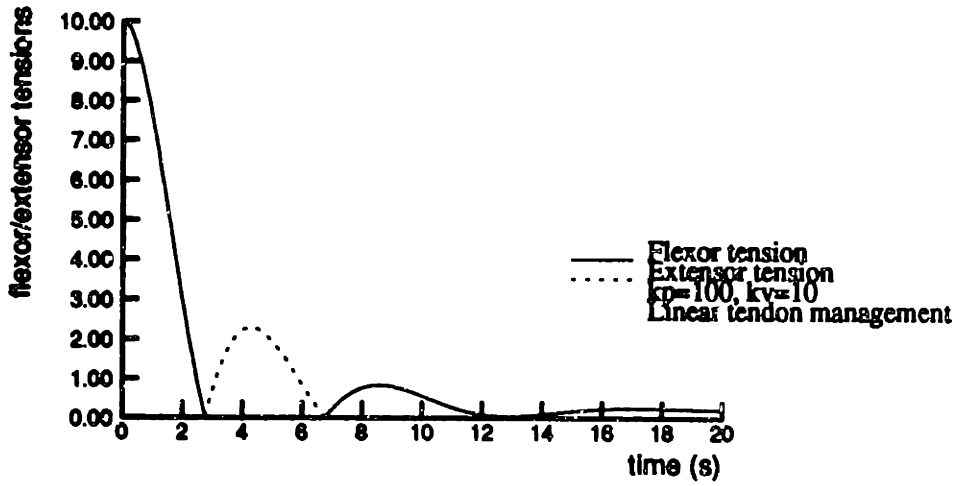


Figure 4.5: Tendon tensions generated.

A plot of the tendon tensions produced by such an equation is shown in Figure 4.6. The plot of actual tensions generated by the controller (which takes into account a coupling effect discussed next) is shown in Figure 4.7.

The joint level controller needs to take into account one further effect, which is the coupling between the various joints. Since there are two tendons per joint and since the hand is tendon driven, the tendons for the distal joints must pass over pulleys installed in the proximal joints. This means that there is coupling between the motions of the proximal joints and the distal joints. This can be seen from a simple two jointed example as shown below (see Figure 4.8).

In practice, the effect of this coupling is rather noticeable. Figure 4.9 demonstrates that while joints 1 through 3 of a finger are moved, joint 3 actually seems to move backward before it recovers and moves forward. Obviously, when the ranges of motion are large, the excursions become larger.

The coupling manifests itself in two ways. Firstly, when a proximal joint is moved, the tendons associated with a distal joint move. Thus, in order to keep a distal joint at the same joint angle relative to a proximal joint, it becomes necessary to introduce decoupling explicitly into the controller. The other way this decoupling can be observed is in the force

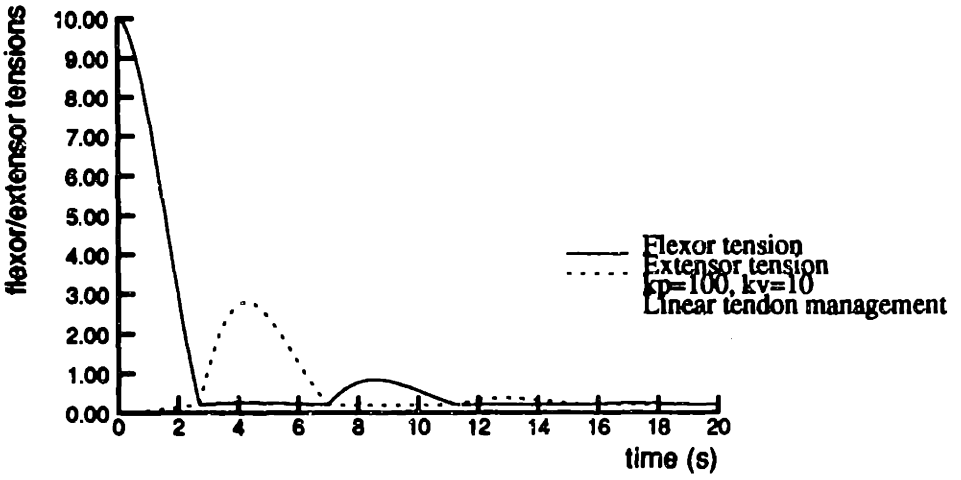


Figure 4.6: Rectification before summing.

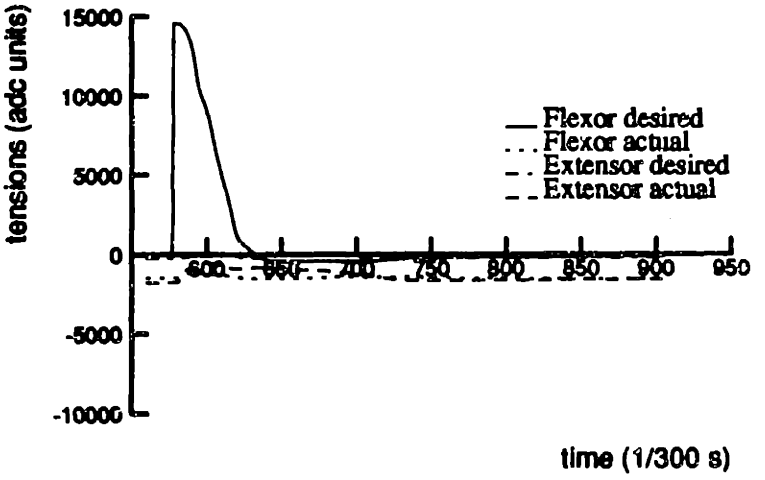
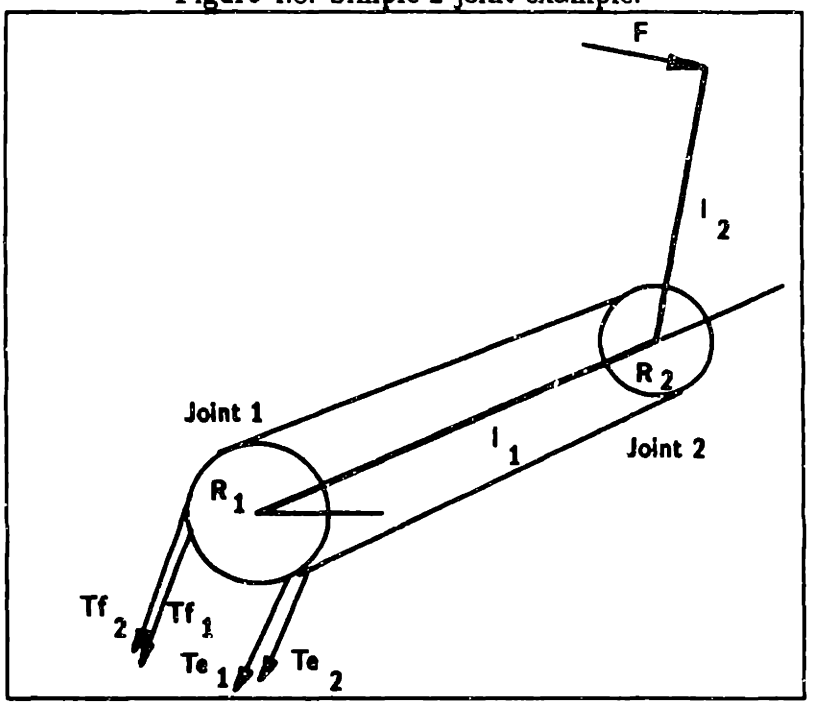


Figure 4.7: Actual tendon tensions generated.

Figure 4.8: Simple 2-joint example.



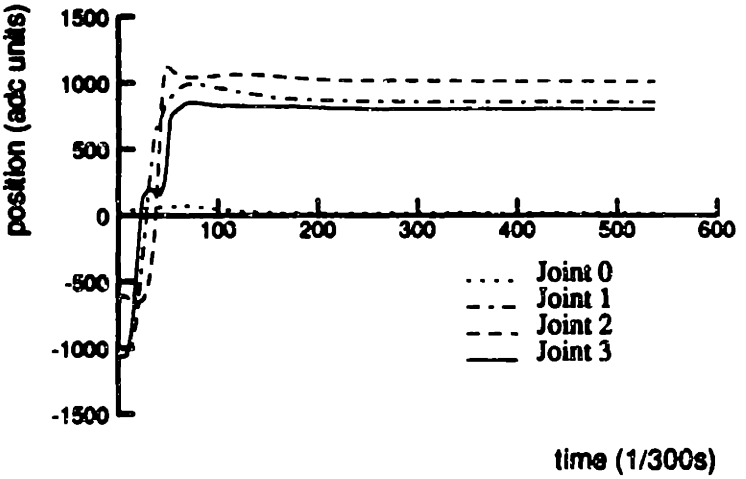


Figure 4.9: Coupling action between joints.

domain – i.e., a torque exerted at the distal joint will mean that the same torque gets exerted at the proximal joint too.

It can be seen that the relationship between tendon velocities and joint angle velocities (or the differential movement relationship), and the relationship between joint torques and tendon tensions is not unlike the relationship expressed conventionally by the Jacobian. Recall that the Jacobian expresses a mapping between joint angle motions and cartesian motions given by

$$\dot{\mathbf{x}} = \mathbf{J}\dot{\theta} \tag{4.8}$$

and relates joint torques to cartesian forces by

$$\tau = \mathbf{J}^T\mathbf{f} \tag{4.9}$$

The mapping expressed by the TM matrix performs a similar function since one can write

$$\tau = \mathbf{TM} \cdot \mathbf{T} \tag{4.10}$$

and

$$\mathbf{T_v} = \mathbf{TM}^T \cdot \dot{\theta} \tag{4.11}$$

where $\mathbf{T_v}$ is a vector representing tendon velocities.

Notice that the columns of this matrix can be easily written down if one uses the differential form of this second relationship.

$$\Delta\mathbf{T_x} = \mathbf{TM}^T \cdot \Delta\theta \tag{4.12}$$

which clearly indicates that if one keeps all the joints fixed and just moves joint i, the corresponding tendon displacement vector ΔT_x will provide the i'th column of the $\mathbf{TM}^T$ matrix.

Owing to the symmetric nature of the TM matrix one can also compute this decoupling in torque space alone by writing

$$\tau_{decoupled} = \mathbf{DC} \cdot \tau \tag{4.13}$$

and the matrix DC now involves ratios of pulley radii, instead of just pulley radii as in the TM matrix. The controller expressed by Eqn. 4.5 can be replaced therefore by:

$$\mathbf{M_p} = \mathbf{DC} \cdot (\mathbf{K_p} \times \Delta\theta + \mathbf{K_v} \times \dot{\Delta\theta}) \tag{4.14}$$

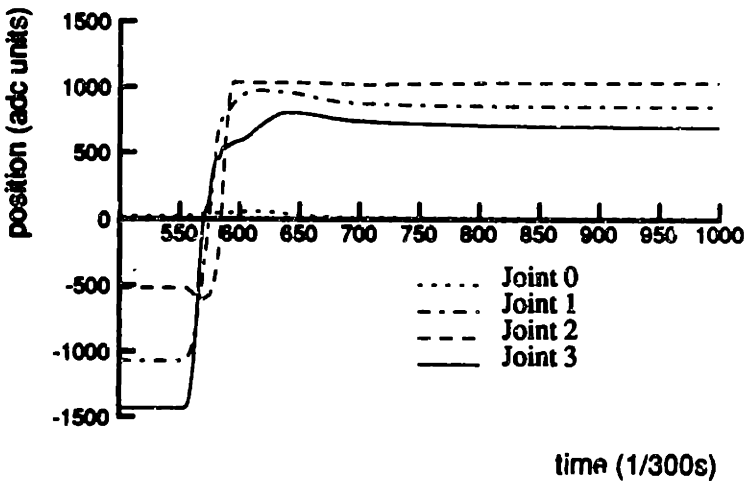


Figure 4.10: Decoupled step response.

In reality, the decoupling matrix depends on the configuration of the joints and their velocities. Expressing such a complex functional dependency would make the controller more complex, and hence we choose to neglect the dynamic effects and concentrate only on the static coupling that exists.

In practice, even such a constant decoupling matrix performs its job rather well. The actual step response output after setting the decoupling matrices appropriately is shown in Figure 4.10, and the effect of the decoupling matrix in the calculation of the torques is shown in Figure 4.11.

We now have derived a controller that takes as an input a joint angle and then computes at the first layer the joint torques needed to achieve the given joint angle. This joint torque is then translated to tendon tensions and servoed by a tendon management servo, after the necessary decoupling and co-contraction levels have been added.

Notice that instead of choosing the joint angle as the desired input (or control variable), we could have chosen the input joint torque as the desired control variable. In this mode of operation, instead of specifying a desired angle to achieve, the higher level controller would directly specify the joint torques that the lower level controller must achieve. The lower level controller can then use this value directly to augment the torque τ_p as computed from

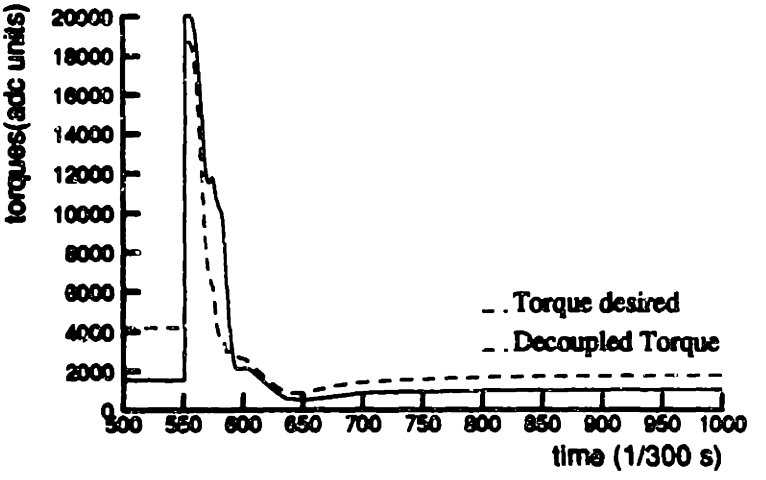


Figure 4.11: Decoupled torques.

$K_p \times \Delta\theta$.

Therefore, as far as a higher level control loop is concerned this joint level control loop can be viewed as a functional box that looks like the one shown in Figure 4.12.

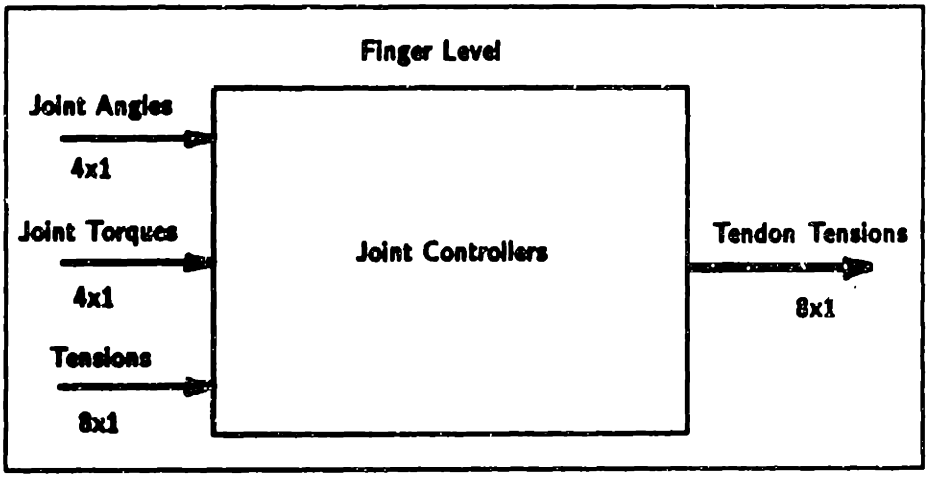


Figure 4.12: Joint controller as viewed from a higher level.

Task Assignment	Speed
4 Joints/Processor	283 Hz
4 Joints/Processor + A/D,D/A	192Hz
2 Joints/Processor	560 Hz

Table 4.1: Performance of the Finger Level Controller - Version I.

4.3 Implementation Issues

We have implemented the hierarchical controller outlined above on two different architectures. In this section we will discuss a few of the issues relevant to implementing such a hierarchical controller. The performance of the two implementations is also described.

Since the two levels of the hierarchy mentioned above (namely, the tendon space level and the joint level) are interwoven and have data dependencies between them, our implementation assigns the so-called *finger-level* controller to a specific processor. The computational architecture designed to implement such control hierarchies is outlined in a later chapter. The important points are that the task is broken up at the finger level and the assignment of tasks to processors is made at compile time and not dynamically.

In the first implementation on the Motorola 68000 based single-board computer, one finger was assigned per processor. In Version II, based on the Motorola 68020 based single board computer, two fingers were assigned to a single processor. The performance of the implementation on the Version I hardware is shown in Table 4.1.

In our first implementation, it was found experimentally that reading and writing the A/D and D/A converters was time consuming. Hence we moved this task to a separate processor. In Version II, the control processors are fast enough to service the A/D and D/A conversions as well.

As can be seen from Table 4.1, reading the analog-to-digital converters and writing to the digital-to-analog converters alone takes approximately half the time in the feedback computation. The performance of the second implementation is indicated in Table 4.2. The last row in this table indicates that the performance of the current implementation is adequate for tracing and monitoring functions as well. In all previous implementations, adding such functionality slowed the performance of the servo loops considerably. Both the two implementations use scaled integer calculations for their control loop computations.

Task Assignment	Speed
8Joints/Processor	550 Hz
8Joints/Processor + A/D,D/A	390Hz
8Joints/Processor + A/D,D/A + Tracing	340Hz

Table 4.2: Performance of the Finger Level controller - Version II.

4.4 Higher Level Control

The third level of control forms the highest level in the current control hierarchy. There are different ways of implementing this level, and a number of different choices that have to be made in the process. Since this level directly interacts with the joint level controller, it is clear that the output of this level should be either a sequence of joint angles (as is usually associated with position control), or a sequence of joint torques (as is usually associated with force control). We will first deal with the issues associated with pure position control in this section. Issues devoted to force control will be addressed later.

In order to do pure position control, a higher level controller could send down a stream of joint angle trajectories to be achieved by the lower level controller. This higher level controller might require information as to when a particular motion that it has asked to be executed has completed.

4.4.1 Hand Primitive Motions

We will first deal with what I like to call *open loop* object level control. In this type of control the position and orientation of a grasped object is controlled with little or no feedback. The idea is essentially to specify co-ordinated motion of the finger joints relatively independently, without worrying too much about the interaction between the finger joints and a grasped object. This style of control follows mainly from the approaches outlined in Mason [1982] and Fearing [1983].

There are advantages and disadvantages associated with this type of control. Very little computation is required in this type of control, and hence the higher level control that interfaces with the joint controllers becomes extremely easy to implement. It is surprising that such types of control are robust in the presence of object and environment uncertainties,

and in the acquisition and grasping of objects. In fact, open loop control essentially means that the final position of a grasped object within the hand may not be known, and hence sophisticated manipulation operations that require such detailed information may not be possible. Planning such operations so that they are guaranteed to succeed is a difficult problem and requires a sophisticated knowledge of both the environment and the grasped object (see Brost et al. [1983]).

In practice, however, this type of control can accomplish a wide variety of tasks. We have been able to perform a number of demonstrations by using a higher level loop wherein the success or failure of an attempted operation is indicated to the control loop. The Utah-MIT hand's kinematic structure that resembles the human hand also makes it easier to define such primitives and actually use them in tasks.

The specification of such motion sequences would be verbose if the position trajectory of every joint needed to be specified. To avoid this, we have chosen to specify the co-ordinated movement of a number of joints via so-called *hand motion primitives.* Hand primitives are essentially a way to avoid specifying a large numbers of joint parameters repeatedly. To grasp a cylindrical object, for example, the simultaneous movement of many of the hand's joints is required. Conventional approaches of using a teach pendant would be hopelessly tedious for programming sixteen joints. Some researchers have proposed using a master/slave system and then operating hands in a teleoperative mode. While this suggestion is not without merit, some rather complex problems have to be overcome to make such a master/slave device. The problems are further compounded by hands with non-anthropomorphic kinematics, where the mapping between the master and slave is non-obvious and may require considerable operator training.

These primitives provide a way of treating entire sets of joints as a single controllable entity. Some of the implemented hand primitives are shown in Table 4.3.
The table is by no means meant to be all-inclusive since the user has the ability to define his own primitives. Primitives can be classified into two categories.

1. *Servo Level* primitives that affect the performance of the control hierarchy by changing the servo parameters (these primitives have not been mentioned in the table).

2. *Joint Subset Level* primitives that affect some given subset of joints of the hand.

Each entry in the table above specifies a primitive and the corresponding joints affected by that primitive. For example, the *THUMB* primitive specifies a motion that is to be executed on joint zero of finger zero, in the positive direction. The *TWIST* primitive causes

Primitive	Fingers affected	Joints Affected
CLOSE	0[+],1[+],2[+],3[+]	1
CURL	0[+],1[+],2[+],3[+]	2,3
CUP	1[+],3[+]	1
SPREAD	1[+],3[-]	0
GRASP	0[+],1[+],2[+],3[+]	1,2,3
TWIST	1[+],3[-]	1
THUMB	0[+]	0
WAIT	All	All
RESET	All	All
START	All	All
SINGLEJOINT	Specified	Specified

Table 4.3: Hand primitive operations.

joint number one of fingers one and three to move in opposite directions simultaneously. Servo level primitives can likewise be specified for subsets of joints. Thus a primitive essentially specifies a co-ordinated motion of a chosen subset of joints. A primitive is not considered to have finished executing until all the joints it affects have moved to their final positions as required by the primitive's single argument. In a conventional robot programming language these primitives could be implemented as a series of `MOVE` statements.

The primitives and the names chosen for them are not important. However, what should be noted is that by treating a number of joints to be controlled as a single parameter, a large amount of the complexity involved in programming the hand can be overcome.

In the current implementation, these hand motion primitives have been extended to form a sort of threaded-interpreted language in which the hand can be programmed. The user can define his own primitives by specifying a name, a set of joints that the primitive affects and the direction of motion associated with each of those joints. At the hand primitive level we have implemented a *motion editor* that allows one to define and operate on entire sequences of hand primitive motions, much like one would operate on pieces of text in a conventional text editor. Using this editor one can enter sequences of hand primitive motions which can then be stored and retrieved from files.

A sequence of motions, once programmed and tested, can itself be treated as a primitive in other motions, enabling the stringing together of long and complicated motion sequences.[2]

Given below are two short examples of such motion sequences, both of which were developed interactively to perform simple tasks:

```
01.  START(+00)  02.   WAIT(+05)  03.  SPEED(+10)
04.  GRASP(-40)  05.   WAIT(+05)
```

The first sequence is an example of a simple grasping operation. As mentioned before, since the hand is controlled completely as a position controlled device, the sequence succeeds in grasping cups or coke cans by relying on the passive compliance provided by the underlying mechanical device.

The second example given below is performed after the hand has successfully grasped a pair of scissors. When executed, the sequence simulates a cutting action.

```
01.   CURL(+10)  02.   WAIT(+05)  03.  CLOSE(-40)
04.   WAIT(+05)  05.  CLOSE(+45)  06.   WAIT(+05)
07.  CLOSE(-45)  08.   WAIT(+05)  09.  CLOSE(+45)
10.   WAIT(+05)  11.  CLOSE(-45)  12.   WAIT(+05)
```

Notice that other ways of programming the hand would involve specifying a trajectory for each of the joints in a more tedious way involving end points and via points.

Both the above examples did not use any *servo-level* primitives to change the servo characteristics as they were being executed. It is easy to see, however, that by judiciously altering the servo parameters like gain and damping, the performance of the hand can be altered during the course of a motion sequence. This is useful for changing between the two extremes of performance achievable with the hand, namely, that of force accuracy and positional accuracy.

It must be mentioned that these motion sequences are parameterizable in terms of speed. This is so because each primitive motion actually causes a series of joint level trajectories to be enqueued. The trajectory generator then uses a simple joint level interpolation scheme

[2]Since sequences of primitives can themselves be treated as primitives, it becomes necessary to do a full topological sort of the sequences before they are written out to a disk file. My approach to solving this problem was essentially gotten from a hierarchical editor called HPEDIT for doing VLSI circuit layouts which maintains a hierarchy of component circuit cells in disk files.

(linear, quadratic and sinusoidal interpolation are the three types of interpolation schemes presently supported), to generate the sequence of joint angles that the lower levels in the control hierarchy must servo to. The linear interpolation is given by the equation

$$\theta_d(t) = \theta_i + t(\theta_f - \theta_i) \tag{4.15}$$

where t ranges from 0 to 1, with a specified step size t_k. This step size is what is altered by the SPEED primitive. By increasing the step size the same primitive motion is performed in a shorter time. The actual time it takes for the primitive motion to complete, of course, is a function of the servo rate and is given by:

$$t_{prim} = \frac{1}{servorate \times t_k} \tag{4.16}$$

4.4.2 Cartesian Space Control

While hand primitives and other such schemes of open loop position control of the joints do accomplish a wide variety of tasks, these suffer from the disadvantage that they fail often owing to a variety of reasons (see Fearing [1987] for a discussion of the baton twirling task). Ensuring that the failure rate goes down would involve a great deal of modeling – both in terms of the geometry of the grasped object and in terms of the physics of the grasping operations.

In this section we present a method for solving the trajectory planning problem during manipulation tasks. This method relies on a number of restrictive assumptions. Some of these will be relaxed in later sections and we will see that planning trajectories in cartesian space for robot hands has quite a different flavor from conventional robot trajectory planning.

In conventional robotics, trajectory planning is usually considered a problem of specifying the positions, velocities and accelerations of the end-effector to carry out a particular task. One way of finding such a specification given the initial and final positions of the end-effector is by using constraint-satisfaction techniques. A variety of constraint polynomials have been used, including cubics to ensure continuity of positions and velocities at the end-points of the motion, quintics to ensure position, velocity and acceleration constraints (see for example Mujtaba [1977]). Resolved motion rate control proposed by Whitney [1969] has been used to repeatedly compute the velocity commands to an underlying velocity controller from the equation:

$$\dot{\theta} = \mathbf{J}^{-1}\dot{\mathbf{x}} \tag{4.17}$$

This method assumes that the cartesian velocity of the end-effector is known or can be computed from the task specification. A method based on computing the successive positions and orientations of the end-effector using homogenous transforms has also been proposed by Paul [1981]. A recursive way of computing the joint space trajectories associated with straight-line cartesian space trajectories has been proposed by Taylor [1979]. This *bounded-deviation* method relies on computing the error in cartesian space of a given trajectory using forward kinematics.

Computing the trajectories to be followed by fingertips of a robot hand manipulating a grasped object is slightly more complicated. As can be seen easily, if one computes straight-line trajectories from the initial positions of the finger tips to their final positions, one could end up crushing a grasped object. A grasped object basically imposes kinematic constraints on where the finger tips or the contacting surfaces can be. Throughout the motion of such a grasped object these constraints have to be satisfied so that the relative distances between the grasp surfaces does not change. A three-fingered hand like the Salisbury hand can be said to make three point contacts with any object it grasps. These three points define a triangle in space which must not change shape as the object is moved by the hand. In the case of the Utah-MIT hand one can think of the tetrahedron formed by the contact points as being manipulated in space instead of the grasped object.

The goal, therefore, is to compute a time history of cartesian positions x_{ik} which denotes the cartesian position of the i'th finger tip at the k'th time step. Once we have these cartesian positions, using the inverse kinematics relations, we can compute the stream of required joint angles.

In the following sections we outline two ways of solving the trajectory planning problem during a grasping operation. The two ways differ in their representation of rotations and consequently in how many operations they need to compute the intermediate points. The first method uses homogenous transformations to represent rotations while the second method uses quaternions.

The first scheme of computing the joint space trajectories follows the presentation outlined in Chiu [1983], wherein linear joint interpolation is used to achieve cartesian space trajectories of the finger tips. The scheme relies on computing intermediate knot points of a given trajectory and then uses linear joint interpolation between the knot points.

4.4.3 Motions Specified using Homogenous Transforms

The motion of a grasped object can be specified in a number of ways. We first deal with motions specified using homogenous transformation matrices.

4.4.3.1 Motion Specified Relative to an Absolute Reference

Generally speaking, it will be possible to indicate this motion relative to an absolute reference frame by a homogenous transform matrix A_m^o. In order to satisfy the constraints that at each point in the motion the contact points must not move relative to one another, we have to compute the position of the grasped object at every instant of the motion. Following the notation in Paul [1981], we can see that this can be done by the following equation. Note that the equation indicates a screwing motion that translates and rotates the grasped object at the same time, in contrast to method outlined by Paul wherein the rotation is split into two separate rotations.

$$A_m^o(t) = \begin{bmatrix} Rot(\hat{\mathbf{n}}, \theta(t)) & \mathbf{p}(t) \\ 0 \quad 0 \quad 0 & 1 \end{bmatrix} \tag{4.18}$$

The above equation essentially expresses the famous theorem by Chasles, that any motion can be considered to be a combination of a translation $\mathbf{p}$ with a unique rotation of angle θ about a unique rotation axis $\hat{\mathbf{n}}$. If a linear interpolation is used, and if this motion needs to be performed in N steps, then after k steps the body will have translated to a position given by

$$p_k = \frac{k\,\mathbf{p}}{N}$$

and will have rotated by $k\,\theta/N$ about the rotation axis $\hat{\mathbf{n}}$. However, the orientation of the object will be indicated by

$$R(t_k) = R^k(\hat{\mathbf{n}}, \frac{\theta}{N})$$

since rotations are composed by multiplying the rotation matrices together.

Now consider an object grasped by the finger-tips at n contact points. The cartesian co-ordinates of the finger-tips can be represented by $\mathbf{x_i}$. At any given instant during the specified motion therefore

$$\mathbf{x_i}(t) = A_m^o(t)\ \mathbf{x_i}(t_0)$$

where $\mathbf{x_i}(t_0)$ specifies the position of the finger tips at the beginning of the motion.

4.4.3.2 Motion Specified Relative to an Alternate Frame

In some cases it may be necessary to specify the motion not in a global reference frame but relative to some other frame, say A_r. The contact points $\mathbf{x}_i(t_0)$ can be expressed relative to this initial frame by $A_r^{-1}\ \mathbf{x}_i(t_0)$. If the motion expressed relative to the A_r frame is $A_m^r(t)$, then the finger tip positions at any given instant during the motion can be seen to be

$$A_r\ A_m^r(t)\ A_r^{-1}\ \mathbf{x}_i(t_0)$$

relative to the global reference frame.

As outlined in the previous section, motions can be specified relative to an absolute reference frame. Such motions are meaningful only in a few situations. More often one desires to specify these motions relative to the grasped object; For example, the operation of a screw driver involves twisting about an axis defined by the screw driver, while exerting a force about that axis. The problem therefore becomes one of determining a meaningful frame relative to which the motion ought to be specified. In other words, what is a meaningful value for the frame A_r in the last equation?

Location of this object specific reference frame A_r can be referred to as the *grasp localization* problem, since it refers to identifying the position and orientation of a grasped object within a dexterous hand. This can be solved in three different ways.

1. The first method is to use external sensing modalities like vision or tactile sensing, to identify the frame A_r directly from identifying visually or tactually observable features on the grasped object.

2. The second method is to assume that the grasp points are known relative to this frame (usually this is the form that the output of a planner will generally take). This implies solving for the elements of A_r from:

   $$\mathbf{x}_i(t_0) = A_r \mathbf{x}_i^r$$

   The left hand side of the above equation can be gotten from the forward kinematics at instant t_0. $\mathbf{x}_i^r$ specifies the grasp points relative to this frame A_r. The first method which relies on sensors could be affected by sensor noise but is unaffected by uncertainties in the initial positions and orientations of the grasped object. The second method on the other hand requires a detailed knowledge of where the fingers are placed relative to this frame and could compute the wrong value for A_r if the fingers are not where they are supposed to be.

3. A third method outlined in Chiu [1983] is to compute a frame known as the *grasp frame* relative to the positions of the finger tips $A_g = \mathcal{F}(\mathbf{x}_i(t_0))$. Then the frame A_r can be specified relative to this grasp frame by $A_r = A_g\ A_{c'}$. This method computes *forward* from the current finger tip positions and is a compromise between the previous two. It will be affected both by sensor noise and by uncertainties in the initial position and orientation of the grasped object.

To summarize therefore, we present the iterative algorithm for actually computing the intermediate knot-points during a cartesian trajectory.

Input:

1. A motion spec of the form $[\hat{\mathbf{n}}, \theta, \mathbf{p}]$ specifying a rotation of θ about an axis $\hat{\mathbf{n}}$, and a translation $\mathbf{p}$ along the axis.
2. N The number of intermediate knot points to compute.
3. The frame A_r relative to which this motion spec has to be applied.

Ouiput:

1. A stream of cartesian positions $\mathbf{x_{ik}}$ for all the finger tips. For the Utah-MIT hand i ranges from 0 to 3, and k ranges from 0 to $N-1$.

Computation:

1. Compute A_r^{-1}, $\mathbf{x}_i(t_0)$, and $\mathbf{x_{start}} = A_r^{-1}\ \mathbf{x}_i(t_0)$.
2. Compute $\Delta\theta = \theta/N$ and $\Delta\mathbf{p} = \mathbf{p}/N$.
3. Let A_r represent the incremental translation and rotation represented by $[\hat{\mathbf{n}}, \Delta\theta, \Delta\mathbf{p}]$.

$$A_r = \begin{bmatrix} Rot(\hat{\mathbf{n}}, \Delta\theta) & \Delta\mathbf{p} \\ 0\ \ 0\ \ 0 & 1 \end{bmatrix}$$

4. Let $A_0 = A_r$. For i from 0 to N by 1 do compute:

$$A_{i+1} = A_i \cdot A_r$$

$$\mathbf{x}_{i+1} = A_{i+1}\ \mathbf{x_{start}}$$

4.4.4 Motions Specified using Quaternions

In the previous section, we used homogenous transforms to represent motions. We presented algorithms for two cases. First, the motion was specified relative to an absolute frame, and then, an alternate frame of reference was used to specify the motion. The algorithms outlined above essentially rely on computing the position and orientation of a grasped object at different points during a motion, and then computing the trajectories traced by the finger tips relative to this frame. The computational complexity of doing this in real time can be seen to be essentially

$$C(T_1 \circ T_2) + n\, C(T \circ \mathbf{v})$$

where $C(T_1 \circ T_2)$ represents the complexity of composing one transformation matrix with another, and $C(T \circ \mathbf{v})$ represents the complexity of computing a vector transformed by a motion transformation.

If one represents the configuration of an object using homogenous transformations, it would take a total of **33M+24A** to compose two transforms. (The rotations alone would take **24M+15A** – since the rotation matrix part of a homogenous transform is orthonormal the third column can be computed as the cross product of the previous two). The operation of computing the transformation of a single point $\mathbf{v}$ by a transformation matrix T can be expressed as

$$\mathbf{v}' = Rot(T) \circ \mathbf{v} + Trans(T)$$

Computing the first term in the above equation can be done in 9M+6A operations, while the second term takes 3 more additions. Therefore a total of 9M+9A operations are required to do the second operation.

The total number of operations needed to perform the online trajectory computation neglecting the precomputations performed is given by **(33+9n)M + (24+9n)A**. For $n = 4$ as is the case with the Utah-MIT hand this turns out to be **60M+51A**.

In this section, we will essentially use the same algorithm as in the previous section, but use a different representation for rotations. Unit quaternions provide a compact representation of rotations. In fact, such quaternions have been used by Taylor [1979] in his trajectory planning scheme for conventional robots. Such a quaternion represents a rotation of θ about an axis $\hat{\mathbf{n}}$ as $[cos(\frac{\theta}{2}), \hat{\mathbf{n}} sin(\frac{\theta}{2})]$. We will denote a quaternion as the a scalar part q_0 taken together with a vector part $\mathbf{q_0}$ to form the quaternion $\mathbf{Q_0} = [q_0, \mathbf{q_0}]$.

The configuration of the object can be described by $C_o = [\mathbf{x_o}, \mathbf{Q_o}]$. The position of the object is represented by the position vector $\mathbf{x_o}$ while the orientation is represented by the unit quaternion $\mathbf{Q_o}$. A short description of the quaternion representation for rotations and its use can be found in Horn [1986].

In the standard representation quaternion composition of rotations takes only 16M+9A. Computing the transformation of point vectors by a quaternion takes 22M+12A in the standard representation of this composition which is:

$$\mathbf{v}' = \mathbf{v}(q^2 - \mathbf{q} \cdot \mathbf{q}) + 2(\mathbf{v} \cdot \mathbf{q})\mathbf{q} + 2\, q\, \mathbf{q} \times \mathbf{y}$$

As suggested in Canny [1984], we can express the above equation as

$$\mathbf{v}' = \mathbf{v} + 2\, \mathbf{q} \times (\mathbf{q} \times \mathbf{v}) + 2\, q\, \mathbf{q} \times \mathbf{v}$$

which can be computed using only 15M+12A operations. [3].

The total complexity of doing the online trajectory computation using such a representation of rotation is given therefore by (16+15n)M + (9+12n)A operations. For the Utah-MIT hand this figure turns out to be 76M+57A. The complexity of operating with quaternions can be seen to be slightly higher.

The above two ways of computing the trajectories assumed that the object was translating uniformly and rotating at a uniform angular velocity as it made the required rotation. If one relaxes this requirement that the angular velocity be uniform, an alternate representation for rotations that uses the scaling property of quaternions can be used to denote Q_0 by $\mathbf{Q_0} = [1, \hat{\mathbf{n}} tan(\frac{\theta}{2})]$. Using this a linear representation for intermediate points during a rotation can be derived as

$$\mathbf{Q(t)} = [1, t\hat{\mathbf{n}} tan(\frac{\theta}{2})]$$

where t could range continuously from 0 to 1.

This allows the intermediate orientations of an object be computed essentially with three multiplications and renormalizations. Such a trajectory through orientation space will not have a uniform angular velocity as noted by Canny [1984], although it will accomplish the required rotation about the required axis. The desirable feature about this representation is that since it relies on a continuous representation it allows a planner to be able to compute the intermediate orientation of the grasped object at any point during the trajectory. Needless to say, there are other ways of defining $\mathbf{Q(t)}$, but a linear interpolation as defined by the equation suggested above, is perhaps the simplest.

[3] Multiplication by 1 or 2 is counted trivially as an addition operation – see Salamin [1979]

Pure position control requires accurate manipulators. This is especially true in assembly tasks that require fine manipulation capabilities. Typical part dimensions in optical fiber assemblies or magnetic core assemblies for computers are less than a millimeter while the tolerance on the parts is of the order of 0.25 micrometers. More recent integrated circuit manufacturing tasks require positioning stages that are accurate to 0.02 micrometers (see Toumi et al. [1986]).

It is clear that to achieve such accuracies, much higher standards need to be achieved in the manufacturing and control of robots.

In some tasks, however, merely increasing the accuracy of the robot performing the task is not enough. If such tasks are to be performed by robots under conventional position control, the models that the robot uses need to be near perfect. The model of the environment needs to be accurate so that the motion planner can generate the right positions that the robot must move to. The model of the robot in terms of kinematic and dynamic parameters needs to be accurate too, so that a controller can generate the right signals to the actuators while making the motions that the planners generate in the presence of unknown disturbances. In reality, perfect robot models, and even good robot models, are hard to come by. The changing and unstructured environment is again impossible to model and hence pure accurate position control for robot manipulators faces fundamental limitations for certain kinds of tasks.

In this section, we present the force control algorithm presented originally in Hollerbach, Narasimhan and Wood [1985] that forms the basis for the implementation of the force controller for the Utah-MIT hand.

5.1 Introduction and Previous Work

A force controller can be implemented in a number of different ways. In general, using a force sensor to monitor the forces of interaction that a manipulator experiences, and then using this force information to modify the controlled parameters like position or velocity of the robot has been termed *force control* (see Whitney [1985]). It has also been applied to the process of generating force signals, in response to deviations from some nominal

trajectory (see Salisbury [1982]).

It is instructive to consider the different alternatives that have been proposed in the literature (Whitney [1985] presents a survey of current force control methods and Maples and Becker [1986] provide a further classification). We will deal exclusively with *active* force control schemes, as opposed to *passive* force control schemes wherein the inherent mechanical compliance of a manipulator is used to perform tasks that require fine manipulation. It must be mentioned that no real manipulator is perfectly stiff, and every robot has some amount of inherent mechanical compliance. This is especially true in the case of a robot like the Utah-MIT hand wherein the inherent damping and mechanical compliance provided by the underlying mechanical system comes very close to providing an ideally stable system for tasks that involve contact with the environment.

Damping control was proposed by Whitney [1977]. In this form of force control, the inverse of a damping matrix is used to modify the desired velocity vector based on the sensed forces. Such a form of control has also been assumed by planners which use the so-called *generalized damper* formulation (see Lozano-Pérez et al. [1984], Donald [1987], Erdmann [1984]).

Salisbury et al. [1982] proposed an alternative to damping control, which has now come to be known as *stiffness control*. In this method, the underlying controller is one that generates forces in response to deviations from some nominal trajectory according to a *generalized-spring* model. His driving equations then could be expressed as

$$\begin{aligned} \mathbf{f} &= \mathbf{K}_{p_x} \cdot \Delta\mathbf{x} \\ \tau &= \mathbf{J}^{\mathbf{T}}\, \mathbf{K}_{p_x}\, \mathbf{J}\Delta\theta \end{aligned} \tag{5.1}$$

The quantity $\mathbf{J}^{\mathbf{T}}\, \mathbf{K}_{p_x}\, \mathbf{J}$ can be denoted as $\mathbf{K}_{p\theta}$ and represents the cartesian stiffness matrix mapped to joint space.

Hogan [1985] provides yet another alternative called *impedance control* which uses damping and stiffness matrices to control a so-called virtual manipulator. Raibert and Craig [1981] propose a scheme called the *hybrid* force control scheme wherein certain axes of the manipulator are force controlled while others are controlled using conventional position control schemes. Mason [1982] presents a formal theory of how natural and artificial constraints arising out of a given task situation can be used to provide information to a hybrid controller.

Perhaps the most important choice that has to be made in designing a force control

scheme is to decide the co-ordinate system in which the error to the controller will be measured. In the scheme proposed by Salisbury et al. [1982] the cartesian error $\Delta \mathbf{x}$ can be mapped to joint space using the cartesian stiffness matrix mapped to joint space. The errors in this method are computed in joint space ($\Delta\theta$). In the hybrid position and force control scheme (Raibert and Craig [1981]), and in the generalized damper scheme (Whitney [1977]) the errors are computed in cartesian space.

There are many problems associated with computing the error in joint space. First, it is more natural to describe tasks in cartesian space and describe the partitioning of which joints are to be force controlled and which are to be position controlled in this task space (see Mason [1982]). Specifying these task level constraints in joint space is certainly cumbersome. If the task level constraints are specified in cartesian space but the underlying controller is written to operate in joint space, then usually some scheme for interpolating has to be designed. These interpolation schemes in joint space usually result in curved axes in cartesian space (see Chiu [1983] for an interpolating controller based on the generalized stiffness method).

The advantage to joint based schemes that is often cited in the literature is that the computation associated with such schemes is often small when compared with cartesian based schemes. But as we will see later on in this chapter, this is not necessarily true. In practice, as reported in Maples and Becker [1986] specialized fine tuning of joint based controllers often adds to system complexity.

These considerations led to our choice for computing the error in cartesian space instead of joint space.

A number of papers on force control published recently have dealt with the stability issues associated with such controllers, since most force controllers have had problems with stability when the robot comes into contact with a stiff environment. An [1986] identifies the kinematic and dynamic instabilities that can occur when a robot contacts its environment, and presents the results of a number of experiments performed with a high performance direct drive robot. The importance of modelling and estimation coupled with experimentation to test the validity of theoretical results is emphasized in this significant work. To solve the stability problem, Whitney [1985] and Roberts [1984] have proposed using a compliant covering to increasing the damping at the end-effector. An [1986] has proposed adapting to the stiffness of the environment by using an identified model of the dynamics of the environment. Using joint torque sensing in lieu of a tip force sensor in the feedback loop has been proposed by Wu and Paul [1980] and studied in detail by An [1986].

5.2 The Force Control Algorithm

Before going into the details of the force control algorithm, it is useful to briefly outline what needs to be computed and why.

Consider an object being grasped by the finger tips. (Note that we are making the assumption that no *non-tip* contact is made, which in some cases could be overly restrictive). To manipulate the object (i.e. to *move* it) or to exert forces on the environment with it, one could say that one has to exert a wrench on the object. These wrenches can be exerted on the object only through the contacting surfaces. If the wrench to be exerted on the object is specified, then the problem can be seen to comprise of two subproblems, viz.,

1. computing the wrenches to be exerted at the contacting surfaces and then
2. computing the joint torques so that these wrenches will be exerted at these contacting surfaces.

5.2.1 Lines, Screws, Wrenches, and Twists

The following section presents briefly some necessary mathematical groundwork, so that the algorithm presented later on in this chapter can be easily understood.

A line in space can be represented by four quantities, for example, the slope and intercept of the projections of the line onto two different orthogonal planes. A line can also be represented by its so-called Plücker co-ordinates, wherein a line is represented by six numbers $L = [l, m, n, p, q, r]^T$. The numbers l, m, n are the direction cosines of the line and p, q, r represent the moment of the line about the origin of some reference co-ordinate system. This can also be denoted as $[\mathbf{l}, \mathbf{r} \times \mathbf{l}]$. From this it follows that

$$\begin{aligned} l^2 + m^2 + n^2 &= 1 \\ lp + mq + nr &= 0 \end{aligned} \tag{5.2}$$

A *screw* is a five dimensional quantity, which is basically a line in space augmented with a quantity called the *pitch*. If the screw is represented by $\mathbf{S} = [S_0, S_1, S_2, S_3, S_4, S_5]^T$, the pitch is given by:

$$p_s = \frac{S_0 S_3 + S_1 S_4 + S_2 S_5}{S_0^2 + S_1^2 + S_2^2}$$

If a screw is given by $\mathbf{S} = [\mathbf{S}_v, \mathbf{S}_o]$ the Plücker line co-ordinates of $\mathbf{S}$ are given by

$$\mathbf{S}_p = [\mathbf{S}_v, \mathbf{S}_o - p_s \mathbf{S}_v]$$

Owing to a famous theorem by Poinsot, any system of forces and torques acting on a body can be reduced to a single force and torque, in effect, a wrench. This means that the effect of all the contact wrenches will be the same as if a single *body wrench* were applied to the body as a whole. An analogous theorem due to Chasles states that a displacement of a rigid body can be considered to be a unique rotation about an axis combined with a unique translation about that axis.

A *twist* (or *wrench*) is a six dimensional quantity used to represent generalized motion (or forces). The primary difference between the two is that wrenches add just like vectors while twists do not (only infinitesimally small twists do). The six dimensional quantities are basically a screw augmented with an *amplitude*. In the case of five-dimensional screws

$$S_0^2 + S_1^2 + S_2^2 = 1$$

but in the case of a *twist* (*wrench*), the sum of the square of the primary component forms the amplitude.

Twists and wrenches provide a convenient representation for thinking about contacts and grasping with articulated hands.

5.2.2 Internal Forces

Consider an object being held by the fingers of a dexterous hand. Each contacting surface exerts a particular wrench on the object, by restricting its motion in certain directions. This set of wrenches exerted by the contact points can be termed the *contact wrenches*.

Let the generalized force exerted by the grasped object on the environment be denoted by

$$\mathbf{w}_o = (\mathbf{f}_o, \mathbf{n}_o)$$

and the sum of the contact wrenches be denoted by

$$\mathbf{w}_e = (\mathbf{f}_e, \mathbf{n}_e)$$

Such an object force can be synthesized from the task requirements using artificial and natural constraints, but for now we will assume that it is specified in some fashion (see for example Mason [1982]).

Each contact point i exerts a wrench $\mathbf{w}_i$ through the point of contact. This wrench is made up of a number of components parallel to the basis wrenches, the composition of which depends on the *type of contact* model assumed. For example, a point contact with

friction in three dimensions prevents relative translation between the contacting surface and the grasped object, but does not prevent relative rotation. Relative translation will be opposed by a frictional force. This frictional force will have one component normal to the surface of contact, and two parallel to a plane tangential to the contacting surface. The force normal to the surface is unisense, but the other two components can be bidirectional. If a *soft* finger contacting model is assumed, then there can be a fourth component in the wrench exerted by the contacting surface, namely, a torque arising about the contact normal.

Salisbury [1982] classifies the different types of contact possible based on the wrench systems they exert on the grasped object in three dimensions. Let $\mathbf{w}_e$ be the external wrench exerted on the grasped object as the result of all the tip contacts.

$$\mathbf{w}_e = \sum_{i=1}^{n} \sum_{j=1}^{n_i} \lambda_{ij} \mathbf{w}_{ij} \tag{5.3}$$

where n is the number of contacts and n_i refers to the number of wrenches belonging to the set of wrenches caused by contact surface i. More compactly,

$$\mathbf{w}_e = \mathcal{W}\mathbf{c}$$

where $\mathcal{W} = (\mathbf{w}_{11}, \ldots, \mathbf{w}_{1n_1}, \mathbf{w}_{2n_1}, \ldots, \mathbf{w}_{nn_n})$ and $\mathbf{c} = (\lambda_{11}, \ldots, \lambda_{1n_1}, \lambda_{21}, \ldots, \lambda_{nn_n})$.

It can be seen that this system of equations is underconstrained. In the case of three fingers assuming a model of point contact with friction, $\mathcal{W}$ is a 6×9 matrix, while under a soft finger contact model, it is a 9×12 matrix.

In the above equation λ_{ij} refers to the intensity or magnitude of the j'th component of the i'th contact wrench. Clearly, since some of the contact wrench components need to be unisense, these wrench intensities have to be positive in any solution.

In its most general form, the solution to the above equation can be expressed as:

$$\mathbf{C} = \mathbf{C_p} + \mathbf{C_h} \tag{5.4}$$

where $\mathbf{C_p}$ is the particular solution, and $\mathbf{C_h}$ is the homogenous solution that lies in the null space of $\mathcal{W}$. Salisbury [1982] and Kerr [1984] have both suggested the use of the right generalized inverse to solve for $\mathbf{C_p}$. The components of $\mathbf{C_h}$ are the internal grasp forces. Such a combination of wrench intensities lying in the nullspace of $\mathcal{W}$ that will result in no net wrench on the object can be chosen a number of different ways.

Salisbury [1985] augmented this equation with additional *internal grip force equations* to make the system of equations solvable, as follows:

$$f_{ij} = (\mathbf{f}_i - \mathbf{f}_j) \cdot \mathbf{r}_{ij} \tag{5.5}$$

where $\mathbf{f}_i$ is the force exerted by finger i, $\mathbf{r}_{ij}$ is the unit vector from the ith to the jth contact point, and f_{ij} is a specified scalar internal grasping force.

In general, there is a question as to how these internal forces are defined and how they are specified. Kerr and Roth [1986] present an algorithm to choose these internal forces so that an optimally stable grasp can be achieved. This algorithm relies on simplifying the friction constraints to four linear constraints (the *friction cone* is considered to be bounded by some *friction pyramid*), and then solving the resulting linear programming problem using standard techniques.

Notice that from the above equation we have one internal force component per pair of contacts. This means that for three degree of freedom fingers there will be ${}^3C_2$ or three such internal force components. If we assume a point contact with friction model, each contact exerts a force on the object but no torque at the contacting point. The specification of all the finger tip forces would involve specifying the nine components of these three force vectors. Augmenting the wrench to be exerted on the object (which involves six components) with the internal forces, we see that the resulting system of equations becomes solvable.

The same result holds for point contact with friction models, for four contact points as well as can be seen from Table 5.1.

no. of contacts	tipforce components	w_o	internal forces
2	6	6	1
3	9	6	3
4	12	6	6
5	15	6	10

Table 5.1: Summary of external equations required - point contact with friction.

We can see from the table that for point contacts with friction, the above definition of internal forces gives rise to an overdetermined system in the case of two finger contacts, and an overdetermined system for five fingered contacts. It has been pointed out by Salisbury [1982] that to achieve complete restraint with any of the three degree of freedom contact

types is not possible. For example, two point contacts with friction grasping an object will still not be able to restrict twists about an axis between the two point contacts.

In the case of three soft-fingered contacts there are twelve components to be solved for. By using the above definition of internal forces we can get three additional constraints. The remaining three constraints can be defined to be the linear forces along lines formed by the intersection of the planes normal to the contacting surfaces.

$$\begin{aligned} \mathbf{f_4} &= \lambda_4(\mathbf{n_1} \times \mathbf{n_2}) \\ \mathbf{f_5} &= \lambda_5(\mathbf{n_1} \times \mathbf{n_3}) \\ \mathbf{f_6} &= \lambda_6(\mathbf{n_2} \times \mathbf{n_3}) \end{aligned} \tag{5.6}$$

where $\mathbf{n_i}$ represents the surface normal vector at the i'th contact point, λ_i represents the force component and $\mathbf{f_i}$ the i'th additional force. These three equations when added to the others make the system of equations solvable in the three fingered contact case, with soft finger contacts. Notice that using such a definition implies knowledge of the contact surface geometry. In the algorithm presented below, we solve the three and four fingered case assuming a point contact model with friction but do not solve the soft-finger contact model.

5.3 Algorithm

In general therefore, the computation of the control law can be broken into the following steps.

1. Sense joint positions and contact points at finger surfaces.
2. Compute present cartesian position of the grasped object.
3. Compute the cartesian position error $\Delta\mathbf{x}$ of the object from its desired position.
4. Compute the control wrench $\mathbf{w_o}$ to be exerted on the object. This is done by some *control law* (usually the *Generalized Stiffness* equation is used), using the position error computed in the previous step.
5. Compute from this wrench the wrenches that need to be exerted at the finger tips, or other contacting surfaces.
6. From the finger tip force compute the joint torques that need to be exerted by the actuators.

There are a number of assumptions implicit in the above mentioned algorithm. It is instructive to enumerate them, so as to get an idea of the algorithm's general applicability. It should also be mentioned that while certain steps of the algorithm are inherently serial, others are not. Such portions of the algorithm can be implemented very conveniently in parallel on the computational architecture that we have developed.

1. We have assumed that the contact points stick. No sliding or relative motion is assumed to occur between the contact points and the grasped object.

2. Usually some kind of contact model has to be assumed, which remains fixed for a specified duration. A *soft-finger* type of contact is very different from a planar contact with friction, and the solvability of some of the equations will depend on the type of contact model chosen. Also, a planar contact may change to a line contact when the object moves around within the hand.

3. The entire operation is assumed to be quasi-static. If fingers move fast enough, then the dynamics of the contact situation need to be considered and will complicate the analysis to a very great extent.

4. We have also implicitly assumed that accurate joint torque control is possible. This assumption is usually implicit in most force control schemes. Almost all of these formulations whose output involves forces or torques assume that the underlying controller and actuation systems will be able to achieve the forces or torques asked of them. This may not be necessarily true.

5. Objects are assumed to be rigid as are fingertips. In reality most finger tips have soft coverings on them to prevent them from damage and to increase grip stability, and objects can be made of non-rigid materials too.

5.4 Computing the Object Displacement

The first two steps of the algorithm essentially involve the computation of the forward kinematics which we have already covered. As a result of this computation, we get the cartesian positions of the contact points (which are usually at the fingertips).

Given the cartesian positions of the finger tips, the next step is to compute the differential displacement of the grasped object. This differential displacement can be computed in

one of two ways. The first is conceptually easy to understand but suffers from the inability to recognize a number of special cases; the second yields a simple and compact solution.

It is easily shown that one can compute the displacement of a body, given the displacement of three non-collinear points on the body. We can express the position vector of a contact point as

$$\mathbf{r}_i = \mathbf{r} + \mathbf{l}_i$$

From this the displacement of each of the fingertips can be written as:

$$\mathbf{dr}_i = \mathbf{dr} + \omega \times \mathbf{l}_i$$

where **dr** represents the translation of the grasped object, ω represents the rotation of the grasped object and $\mathbf{l}_i$ represents a vector drawn from the representative point to a contact point. Subtracting one equation from the other and expressing them in matrix form we get:

$$\begin{bmatrix} \mathbf{dr}_1 - \mathbf{dr}_2 \\ \mathbf{dr}_1 - \mathbf{dr}_3 \\ \mathbf{dr}_2 - \mathbf{dr}_3 \end{bmatrix} = \omega \times \begin{bmatrix} \mathbf{l}_1 - \mathbf{l}_2 \\ \mathbf{l}_1 - \mathbf{l}_3 \\ \mathbf{l}_2 - \mathbf{l}_3 \end{bmatrix} \tag{5.7}$$

for three contact points. The above equation can be easily solved for the components of ω by using the fact that

$$\omega\times = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & \omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$$

To solve the vector equation, one can solve for ω and then recover the components of **dr** using one of the above equations. There are many special cases involved in the above computation which takes 17M+38A steps. These have to do with cases wherein a division by zero would occur. Taking care of such cases would require further computation. Whether the motion is really a translation or a rotation is also not readily apparent in the above formulation.

Using the theorems mentioned above, one can formulate another algorithm that recovers the components of a screw displacement, given the displacements of the finger tips. The algorithm is a slightly modified version of the algorithm given in Angeles [1982]. Let **a**, **b** and **c** denote three displacement vectors $\mathbf{dr}_1, \mathbf{dr}_2$ and $\mathbf{dr}_3$. The objective is to compute $\hat{n}$, the vector about which the rotation occurs, the translation along this vector and the angle of rotation. The algorithm proceeds as follows:

1. Compute differences between pairs of displacement vectors. Let

$$\begin{aligned} \delta_a &= \mathbf{a} - \mathbf{b} \\ \delta_b &= \mathbf{b} - \mathbf{c} \\ \delta_c &= \mathbf{a} - \mathbf{c} \end{aligned} \tag{5.8}$$

2. Check for collinearity. Compute $(\mathbf{a} - \mathbf{c}) \times (\mathbf{b} - \mathbf{c})$, and check if this vector is equal to the null vector. If the points are collinear, then choose another point and try again.

3. If all of them are identical to zero, then the motion is a pure translation which is equal to any of the displacement vectors.

4. If two of these differences are identical to zero and if these are δ_a and δ_b, then define a new motion arising out of subtracting δ_c from the other two vectors. This will mean that there will now be two non-vanishing difference vectors and one can use the following case.

5. If one of them is identical to zero[1], then check if the two remaining are parallel by checking if the cross product $\delta_b \times \delta_c$ is the null vector. If the two difference vectors are non-parallel, then $\hat{\mathbf{n}}$ can be computed as their cross product. If they are parallel, then one of them can be expressed as a constant multiple of the other.

$$\delta_b = \beta \delta_c$$

The vector $\hat{\mathbf{n}}$ can then be computed as the normalized form of the vector given below:

$$\mathbf{b} - \mathbf{c} - \beta(\mathbf{a} - \mathbf{c})$$

6. If none of the differences are identical to zero, then we compute $\delta_c \cdot (\delta_a \times \delta_b)$ to see if they are co-planar. If they are not coplanar then the vector $\hat{\mathbf{n}}$ can be computed as the cross produce of two of the displacement vectors as before. If they are co-planar, then we compute if the differences are parallel, by computing $(\delta_a - \delta_c) \times (\delta_b - \delta_c)$. If they are parallel, then the motion is once again a pure translation. If they are not parallel, then $\hat{\mathbf{n}}$ can be computed as the cross product of these two difference vectors as before.

[1] Without loss of generality, we can assume that δ_a is the vector that vanishes.

7. Compute the angle of rotation and the linear displacement if they haven't still been computed. This is done using

$$\begin{aligned} cos(\theta) &= \frac{ac \cdot a'c}{|a'a|^2} \\ \delta &= \delta_a \cdot \hat{\mathbf{n}} \end{aligned} \tag{5.9}$$

where a is the initial position of one of the points, a' is its final position after the move has been made and c is a point on the screw axis.

The complexity of the above algorithm is higher than the previous one mentioned in the worst case. As can be seen, depending on the nature of the motion involved, the computations can range between **6M+19A** to **31M+28A**. The average case complexity will of course be much better in this algorithm, and we prefer it since it recognizes all the special cases involved.

The next step in the algorithm calculates the force to be exerted on the object based on its displacement. The usual form of a *generalized stiffness formulation is used.* The stiffness matrix $\mathbf{K_p}$ is specified in cartesian space as mentioned before. The force on the object is calculated using:

$$\mathbf{w}_o = \mathbf{K}_p \cdot \Delta\mathbf{x} + \mathbf{K}_v \cdot \dot{\Delta\mathbf{x}} + \mathbf{w}_b \tag{5.10}$$

where $\mathbf{K}_p$ represents the stiffness matrix, $\mathbf{K}_v$ represents the damping matrix and $\mathbf{w}_b$ is a bias wrench. Usually, both the 6×6 matrices $\mathbf{K}_p$ and $\mathbf{K}_v$ are diagonal. Such matrices can be specified by a programmer or synthesized from the nature of a task. This computation, assuming diagonal matrices, takes **12M+12A** operations.

5.5 Computing the Fingertip Forces

The next step is perhaps the most complicated one in the algorithm. Given the wrench to be exerted on the object, our objective is to compute the wrenches that need to be exerted by the fingertips in order that the desired object wrench be realized. There are many ways of performing this computation, the most efficient of which is to use a method that relies on representing the forces as vectors and uses the definition of internal forces as given by Salisbury [1982].

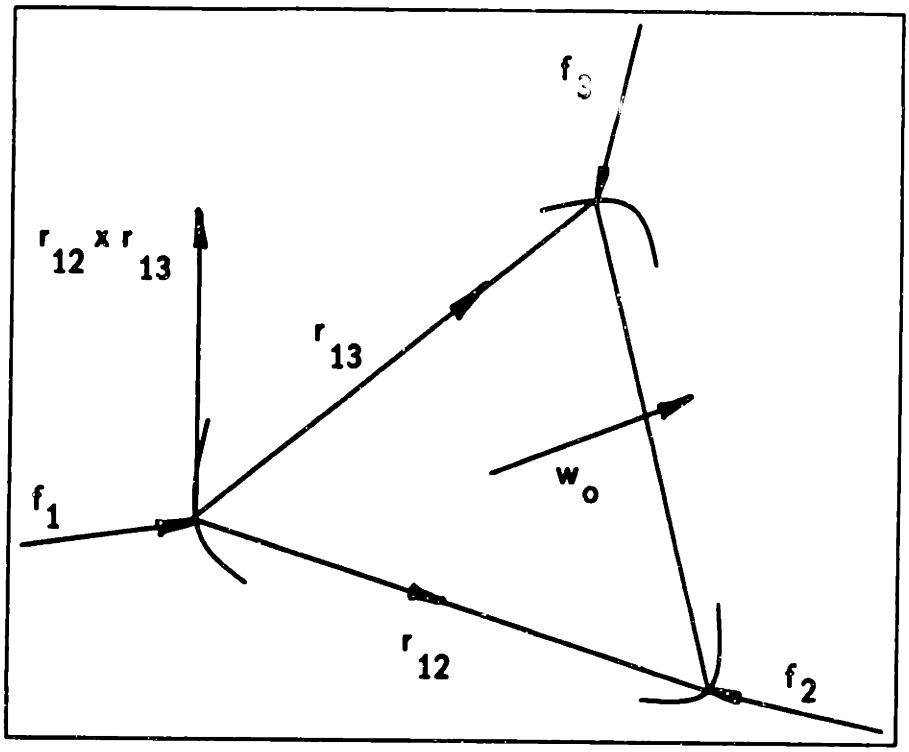


Figure 5.1: Three point contact with friction.

5.5.1 Three Point Contact with Friction

The single observation that helps in speeding up this part of the computation considerably is that one can take advantage of the inherent geometry involved in the situation. This can be done by solving not for the components of the tip forces themselves but for the projections of these forces along the interfinger position vectors, along which the internal forces have been defined to act (see Figure 5.1).

The internal gripping forces are specified as:

$$\begin{aligned} (\mathbf{f}_1 - \mathbf{f}_2)\cdot\mathbf{r}_{12} &= f_{12} \\ (\mathbf{f}_1 - \mathbf{f}_3)\cdot\mathbf{r}_{13} &= f_{13} \\ (\mathbf{f}_2 - \mathbf{f}_3)\cdot(\mathbf{r}_{12} - \mathbf{r}_{13}) &= f_{23} \end{aligned} \tag{5.11}$$

where these quantities have been defined previously. The force and torque balance equations are:

$$\begin{aligned} \mathbf{f}_e &= \mathbf{f}_1 + \mathbf{f}_2 + \mathbf{f}_3 \\ \mathbf{n}_e &= \mathbf{r}_{12} \times \mathbf{f}_2 + \mathbf{r}_{13} \times \mathbf{f}_3 \end{aligned} \tag{5.12}$$

The torque $\mathbf{n}_e$ exerted on the object is expressed about the contact point made at the first finger. This can be derived from the torque $\mathbf{n}_o$ specified with respect to some other reference point O on the object by:

$$\mathbf{n}_e = \mathbf{n}_o + \mathbf{r}_o \times \mathbf{f}_e \tag{5.13}$$

where $\mathbf{r}_o$ is the vector connecting the first finger contact point to the reference point O.

The force balance equation can be projected along the non-orthogonal co-ordinate system formed by the vectors $\mathbf{r}_{12}$, $\mathbf{r}_{13}$, and $\mathbf{r}_{12} \times \mathbf{r}_{13}$. This gives:

$$\begin{aligned} \mathbf{f}_e \cdot \mathbf{r}_{12} &= \mathbf{f}_1 \cdot \mathbf{r}_{12} + \mathbf{f}_2 \cdot \mathbf{r}_{12} + \mathbf{f}_3 \cdot \mathbf{r}_{12} \\ \mathbf{f}_e \cdot \mathbf{r}_{13} &= \mathbf{f}_1 \cdot \mathbf{r}_{13} + \mathbf{f}_2 \cdot \mathbf{r}_{13} + \mathbf{f}_3 \cdot \mathbf{r}_{13} \\ \mathbf{f}_e \cdot (\mathbf{r}_{12} \times \mathbf{r}_{13}) &= \mathbf{f}_1 \cdot (\mathbf{r}_{12} \times \mathbf{r}_{13}) + \mathbf{f}_2 \cdot (\mathbf{r}_{12} \times \mathbf{r}_{13}) + \mathbf{f}_3 \cdot (\mathbf{r}_{12} \times \mathbf{r}_{13}) \end{aligned} \tag{5.14}$$

Similarly for the torques,

$$\begin{aligned} \mathbf{n}_e \cdot \mathbf{r}_{12} &= (\mathbf{r}_{12} \times \mathbf{f}_2) \cdot \mathbf{r}_{12} + (\mathbf{r}_{13} \times \mathbf{f}_3) \cdot \mathbf{r}_{12} \\ \mathbf{n}_e \cdot \mathbf{r}_{13} &= (\mathbf{r}_{12} \times \mathbf{f}_2) \cdot \mathbf{r}_{13} + (\mathbf{r}_{13} \times \mathbf{f}_3) \cdot \mathbf{r}_{13} \\ \mathbf{n}_e \cdot (\mathbf{r}_{12} \times \mathbf{r}_{13}) &= (\mathbf{r}_{12} \times \mathbf{f}_2) \cdot (\mathbf{r}_{12} \times \mathbf{r}_{13}) + (\mathbf{r}_{13} \times \mathbf{f}_3) \cdot (\mathbf{r}_{12} \times \mathbf{r}_{13}) \end{aligned} \tag{5.15}$$

which is more conveniently rewritten as

$$\begin{aligned} \mathbf{n}_e \cdot \mathbf{r}_{12} &= \mathbf{f}_3 \cdot (\mathbf{r}_{12} \times \mathbf{r}_{13}) \\ \mathbf{n}_e \cdot \mathbf{r}_{13} &= -\mathbf{f}_2 \cdot (\mathbf{r}_{12} \times \mathbf{r}_{13}) \\ \mathbf{n}_e \cdot (\mathbf{r}_{12} \times \mathbf{r}_{13}) &= (\mathbf{f}_2 \cdot \mathbf{r}_{13})(\mathbf{r}_{12}^2) - \mathbf{f}_2 \cdot \mathbf{r}_{12}(\mathbf{r}_{12} \cdot \mathbf{r}_{13}) + \mathbf{f}_3 \cdot \mathbf{r}_{13}(\mathbf{r}_{12} \cdot \mathbf{r}_{13}) \\ &\quad -\mathbf{f}_3 \cdot \mathbf{r}_{12}(\mathbf{r}_{13}^2) \end{aligned} \tag{5.16}$$

With the above equations the solutions for the fingertip forces can be found first in the non-orthogonal co-ordinate system given by the vectors $\mathbf{r}_{12}$, $\mathbf{r}_{13}$, and $\mathbf{r}_{12} \times \mathbf{r}_{13}$ as follows:

$$\mathbf{f}_3 \cdot \mathbf{r}_{13} = \frac{(\mathbf{r}_{12} \times \mathbf{r}_{13}) \cdot \mathbf{n}_e + \mathbf{r}_{12}^2(f_{23} - \mathbf{f}_e \cdot \mathbf{r}_{13}) + \mathbf{r}_{13}^2(\mathbf{f}_e \cdot \mathbf{r}_{12} - f_{12}) - \mathcal{K}(2\,\mathbf{r}_{13}^2 - \mathbf{r}_{12} \cdot \mathbf{r}_{13})}{2\,(\mathbf{r}_{12} \cdot \mathbf{r}_{13} - \mathbf{r}_{12}^2 - \mathbf{r}_{13}^2)} \tag{5.17}$$

where $\mathcal{K} = \frac{1}{3}(f_{13} - f_{12} - f_{23} + \mathbf{f}_e \cdot \mathbf{r}_{12} + \mathbf{f}_e \cdot \mathbf{r}_{13})$. Using the above, and solving for the projection of $\mathbf{f}_2$ along $\mathbf{r}_{12}$ and $\mathbf{r}_{13}$:

$$\begin{aligned} \mathbf{f}_2 \cdot \mathbf{r}_{12} &= \mathcal{K} - \mathbf{f}_3 \cdot \mathbf{r}_{13} \\ \mathbf{f}_2 \cdot \mathbf{r}_{13} &= \mathbf{f}_e \cdot \mathbf{r}_{13} - f_{23} - 2\,\mathbf{f}_3 \cdot \mathbf{r}_{13} \\ \mathbf{f}_3 \cdot \mathbf{r}_{12} &= \mathbf{f}_e \cdot \mathbf{r}_{12} - f_{12} - 2\,\mathbf{f}_2 \cdot \mathbf{r}_{12} \\ \mathbf{f}_2 \cdot \mathbf{r}_{13} &= \mathbf{f}_e \cdot \mathbf{r}_{13} - f_{23} - 2\,\mathbf{f}_3 \cdot \mathbf{r}_{13} \end{aligned} \tag{5.18}$$

One can now finally solve for the projections of $\mathbf{f}_1$ along the two vectors $\mathbf{r}_{12}$ and $\mathbf{r}_{13}$:

$$\begin{aligned} \mathbf{f}_1 \cdot \mathbf{r}_{12} &= f_{12} + \mathbf{f}_2 \cdot \mathbf{r}_{12} \\ \mathbf{f}_1 \cdot \mathbf{r}_{13} &= f_{23} + \mathbf{f}_3 \cdot \mathbf{r}_{13} \end{aligned} \tag{5.19}$$

From the above equations it is apparent that computing the components of the fingertip forces along $\mathbf{r}_{12}$, $\mathbf{r}_{13}$, and $\mathbf{r}_{12} \times \mathbf{r}_{13}$ can be done extremely efficiently. Once these components have been computed it may be necessary to orthogonalize them. To convert these components into orthogonal force components, Gaussian elimination of the following matrix equation can be used:

$$\begin{bmatrix} \mathbf{r}_{12}^T \\ \mathbf{r}_{13}^T \\ (\mathbf{r}_{12} \times \mathbf{r}_{13})^T \end{bmatrix} \mathbf{f}_i = \mathbf{b}_i \tag{5.20}$$

where the constants $\mathbf{b}_i$ are the actual finger force components in the non-orthogonal co-ordinate system just computed.

A summary of the computational complexity of the method is presented in Table 5.2. Performing the transformation to the non-orthogonal co-ordinate system in the three point contact case can be seen to reduce the computational burden by almost a factor of four.[2]

5.5.2 Four Point Contact with Friction

The four point contact case with friction is an interesting case, since it corresponds more closely to the case of the Utah-MIT hand. In this case one has to compute three orthogonal force components for four fingertip contact forces. The specified external force

[2]It can be shown that to perform simple Gaussian Elimination on an $n \times n$ matrix it takes $\frac{n(n+1)(2n+1)}{6}$ multiplies, n divides and $\frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n$ additions as in Hovannessian et al.[1969]. In our analysis, divisions have been assumed to take the same amount of time as multiplies.

Computation	Multiplies	Additions
Coefficients needed to compute $\mathbf{f}_3 \cdot \mathbf{r}_{13}$	30	25
Recovering the other components	12	10
Orthogonalization	36	30
Conversion of n_o to n_e	6	6
Grand Total	84	71
Direct Gaussian-Elimination	295	183

Table 5.2: Summary of computational requirements: Three point contact.

has six components and therefore an additional six internal grasping force components need to be specified.

Using the same definition of internal force as before, we can see that it is always unambiguously possible to get six components taking the inter-finger forces pairwise, providing that all the four grasp points are non-coplanar.

The force and torque balance equations for four-point contact are:

$$\begin{aligned} \mathbf{f}_e &= \textstyle\sum_{i=1}^{4} \mathbf{f}_i \\ \mathbf{n}_e &= \mathbf{r}_{12} \times \mathbf{f}_2 + \mathbf{r}_{13} \times \mathbf{f}_3 + \mathbf{r}_{14} \times \mathbf{f}_4 \end{aligned} \tag{5.21}$$

The vectors $\mathbf{r}_{12}$, $\mathbf{r}_{13}$, and $\mathbf{r}_{14}$ form a non-orthogonal co-ordinate system. These three vectors provide a natural system in which to solve for the force components (see Figure 5.2). The internal force constraints written in terms of these vectors are:

$$\begin{aligned} (\mathbf{f}_1 - \mathbf{f}_2) \cdot \mathbf{r}_{12} &= f_{12}. \\ (\mathbf{f}_1 - \mathbf{f}_3) \cdot \mathbf{r}_{13} &= f_{13}. \\ (\mathbf{f}_1 - \mathbf{f}_4) \cdot \mathbf{r}_{14} &= f_{14}. \\ (\mathbf{f}_2 - \mathbf{f}_3) \cdot (\mathbf{r}_{12} - \mathbf{r}_{13}) &= f_{23}. \\ (\mathbf{f}_4 - \mathbf{f}_2) \cdot (\mathbf{r}_{14} - \mathbf{r}_{12}) &= f_{42}. \\ (\mathbf{f}_4 - \mathbf{f}_3) \cdot (\mathbf{r}_{14} - \mathbf{r}_{13}) &= f_{43}. \end{aligned} \tag{5.22}$$

Since $\mathbf{f}_1$ does not appear in the torque balance equation because the torques were referenced to the first finger contact point, we begin the process of eliminating variables by removing $\mathbf{f}_1$ from the first three equations in (5.22) through substitution from (5.21). Also, expanding the last three equations in (5.22) yields:

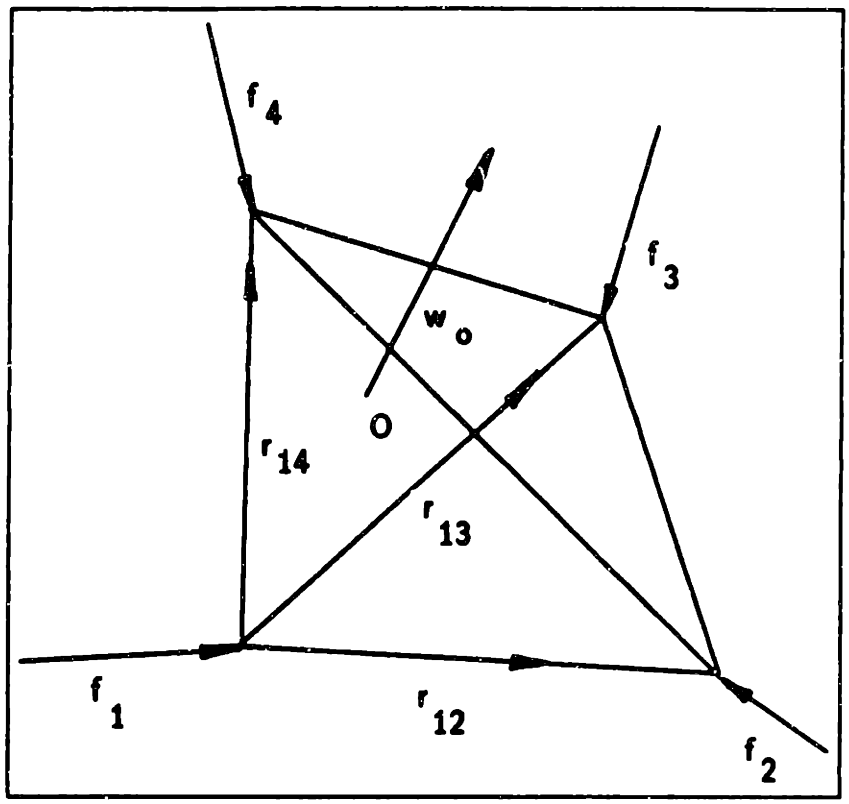


Figure 5.2: Four point contact with friction.

$$
\begin{aligned}
f_{12} + 2\,\mathbf{f}_2 \cdot \mathbf{r}_{12} + \mathbf{f}_3 \cdot \mathbf{r}_{12} + \mathbf{f}_4 \cdot \mathbf{r}_{12} &= \mathbf{f}_e \cdot \mathbf{r}_{12} \quad \text{(a)} \\
f_{13} + \mathbf{f}_2 \cdot \mathbf{r}_{13} + 2\,\mathbf{f}_3 \cdot \mathbf{r}_{13} + \mathbf{f}_4 \cdot \mathbf{r}_{13} &= \mathbf{f}_e \cdot \mathbf{r}_{13} \quad \text{(b)} \\
f_{14} + \mathbf{f}_2 \cdot \mathbf{r}_{14} + \mathbf{f}_3 \cdot \mathbf{r}_{14} + 2\,\mathbf{f}_4 \cdot \mathbf{r}_{14} &= \mathbf{f}_e \cdot \mathbf{r}_{14} \quad \text{(c)} \\
\mathbf{f}_2 \cdot \mathbf{r}_{12} - \mathbf{f}_2 \cdot \mathbf{r}_{13} - \mathbf{f}_3 \cdot \mathbf{r}_{12} + \mathbf{f}_3 \cdot \mathbf{r}_{13} &= f_{23} \quad \text{(d)} \\
\mathbf{f}_4 \cdot \mathbf{r}_{14} - \mathbf{f}_4 \cdot \mathbf{r}_{12} - \mathbf{f}_2 \cdot \mathbf{r}_{14} + \mathbf{f}_2 \cdot \mathbf{r}_{12} &= f_{42} \quad \text{(e)} \\
\mathbf{f}_4 \cdot \mathbf{r}_{14} - \mathbf{f}_4 \cdot \mathbf{r}_{13} - \mathbf{f}_3 \cdot \mathbf{r}_{14} + \mathbf{f}_3 \cdot \mathbf{r}_{13} &= f_{43} \quad \text{(f)}
\end{aligned}
\tag{5.23}
$$

Adding the above set of equations and rearranging:

$$
\mathbf{f}_2 \cdot \mathbf{r}_{12} + \mathbf{f}_3 \cdot \mathbf{r}_{13} + \mathbf{f}_4 \cdot \mathbf{r}_{14} = \frac{1}{4}\left(f_{23} + f_{42} + f_{43} - f_{12} - f_{13} - f_{14} + \mathbf{f}_e \cdot (\mathbf{r}_{12} + \mathbf{r}_{13} + \mathbf{r}_{14})\right) \tag{5.24}
$$

To reduce the vector equation for the torque-balance into its scalar components, it is projected along the vectors $\mathbf{r}_{12} \times \mathbf{r}_{13}$, $\mathbf{r}_{13} \times \mathbf{r}_{14}$, and $\mathbf{r}_{13} \times \mathbf{r}_{14}$. This yields the following

three equations after the appropriate expansions have been performed:

$$\begin{aligned}
(\mathbf{r}_{12}\times\mathbf{r}_{13})\cdot\mathbf{n}_e &= (\mathbf{f}_2\cdot\mathbf{r}_{13})(\mathbf{r}_{12}^2)-(\mathbf{f}_2\cdot\mathbf{r}_{12})(\mathbf{r}_{12}\cdot\mathbf{r}_{13})+(\mathbf{f}_3\cdot\mathbf{r}_{13})(\mathbf{r}_{12}\cdot\mathbf{r}_{13})\\
&\quad -(\mathbf{f}_3\cdot\mathbf{r}_{12})(\mathbf{r}_{13}^2)+(\mathbf{f}_4\cdot\mathbf{r}_{13})(\mathbf{r}_{12}\cdot\mathbf{r}_{14})-(\mathbf{f}_4\cdot\mathbf{r}_{12})(\mathbf{r}_{13}\cdot\mathbf{r}_{14})\\
(\mathbf{r}_{12}\times\mathbf{r}_{14})\cdot\mathbf{n}_e &= (\mathbf{f}_2\cdot\mathbf{r}_{14})(\mathbf{r}_{12}^2)-(\mathbf{f}_2\cdot\mathbf{r}_{12})(\mathbf{r}_{12}\cdot\mathbf{r}_{14})+(\mathbf{f}_3\cdot\mathbf{r}_{14})(\mathbf{r}_{12}\cdot\mathbf{r}_{13})\\
&\quad -(\mathbf{f}_3\cdot\mathbf{r}_{12})(\mathbf{r}_{13}\cdot\mathbf{r}_{14})+(\mathbf{f}_4\cdot\mathbf{r}_{14})(\mathbf{r}_{12}\cdot\mathbf{r}_{14})-(\mathbf{f}_4\cdot\mathbf{r}_{12})(\mathbf{r}_{14}^2)\\
(\mathbf{r}_{13}\times\mathbf{r}_{14})\cdot\mathbf{n}_e &= (\mathbf{f}_2\cdot\mathbf{r}_{14})(\mathbf{r}_{12}\cdot\mathbf{r}_{13})-(\mathbf{f}_2\cdot\mathbf{r}_{13})(\mathbf{r}_{12}\cdot\mathbf{r}_{14})+(\mathbf{f}_3\cdot\mathbf{r}_{14})(\mathbf{r}_{13}^2)\\
&\quad -(\mathbf{f}_3\cdot\mathbf{r}_{13})(\mathbf{r}_{13}\cdot\mathbf{r}_{14})+(\mathbf{f}_4\cdot\mathbf{r}_{14})(\mathbf{r}_{13}\cdot\mathbf{r}_{14})-(\mathbf{f}_4\cdot\mathbf{r}_{13})(\mathbf{r}_{14}^2)
\end{aligned} \tag{5.25}$$

Since $\mathbf{f}_1$ has been eliminated, nine variables remain. The solution proceeds by reducing (5.25) to four variables, $\mathbf{f}_2\cdot\mathbf{r}_{12}$, $\mathbf{f}_3\cdot\mathbf{r}_{13}$, $\mathbf{f}_4\cdot\mathbf{r}_{14}$ and $\mathbf{f}_2\cdot\mathbf{r}_{13}$. With (5.24) the original system of equations is reduced to a 4×4 matrix equation solvable by Gaussian elimination. The remaining components are then recovered, and finally the components of the fingertip forces along the inter-finger vectors are orthogonalized as before.

By simple manipulations on (5.23), the remaining five variables are expressed in terms of these four variables:

$$\begin{aligned}
\mathbf{f}_4\cdot\mathbf{r}_{13} &= -\mathbf{f}_2\cdot\mathbf{r}_{13}-2\,\mathbf{f}_3\cdot\mathbf{r}_{13}-f_{13}+\mathbf{f}_e\cdot\mathbf{r}_{13}\\
\mathbf{f}_4\cdot\mathbf{r}_{12} &= -3\,\mathbf{f}_2\cdot\mathbf{r}_{12}+\mathbf{f}_2\cdot\mathbf{r}_{13}-\mathbf{f}_3\cdot\mathbf{r}_{13}+f_{23}-f_{12}+\mathbf{f}_e\cdot\mathbf{r}_{12}\\
\mathbf{f}_3\cdot\mathbf{r}_{14} &= \mathbf{f}_2\cdot\mathbf{r}_{13}+3\,\mathbf{f}_3\cdot\mathbf{r}_{13}+\mathbf{f}_4\cdot\mathbf{r}_{14}+f_{13}-f_{43}-\mathbf{f}_e\cdot\mathbf{r}_{13}\\
\mathbf{f}_3\cdot\mathbf{r}_{12} &= \mathbf{f}_2\cdot\mathbf{r}_{12}-\mathbf{f}_2\cdot\mathbf{r}_{13}+\mathbf{f}_3\cdot\mathbf{r}_{13}-f_{23}\\
\mathbf{f}_2\cdot\mathbf{r}_{14} &= -\mathbf{f}_2\cdot\mathbf{r}_{13}-3\,\mathbf{f}_3\cdot\mathbf{r}_{13}-3\,\mathbf{f}_4\cdot\mathbf{r}_{14}+f_{13}-f_{14}+f_{43}+\mathbf{f}_e\cdot(\mathbf{r}_{13}+\mathbf{r}_{14})
\end{aligned} \tag{5.26}$$

Substituting into (5.25),

$$\begin{aligned}
&(\mathbf{r}_{12}^2+\mathbf{r}_{13}^2-\mathbf{r}_{12}\cdot\mathbf{r}_{14}-\mathbf{r}_{13}\cdot\mathbf{r}_{14})(\mathbf{f}_2\cdot\mathbf{r}_{13})+(3\mathbf{r}_{13}\cdot\mathbf{r}_{14}-\mathbf{r}_{13}^2-\mathbf{r}_{12}\cdot\mathbf{r}_{13})(\mathbf{f}_2\cdot\mathbf{r}_{12})+(\mathbf{r}_{12}\cdot\mathbf{r}_{13}\\
&-\mathbf{r}_{13}^2-2\mathbf{r}_{12}\cdot\mathbf{r}_{14}+\mathbf{r}_{13}\cdot\mathbf{r}_{14})(\mathbf{f}_3\cdot\mathbf{r}_{13})=\mathbf{n}_e\cdot(\mathbf{r}_{12}\times\mathbf{r}_{13})+(\mathbf{r}_{13}\cdot\mathbf{r}_{14})(\mathbf{f}_e\cdot\mathbf{r}_{12}+f_{23}-f_{12})\\
&+(\mathbf{r}_{13}\cdot\mathbf{r}_{14})(f_{13}-\mathbf{f}_e\cdot\mathbf{r}_{13})-f_{23}(\mathbf{r}_{13}^2)
\end{aligned} \tag{5.27}$$

Computation	Multiplies	Additions
Coefficients of the 4×4 matrix	65	91
Gaussian Elimination of the matrix	40	32
Recovering other components	4	28
Orthogonalization	48	40
Conversion of n_o to n_e	6	6
Grand Total	163	197
Direct Gaussian-Elimination	662	446

Table 5.3: Summary of computational requirements: Four point contact.

$$(\mathbf{r}_{12}\cdot\mathbf{r}_{13}+\mathbf{r}_{13}\cdot\mathbf{r}_{14}-\mathbf{r}_{12}^2-\mathbf{r}_{14}^2)(\mathbf{f}_2\cdot\mathbf{r}_{13})+(3\mathbf{r}_{14}^2-\mathbf{r}_{12}\cdot\mathbf{r}_{14}-\mathbf{r}_{13}\cdot\mathbf{r}_{14})(\mathbf{f}_2\cdot\mathbf{r}_{12})$$
$$+(\mathbf{r}_{14}^2-3\mathbf{r}_{12}^2+3\mathbf{r}_{12}\cdot\mathbf{r}_{13}-\mathbf{r}_{13}\cdot\mathbf{r}_{14})(\mathbf{f}_3\cdot\mathbf{r}_{13})+(\mathbf{r}_{12}\cdot\mathbf{r}_{13}+\mathbf{r}_{12}\cdot\mathbf{r}_{14}-3\mathbf{r}_{12}^2)(\mathbf{f}_4\cdot\mathbf{r}_{14})$$
$$=\mathbf{n}_e\cdot(\mathbf{r}_{12}\times\mathbf{r}_{14})+(\mathbf{f}_e\cdot\mathbf{r}_{13})(\mathbf{r}_{12}\cdot\mathbf{r}_{13}-\mathbf{r}_{12}^2)+\mathbf{r}_{12}^2(f_{14}-f_{43}+f_{13}-\mathbf{f}_e\cdot\mathbf{r}_{14})-f_{23}(\mathbf{r}_{13}\cdot\mathbf{r}_{14})$$
$$+\mathbf{r}_{14}^2(f_{23}-f_{12}+\mathbf{f}_e\cdot\mathbf{r}_{12}) \tag{5.28}$$

$$(\mathbf{r}_{14}^2+\mathbf{r}_{13}^2-\mathbf{r}_{12}\cdot\mathbf{r}_{14}-\mathbf{r}_{12}\cdot\mathbf{r}_{13})(\mathbf{f}_2\cdot\mathbf{r}_{13})+(3\mathbf{r}_{13}^2+2\,\mathbf{r}_{14}^2-3\mathbf{r}_{12}\cdot\mathbf{r}_{13}-\mathbf{r}_{13}\cdot\mathbf{r}_{14})(\mathbf{f}_3\cdot\mathbf{r}_{13})$$
$$+(\mathbf{r}_{13}^2+\mathbf{r}_{13}\cdot\mathbf{r}_{14}-3\mathbf{r}_{12}\cdot\mathbf{r}_{13})(\mathbf{f}_4\cdot\mathbf{r}_{14})=\mathbf{n}_e\cdot(\mathbf{r}_{13}\times\mathbf{r}_{14})+(\mathbf{f}_e\cdot\mathbf{r}_{13})(\mathbf{r}_{13}^2+\mathbf{r}_{14}^2-\mathbf{r}_{12}\cdot\mathbf{r}_{13})$$
$$-(\mathbf{r}_{12}\cdot\mathbf{r}_{13})(\mathbf{f}_e\cdot\mathbf{r}_{14}-f_{14}+f_{43}-f_{13})-\mathbf{r}_{13}^2(f_{23}-f_{43})-f_{13}(\mathbf{r}_{14}^2) \tag{5.29}$$

Equations (5.27)-(5.29) and (5.24) form a 4×4 matrix equation, that can be solved by Gaussian elimination. Once this has been done, the remaining components can be recovered using (5.26). This procedure will result in finally solving for the fingertip forces along the three vectors formed by $\mathbf{r}_{12}$, $\mathbf{r}_{13}$, and $\mathbf{r}_{14}$. These components can be orthogonalized in a manner similar to that outlined for the three point contact case.

The computational requirements for the four point contact case are summarized in Table 5.3. Similar to the three point case, a factor of four improvement in efficiency over Gaussian elimination is obtained.

5.6 Computing the Joint Torques

As we saw on the section on position control, the lower level controller is designed to take as input either joint positions or joint torques. In this chapter, we have shown how one can compute the forces to be exerted at the fingertips based on a desired force to be exerted on a grasped object, and a desired stiffness matrix. The actual torques to be exerted at the joints can be computed from these values using the relation $\tau_i = \mathbf{J_i}^T \mathbf{f_i}$.

In the case of the Utah-MIT hand, however, one can use an important kinematic feature of each finger to reduce the computations needed for this part of the algorithm. The last three joints of each finger of this hand form a 3R planar pair; also, the first axis of rotation is perpendicular to the axes of the planar pair.

For point contact with friction, only the force component $\mathbf{f}_e$ of the contact wrench $\mathbf{w}_e$ acts on the finger joints. To take advantage of the 3R planar pair formed by the last three joints, $\mathbf{f}_e$ is first rotated by θ_1 using the following **A** matrix[3]:

$$\mathbf{A}_1 = \begin{bmatrix} C_1 & 0 & S_1 \\ S_1 & 0 & -C_1 \\ 0 & 1 & 0 \end{bmatrix} \tag{5.30}$$

This yields:

$${}^1\mathbf{f}_e = \begin{bmatrix} f_{ex}C_1 - f_{ey}S_1 \\ f_{ez} \\ f_{ex}S_1 - f_{ey}C_1 \end{bmatrix} \tag{5.31}$$

The contact point made at the fingertip is located by the vector $\mathbf{l}_4$, as measured from the co-ordinate system affixed to the last finger joint. This quantity can be sensed using a tactile sensor (Siegel [1986]) and expressed with respect to a co-ordinate system affixed to the first joint of the finger using the following equation:

$${}^1\mathbf{l}_4 = \begin{bmatrix} {}^4l_{4x}C_{234} - {}^4l_{4y}S_{234} \\ {}^4l_{4x}S_{234} - {}^4l_{4y}C_{234} \\ {}^4l_{4x} \end{bmatrix} \tag{5.32}$$

Expressing $\mathbf{w}_e$ about the most distal joint p:

[3]Notation: a) C_{xyz} stands for $cos(\theta_x + \theta_y + \theta_z)$. b) The left superscript before any quantity indicates the frame of reference in which that quantity has been expressed. c) Joint numberings are 1 to 4 instead of 0 to 3 as indicated in the kinematics calculations.

Computation	Multiplies	Additions
Forward Kinematics	94	63
Computing object displacement	40	34
Control Equation	12	12
Fingertip force computation	167	210
Joint torque computation	23	14
Total	336	333

Table 5.4: Summary of computations for the force control algorithm.

$$\begin{aligned} {}^1\mathbf{f}_p &= {}^1\mathbf{f}_c \\ {}^1\mathbf{n}_p &= {}^1\mathbf{l}_4 \times {}^1\mathbf{f}_c \end{aligned} \tag{5.33}$$

It is now fairly straightforward to solve for the joint torques. Denoting τ_i to be the torque to be exerted at joint i, and a_i as the various link lengths, one gets:

$$\begin{aligned} \tau_4 &= {}^1n_{pz} \\ \tau_3 &= {}^1n_{px} + a_3\left({}^1f_{py}C_{23} - {}^1f_{px}S_{23}\right) \\ \tau_2 &= \tau_3 + a_2\left({}^1f_{py}C_2 - {}^1f_{px}S_2\right) \\ \tau_1 &= {}^1n_{py} - {}^1f_{px}\left(a_1 + a_2C_2 + a_3C_{23}\right) \end{aligned} \tag{5.34}$$

These equations can be evaluated quickly with only 23 multiplications and 14 additions. Other methods that rely on the transpose of the Jacobian matrix are computationally not as efficient.

The total for the entire algorithm is therefore summarized in Table 5.4.

5.7 Implementation

The implementation of this force control algorithms has been completed on the architecture to be described in the next chapter. We have just begun performing various experiments with this implementation and hope to continue these experiments in the near future. Force controllers present in the literature often stop at the point when the torques have been computed. It is assumed to be the job of the underlying real time controller to ensure that these torques are accurately achieved.

In reality, this may be true of robots driven by electric motors where the current applied to the motors is a very good approximation of the torques applied by the motors at the robot's joints. Even with such robots, Paul [1987] indicates that simply closing a torque loop may not be the best one can do. In the case of robots like the Utah-MIT hand the voltages applied to the actuators affect the tendon tensions in a very complicated fashion. The hysteresis present in the magnetic field that causes the deflection of the primary jet in the pneumatic valves causes *stiction* which is very hard to model. The *coulombic* friction present in the transmission as the tendons pass over the pulleys is another source of problems although it is not quite so severe.

Another practical issue that has to be dealt with, since the system is dual-acting, is the effect of the controller on one of the tendons (on the flexor say), which shows up as a load on the other (on the extensor). Modelling the effect of the tendon dynamics present in such a system would enable much higher performance to be attained.

5.8 Future Work

A lot of work remains to be done with respect to the force controller implementation. It is imperative that a good understanding of the underlying actuator and transmission system be attained first. Without such a model, it would be very hard to ensure the accurate exertion of torques, which is what the algorithm presented in this chapter ultimately relies on. As indicated by our initial grasping experiments, it is clear that the kinematic structure of the hand interacts with a grasped object in a number of interesting ways. An understanding of how the passive compliance in such mechanisms as the Utah-MIT hand could be used to sense contact, grasp objects more stably and manipulate them still seems like a long term goal to achieve, but we hope that these experiments will indicate the steps to take in the future.

Experimentation and validation of theoretical results is important in any applied science. Consequently, research publications in robotics often contain sections devoted to implementation. The value of such experiments cannot be overemphasized.

For performing such research coupled with experimentation, powerful tools are necessary. Robots like the Utah-MIT hand that push the state of the art in terms of mechanisms, actuators and transmission systems allow investigations into new levels of performance that previous generation robots could not hope to achieve. These robots are characterized by a number of joints and consequently demand powerful computer architectures to be controlled and utilized effectively.

An unusually large portion of time spent in working on this thesis was actually spent on the implementation and testing of the ideas presented herein. As we saw earlier, controlling a complex device like the Utah-MIT dexterous hand in real-time is an extremely compute intensive task. One of the first problems we addressed therefore, (and one that took the longest to find a good solution to) was the development of a suitable computational architecture which would be adequate for the task. As a tool for doing useful research in robotics, we feel that the computational architecture described in the following sections will be adequate for some time to come.

The term "computational architecture" is used to denote both the hardware architecture and the software systems that have been developed as a solution to the real time control problem in a research environment. The first version of the solution was implemented on Motorola 68000 single board computers running on the Multibus-I. The second and present version is based on Motorola 68020 boards running on the VME-Bus. The entire real time development system has come to be known as the CONDOR programming environment.

We feel that the amount of time spent on implementing the software and hardware systems has been worthwhile. One of the glaring deficiencies in the state of current research in robotics control is the diversity of the primitive control architectures that are used. The diversity makes it harder for researchers to share implementations, to build on previous work and to some extent even hampers the reproducibility of experimental results. The primitive nature of the hardware on the other hand, restricts the complexity of the devices

that can be controlled by such systems.

We must mention that our effort in developing the computational architecture has been more successful than we had anticipated. The system has been used to control other robotic devices beside the Utah-MIT dexterous hand, notably the MIT Serial Link Direct Drive Arm. A number of similar systems are in the process of being built inside MIT's Artificial Intelligence Laboratory to control various real-time devices and other research institutions that have the Utah-MIT hand.

6.1 Design Motivation

Controlling the hand, or any other high-performance real-time robotic device, is essentially a compute bound task. If one uses a single processor to control all the joints and process all the sensory information, one would need an extremely fast (and consequently, a very expensive) cpu to perform the lowest levels of control.

Two desirable goals for any real-time controller to be used as an experimental tool are flexibility and efficiency. The first of these ensures that adequate mechanisms will be provided by the system to perform the necessary experiments, while the second ensures that the needed performance in terms of computational horsepower to control these complex experimental devices will also be provided. Computational architectures found in industry often provide performance at the expense of flexibility, while university efforts have resulted in architectures that sometimes do not provide adequate real-time response.

To illustrate this problem, consider the fact that the Utah-MIT hand has four fingers each with four-degrees of freedom. If the sixteen joints need to be servo'ed at a rate of one thousand hertz, then using a one MIP[1] processor like a VAX 11/780 would leave only 63 instructions per joint per servo in which all the computations have to be performed. The cost of a uniprocessor whose performance is adequate could be as high as one million dollars. Clearly, less expensive solutions are desirable.

The first observation that makes this problem easier is that

- At the level of actuators and joints, robot control exhibits coarse-grain parallelism.

By the above statement we mean that the parallelism inherent in most robotics control algorithms (see Hollerbach [1980], Raibert and Craig [1981] for typical examples) is not at

[1]One MIP is one million instructions per second – a measure over which there has been considerable controversy in the recent past with the advent of RISC machines.

the level of individual instructions or arithmetic statements but at the level of functions and processes (see Lathrop [1983] for a systolic architecture approach to exploit the parallelism in manipulator dynamics).

What this observation transfers down to at the architecture level is that having multitudes of processors, each of which can execute only a small repertoire of instructions, will probably not help very much for implementing robot control algorithms. By *coarse-grain parallelism* I also mean that the amount of processing power required at each node could be substantial. In fact, Version I of the CONDOR had no provisions for floating point arithmetic since it was felt at the time that scaled integers would suffice for most of our computations. Soon, however, the complexity of implementing digital control loops with such scaled arithmetic forced us to realize that we would benefit substantially with the addition of floating point hardware. Version II of the CONDOR therefore has the fast Motorola 68681 floating point unit in addition to the Motorola 68020 processor.

The second observation of controllers for real time systems is that

- Real time control involves i/o with external peripherals, and the word "real" in real-time translates to servicing hardware interrupts with low latency.

This constraint is often a very severe one, especially when the robot to be controlled is even moderately complex. The need for low-overhead and efficient schemes for implementing control programs often is in direct conflict with the goals of flexibility and ease of program development. Servicing interrupts on a single processor is not a significant problem, and if the scheduler has been designed correctly, very complex servoing can be done. However, on a multiple processor system, this raises rather complex issues. First, the system needs to be modeled as a multi-rate system if a hierarchical controller with various routines running at different rates on different processors is implemented. Secondly, the issue of synchronization becomes harder to address than in a SIMD environment. However, such issues are far outweighed by the advantages of having a multiple microprocessor system, which are:

(a) The implementation of hierarchical controllers is possible in a flexible and modular way.

(b) More processing power can be added if needed, by adding more processors to the system.

Processor Type	Speed	Cost	Comments
Microvax II	1 MIP	mod	interconnect problems
Vax 11/750	0.6 MIP	high	interconnect problems
Vax 11/780	1 MIP	high	interconnect problems
Symbolics 3600	1 MIPS	high	lacks real time support
National 32032	1 MIPS	low	
Motorola 68000	1 MIPS	low	lacks floating point
Motorola 68020	2.5 MIPS	low	has a ffp co-processor

Table 6.1: Comparisons of processing power available from alternative hardware configurations.

Interconnect Type	Speed	Bandwidth	Comments
Serial Lines	300-38Kbaud	low	Too slow
Parallel Ports	100K-200Kbaud	low	Slow
Ethernet	10 Megabits/s	fairly high	Complicated software but fast
DMA-DR11-W	1 Megabyte/s	fairly high	Complicated software but fast
Bus-to-Bus Adaptor	Bus Speeds	high	Simple software and fast

Table 6.2: Comparisons of different types of interconnect.

The hardware design that we finally chose has fairly sophisticated processors in the form of single-board computers plugged into a standard bus. The Version-I design was based on computers based on the Motorola 68000, while the Version-II design was based on the Motorola 68020 cpu, coupled with the Motorola 68881 floating point unit. The interconnect scheme chosen for the different processors was the Multibus for Version-I hardware and VME-Bus for the next version.

In Table 6.1 we give a performance comparison of the different cpu's we looked at, and in Table 6.2 we give a summary of the different interconnect schemes we considered.

Another major design decision we made was to use off-the-shelf hardware. A number of man years have been wasted in robotics laboratories, building custom solutions to hardware

problems, which we wanted desperately to avoid. Developing high performance hardware is a task best left to companies that specialize at it. Integration at the board level might have been appropriate had board level products not been available to satisfy our requirements, but as it turned out, powerful single board computers were available at low costs. Hence we chose to concentrate our efforts at the system integration level, using board level products.

The use of an industry standard interconnect scheme cannot be overemphasized. The benefits of using a standard interconnect bus are that

(a) One can get other peripherals (like A/D and D/A converters) for such interconnects relatively easily.

(b) The migration path to newer hardware as faster boards become available is relatively painless.

(c) Replication and trouble shooting are much easier with published specifications.

(d) An objective comparison of such interconnect schemes can be made.

The block diagram of the resulting hardware is shown in Figure 6.1 for the Version-I hardware, while Figure 6.2 shows the final hardware configuration.

The selection of the right pieces of hardware certainly took more time than anticipated. This was mainly caused by the fact that although we had chosen the industry standard VME bus as our interconnect, the concept of linking such busses together using a bus-to-bus adaptor had not been put to test fully at the time we put our system together. The use of a 16 MHz 68020, with 1 megabytes of no wait-state, dual port ram also was state of the art at the time. This meant that we had our bit of hardware trouble shooting to do, even though the individual components we were working with were all supposed to conform to a standard.

A more detailed discussion of the design decisions pertaining to the Version I hardware can be found in Siegel et al. [1985]. This paper also details some of the software issues involved.

It must be pointed out that other research efforts have also recognized the cost benefits in using such a microprocessor architecture for their control laboratories. Of these, the NYMPH system described in Chen et al. [1986] and the system being built at IBM described by Korein et al. [1986] are patterned after our CONDOR system. Besides these systems, there have been other efforts that have tried to couple a real time microprocessor

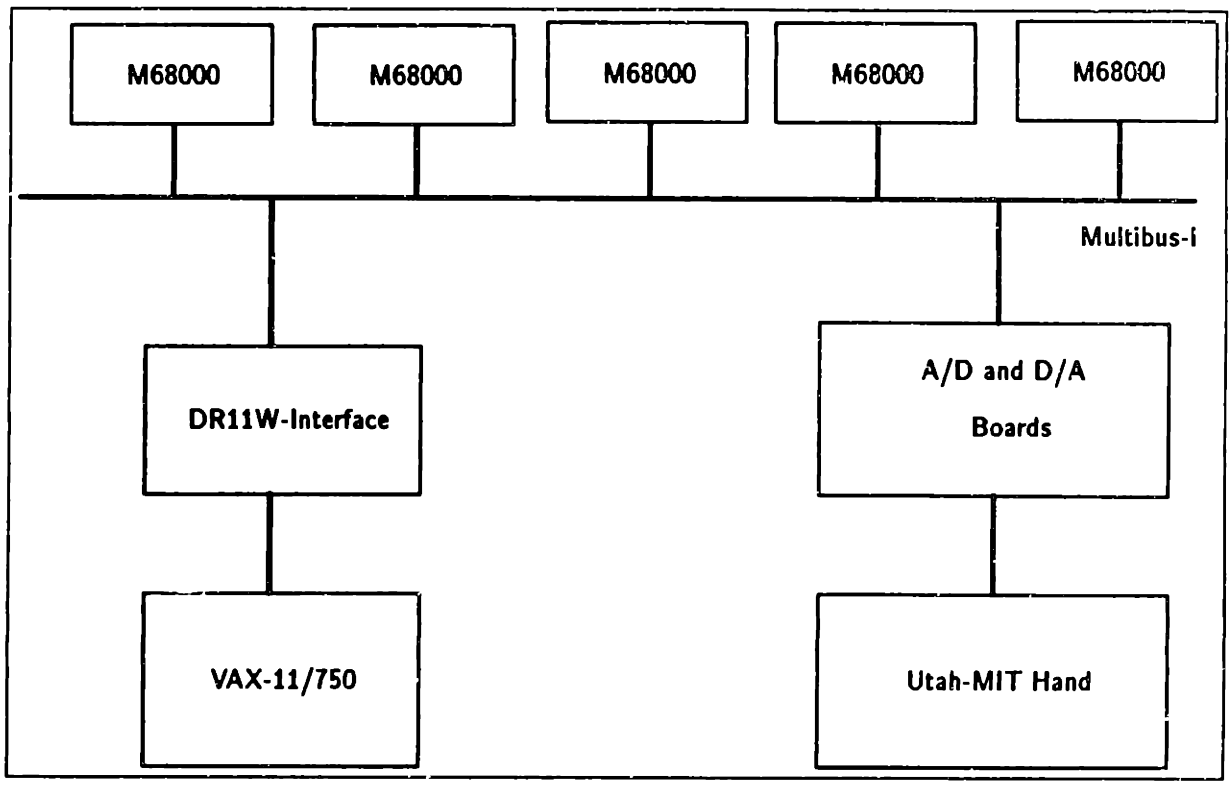


Figure 6.1: Block Diagram of Version I of the hardware.

architecture along with a development host. Our system differs from these controllers in two important ways. Firstly, we attempt to avoid duplicating software and hardware components that can be found on conventional, non real-time systems. Essentially, our system is partitioned into a real-time processing component and a conventional time-sharing computer development component. Secondly, we provide an extremely efficient computation environment. Only the minimal set of features necessary to provide a reasonable and convenient environment were included. This insures highly efficient operation of the CONDOR controller.

For example, Kim et al. [1987] describe a Multibus based real time system for controlling the Puma/RAL hand system. Their system is based on single board computers connected with a Multibus. Their development processor is one of the single board computers, and it provides access to disk and other system resources. We chose not to use this approach for several reasons. Most importantly, we feel that conventional computer systems offer adequate file serving and user interface capabilities, and did not want to duplicate such

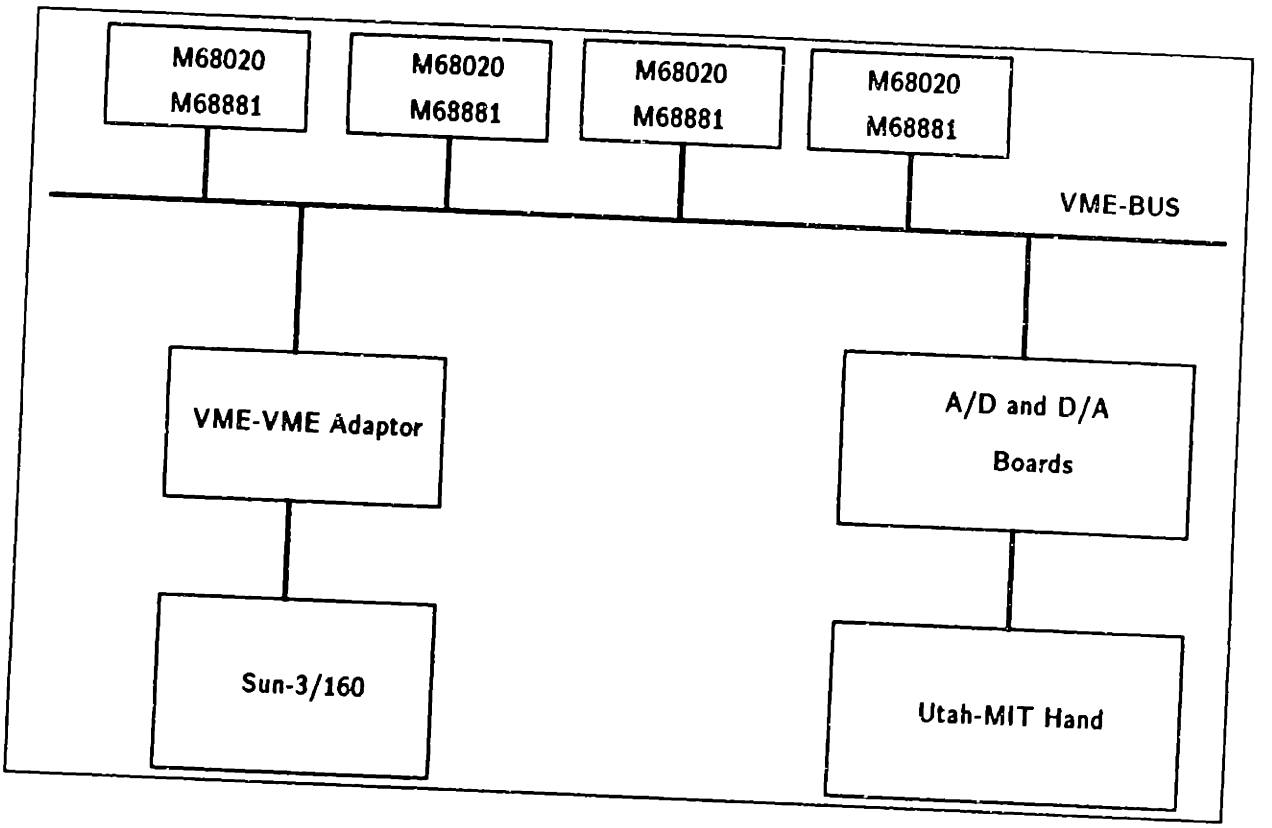


Figure 6.2: Block Diagram of Version II of the hardware.

facilities in our real-time system. Secondly, bus bandwidth on the real-time system is best left for real-time uses, and adding the development host's traffic on the same bus only exacerbates the contention for this resource.

Paul et al. [1986] describe another system built to control a Puma robot. This system is similar to our Version I architecture in that it uses the Multibus-I for its interconnect scheme. Connection to the development host was made initially through an ethernet communication link, but the complicated software needed to drive such an interface has necessitated the addition of boards that provide DMA access, much like in our Version I hardware.

6.1.1 Comparison between Version I and Version II

As is obvious from the figures, perhaps the main difference between the two different versions of the hardware has been the inclusion of the bus-to-bus adaptor which makes

possible a shared-memory implementation of a message passing scheme, which has been the key to a fast and extensible control hierarchy.

There are a number of desirable features present in the new hardware that were completely absent in the earlier version. The bus-to-bus adaptor enables transparent access from the development host into the dual-ported shared memory on the control microprocessors. In the Version I hardware, the DMA connection was being made between widely disparate hardware. Consequently, the software had to contend with different data formats, and this resulted in wasteful format conversions that had to be performed on every data transfer. The connection was slow, at times unreliable and of low bandwidth.

The individual microprocessors on the Version II hardware are fast processors, and coupled with the floating point engine they come equipped with, provide a much faster environment for scientific and control computing. The Version I software performed most of its computation using scaled integer arithmetic. Although such code is highly efficient, it is also unreadable and unmaintainable in some instances. Floating point on the other hand may be significantly slower if implemented in software.

The Sun-3 development host also provides a much better environment in terms of bit-mapped graphics, an industry standard window system and a transparent compiler (in Version I a cross compiler had to be used) that generates compact code. Programs that run on the controller processors can typically be run on the Sun-3 development host without recompilation.

To summarize, the features common to both versions of our hardware design have been:

1. Low cost, but powerful, multiple, single board computers performing the real time control function.
2. A *development host* that runs 4.2 BSD Unix to perform the software development on.
3. A relatively high bandwidth *interconnect* between the two systems.
4. An industry standard *bus* that connects up the slave microprocessors.
5. Dual-ported ram on the control processors, that facilitates control programs to take advantage of the shared-memory architecture.
6. A Mailbox interrupt that provides a hardware mechanism for interprocessor communication.

The differences have been:

1. The Version II hardware uses the same processor for the development host and the control processor, obviating the need for a cross compiler.

2. The Version II interconnect is the VME-Bus which is faster, provides support for full 32 bit transfers and is now the industry standard. The Multibus-I which was the interconnect used in the Version I hardware is now outdated.

3. The single board computer used in the new hardware runs at 16 megahertz, and contains a floating point co-processor.

4. The *development host* in the new system, a Sun-3, provides bit-mapped graphics, a fully networked file system and a much better software environment.

5. The connection between the development host and the control processors is made via a bus-to-bus adaptor in the Version II hardware which provides a more transparent form of access and communication.

6.2 Software

An important piece in any computational architecture is the software that is available to run on the hardware. Our comments regarding flexibility and efficiency of robot control hardware apply equally as well to the software components.

The software system designed to control the hand and other such robotic devices is relatively large and is adequately described in Appendix B. In this section we will merely provide an overview of what we feel are the innovative aspects of the software system.

The design goals of the software system were to:

(a) provide a flexible environment in which control programs can be written, debugged and run,

(b) provide efficient, low overhead means of doing often required tasks,

(c) provide easy to use programmer libraries for dealing with data transfer hardware, and real-time interaction with robotic devices and

(d) provide a sophisticated user interface with support for bit-mapped graphics.

To achieve these goals, the CONDOR system is structured around a few relatively simple organizing principles. In a typical program development scenario, the user is expected to

write and compile his program on a development machine. This development host is a Sun-3 workstation in our system. Since the Sun runs the Unix operating system[2], this means that the programmer has at his disposal all the software tools that he needs for writing and compiling his program. Once the program has been compiled, it is downloaded onto the slave microprocessors, where it is actually run. In this environment, which is also called the *run-time* environment, the user has access to a number of program libraries at his disposal, which enable him to do a number of tasks in a flexible and portable fashion.

Thus, the CONDOR environment is really *two* different environments. First, there is the environment on the Sun workstations known as the *development* environment, and then there is the *run-time* or *real-time* environment on the slave microprocessors. Much effort was put into making the Sun and the slave microprocessor run-time environments as compatible as possible. In fact, most programs that run on the real-time controllers will run on the Sun simply by relinking them with the appropriate library.

In the following sections, we provide a brief overview of some of the important subsystems of the CONDOR programming environment. The discussion is restricted to pieces of software that actually took substantial time to implement and that contain innovative contributions in terms of software engineering and system development.

We must mention that the hardware architecture is a tightly-coupled, truly MIMD machine. As such, it requires that a programmer write programs that run on the different processors. Our approach to managing this multiplicity problem in terms of software engineering has been largely to ignore it. We felt that the coarse-grain parallelism existing in current control programs and algorithms could be managed by the programmer himself. Therefore, we chose not to impose any restrictions on what a programmer could do. Furthermore any automatic scheduling or load-balancing scheme necessitates some overhead, impairing the performance of the system as a whole. The effort involved in implementing such a scheme is often not commensurate with its benefits.

Typically, control programs partition fairly cleanly, and hence manually deciding upon a partitioning scheme has not proved to be a problem in practice.

The number of processors in our control development scenario rarely exceeds five or six and we do not expect this architecture to be applied to problems requiring more than a dozen processors. Under this number, managing the different programs that need to run

[2]Most of the system runs on Sun O.S. 3.2 which is closely compatible with and is based on Berkeley 4.2 BSD Unix.

on the different processors can be done with existing software tools like *make*[3].

6.2.1 Devices

Interacting with robots usually means involvement with different types of hardware devices. Most robots have idiosyncratic controllers at their front ends, and a diverse array of sensor devices. A number of schemes exist for actually performing data transfers; polling, vectored and non-vectored interrupts, direct memory access are but few of many.

Any computational architecture that expects to deal with these robots must have mechanisms to support these different ways of interacting with devices.

The design of the device system was motivated partly by our experience with an earlier version of the system. In Version I of the CONDOR there was no systematic way of dealing with devices. In fact, what existed was an ad-hoc interface between the `read` and `write` system calls and various device specific routines. Even this interface was not strictly adhered to by most of our device drivers. This meant that to use a parallel port board that had been plugged into the system a user had to be aware of the procedures needed to initialize it, and be aware of what addresses the device's registers occupied and a myriad of other low level details. In Version II of the CONDOR we distinctly felt a need for a clean design at the lowest level.

The CONDOR real-time environment was therefore designed to provide an extensible way of writing device drivers which are program modules that control real-time devices that the CONDOR environment is aware of. Only the writer of a device driver need be aware of low level details like hardware addresses and interrupt mechanisms. The CONDOR system also provides a standard and well-documented manner in which device drivers can be written and automates much of the bookkeeping previously done by the device drivers themselves.

The CONDOR system's device mechanisms have been designed to be extremely fast and portable. The mechanism is static. All that is required to install a new hardware device is to recompile the system libraries used by the programmers. This is in contrast to other complicated systems that provide support for downloadable device drivers at run time rather than compile time. The overhead associated with recompilation is significantly lower than the overhead and complexity associated with any dynamic loading scheme.

The functions that these mechanisms provide are as follows:

[3] *make* is a Unix program for managing complex programs comprising a number of different modules.

1. Provide for automatic device initialization.
2. Provide capabilities to handle interrupting devices.
3. Provide capabilities to handle shared interrupt vectors.
4. Provide standard system calls like **open**, **read** and **write**, where appropriate.
5. Provide extensibility to handle devices that may require more than the standard set of routines.

The mechanisms for handling devices is modeled after the style found in early versions of Unix although there are a few significant differences. Each device is essentially modeled as an abstract data type, on which a few standard operations can be performed. When such a device is opened, an integer object called a **file descriptor** is returned whose semantics are close to that of the conventional Unix file descriptor. Users can then use this object as an argument to other operations that they need performed on the device.

The standard file descriptor usually maps to a device structure that has the following fields:

```
struct devsw {
    char *dvname;
    int (*dvopen)();          /* open routine */
    int (*dvclose)();         /* close */
    int (*dvread)();          /* read */
    int (*dvwrite)();         /* write */
    int (*dvcntl)();          /* ioctl */
    int (*dvinit)();          /* init - probe */
    int (*dvputc)();          /* put a char */
    int (*dvgetc)();          /* get a char */
    int (*dvseek)();          /* seek */
    int (*dviint)();          /* input interrupt routine */
    int (*dvoint)();          /* output interrupt routine */
    char *dvdata;             /* device specific data */
    char *dvbuf;              /* device's buffer */
    int dvno;                 /* device's no */
    int *dvcsrs;              /* device csr array */
```

```
        int *dvvectors;             /* device's vector array */
        int dvnumvectors;           /* number vectors for a single device*/
    };
```

The configuration for the entire system is initialized statically at compile time. For example, a parallel port board may be described by an entry similar to the one given below in the configuration file:

```
/* 4 Motorola Parallel Port Board */
    {
        "mpp", mpp_open, mpp_close, mpp_read, mpp_write,
        mpp_cntl, mpp_xinit, NULL, NULL, ionull,
        mpp_xint, mpp_xint, NULL, NULL, 0, mpp_csr, mpp_vec, 2,
    },
```

The following piece of code shows how a programmer can for example open a parallel port device and set it up for further operations:

```
int
parallel_port_startup(board_number)
int board_number;
{
    int fd;
    if((fd = open(''':mpp'', board_number, 2)) < 0){
        printf(''Couldn't open device?\r\n'');
        exit(0);
    }

    /* Reset the board */
    mpp_reset(fd);

    /* Configure the board to be in raw 16-bit mode */
    mpp_config_16bit_raw(fd);

    /* return the fd, so that the user can use it later */
    return(fd);
```

}

As the above usage illustrates, each device has an `open()` routine, a `close()` routine etc. There is one system-specific configuration file that tells the run-time system, what types of devices are supposed to exist and other required parameters. The CONDOR run-time system will, upon startup on the slave microprocessors, try to find all such devices and initialize them as specified by the device's `init()` routine. This is analogous to the `probe()` routine in conventional Unix systems.

When the user-level program then begins running on the slave microprocessors, they can perform operations like **open**, **read**, or **write** on these devices. The system maps these device independent calls, through the file descriptor objects to device specific routines. Although the **file descriptors** are allocated each time a device is open'ed, they will map to the same device level buffers, since the CONDOR is not multi-tasking. The lower level interrupt routines, which we will mention briefly later, operate on these device-specific buffers.

Thus it becomes extremely easy to add devices onto an existing CONDOR system. The writer of the device driver merely has to write his individual routines that correspond to entries in the device configuration file, and recompile the system.

There are a number of different operations that can usually be performed on devices besides the usual **read**, **write** set. The common Unix way of handling such unusual operations is to overload the common system call known as `ioctl()`. But this approach can soon get out of hand, with a large number of ioctl options being defined, each with its own peculiarity. In the CONDOR system we chose to use the following conventions to deal with this problem, and in practice, this has proved effective.

1. Device specific routines will be uniquely named by prefixing the routine with the name of the device (for example, a routine for configuring the **mpp** device will be called **mpp_configure**).

2. Routines that are peculiar to a device will take the file descriptor as the first argument and map it to a device specific structure. Once this mapping has been performed, that driver can then do any kind of operation that it desires. This essentially provides entry points into the device driver through a back door – not through the standard device structure that contains device specific versions of **read**, **write** etc.

The device driver interface also provides the low-level glue for the interrupt mechanisms,

the file server interface and the buffered stdio libraries.

6.2.2 Interrupts

Besides the ability to deal with devices, another capability that control system architectures require is the ability to service interrupts in real-time. These interrupts usually correspond to events that require attention, or periodic interrupts from the timer at some sampling rate.

The VME-bus provides support for eight levels of prioritized vectored interrupts, and the Motorola 68020 processor has the capability to support 256 different interrupt vectors. Interrupts on the VME-Bus are vectored, in contrast to the Multibus-I which supports non-vectored interrupts. The Multibus-II supports a different notion of interrupts which essentially increases the number of different levels of interrupt possible on the bus.

Interrupts can come from a variety of sources: interval timers, a-d and d-a converters, parallel and serial i/o devices, etc. It is important that all such interrupts be handled in a uniform and extensible fashion. In Version I of the CONDOR, interrupts were handled in a rather ad-hoc fashion. This resulted in a lot of assembler code sprinkled throughout the system, since most interrupt servicing routines have a small assembler portion that dispatches to a routine written in a higher level language.

In Version II of the CONDOR, all interrupts vector to the same higher level routine. The system has a software data structure that maps vector numbers to interrupt servicing routines. This scheme has resulted in a single assembler routine that services all interrupts.

There are a few complications that must be dealt with. When an interrupt is generated and needs to be serviced by a processor, somehow enough information must be found that allows this interrupt to be mapped to a file descriptor corresponding to a device. Since the CONDOR is not multi-tasking, no distinction exists between system space and user space. When a serial chip interrupts the system telling it that it has a character to transmit, the system has to be able to tell which serial-i/o driver's input buffer the character must be sent to.

The problem is further complicated by the fact interrupt vectors can be shared. This means that two devices can both interrupt the system using the same level and same interrupt vector. The CONDOR system solves such complications by maintaining a list of all devices that express interest in receiving interrupts on a particular vector. When more than one device can vector using the same interrupt vector, the CONDOR system maps this

interrupt vector to a generalized *device-level interrupt* vector. This routine goes through the list of interested devices and invokes the interrupt routine of each of those devices one by one.

Each interrupt routine when invoked by the system is passed two arguments telling it the vector number and a pointer to a saved copy of the registers that reflect the state of the machine when the interrupt occurred. The device level interrupt on the other hand invokes its interrupt routines with arguments indicating the file descriptor corresponding to the interrupt, and the minor device number (since more than one device may use the same interrupt vector number), in addition to these two.

6.2.3 Message Passing

The message passing sub system is another low level system of the CONDOR. While the device driver and support routines are intended mainly to provide support for bootstrapping and running a program on a single processor, the message passing system tries to address the issue of multiple processors and the need for communication between them.

6.2.3.1 Introduction

The message passing system provides a simple, low-overhead manner in which communication of data can occur between processors that comprise the CONDOR controller. A typical application running on the CONDOR controller comprises of a set of processes running on the microprocessors. These processes can communicate with one another using the message passing system. Since the servos are always compute bound tasks, any system for communication between such tasks has to be extremely time-efficient in order not to impair the real time performance of the system noticeably. The primary design goal of the message passing system was therefore efficiency.

The present system is to a large extent, a redesign of the system described in Narasimhan et al. [1986]. Although the functionality provided by that system was far greater than that provided by the present version, I feel that this second attempt has been made substantially more efficient.

6.2.3.2 Messages

In any multiprocessing environment interprocessor communication is a necessity. Since the Ironics processors and the Sun host computer are all bus masters on a common VME

bus, each machine has access to each other's dual ported memory. Interprocessor communication occurs over the bus and directly uses shared memory. This allows, for example, any processor to directly access data in another processor's memory. The most basic form of interprocessor communication possible would be direct memory reads and writes from another processor storage. Unfortunately, this unrestricted access, though highly efficient, is hard to control.

To overcome the problems of unrestricted memory access, a mailbox-based message passing system is supported. Mailbox interrupts can be thought of as a software extension to the processor's hardware level interrupts. Another way of thinking about them conceptually is to regard mailbox numbers as port numbers that map to specific remote procedure calls.

A mailbox interrupt has a vector number and a handler routine. When a particular mailbox vector arrives, its appropriate handler is invoked. The handler is passed the processor number that initiated the mailbox interrupt and a one integer data value. This integer data value is the message data[4]. To summarize, there are two pieces of data that get transmitted for every message – a message handler number and a piece of integer data that can be used as the user sees fit.

The Version I message passing system was substantially more complex. Messages could be of arbitrary size, and they could be addressed not to individual processors but to so-called virtual devices that corresponded to Version II's remote procedures. These remote procedures were assigned to processors by a preprocessor that took as input an assignment file, that mentioned which routines were to be available on which processor. The preprocessor then generated a routing table that had to be linked in with each program running on the individual processors so that the **send** routine on that processor could decide based on the recipient virtual device number alone which processor it ought to send the message to.

Another important capability that the Version I implementation lacked was the ability to reply to messages. This meant that application programmers had no way of knowing when a particular message succeeded or failed. Implementation of messages with replies could be done in the Version I system through busy-waiting or spin locks, which was also very undesirable. The Version II protocol provides support for replying to every message, at a very basic level, should the user require reliable and synchronous transmission of messages. The implementation uses a reverse send from the recipient processor to the

[4] An integer in the present implementation refers to any quantity that can fit in 32 bits.

sending processor to acknowledge the receipt of a message. This addition has proven to be extremely useful.

The Version II redesign was done mainly because of the complexity and the overhead of the earlier system, that prevented most control programs from hardly ever using the message passing system.

To illustrate the present system, let us say for example, that the user wants to write an address handler that will receive a message composed of a memory address, and will respond with the value found at that particular memory address. This simple example will illustrate not only how messages are received, but also how messages are sent and how certain messages can be replied to.

The conceptual design for such a user level message passing system is simple. Each mailbox handler is invoked with the first argument being the processor number that sent the message and the second argument being a piece of data that is wide enough to fit into a 32 bit quantity. We can therefore design the message handler such that when invoked, the memory address it needs to look at will be passed to it as the data associated with the message (since we do not expect our addresses to be longer than this). The message handler will essentially decode the message to find out what this address is, and return it.

The handler can therefore be written thus:

```
simple_decoder(proc, data)
int proc;
int data;
{
    return(*(unsigned int *)data);
}
```

After writing the handler one has to associate this handler with a vector number so that when other processors request this service, it will be performed correctly. This is done by the following piece of code.

```
int simple_decoder();
mbox_set_vector(12, simple_decoder, ''A test handler'');
```

Once this call has been executed messages that arrive for vector numbered 12 will cause the simple decoder routine to be invoked automatically.

But how does another processor invoke this routine? This is done by the **mbox_send** routine. If the **simple_decoder** routine is available on processor 0 one can execute the following piece of code on any of the processors (including 0) to invoke the service.

```
value = mbox_send_with_reply(0, 12, address);
```

This will cause the handler that corresponds to the number 12 to be invoked on processor 0 with the second argument being **address**. The call will not return until the other processor has responded with the value found at the given address. This call can therefore be used to provide synchronization.

There does exist another version of message sends that does not require the sender to wait until the handler for the message has completed executing on the recipient processor (called **mbox_send**).

There are several important points to be noted about using the mailbox handler for communication:

(a) Since message sending happens asynchronously, the execution of a handler resembles an interrupt. All caveats that apply to interrupts and interrupt handlers therefore apply to message handlers too.

(b) The base system is extensible in the sense that more complicated protocols can be built on top of it. For example, the underlying system does not provide any support for queueing, although one can very easily build one for mailboxes that require this.

(c) On the Sun, message handling is arranged to always happen on the process that sets up the handler using the Unix **select** system call.

(d) Since the message system is based on shared memory, sending long messages is usually handled by sending a pointer to the beginning of a long piece of data. If the communication mechanism is serial, wherein a stream of data needs to be sent, sending a large number of messages may cause unwanted system overhead (although message sending is usually a single subroutine call). This can be avoided via a packet based interface to the message passing system the details of which are beyond the scope of this document. It is our opinion, that such links will be of such low bandwidth that they will rarely be used, if ever at all.

(e) Most often, one may need a variety of functions that need to be performed in response to a message. Instead of defining ONE message handler for each such function, it may

be helpful to collect related groups of handlers into a single handler, that dispatches to separate routines based on the data item sent along with the message.

(f) We have thus far used the convention that related messages thus grouped will be found in a single file, and that the vector numbers that correspond to message handlers be localized to a single .h file. This allows the symbolic use of handler names rather than numbers.

(g) Where efficiency is important, the message handling system can be used just to set up pointers from one processor into another's memory. After such a set up is complete, the processors can read and write this shared memory as they please. This removes the burning of addresses in programs, but maintaining the integrity of shared data structures will entirely be the programmer's responsibility.

Message sending and the invoking of message handlers is implemented under the present system using a so-called *mailbox interrupt* which is a hardware interrupt that allows one processor to interrupt another processor on the VME-Bus. This hardware support is critical to our current implementation's efficiency. To protect the integrity of certain implementation dependent data structures the TAS or *test-and-set* instruction is used. It is important that this instruction be supported truly indivisibly by the hardware even across the bus, for our implementation to work correctly.

6.2.3.3 Support for Message Passing on the Sun End

The development host in the system must have and support the basic primitives needed to do message passing with the slave or control microprocessors. The Sun on the other hand, runs a time-sharing operating system, and at any given moment there may be a number of processes on this machine all communicating with one or more slave processors across the bus-to-bus adaptor.

As we mentioned earlier, message handling resembles the occurrence of a hardware interrupt, in that a routine is invoked in response to a **send** from another processor. On the Sun, however, when a message comes in from the slave processors, a decision needs to be made as to which process must be woken up or interrupted. The CONDOR system supports the notion of *virtual processor*. Every process on the Sun end, which desires to communicate with the slave microprocessors, must express its interest to the kernel. Since interrupts on the VME-Bus are vectored, this is done by the process expressing its interest

relative to a particular vector. When a slave processor desires to interrupt the Sun, it does so using this vector. The Unix kernel traps on this interrupt vector and signals all processes that have expressed an interest in receiving the interrupt. Currently, only one process can *own* an interrupt vector on the Sun, although a low level mechanism does exist for multiple processes to be woken up on the same interrupt vector.

An Ironics processor's memory is entirely dual ported, and hence totally accessible over the bus. The Sun has a region of memory called DVMA space (for Direct Virtual Memory Access). The DVMA area occupies the lowest megabyte of the VME 24D16 and 24D32 address space of the VME bus. However, it is not convenient for an Ironics processor to communicate with the Sun using this space, since Unix memory management issues become complex.

Instead, an extra 1 megabyte dual ported ram board was installed in the VME bus for use primarily by the Sun. This board can be thought of as the local memory that the Sun has control over. Ironics processors can directly communicate with this storage, instead of using the DVMA space on the Sun. If need be, this area of memory that the Sun uses for receiving messages intended for it, can be allocated on any of the Ironics single board computers' onboard dual-ported memory.

From the Sun, the CONDOR system maps the entire VME 24D16 space (or VME 24D32 space if the adaptor used is the **hve-2000**) into the user address space of the control process. Memory references to any of the Ironics or the Sun's 1 megabyte board on the VME bus become simple array references from a user's program. The PROC_RAM(processor) macro returns the pointer to the bottom of memory for the particular processor. For example, to write a value to location 100 in processor 3's memory one could use the following code:

```
int *processor3_ram = (int *)(PROC_RAM(3));
processor3_ram[100] = value;
```

The PROC_RAM macro is also used for programs running on Ironics processors to access memory of other Ironics processors. The code above would work, in fact, on any processor in the system.

Even though the mailbox routines in the CONDOR system are highly efficient, it is sometimes desirable to directly use the VME bus shared memory for interprocessor communications. For example, consider a data structure on processor A:

```
struct data {
```

```
    int a;
    int b;
} data_A;
```

Consider the following mailbox handler also present on processor A:

```
get_data_pointer(proc, data)
int processor;
int data;
{
    return ((PROC_RAM(PROC_A) + (int)(&data_A));
}
```

Processor B could define a pointer to such a structure, and initialize it's value using the mailbox handler made available on processor A:

```
struct data {
    int a;
    int b;
} *data_A = (struct data *)
            mbox_send_with_reply(PROC_A,GET_A_DATA_PTR, 0);
```

The mailbox handler on A will pass the VME bus address of its data structure back to processor B.

6.2.3.4 Message Passing and its Implication for Control

The previous section has briefly outlined the message passing scheme present in Version II of the CONDOR system. The scheme is extremely efficient and fast since it exploits the shared dual-ported memory across the VME-Bus to the maximum.

Table 6.3 summarizes the performance of the message passing system as benchmarked by a variety of routines.

As can be seen from the table, the performance of the message passing system is extremely good between two control processors and slows down only when messages are being sent up to the development host owing to the overhead caused by the Unix timesharing operating system running on this host.

Type of Operation	MilliSecs/Message	Variation (secs)
Ironics to Sun	34	5
Ironics to Sun + Reply	38	10
Sun to Ironics	4	0
Sun to Ironics + Reply	4	0
Ironics to Ironics	0.2	0
Ironics to Ironics + Reply	0.25	0

Table 6.3: Performance of the Message Passing System.

The reliability of the Version II implementation has also been unusually good. A number of other protocols have been implemented on top of the base message passing system, which will be described shortly, and all these other services perform extremely reliably.

It must be mentioned that the message rates are not as high as servo rates. Consequently, messages are used only as signals to start and stop processes or control the flow of computation and are not used as timing pulses.

Using the facilities provided by the message passing system judiciously it becomes possible to treat a hierarchical controller as a truly object-oriented system that will respond to certain messages in certain ways. For example a low-level joint level PID controller could be described as an object that responds to different kinds of messages. These could be messages that

(a) alter internal parameters like gains, damping co-efficients, position set points, etc., and

(b) others that start and stop the execution of the servo loop or change the servo rate.

Once the necessary code has been written to respond to these messages, and the program is running on a particular control processor, any *other* processor can now act as a supervisor of the lower level controller. This master processor can now *send* commands down to the lower processor telling it what to do. Since the protocol is clearly implemented by the message handler in the lower control processor, it is clearly specified and easy to change and hides all implementation details of how the messages are actually implemented, just as a good *data abstraction* requires. Such an implementation could be nested, with one control processor sending a message down to a lower level processor which in turn communicates with another processor to perform its task and so on.

In fact, most of the control implementations described in the previous sections have been implemented using such an object-oriented paradigm.

6.2.4 Virtual Terminals, The File Server, Debugging

The CONDOR system requires the concurrent operation of several control microprocessors to perform its task. Programming such a complex MIMD machine would certainly be a nightmare were it not for the numerous services that were built on top of the base system using the message passing facilities.

These subsystems enable flexible interaction with a number of slave microprocessors at a time, provide file server capabilities on the development host, and enable symbolic debugging.

It is in the implementation of such rather complex subsystems that our message passing design has truly paid off. Although the Version I software was flexible enough to implement these facilities, the system was complex enough to warrant most programmers to actually use only a few of its capabilities.

6.2.4.1 The Pseudo Terminal Emulator

The pseudo terminal emulator is a prime example of such a complex facility. It was originally intended to provide a terminal interface to the multiple slave microprocessors controlling the hand. Such a real-time facility to look at the states of the different machines was required to develop and debug the rest of the software.

In the Version I hardware, this would have meant either attaching multiple terminals to the different nodes or having the programmer switch modes actively. Complex issues of buffering, forwarding and flow control needed to be addressed. The Version II hardware provides a simple solution. Different areas or windows on the bit mapped screen are used to indicate different processors. A multiplexor/demultiplexor process that routes keyboard input to the designated processor's input queue and routes the output from a processor to a particular window was all that was needed to facilitate a rather sophisticated user interface that enables the user to interact with any of the control microprocessors simultaneously. The protocol used for the transmission of the data is extremely simple and could be built and tested in a day.

6.2.4.2 File Server

Another such complex subsystem is the file server. One of the predominant needs for such a service arises from the fact that control programs, especially in a research environment need to collect data and store this data in disk files to facilitate further analyses and examinations. If programmers could write their programs to run on the slave control processors just as though they were writing their programs to run on the development host:

1. They can debug their code faster since they can test it out on the development host, since it eliminates a few stages in the *edit, compile, download, run, debug* cycle that would otherwise be required.

2. They can use the same model they use in programming conventional Unix machines as far as the file system is concerned.

The file server on the Sun end provides such a transparent service to control programs running on the control microprocessors. For example, when the following program is run on the control processors, the **open()** call essentially gets translated to an **mbox_send** that sends up a message to the Sun, asking it to service the request.

```
FILE *fp;
if((fp = fopen(filename, ''r'')) == NULL) {
      printf(''Couldn't open file %s\r\n'', filename);
      return(-1);
}
```

Other system calls related to the file system are also sent up to the Sun in a similar way.

To make the file server efficient, shared memory is extensively used to avoid copying data. When a microprocessor performs a read or a write, the file server process running on the Sun reads or writes the data directly to or from the microprocessor's memory. No intermediate data copy is required. To perform a read, for example, the microprocessor would send a message to the Sun giving the address to read the data into, and the number of bytes to read. The data is directly transferred into the processors buffer.

The file server is designed to operate in a stateless manner. All data necessary is stored on the microprocessor. The Sun server does cache some information, but if necessary, it can request the information from the microprocessor. Thus, if the Sun file server process is terminated and restarted, the microprocessor can continue its file operations without any noticeable effect.

6.2.4.3 The Debugger

Perhaps the most complex of all the subsystems to be implemented using the message passing system is the **ptrace/wait** emulation facility that enables sophisticated symbolic debugging of C programs running on the control processors from the development host remotely. In conventional Unix systems, debugging is done in the following manner: A parent process is set up to communicate with a slave process, which is the process being debugged. The user interacts with the parent and the parent carries out his requests, by controlling the slave process via a system call known as **ptrace**. This system call essentially is a message passing specification, which enables one process to control and trace another process's actions in terms of single stepping, setting up break points, examining register values and so on.

Debugging is a rather unique activity associated with programming and programmers. In practice, even after good system design and careful coding, it is in debugging an initial system that the skill level and productivity of a programmer usually shows up. As the complexity of the system goes up, the need for such debugging aids becomes important. Before designing the second version of the CONDOR system, while we were evaluating our experiences with the first version, it was the debugger that occupied most of our thinking. This was an area we felt sorely needed addressing. The solution we came up with, was to avoid writing a new debugger, since most conventional Unix debuggers did have the capabilities we needed. What they lacked was the ability to debug a process running independently, perhaps on another machine. Hence, we decided to implement an emulation facility for the **ptrace** system call using the Version II message passing system. In this manner, any conventional Unix debugger could act as the master during a debugging session and could communicate with a slave process running not on the same machine, but on one of the control microprocessors.

A short table of the message passing operations needed to implement such a scheme is shown in Table 6.4.

This emulation facility can be linked into any conventional Unix debugger. Once this linking has been done the debugger can be used to debug control processes running on the slave microprocessors across the bus to bus adaptor.

Such a remote debugger interface can be used in many ways. The first allows the user to start executing a program directly under the debugger's control. This mode would be used when a bug is actively being tracked down, and setting initial break points might

TRACEME	Start tracing the process
ATTACH	Attach to a particular processor
DETACH	Detach from a processor being debugged
PEEKTEXT	Look at a program running on a slave
PEEKDATA	Look at the data on a slave
POKETEXT	Modify the program running on a slave
POKEDATA	Modify the data running on a slave
KILL	Kill the slave process
SINGLESTEP	Single step the slave process
CONT	Continue the slave process
GETREGS	Get registers from a slave process
SETREGS	Set the registers on a slave
START	Start up the slave process
STOP	Stop the slave process

Table 6.4: Opcodes for messages to implement the ptrace emulation.

be necessary. The second mode allows the debugger to be attached to a processor after execution has begun. For example, if a program were in an infinite loop, the debugger could be attached to the running program to determine what went wrong. Finally, if a program receives a fatal exception, for example an addressing error occurred, the debugger can be attached after the error occurred to help analyze the problem.

Once the debugger has attached to a process on the slave control processor, all the capabilities of the debugger can be used to debug the program running on the remote host, just as if it were running on the development host.

Besides these important utilities a host of other programs have been written to aid in program development on the CONDOR real time system. These include plotting programs to graph real time data collected on the slave processors, simulators that provide a three-dimensional graphics capability to the industry standard X window system, a number of programmer libraries that can be used on both environments, and diagnostic programs for the different pieces of hardware that exist in the system.

Taken together, the system comprises of an architecture that we feel comes very close to satisfying our initial design goal of providing a flexible software environment in which a researcher can design his real-time control programs without sacrificing performance.

6.2.5 The CONDOR User Interface

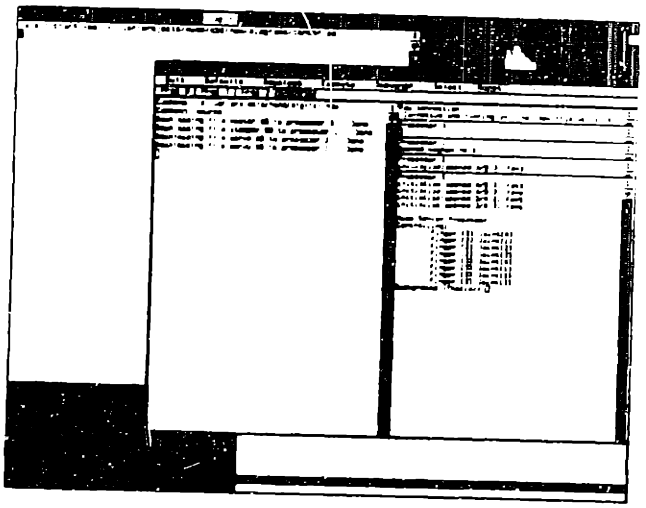


Figure 6.3: The CONDOR running under the X Window System.

The file server, debugger and virtual terminals combined together form the CONDOR user interface. This program is an X-window system based application that programmers utilize to interact with the slave microprocessors. The user interface provides one virtual terminal for each slave processor, it runs a file server, and it can start debuggers. Figure 6.3 shows the screen of a typical CONDOR user interface session.

6.2.6 Controller Implementation

To make the above abstract discussion of the computational architecture more concrete, in this section we outline how the actual partitioning has been done for the controller implementations that have been described in the previous chapters.

The current digital controller implementations use four processors. One processor is allocated for performing the joint-torque and tendon tension level servoing of eight joints at 400 hertz. Another processor is dedicated to controlling the *xyz* table. The fourth processor is a master controller.

An abstraction similar to that used for device configuration is also used for configuring the hand control programs. There is one central configuration file, which specifies which joints are allocated to which processor. This configuration file also specifies which A/D's and

D/A's are allocated to a particular joint. This permits hand control programs to developed in a modular fashion. Changes in hardware occur in terms of changed connectors, boards, and hardware addresses. To cope with these changes, no changes to the actual control programs is necessary. All that is needed are small changes to the controller file and recompiling the entire hand controller system.

The trajectory generator currently executes at 50 Hz on the master processor. Besides executing this slower loop, the master processor also performs a number of other functions, including:

(a) It forms the primary interface to the Sun, to load and save entire sequences of motions.

(b) It provides a sophisticated interface to the user, who can select a subset of joints to monitor at any given moment. The required controller variables can be tuned from this level, and the status of the hand at any given moment can be ascertained.

(c) It controls the execution of the servos on the slaves. Using the message passing system, it can enable or disable servos on the other processors, find out their status, or restart errant programs.

The current implementation is also very modular, in that the actual servo portion of a program that implements the control computation is contained in a single file. By replacing this file with another of his own choosing, a prospective user can begin to experiment with low, medium and higher level control algorithms, without having to rewrite or even look at the rest of the user interface programs.

Besides the actual real-time programs, the hand control environment also comprises of programs that run on the Sun. These programs include a real-time plotter that allows a user to monitor any of the controller variables in real time, and a kinematic simulator that can be used to generate input to the trajectory generator running on the master processor.

This thesis has looked at different problems that arise in actually controlling a robot hand to perform simple tasks. The bottom-up approach taken in this thesis is perhaps not representative of other efforts that have concentrated on a single theoretical issue. However, it has resulted in valuable tools and experience that can now be used to deal with the truly important and relevant higher level issues.

To complete this thesis, I would like to briefly describe representative problem areas that need to be looked at in the near future. These include extensions to the work presented here in this thesis, and new problem areas that have been suggested during the initial experiments that I have begun performing with the controller implementations.

7.1 Control

There are a number of problems that still need to be resolved with respect to the control of dexterous robot hands. With respect to position control, we have shown that stable control of such hand-arm systems is indeed possible, and that simple hierarchical controllers can perform the required tasks. The interesting task that remains to be done is to mount such a dexterous hand on a robot that has a multi degree of freedom wrist. This will increase the versatility of the hand by increasing the number of tasks it can be used for tremendously.

Achieving stable force control in manipulation tasks however, does not seem to be so simple. In particular the experiments performed with the force controller implementation seem to indicate the following areas in which much fruitful research can be expected in the future.

Actuator Modeling: Part of the problem involved in the current implementation of the force controller implementation is that there is no good underlying model of the actuation and transmission systems that the controller can use. The pneumatic actuator is in fact a highly non-linear system with stiction and coulombic friction.

Torque Estimation: Since the Utah-MIT hand does not have any torque sensors, the torque about a joint must be estimated. Without such information, there exists no good way of determining how well the force controller is performing. In practice these torques can be

estimated in many different ways:

1. From the sensed tendon tensions:

$$\tau_i = (T_{fi} - T_{ei})r_i$$

This is equivalent to making an estimate of the joint torques, from the input variables to the controller. Since the tendon tensions are measured fairly close to the joint, the friction between the actuators and the sensors can be neglected.

2. From the outgoing actuator voltages:

$$\tau_i = (V_{fi} - V_{ei})k_i$$

This assumes that the estimate of the joint torque is made from the output variables of the controller.

3. Use a sensed finger tip force and use the relation $\tau = \mathbf{J}^T\mathbf{f}$. This method is feasible if there is some kind of sensor at the finger tip that can output sensed force information. This is true of the Salisbury Hand which has a ball sensor mounted on its finger tips, which provides force and torque information, but not of the Utah-MIT hand.

4. From the position error:

$$\tau_i = K_i\delta\theta_i$$

where K_i is some specified joint stiffness.

5. From estimated accelerations.

$$\tau_i = M_i\ddot{\theta}_i$$

7.2 Planning

Currently there does not exist any flexible manner in which a high level task can be planned on the Utah-MIT hand. It would be interesting to apply the algorithms presented in the literature to choose grasp points and evaluate if indeed they enable achievement of stable grasps.

The higher level problems of feasibility and reachability mentioned in the introduction certainly need to be solved before such algorithms can be put to use in practice. Trying

to use such high level planners could indicate the importance and relevance of the various issues involved.

There are many issues with respect to planning that we have only hinted at in this thesis. The level of geometrical modeling needed to achieve stable and dexterous manipulation and the sophistication of current algorithms both need to be enhanced for the development of a more or less automatic programming environment for dexterous hands.

7.3 Sensing

It would certainly be instructive to mount tactile sensors on the hand and try to use the information coming off of these sensors while performing grasping and manipulation operations as well. In particular, the use of such information in the detection and prevention of failures during such tasks needs to be studied. The interface between such high-bandwidth sensors and the hierarchical controller needs to be made in a principled way.

The interaction between such sensors and higher level planning or learning algorithms that learn to plan over successive trials is also a subject that could bear fruitful results.

7.4 Conclusion

In conclusion therefore, I would like to mention that this thesis has looked at problems from widely different areas. We have looked at problems involving kinematics, control, planning and computer architecture. The problems involving kinematics and control were solved and a novel computational architecture was presented as a tool with which to address the experimental issues that arise in controlling dexterous robotic hands. A new force control algorithm was presented that is computationally very efficient. Future effort will concentrate on experiments that look at other issues pertaining to force control and higher level issues involved in the planning of grasping and manipulation operations.

References

1. **Abel, J. M., Holzmann, W., and McCarthy, J. M.,** *"On Grasping Planar Objects with Two Articulated Fingers"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 576-581, 1985.*

2. **Abramowitz, J.,et al,** *"The Pennsylvania Articulated Mechanical Hand"*, The University of Pennsylvania, *1982.*

3. **Abramowitz, J. D., Goodnow, J., and Paul, B.,** *"The Pennsylvania Articulated Mechanical Hand"*, Proc. ASME Conference on Robotics, Chicago, *1983.*

4. **An, C. H.,** *"Trajectory and Force Control of a Direct Drive Arm"*, Ph. D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *1986.*

5. **Angeles, J.,** *"Spatial Kinematic Chains - Analysis, Synthesis and Optimization"*, Springer-Verlag, *1982.*

6. **Angeles, J.,** *"On the Numerical Solution of the Inverse Kinematic Problem"*, International Journal of Robotics Research, *Vol. 4, No. 2, pp. 21-37, 1985.*

7. **Angeles, J.,** *"Automatic Computation of the Screw Parameters of Rigid Body Motions. Part I: Finitely Separated Motions"*, Journal of Dynamic Systems, Measurement, and Control, *Vol. 108, pp. 32-38, 1986.*

8. **Angeles, J.,** *"Automatic Computation of the Screw Parameters of Rigid Body Motions. Part II: Infinitesimally Separated Motions"*, Journal of Dynamic Systems, Measurement, and Control, *Vol. 108, pp. 39-43, 1986.*

9. **Arnold,** *"Mathematical Methods of Classical Mechanics"*, Springer-Verlag, *1978.*

10. **Asada, H., and Slotine, J. J.**, *"Robot Analysis and Control"*, Wiley-Interscience, New York, *1986.*

11. **Aström, K. J., Wittenmark, B.**, *"Computer Controlled Systems: Theory and Design"*, Prentice Hall, *1984.*

12. **Bajpai, A., and Roth, B.**, *"Workspace and Mobility of a Closed Loop Manipulator"*, International Journal of Robotics Research, *Vol. 5, No. 2, pp. 131-142, 1986.*

13. **Baker, D. R., and Wampler, C. W.**, *"On the Inverse Kinematics of Redundant Manipulators"*, General Motors Research Laboratories, *No. GMR-5478, July, 1986.*

14. **Begej, S.**, *"An Optical Tactile Array Sensor"*, Proc. SPIE Conference on Intelligent Robots and Computer Vision, *pp. 271-280, 1984.*

15. **Biggers, K. B., Gerpheide, G. E., Jacobsen, S. C.**, *"Low Level Control of the Utah-MIT Hand"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 61-66, 1986.*

16. **Brock, D., and Chiu, S.**, *"Environment Perception of an Articulated Robot Hand using Contact Sensors"*, ASME Winter Annual Meeting: Robotics and Manufacturing Automation, *pp. 89-96, 1985.*

17. **Brockett, R. W.**, *"Robotic Hands with Rheological Surfaces"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 942-947, 1985* .

18. **Brooks, R. A.**, *"Achieving Artificial Intelligence Through Building Robots"*, **AIM No. 899**, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, *May, 1986.*

19. **Brost, R. C.**, *"Automatic Grasp Planning in the Presence of Uncertainty"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 1575-1581, 1986.*

20. **Canny, J. F.**, *"Collision Detection for Moving Polyhedra"*, Pattern Analysis and Machine Intelligence, *Vol. 8, No. 2, 1986.*

21. **Canny, J. F.**, *"The Complexity of Robot Motion Planning"*, Ph. D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *1987.*

22. **Chang, P.**, *"A Closed Form Solution for the Kinematics of Redundant Manipulators"*, **AIM No. 854**, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, *March, 1986.*

23. **Checinski, S. S., Agrawal, A. K.**, *"Magnetoelastic Tactile Sensor"*, Robot Sensors, Volume 2 - Tactile and Non-Vision, Springer-Verlag, *pp. 229-235, 1986.*

24. **Chelpanov, I. B., Kopashnikov, S. N.**, *"Problems With the Mechanics of Industrial Robot Grippers"*, Mechanism and Machine Theory, *Vol. 18, No. 4, pp. 295-299, 1983.*

25. **Chen, F. Y.**, *"Gripping Mechanisms for Industrial Robots"*, Mechanism and Machine Theory, *Vol. 17, No. 5, pp. 299-311, 1982.*

26. **Chen, J. B., Fearing, R. S., Armstrong, B. S. and Burdick, J. W.**, *"NYMPH: A Multiprocessor for Manipulation Applications"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 1731-1736, 1986.*

27. **Chiu, S. L.**, *"Generating Compliant Motion of Objects with an Articulated Hand"*, S. M. Thesis, Department of Mechanical Engineering, Massachusetts Institute of Technology, *1983.*

28. **Colson, J. C., and Perreira, D. N.**, *"Kinematic Arrangements Used in Industrial Robots"*, Proceedings of the 13'th International symposium on Industrial Robots, Soceity of Manufacturing Engineers, *pp. 20-1-20-18, 1983.*

29. **Craig, J. J.**, *"Introduction to Robotics: Mechanics and Control"*, Addison-Wesley, *1986.*

30. **Crosnier, J. J.**, *"Grasping Systems with Tactile Sense using Optical Fibres"*, Robot Sensors, Volume 2 - Tactile and Non-Vision, Springer-Verlag, *pp. 209-217, 1986.*

31. **Crossley, F. R. E., and Umholtz, F. G.**, *"Design of a Three-Fingered Hand"*, Mechanism and Machine Theory, *Vol. 12, pp. 85-93, 1977.*

32. **Cutkosky, M. R.**, *"Mechanical Properties for the Grasp of a Robotic hand"*, CMU-RI-TR-84-24, Carnegie Mellon University, *1984.*

33. **Cutkosky, M. R., and Wright, P. K.**, *"Modelling Manufacturing Grips and Correlations with the Design of Robotic Hands"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 1533-1539, 1986.*

34. **Cutkosky, M. R., Jourdain, J. M., and Wright, P. K.**, *"Skin Materials for Robotic Fingers"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 1649-1654, 1987.*

35. **Cutkosky, M. R.**, *"Friction, Stability, and the Design of Robotic Fingers"*, International Journal of Robotics Research, *Vol. 5, No. 4, pp. 20-37, 1986.*

36. **Dario, P., Bicchi, A., Vivaldi, F., Pinotti, P. C.**, *"Tendon Actuated Exploratory Finger with Polymeric Skin-like Tactile Sensor"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 701-706, 1985.*

37. **Denavit, J. and Hartenberg, R. S.**, *"A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices"*, Journal of Applied Mechanics, Transactions of the ASME, *Vol. 77, No. Series E, pp. 215-221, June 1955.*

38. **Donald, B. R.**, *"Motion Planning with Six Degrees of Freedom"*, **AI-TR-791**, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, *1985.*

39. **Donald, B. R.**, *"Error Detection and Recovery for Robot Motion Planning with Uncertainty"*, Ph. D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *1987.*

40. **Driels, M. R.**, *"Pose Estimation using Tactile Sensor Data for Assembly Operations"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 1255-1261, 1986.*

41. **Ellis, R. E.**, *"Acquiring Tactile Data for the Recognition of Planar Objects"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 1799-1805, 1987.*

42. **Erdmann, M., and Lozano-Pérez, T.**, *"On Multiple Moving Objects"*, **AIM No. 883**, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, *May, 1986.*

43. **Erdmann, M. E.,** *"On Motion Planning with Uncertainty"*, S. M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *1984.*

44. **Fearing, R. S.,** *"Simplified Grasping and Manipulation with Dexterous Robot Hands"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 188-195, 1986.*

45. **Fearing, R. S.,** *"Some Experiments with Tactile Sensing during Grasping"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 1637-1643, 1987.*

46. **Fearing, R. S.,** *"Touch Processing for Determining a Stable Grasp"*, S. M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *September 1983.*

47. **Grahn, A. R. and Astle, L.,** *"Robotic Ultrasonic Force Sensor Arrays"*, Robot Sensors, Volume 2 - Tactile and Non-Vision, Springer-Verlag, *pp. 297-315, 1986.*

48. **Grimson, W. E. L., and Lozano-Pérez,** *"Model-based Recognition and Localization from Sparse Range or Tactile Data"*, International Journal of Robotics Research, *Vol. 3, No. 3, pp. 3-35, 1984.*

49. **Hanafusa, H., and Asada, H.,** *"An Adaptive Control of Robot Hand Equipped with Pneumatic Proximity Sensors"*, Proc. 6'th International Symposium on Industrial Robots, *pp. D4-31-42, 1976.*

50. **Hanafusa, H., and Asada, H.,** *"Stable Prehension by a Robot Hand with Elastic Fingers"*, Proc. 7'th International Symposium on Industrial Robots, *pp. 361-368, 1977.*

51. **Harmon, L. D.,** *"Automated Tactile Sensing"*, The International Journal of Robotics Research, *Vol. 1, No. 2, pp. 3-32, 1982.*

52. **Harmon, L. D.,** *"Touch-Sensing Technology: A Review"*, Tech. Rep. MSR 80-83, Society of Manufacturing Engineers, *1980.*

53. **Harmon, L. D.,** *"Automated Touch Sensing: A Brief Perspective and Several New Approaches"*, Proc. 1'st IEEE Computer Society International Conference on Robotics, *pp. 326-331, 1984.*

54. **Harmon, L. D.**, *"Tactile Sensing for Robots"*, Recent Advances in Robotics, Wiley and Sons, *pp. 389-421, 1985.*

55. **Hogan, N**, *"Impedance Control: An Approach to Manipulation: Part I - Theory"*, ASME Journal of Dynamic Systems, Measurement, and Control, *Vol. 107, pp. 1-7, March 1985.*

56. **Hogan, N**, *"Impedance Control: An Approach to Manipulation: Part II - Implementation"*, ASME Journal of Dynamic Systems, Measurement, and Control, *Vol. 107, pp. 8-16, March 1985.*

57. **Hogan, N**, *"Impedance Control: An Approach to Manipulation: Part III - Applications"*, ASME Journal of Dynamic Systems, Measurement, and Control, *Vol. 107, pp. 17-24, March 1985.*

58. **Hovannessian, S. H., and Pipes, L. A.**, *"Digital Computer Methods in Engineering "*, McGraw-Hill, *1969.*

59. **Hollerbach, J. M.**, *"Computers, Brains and the Control of Movement"*, Trends in NeuroSciences, *Vol. 5, No. 6, pp. 189-192, June 1982.*

60. **Hollerbach, J. M., and Suh, K. C.**, *"Redundancy Resolution of Manipulators Through Torque Optimization"*, Proc. of the 1985 IEEE International Conference on Robotics and Automation, *pp. 1016-1022, March 1985.*

61. **Hollerbach, J. M., Narasimhan, S., Wood, J. E.**, *"Finger Force Computation Without the Grip Jacobian"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 871-875, 1986.*

62. **Horn, B. K. P.** , *"Machine Vision"*, MIT Press, Cambridge, *1986.*

63. **Iberall, T.**, *"The Nature of Human Prehension: Three Dexterous Hands in One"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 396-401, 1987.*

64. **Jacobsen, S. C., Knutti, D. F., Biggers, K. B., Iversen, E. K., and Wood, J. E.**, *"An Electropneumatic Actuation System for the Utah/MIT Dextrous Hand"*, Theory and Practice of Robots and Manipulators, Proceedings of RoManSy '84: the

Fifth CISM-IFToMM Symposium, MIT Press, Cambridge, Mass, *pp. 271-280, 1985.*

65. **Jacobsen, S. C., Iversen, E. K, Knutti, D. F., Johnson, R. T, Biggers, K. B.,** *"Design of the Utah-MIT Dexterous Hand"*, Proc. IEEE International Conference on Robotics and Automation, *April 7-10, 1986.*

66. **Jacobsen, S. C., Wood, J. E., Knutti, D. F., Biggers, K. B.,** *"The Utah/MIT Dexterous Hand: Work in Progress"*, International Journal of Robotics Research, *Vol. 3, No. 4, pp. 21-50, 1984.*

67. **Jameson, J. W.,** *"Analytic Techniques for Automated Grasp"*, Ph. D. Thesis, Department of Mechanical Engineering, Stanford University, *June 1985.*

68. **Kato Ichiro,** *"Mechanical Hands Illustrated"*, Hemisphere Publishing Corporation, New York, *1982..*

69. **Keller, A. D., Taylor, C. C., Zahm, V.,** *"Studies to Determine the Functional Requirements for Hand and Arm Prosthesis"*, Dept. of Engg. UCLA, *1947.*

70. **Kemper, A., and Wallrath, M.,** *"An Analysis of Geometric Modelling in Database Systems"*, Computing Surveys, *Vol. 19, No. 1, pp. 47-91, March 1987.*

71. **Kerr, J.,** *"Analysis of Multifingered Hands"*, Ph. D. Thesis, Department of Mechanical Engineering, Stanford University, *1985.*

72. **Kerr, J., Roth, B.,** *"Analysis of Multifingered Hands"*, International Journal of Robotics Research, *Vol. 4, No. 4, pp. 3-17, 1985.*

73. **Kerr, J., Roth, B.,** *"Special Grasping Configurations with Dextrous Hands"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 1361-1367, 1986.*

74. **Khalil, W., and Kleinfinger, J. F.,** *No. A New Geometric Notation for Open and Closed-Loop Robots*, Proc. IEEE International Conference on Robotics and Automation, *Vol. 2, pp. 1174-1179, April, 1986..*

75. **Klein, C., and Blaho, B. E.,** *"Dexterity Measures for the Design and Control of Kinematically Redundant Manipulator"*, International Journal of Robotics Research, *Vol. 6, No. 2, pp. 72-83, Summer 1987..*

76. **Kobayishi, H.**, *"Grasping and Manipulation of Objects by Articulated Hands"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 1514-1519, 1986.*

77. **Korein, J. U., Maier, G. E., Taylor, R. H., and Durfee, L. F.**, *"A Configurable System for Automation Programming and Control"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 1871-1877, 1986.*

78. **Lathrop, R.**, *"Parallelism in Arms and Legs"*, S. M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *1982.*

79. **Lee, C. S. G.**, *"Robot Arm Dynamics"*, Tutorial on Robotics, IEEE Computer Society, *pp. 93-101, 1984.*

80. **Lee, C. S. G.**, *"Robot Arm Kinematics"*, Tutorial on Robotics, IEEE Computer Society, *pp. 47-65, 1984.*

81. **Lee, C. S. G., Mudge, T. N., Turney, J. L.**, *"Hierarchical Control Structure Using Special Purpose Processors for the Control of Robot Arms"*, Proceedings of the 1982 Pattern Recognition and Image Processing Conference, IEEE, *pp. 634-640, 1982.*

82. **Liegeois, A.** , *"Automatic Supervisory Control for the Configuration and Behavior of Multibody Mechanisms"*, IEEE Transactions on System, Man, and Cybernetics, *Vol. SMC-7, No. 12, pp. 842-868, 1977.*

83. **Loucks, C. S., Johnson, V. J., Boissiere, P. T., Starr, G. P., Steele, J. P. H.**, *"Modeling and Control of the Stanford/JPL Hand"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 573-578, 1987.*

84. **Lozano-Pérez, T.**, *"Spatial Planning: A Configuration Space Approach"*, IEEE Transactions on Computers, *Vol. C-32, No. 2, pp. 108-120, February, 1983.*

85. **Lozano-Pérez, T.**, *"The Design of a Mechanical Assembly System"*, S. M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *1976.*

86. **Lozano-Pérez, T.**, *"Robot Programming"*, Proceedings of the IEEE, *Vol. 71, No. 7, pp. 821-841, July, 1983.*.

87. **Lozano-Pérez, T., Mason, M. T., and Taylor, R. H.**, *"Automatic Synthesis of Fine-Motion Strategies for Robots"*, International Journal of Robotics Research, *Vol. 3, No. 1, 1984.*

88. **Lozano-Pérez, T.**, *"A Simple Motion Planning Algorithm for General Robot Manipulators"*, **AIM No. 896**, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, *June, 1986.*.

89. **Lozano-Pérez, T.**, *"Handey: A Robot System that Recognizes, Plans and Manipulates"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 843-849, 1987.*

90. **Luo, R. C., Grande, D.**, *"Servo-Controlled Gripper with Sensors for Flexible Assembly"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 451-460, 1984.*

91. **Luo, R. C., Lin, Min-Hsiung, and Scherp, R. S.**, *"The Issues and Approaches of a Robot Multi-Sensor Integration"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 1941-1946, 1987.*

92. **Maciejewski, A. A., and Klein, C. A.**, *"Obstacle Avoidance for Kinematically Redundant Manipulators in Dynamically Varying Environments"*, International Journal of Robotics Research, *Vol. 4, No. 3, pp. 109-117, 1985.*

93. **Maples, J. A., Becker, J. J.**, *"Experiments in Force Control of Robotic Manipulators"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 695-702, 1986.*

94. **Mason, M. T.**, *"Compliance and Force Control for Computer Controlled Manipulators"*, IEEE Transactions on System, Man and Cybernetics, *Vol. SMC-11, pp. 418-432, 1981.*

95. **Mason, M. T.**, *"Manipulator Grasping and Pushing Operations"*, **AI-TR-690**, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, *June 1982.*

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

97. **Mason, M. T.**, *"Manipulator Grasping and Pushing Operations"*, Ph. D. Thesis, Dept. of Electrical Engg. and Computer Science, Massachusetts Institute of Technology, *1985.*

98. **Montana, D. J.**, *"Tactile Sensing and the Kinematics of Contact"*, Ph. D. Thesis, Division of Applied Sciences, Harvard University, *1986.*

99. **Mujtaba, M. S.**, *"Motion Sequencing of Manipulators"*, STAN-CS-82-917, Dept. of Computer Science, Stanford University, *1982.*

100. **Nakamura, Y., Hanafusa, H., Ueno, N.**, *"A Piezoelectric Film Sensor for Robotic End-Effectors"*, Robot Sensors, Volume 2 - Tactile and Non-Vision, Springer-Verlag, *pp. 247-257, 1986.*

101. **Narasimhan, S., Siegel, D. M., Jones, S. A.**, *"Controlling the Utah-MIT Hand"*, Proc. of SPIE symposium on Advances in Intelligent Robotic systems, *Fall, 1986.*

102. **Nguyen, V.**, *"Constructing Stable, Force-Closure Grasps"*, S. M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *1986.*

103. **Nguyen, V.**, *"The Synthesis of Stable Grasps in the Plane"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 884-889, 1986.*

104. **Nguyen, V.**, *"Constructing Force-Closure Grasps"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 1368-1373, 1986.*

105. **Nguyen, V.**, *"Constructing Stable Grasps in 3-D"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 234-239, 1987.*

106. **Nguyen, V.**, *"Constructing Force-Closure Grasps in 3-D"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 240-245, 1987.*

107. **Ogata, K.**, *"Discrete-Time Control Systems"*, Prentics Hall, *1987.*

108. **Ohwovoriole, E.**, *"On the Total Degree of Freedom of Planar Bodies with Direct Contact"*, American Society of Mechanical Engineers, *Vol. 84-DET, No. 22, pp. 1-6, 1984.*

109. **Ohwovoriole, M. S.**, *"An Extension of Screw Theory and Its Application to the Automation of Industrial Assemblies"*, Ph. D. thesis, Department of Mechanical Engineering, Stanford University, *April 1980.*

110. **Okada, T.**, *"Object Handling system for Manual Industry"*, IEEE Transactions on System, Man and Cybernetics, *Vol. SMC 9, No. 2, pp. 79-89, February 1979.*

111. **Okada, T.**, *"Computer Control of Multijointed Finger System for Precise Object Handling"*, IEEE Transactions on System, Man and Cybernetics, *Vol. SMC 12, pp. 289-299, 1982.*

112. **Okada, T., and Kanade, T.**, *"Appropriate Lengths Between Phalanges of Multi-jointed Fingers for Stable Grasping"*, CMU-RI-TR-83-13, Robotics Institute, Carnegie-Mellon University, *22 July 1983.*

113. **Okada, T., Tsuchiya, S.**, *"Object Recognition by Grasping"*, Pattern Recognition, *Vol. 9, pp. 111-119, 1977.*

114. **Palm, W. J., Datseris, P.**, *"Pose Seeking Algorithms for the Control of Dexterous Robot Hands"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 582-587, 1985.*

115. **Paul, B.**, *"A Systems Approach to the Torque Control of a Permanent Magnet Brushless Motor"*, S. M. Thesis, Department of Mechanical Engineering, Massachusetts Institute of Technology, *1987.*

116. **Paul, R., and Shimano, B.**, *"Compliance and Control"*, Proceedings of the Joint Automatic Control Conference, The American Society of Mechanical Engineers, *pp. 694-699, 1976.*

117. **Paul, R. P., and Zhang, H.**, *"Design of a Robot Force/Motion Server"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 1878-1883, 1986.*

118. **Paul, R. P.**, *"Robot Manipulators: Mathematics, Programming and Control"*, MIT Press, Cambridge, Massachusetts, *1981.*

119. **Pieper, D. L.**, *No. The Kinematics of Manipulators Under Force Control*, Ph. D. Thesis, Computer Science Department, Stanford University, *October 1968.*

120. **Raibert, M. H., Craig, J. J.,** *"Hybrid Position/Force Control of Manipulators"*, ASME Journal of Dynamic Systems, Measurement, and Control, *Vol. 102, pp. 126-133, June 1981.*

121. **Roberts, R. K., Paul, R. P., and Hillberry, B. M.,** *"The Effect of Wrist Force Sensor Stiffness on the Control of Robot Manipulators"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 269-274, 1984.*

122. **Roth, B.,** *"Overview on Advanced Robotics: Manipulation"*, 1985 ICAR, *pp. 559-570, 1985.*

123. **Salamin, E.,** *"Application of Quaternions to Computation with Rotations"*, Internal Working Paper, A. I. Laboratory, Stanford University, *1979.*

124. **Salisbury, J. K.,** *"Kinematic and Force Analysis of Articulated Hands"*, Ph. D. Thesis, Department of Mechanical Engineering, Stanford University., *July, 1982.*

125. **Salisbury, J. K., and Craig, J. J.,** *"Articulated hands: Force Control and Kinematic Issues"*, International Journal of Robotics Research, *Vol. 3, No. 4, pp. 4-17, 1982.*

126. **Salisbury, J. K., and Abramowitz, J. D.,** *"Design and Control of a Redundant Mechanism for Small Motion"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 323-328, March, 1985.*

127. **Seering, W. P.,** *"Robotics and Manufacturing - A Perspective"*, The First International Symposium on Robotics Research, *pp. 973-983, 1984.*

128. **Schlesinger, G.,** *"Der Mechanische Aufbau der Kunstlichen Glieder"*, Ersatuglider und Arbeitshilfen, M. Borchardt et al., Springer, *1919.*

129. **Sheth, P. N. and Uicker, J. J.,** *"A Generalized Symbolic Notation for Mechanisms"*, Journal of Engineering for Industry, Transactions of the ASME, *Vol. 93, No. Series B, No 1., pp. 102-112, Feb 1971.*

130. **Sheth, P. N. and Uicker, J. J. Jr.,** *"IMP (Integrated mechanisms program) - A Computer Aided Design Analysis System for Mechanisms and Linkage"*, Journal of Engineering for Industry, Transactions of the ASME, *Vol. 94, pp. 454-464, May 1972.*

131. **Shimano, B., and Roth, B.,** *"On Force Sensing Information and Its Use in Controlling Manipulators"*, Proceedings of the Eighth Industrial Symposium on Industrial Robots, IFC Publications Limited, *pp. 119-126*, Washington, D.C,

132. **Siegel, D. M.,** *"Contact Sensors for Dexterous Robotic Hands"*, S. M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *1986.*

133. **Siegel, D. M., Narasimhan, S., Hollerbach, J. M, Gerpheide, G. E., Kriegman, D.,** *"Computational Architecture for the Utah-MIT hand"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 918-925, 1985.*

134. **Siegel, D. M., Garabieta, I., Hollerbach, J. M.,** *"A Capacitive Based Tactile Sensors"*, SPIE Conf. Intelligent Robots and Computer Vision, *Sept. 1985.*

135. **Siegel, D. M., and Simmons, L.,** *"A Thermal Based Sensor System"*, SME Sensors '85 Conference, Detroit, MI, *November 1985.*

136. **Skinner, F.,** *"Designing a Multiple Prehension Manipulator"*, Mechanical Engineering, *pp. 30-37, September 1975..*

137. **Speeter, T. H.,** *"Analysis and Control of Robotic Manipulation"*, Ph. D. Thesis, Department of Biomedical Engineering, Case Western Reserve University, *1987.*

138. **Taylor, R. H.,** *"Planning and Execution of Straight-Line Manipulator Trajectories"*, IBM Journal of Research and Development, *Vol. 23, pp. 424-436, 1979.*

139. **Tournassoud, P., Lozano-Pérez, T., Mazer, E.,** *"Regrasping"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 1924-1928, 1987.*

140. **Toumi, Y. K., Ro, P. I.,** *"A Dual-Drive Design Concept for Enhancing the Micro Manipulation of Direct-Drive Arms"*, Proc. of The Winter Annual Meeting of the ASME, *pp. 115-121, December, 1986.*

141. **Tsai, L. W., and Morgan, A. P.,** *"Solving the Kinematics of the Most General Six and Five Degree of Freedom Manipulators by Continuation Methods"*, The American Society of Mechanical Engineers, *Vol. ASME-84-DET-20, pp. 1-12, December, 1984.*

142. **Tubiana, R.**, *"The Architecture and Functions of the Hand"*, The Hand, *Vol. 1*, W. B. Saunders and Co., *pp. 19-93, 1981.*

143. **Uicker, J. J. Jr., Denavit, J., and Hartenburg, R. S.**, *"An Iterative Method for the Displacement Analysis of Spatial Mechanisms"*, Transactions of the ASME, Journal of Applied Mechanics, *Vol. 86, No. Series E, No. 2, pp. 309-314, June 1964.*

144. **Venkataraman, S. T., and Djaferis, T. E.**, *"Multivariable Feedback Control of the JPL/Stanford Hand"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 77-82, 1987.*

145. **Whitney, D. E.**, *"Historical Perspective and State of the Art in Robot Force Control"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 262-268, 1985.*

146. **Whitney, D. E.**, *"Force Feedback Control of Manipulator Fine Motions"*, Transactions of ASME, Journal of Dynamic Systems, Measurement and Control, The American Society of Mechanical Engineers, *pp. 91-97, June 1977.*

147. **Whitney, D. E.**, *"Resolved Motion Rate Control of Manipulators and Human Prostheses"*, IEEE Transactions Man-Machine Systems, MMS-10, *pp. 47-53, 1969.*

148. **Wu, C. H., and Paul, R. P.**, *"Manipulator Compliance Based on Joint Torque Control"*, Proc. IEEE Conference on Decision and Control, *pp. 88-94, 1980.*

149. **Wu, C. H., and Paul, R. P.**, *"Resolved Motion Force Control of Robot Manipulators"*, IEEE Transactions on Systems, Man and Cybernetics, *Vol. SMC-12, No. 3, pp. 266-275, June 1982.*

150. **Yoshikawa, T., Nagai, K.**, *"Manipulating and Grasping Forces in Manipulation by Multi-Fingered Hands"*, Proc. IEEE International Conference on Robotics and Automation, *pp. 1998-2004, 1987.*

151. **Yoshikawa, T.**, *"Manipulatability of Robotic Mechanisms"*, International Symposium of Robotics Research, *Vol. 2, 1984.*

152. **Yoshikawa, T.**, *"Analysis and Control of Robot Manipulators with Redundancy"*, The First International Symposium on Robotics Research, MIT Press, Cambridge, *pp. 735-747, 1984.*

This appendix deals with computing the forward kinematics of a single finger of the Utah-MIT hand.

The numberings of the various fingers and joints are indicated in Fig. A.1.

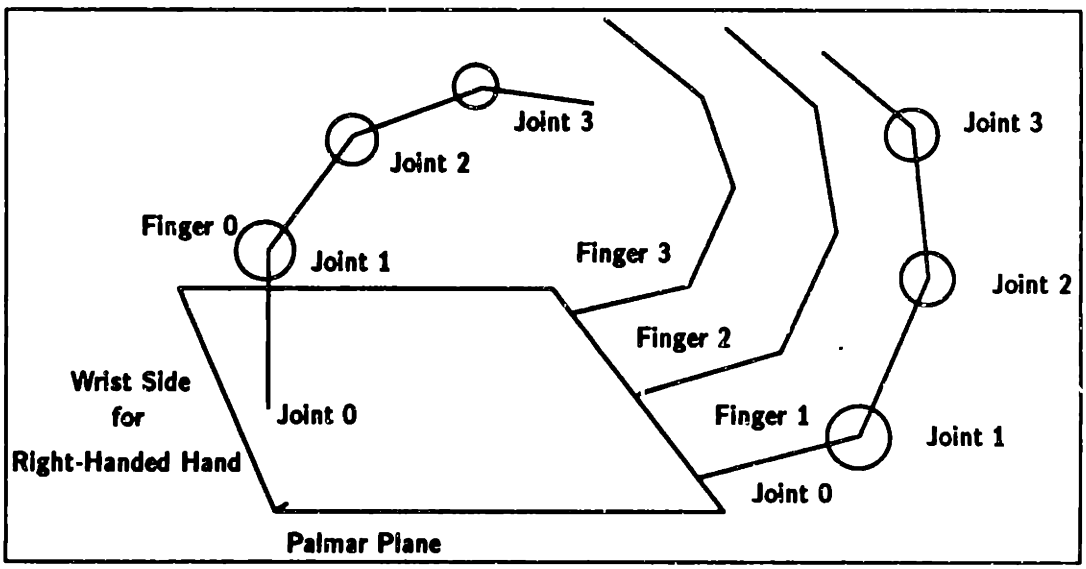


Figure A.1: Numbering of joints on the Utah-MIT hand

A picture of the finger geometry is included in Fig. A.2.

A.1 Co-ordinate transforms based on D-H matrices

The Denavit-Hartenburg notation can be easily applied to denote the various co-ordinate systems associated with each finger. In what follows, the z_i axes are aligned with the axis of rotation of joint i.

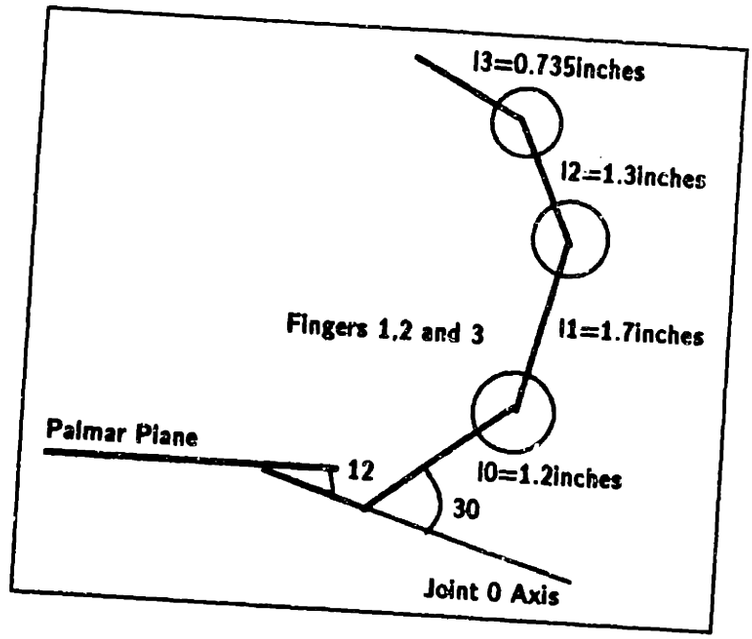


Figure A.2: Single finger - Non thumb

These parameters are included here even though the controller and the rest of the software use vector manipulations rather than matrix calculations. The popularity of the D-H matrices after the publication of Paul [1982] and Craig [1986] has made this notation more accessible and somewhat of a standard.

Let the co-ordinate system whose axes are defined by $[\mathbf{x_0}, \mathbf{y_0}, \mathbf{z_0}]$ denote a co-ordinate frame affixed to the palmar plane of the Hand. This co-ordinate system is the one relative to which all other co-ordinate frames will be expressed. This frame is chosen so that positive x and y values will be on the palmar plane and the z axis is perpendicular to the palmar plane, pointing upward from it.

Then the relation between this frame and the first joint axis frame of a non-thumb finger can be expressed by:

$$Trans(\mathbf{y_0}, r_p) \cdot Trans(\mathbf{z_0}, l_p tan(\phi_p) - h_p) \cdot Rot(\mathbf{y_0}, \pi/2 + \phi_p) \cdot Rot(\mathbf{z}', \pi/2)$$

where $\mathbf{z}'$ denotes the $\mathbf{z_0}$ axis *after* the rotation about $\mathbf{y_0}$ has been performed, ϕ_p denotes the angle made by the axis of rotation of joint 0 with the palmar plane (which is 12 degrees for non-thumb fingers), r_p denots the distance separting the joint 0 axes and l_p denotes the

length of the palmar plane and is equal to 1.05 inches (please refer to Fig. A.3).

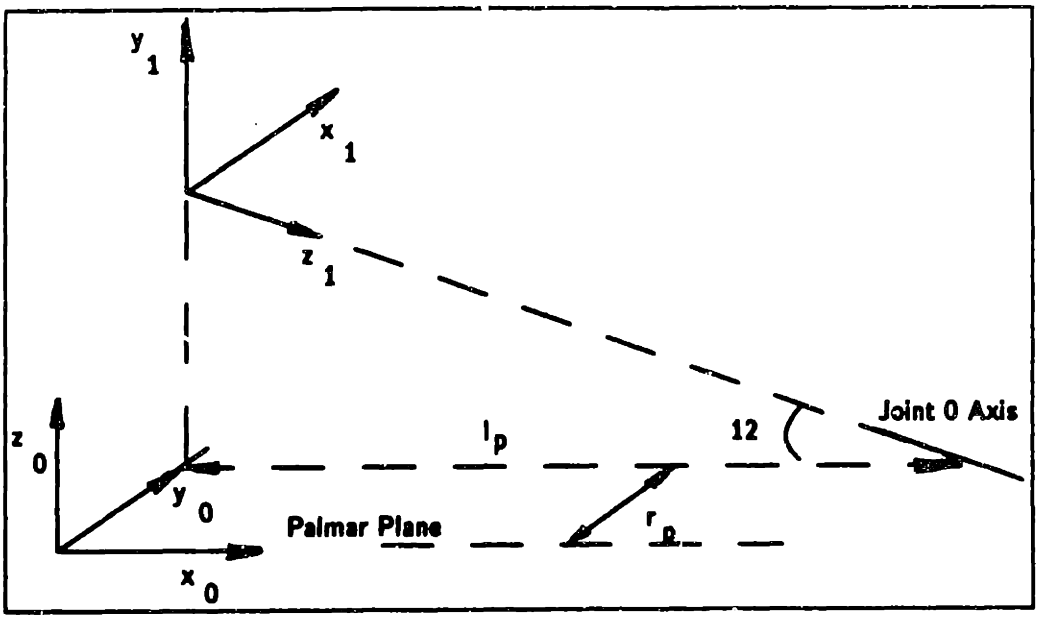


Figure A.3: The relation between the palm and the zero joint frame

$$ {}^{1}A_0 = \begin{bmatrix} 0 & sin(\phi_p) & cos(\phi_p) & 0 \\ 1 & 0 & 0 & r_p \\ 0 & cos(\phi_p) & -sin(\phi_p) & l_p tan(\phi_p) - h_p \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.1} $$

where r_p refers to the distance of the finger from the x_0 axis as measured along y_0 (which is 1.42 inches for finger 2, and 2.69 inches for finger 3), and l_p refers to the length of the palmar plane as mentioned above. h_p is the height of the palmar plane above the joint 0 axes which is equal to 0.95 inches.

Once the frame affixed to joint 1 has been determined the other frames associated with the finger joints can be determined by using the standard D-H matrix given by:

$$ {}^{i}A_{i+1} = \begin{bmatrix} C_i & -S_i C_\alpha & S_i S_\alpha & a_i C_i \\ S_i & C_i C_\alpha & -C_i S_\alpha & a_i S_i \\ 0 & S_\alpha & C_\alpha & d \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.2} $$

where C_i denotes $cos(\theta_i)$ and S_i denotes $sin(\theta_i)$.

The first link of the non-thumb fingers makes an angle of 30 degrees with the axis of rotation of the zeroeth joint of these fingers. This angle is denoted by ϕ_0. l_0 denotes the link length of this first link and is equal to 1.2 inches.

Using the above equation, we can write:

$$ {}^1A_2 = \begin{bmatrix} C_0 & 0 & -S_0 & l_0 sin(\phi_0) C_0 \\ S_0 & 0 & C_0 & l_0 sin(\phi_0) S_0 \\ 0 & -1 & 0 & \frac{l_p}{cos(\phi_p)} + l_0 cos(\phi_0) \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.3} $$

which can be derived from the fact that α_0 is $-\pi/2$, $a_0 = l_0 sin(\phi_0)$, and $d_0 = \frac{l_p}{cos(\phi_p)} + l_0 cos(\phi_0)$. The angle ϕ_0 denotes the constant angle made by the link between the joint 1 axis, and is equal to 30 degrees.

The remaining three axes are simple revolute axes. All of these three axes are parallel and are perpendicular to the plane of finger movement called the *operational plane*. Their A matrices can be seen to be:

$$ {}^2A_3 = \begin{bmatrix} C_1 & -S_1 & 0 & l_1 C_1 \\ S_1 & C_1 & 0 & l_1 S_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^3A_4 = \begin{bmatrix} C_2 & -S_2 & 0 & l_2 C_2 \\ S_2 & C_2 & 0 & l_2 S_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.4} $$

$$ {}^4A_5 = \begin{bmatrix} C_3 & -S_3 & 0 & l_3 C_3 \\ S_3 & C_3 & 0 & l_3 S_3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

where l_1 is 1.70 inches, l_2 is 1.30 inches, and l_3 is the distance from the contact point to

the last joint's axis of rotation, which is 0.735 inches.

From the last of the above equations, it can be seen that the contact point between a grasped object and the finger tip is assumed to be a constant given by l_3. In practice, this is only an approximation. In fact the contact point ought to be expressed as a vector in the last co-ordinate frame of the distal joint. This would be possible for example when tactile sensors are mounted on the finger tip enabling the acquisition of such information.

Table A.1: D-H Parameters for Non-Thumb Fingers

Joint+1	d_i	a_i	α
1	$\frac{l_p}{cos(\phi_p)} + l_0\, cos(\phi_0)$	$l_0\, sin(\phi_0)$	$-\pi/2$
2	0	l_1	0
3	0	l_2	0
4	0	l_3	0

The A matrices given above make the solution of the forward kinematics problem easy. A summary of the D-H parameters for the non-thumb fingers is given in Table A.1. The final mapping is given by Equation A.5.

$$
\begin{aligned}
{}^0A_{5_{00}} &= sin(\phi_p)\, S_0\, C_{123} - cos(\phi_p)\, S_{123} \\
{}^0A_{5_{01}} &= -sin(\phi_p)\, S_0\, S_{123} - cos(\phi_p)\, C_{123} \\
{}^0A_{5_{02}} &= sin(\phi_p)\, C_0 \\
{}^0A_{5_{03}} &= cos(\phi_p)\left[-l_3\, S_{123} - l_2\, S_{12} - l_1\, S_1 + \frac{l_p}{cos(\phi_p)} + l_0\, cos(\phi_0)\right] \\
&\quad + sin(\phi_p)\,[S_0\,(l_3\, C_{123} + l_2\, C_{12} + l_1\, C_1) + l_0\, sin(\phi_0)\, S_0] \\
{}^0A_{5_{10}} &= C_0\, C_{123} \\
{}^0A_{5_{11}} &= -C_0\, S_{123} \\
{}^0A_{5_{12}} &= -S_0 \\
{}^0A_{5_{13}} &= C_0\,[l_3\, C_{123} + l_2\, C_{12} + l_1\, C_1] + l_0\, sin(\phi_0)\, C_0 + r_p \\
{}^0A_{5_{20}} &= sin(\phi_p)\, S_{123} + cos(\phi_p)\, S_0\, C_{123} \\
{}^0A_{5_{21}} &= sin(\phi_p)\, C_{123} - cos(\phi_p)\, S_0\, S_{123} \\
{}^0A_{5_{22}} &= cos(\phi_p)\, C_0 \\
{}^0A_{5_{23}} &= -sin(\phi_p)\left[-l_3\, S_{123} - l_2\, S_{12} - l_1\, S_1 + \frac{l_p}{cos(\phi_p)} + l_0\, cos(\phi_0)\right] \\
&\quad + cos(\phi_p)\,[S_0\,(l_3\, C_{123} + l_2\, C_{12} + l_1\, C_1) + l_0\, sin(\phi_0)\, S_0] - h_p + l_p\, tan(\phi_p) \\
{}^0A_{5_{30}} &= 0 \\
{}^0A_{5_{31}} &= 0 \\
{}^0A_{5_{32}} &= 0 \\
{}^0A_{5_{33}} &= 1
\end{aligned}
\tag{A.5}
$$

where S_{ij} refers to $sin(\theta_i + \theta_j)$, C_{ij} refers to $cos(\theta_i + \theta_j)$, S_{ijk} refers to $sin(\theta_i + \theta_j + \theta_k)$ and C_{ijk} refers to $cos(\theta_i + \theta_j + \theta_k)$. (Note that the numbering scheme is different from the conventional usage of the D-H parameters since we have introduced an extra constant ${}^0A_1$ matrix.

Computationally, it would take 26M+16A+8T operations to compute all elements of the forward kinematics for a non-thumb finger. [1] If we are merely interested in computing the cartesian positions of the fingertips, it would take 12M+14A+8T to perform the above computations. For three fingers, this number works out to 78M+48A+24T operations.

[1] xM+yA+zT refers to x multiplies, y additions and z transcendental function lookups.

A.2 The thumb frames

The Utah-MIT hand has a four degree of freedom thumb whose co-ordinate frames are different from the other non-thumb fingers. The main differences are caused by the fact that the zero'th joint on the thumb rotates about an axis that is parallel to the palmar plane and is aligned along with the x_0 axis. Joint 1 of the thumb is also only 0.4 inches above the joint 0 axis. Furthermore, this axis is perpendicular to the joint 0 axis.

This means that for the thumb the relation between the first two frames can be expressed as

$$Trans(\mathbf{y_0}, r_p) \cdot Rot(\mathbf{y_0}, \pi/2) \cdot Rot(\mathbf{z}', \pi)$$

where r_p for the thumb is 0.695 inches, and $\mathbf{z}'$ denotes the $\mathbf{z_0}$ axis after the rotation about the $\mathbf{y_0}$ axis.

Expanding the matrices, we get:

$$ {}^0A_1 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & r_p \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.6}$$

The derivation of the ${}^2A_1$ frame is relatively straightforward. With a_0 set to a constant h_0 (0.45inches) and α_0 set to $-\pi/2$ and using Equation. A.2 we get:

$$ {}^1A_2 = \begin{bmatrix} C_0 & 0 & -S_0 & h_0C_0 \\ S_0 & 0 & C_0 & h_0S_0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.7}$$

The orientation of the different co-ordinate frames are illustrated in Fig. A.4.

Since the thumb's three distal joints have the same structure as the other non-thumb fingers the other A matrices turn to be the same as indicated above for the non-thumb fingers.

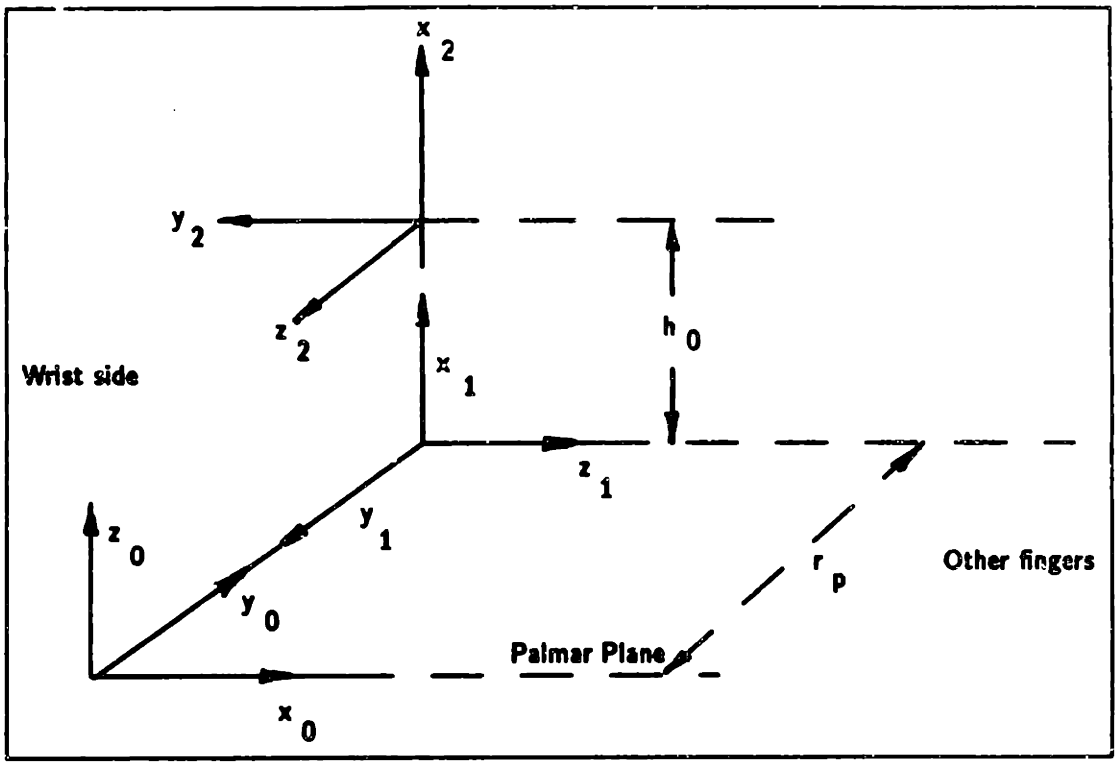


Figure A.4: The first two frames for the thumb

Therefore the remaining transforms can all be lumped together as:

$$ {}^{2}A_5 = \begin{bmatrix} C_{123} & -S_{123} & 0 & l_3C_{123} + l_2C_{12} + l_1C_1 \\ S_{123} & C_{123} & 0 & l_3S_{123} + l_2S_{12} + l_1S_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.8} $$

The D-H parameters for the thumb are summarized in Table.A.2.

Table A.2: D-H Parameters for the Thumb Fingers

Joint+1	d_i	a_i	α
1	0	h_0	$-\pi/2$
2	0	l_1	0
3	0	l_2	0
4	0	l_3	0

Multiplying the individual transforms out, we get:

$$
\begin{aligned}
{}^0A_{5_{00}} &= -S_{123} \\
{}^0A_{5_{01}} &= -C_{123} \\
{}^0A_{5_{02}} &= 0 \\
{}^0A_{5_{03}} &= -l_3\,S_{123} - l_2\,S_{12} - l_1\,S_1 \\
{}^0A_{5_{10}} &= -S_0\,C_{123} \\
{}^0A_{5_{11}} &= S_0\,S_{123} \\
{}^0A_{5_{12}} &= -C_0 \\
{}^0A_{5_{13}} &= -S_0\,(l_3\,C_{123} + l_2\,C_{12} + l_1\,C_1) - h_0\,S_0 + r_p \\
{}^0A_{5_{20}} &= C_0\,C_{123} \\
{}^0A_{5_{21}} &= -C_0\,S_{123} \\
{}^0A_{5_{22}} &= -S_0 \\
{}^0A_{5_{23}} &= C_0\,(l_3\,C_{123} + l_2\,C_{12} + l_1\,C_1) + h_0\,C_0 \\
{}^0A_{5_{30}} &= 0 \\
{}^0A_{5_{31}} &= 0 \\
{}^0A_{5_{32}} &= 0 \\
{}^0A_{5_{33}} &= 1
\end{aligned}
\tag{A.9}
$$

Computationally, to compute all elements of the frame for the thumb, it takes only 12M+9A+8T operations. Computing just the cartesian co-ordinates of the tip of the thumb can be done in 8M+9A+8T operations.

The entire forward kinematics for the Utah-MIT hand can therefore be done in 94M+63A+32T operations.

This appendix is intended to give the reader a brief glimpse into the complexity and power of the CONDOR system. It is not intended as a comprehensive programmer's manual, but is intended merely to provide an index.

Since the system is rather large, the following sections have been broken up into functional categories. I have included a one line description of the functions where I've felt it was necessary. Most of the functions implemented follow standard Unix semantics. I have avoided describing the more complex functions provided like the ptrace emulation library and modifications to the Unix kernel we have had to make hoping to provide a view of what the programmer's interface to the CONDOR system would look like without going into the details of how such a functionality is implemented internally.

B.1 Programs

Like any development environment, the CONDOR system provides a number of programs to assist the programmer in his task. In a real time development environment the important programs are those meant for plotting collected data, programs that allow downloading and execution of other programs, diagnostic programs that debug hardware devices, and debuggers that allow the user to debug his real time programs.

The CONDOR has a full complement of all these types of programs.

1. *CONDOR* – This forms the default user interface that allows users to connect up to the slave microprocessors, download and execute programs on them, and debug an errant program. It uses the X window system to provide a sophisticated user interface.

2. *xplot* – This is the plotter program that plots collected data and has a mode unique to the hand control software wherein it allows monitoring any of the hand control

variables in real time.

3. *conf* – This program shows the current configuration of a system. It uses the standard configuration file for a system that the CONDOR uses, and verifies all the hardware devices mentioned therein do exist at the right addresses.

4. *dl68* – This is a standalone downloader to the *CONDOR* processors. This can function across serial lines, dma devices and across the VME-VME bus to bus adaptor.

5. *icc* – This is a top level interface to the compiler and linker. It runs on the development host and compiles code to run on the slave microprocessors.

6. *raw* – This program provides a raw connection via a serial line to a slave microprocessor. It is intended mainly for bootstrapping the system until the *condor* program can be run.

7. *mraw* – This program is like the *raw* program and allows the user to connect up to any processor using the hve bus to bus adaptor. Using this multiple users can all connect up to and be using, different processors in the CONDOR system.

8. *igdb* – The modified version of the debugger GDB used by the CONDOR system.

B.2 Math Library

sincos(*theta, ptrcos, ptrsin*) *Function*

Calculate both the *sin* and *cos* of the given angle.

Ssincos(*theta, ptrcos, ptrsin*) *Function*

Like the previous routine, but optimized for single precision floating point.

asin(*theta*) *Function*

Returns the arc sine of its argument *theta*.

acos(*theta*) *Function*

Returns the arc cosine of its argument *theta*.

atan(*theta*) *Function*

Returns the arc tangent of its argument *theta*.

B.2.1 Vectors and Matrices

aset(*a, i, j, val*) *Function*

Does what the old lisp function with the same name used to do. It sets element referred to by `a[i,j]` to the float `val`.

aref(*a, i, j*) *Function*

This returns the element referred to by `a[i,j]`.

array_add(*a, b, c*) *Function*

This adds two matrices together, if they are of the same shape.

array_sub(*a, b, c*) *Function*

Subtracts array *b* from *a*, if they are of the same shape.

array_scalar_add(*a, scalarvalue, b*) *Function*

This adds a scalar value to every element of the specified array a.

array_scalar_sub(*a, scalarvalue, b*) *Function*

This subtracts a scalar value from every element of the specified array.

array_scalar_mul(*a, scalarvalue, b*) *Function*

This multiplies every element of the specified array a by the specified scalar value.

array_scalar_div(*a, scalarvalue, b*) *Function*

This divides every element of the specified array a by the specified scalar value.

array_scalar_shift_add(*a, scalar, b*) *Function*

In short doing a

```
array_scalar_shift_add(a, 3.0, b);
```

is equivalent to writing (in pseudo-code)

```
for all_elements_of b
   b[i] = b[i] + a[i] + 3.0;
```

Both arrays should be of the same shape.

array_scalar_shift_sub(*a, scalar, b*) *Function*

This shifts the elements of array a, down instead of up as the add routine does.

array_scalar_shift_mul(*a, scalar, b*) *Function*

This is equivalent to the following:

```
for all_elements_of b
   b[i] = b[i] + a[i] * scalar;
```

array_scalar_shift_div(*a, scalar, b*) *Function*

This is equivalent to the following:

```
for all_elements_of b
   b[i] = b[i] + a[i] / scalar;
```

array_norm_one(*a*) *Function*

Computes the maximum column sum of the specified array (also known as the 1-norm.

array_norm_infinity(*a*) *Function*

Computes the maximum row sum of the specified array. (also known as the inf-norm.

array_norm_frobenius(*a*) *Function*

Computes and returns the frobenius norm of the given array.

array_multiply(*a*, *b*, *c*) *Function*

This is the normal matrix multiply routine.

array_multiply_new(*a*, *b*) *Function*

This routine returns a new array which is equal to a multiplied by b.

array_multiply_destructive(*a*, *b*) *Function*

This routine modifies b to be the array a times b.

array_invert(*a*, *b*) *Function*

The familiar matrix inversion routine.

array_multiple_solve(*a*, *b*, *c*) *Function*

This routine uses gaussian elimination with partial pivoting to solve the system of equations

represented by $Ax = B$. The matrix A is the first argument a. The matrix B is passed in as the second argument. It can have many columns. The solution matrix is returned in c or b destructively if c is null.

array_print(*a*) *Function*

Prints out the array.

array_identify(*a*) *Function*

Turns the square array into an identity matrix.

array_set(*a, value*) *Function*

Sets all the elements of the array to be the specified value.

array_set_subarray(*a, value, startrow, endrow, staricol, endcol*) *Function*

Sets a sub-array of an array to be the specified value.

array_zero(*a*) *Function*

Zeroes the array.

array_zero_subarray(*a, startrow, endrow, startcol, endcol*) *Function*

Zeroes a specified subarray of the specified array.

array_trace(*a*) *Function*

Computes and returns the trace of the matrix *a*.

array_copy(*a, b*) *Function*

This copies the specified array *a* into the array *b*.

array_transpose(*a, b*) *Function*

This transposes the array a into the array b.

array_transpose_new(*a*) *Function*

This creates and returns a new array which is the transpose of the argument array a.

array_solve_linear(*a, b, solution*) *Function*

This routine takes as input the array *a*, and the right hand side column vector in the specified array *b*. It solves the linear equation Ax = b, and returns the solution in the column vector *solution*. Uses the fast gaussian elimination with partial pivoting to find the solution if one exists.

array_row_interchange(*a, row, row1, from, to*) *Function*

array_column_interchange(*a, col, col1, from, to*) *Function*

These routines interchange the specified rows or columns of the arrays.

array_gem(*a, b*) *Function*

This routine performs gaussian elimination on the input array a. RHS is passed in as the second argument array *b*. This routine is destructive and modifies the input array *a* for speed.

array_input(*a*) *Function*

Takes an array as input.

B.2.2 Vectors

Vectors can be column or row vectors. They are created by the routines

```
new_column_vector(rows);
new_row_vector(cols);
new_vector(rows,cols);
new_3vector();
new_4vector();
new_6vector();
new_array(rows, cols);
make_3vector_from(x, y, z);
make_4vector_from(x, y, z, w);
```

vector_dot(*a*, *b*) *Function*

Returns the dot product of the two vectors.

vector_magnitude(*a*) *Function*

This computes and returns the magnitude of a given vector *a*.

vector_angle_between(*a*, *b*) *Function*

This computes the angle between the two column vectors passed in *a* and *b*.

vector_normalize(*a*) *Function*

Normalization function for vectors.

vector_cross3(*a*, *b*, *c*) *Function*

This routine computes the cross product of the two vectors a and b.

B.2.3 Homogenous Transforms

Homogenous routines are implemented using the array routines outlined above. These transforms can be created using

```
make_null_transform();
make_translation(vector);
make_translation_from(x, y, z);
make_rotation(vector, thru);
```

Transforms can be modified using

```
transform_set_translation_from(tfm, x, y, z);
transform_set_translation(tfm, vector);
transform_set_rotation(tfm, vector, thru);
transform_zero(tfm);
transform_zero_rotation(tfm);
transform_zero_translation(tfm);
transform_identify(tfm);
```

They can be copied with `transform_copy_new` and `transform_copy`, and printed out with `transform_print`.

Transforms can be composed and inverted with:

```
transform_compose_new(list_of_transforms);
transform_compose(resulttfm, list_of_transforms);
transform_invert_new(tfm);
transform_invert(tfm, result);
transform_invert_destructive(tfm);
```

A number of routines built on top of the homogenous transform library allow implementation of forward and inverse kinematics of robots, and 3-d graphics using the X window system.

B.3 Storage Management

malloc(*bytes*) *Function*

Allocates *bytes* units of storage.

free(*ptr*) *Function*

Frees the storage previously allocated.

sbrk(*incr*) *Function*

Gets a chunk of memory of size *incr* from the system.

realloc(*ptr, bytes*) *Function*

Reallocates the buffer of storage previously allocated to a larger/smaller size.

B.3.1 Examining and modifying memory locations

Reads a byte, word or a long from a specified address one can use the routines **memory_peekc**, **memory_peekw** and **memory_peekl**. To write a byte, word or long into a specified address one uses **memory_pokec**, **memory_pokew** and **memory_pokel**. Versions of these routines exist that will work across the bus to bus adaptor.

memory_size() *Function*

Indicate how much memory the system is currently configured with.

B.4 The I/O Package

Provides all the standard Unix *stdio* functions. Internally, the device interface implements the following system calls:

open(*name, flags, mode*) *Function*

Opens a file or a device named *name*. Devices currently supported include:

1. **:tty** – The Motorola 68681 serial chip on the Ironics 3273 system controller.
2. **:pty** – Pseudo terminal drivers. The flag argument to this device indicates which processor a pty should be opened to.
3. **:ptysun** – Pseudo terminal driver to the sun. Opening this device results in an **fd** that is connected to a terminal window on the Sun. Reading and writing from this fd, will correspond to doing i/o with the corresponding window on the sun. Notice that for this to work, an appropriate program that provides the pty service must be running on the Sun end.
4. **:magnon** – The magnon stepper motor controller boards.
5. **:mpp** – The motorola parallel port boards. (MVME304).
6. **:dt1401** – The data translation data acquisition boards.
7. **:adc** – The multibus a-d, d-a boards.
8. **:dt1406** – The data translation a-d boards for the vme.

dup(*oldd*) *Function*

Duplicates an existing file descriptor object.

creat(*name, mode*) *Function*

Creates a new file with the specified name and mode.

read(*dev, buf, count*) *Function*
Reads from the specified file descriptor into the specified buffer the specified count number of bytes.

write(*dev, buf, count*) *Function*
Writes to the specified file descriptor from the specified buffer the specified count number of bytes.

lseek(*dev, count, whence*) *Function*
This routine applicable to files open on the suns, moves a file pointer.

ioctl(*dev, code, arg*) *Function*
Specialized device specific operations on the specified file descriptor object.

The **stdio** calls supported include **fopen**, **fclose**, **fprintf**, **fscanf**, **fseek**, **rewind**, **getc**, **putc**, **fflush**, **fgets**, **fputs**, **fdopen**, **ungetc**, **getchar**, **putchar**, **gets**, **puts**, **printf** and **scanf**.

Specific devices like the parallel port device have more operations defined on them, since we chose not to overload the **ioctl** call as is customary within the Unix community.

B.4.1 Strings Library

A large number of functions are available for operating on conventional C ascii strings. These will not be detailed here, since their semantics are exactly the same as that of their Unix counterparts (do a **man string** for details). The functions that we support are **strcpy**, **strcpyn**, **strncpy**, **strlen**, **strcat**, **strncat**, **strcatn**, **strcmp**, **strncmp**, **strcmpn**, **strchr**, **strrchr**, **strpbrk**, **index**, **rindex**. In addition to these **bzero**, **bcopy** and **bcmp** are also supported, as are **atof**, **atoi**, **atol**, **atov**, **ecvt**, **fcvt**, **gcvt**, **modf** and **ldexp**. The standard man page documentation available for these functions applies for their **CONDOR** versions too.

B.5 Data Transfer routines

A flexible interface to the A/D and D/A converters is given by the following routines. These are implemented for all the A/D devices we currently support, so once such a device has been open'ed it becomes very use to actually use the device.

adc_set_gain(*fd, gain*) *Function*
This routine sets the gain on a particular converter.

adc_read_channel(*fd, channelnumber*) *Function*
This routine converts and reads the specified A/D channel.

adc_convert(*fd, start, count, buf*) *Function*
Meant for block conversions of a number of channels.

adc_fill_convert(*fd, channelno, count, buf*) *Function*
This routine fills up a buffer with values gotten from repeated conversions on the same channel.

adc_repeat_convert(*fd, channelno, count, buf*) *Function*
This routine fills up a buffer by repeatedly converting on the channels and performing an averaging function if needed.

adc_poll_convert(*fd, start, count, buf*) *Function*
The conversion is performed via polling instead of being interrupt driven.

adc_poll_read_channel(*fd, channel*) *Function*

The conversion is performed via polling on a single channel.

dac_write(*fd, start, no, buf*) *Function*
This writes to the D/A device the specified values.

B.6 Simple Real Time Tasking

To enable the user to specify a simple servo loop to be executed at a specified sampling rate the following functions are provided.

start_servo(*loop, rate*) *Function*
This function sets up a specified function to execute at a specified rate.

enable_servo() *Function*
This routine actually enables the running of a servo.

disable_servo() *Function*
Disables an enabled and running servo.

servo_status() *Function*
Meant for diagnostics this routine prints out the status of a servo.

servo_ramp(*start*) *Function*
This routine starts from a low servo rate and ramps up to a value where the servo can comfortably run.

stop_servo() *Function*
Used to stop a running servo.

set_servorate(*rate*) *Function*
One can modify the sampling rate, if one desires to do so, with this function.

protect_servo() *Function*
This prevents the servo from being interrupted by any routine.

unprotect_servo(*level*) *Function*
This makes the servo loop interruptible.

B.7 Dealing with multiple processors

The control programmer needs to be aware of the fact that his programs are running on a truly MIMD machine. The following functions provide a simple manner in which he can query or find out the status of this MIMD machine. The more flexible message passing system will be described later. The programmer who wishes to use the CONDOR system may never have to use these functions, since the user interface programs will provide most of the functionality he needs. However, if he wishes to write his own master control programs and user interfaces, they can be done with the following functions:

proc_presentp(*processor*) *Function*
This routine can be used to find out if a processor is existent in the system.

proc_runningp(*processor*) *Function*
This routine can be used to find out if a user program is currently running on a specified processors.

proc_any_runningp() *Function*
This can be used to find out if any processor is running a user program.

proc_print_status() *Function*
The status of all the processors in a system can be printed out by this routine.

imon_go(*processor, address***)** *Function*
This starts execution of a user program on a specified processor.

download_a_file(*filename, processor, offset***)** *Function*
This function downloads a file to the specified processor.

To perform block transfers between files, without using the stdio system one can also use the lower level functions `download_a_block`, `upload_into_file` and `upload`.

B.8 Command Parser Library - Input routines

Perhaps the area in which Unix needs some real work, is in its user interface. The following functions provide an interface which is easier to understand and use, and infinitely more powerful than the `stdio` functions.

exec_command(*parameter_list***)** *Function*
This routine essentially allows the specification of top level command loops. This has been implemented to run on normal ascii terminals as well as under the X window system. It provides emacs-like command line editing, command completions, lookups for partial matches and a sophisticated on-line help system. Other functions built on this interface allow users to define their own command and argument types.

B.8.1 Miscellaneous input routines

Parsing input from the user can be done with `read_int`, `read_valid_int`, `read_int_in_range`, `read_float`, `read_valid_float`, `read_float_in_range`, `read_double`, `read_valid_double`, `read_double_in_range` , `tty_gets`, `read_line` and other such routines.

The syntax for numbers used by these functions follows very closely the lisp machine syntax (which follows the common-lisp specification). This allows dynamically changing the input and output radix and other such useful functions.

Since all of the routines are built on top of the base input editor, all the routines provide input editing capability a-la emacs but on a line by line basis.

B.8.2 Window system functions

Part of the library includes a rather sophisticated library out of which programs like the *condor* user interface can be created. These functions rely on the C version of XLib, and the Sx Toolkit library to provide an interface that includes pop-up menus, command prompters, scrollable text interaction windows and so on. These functions are too numerous to mention even in passing here and are rather dependent on the X window system at this point.

B.9 Hash Tables

A generic hash table mechanism has been built for use by programmers (the primitive command decoder is an example of a control program that uses this library for efficiency).

htinit(*size*) *Function*
Initialize a hash table of the requested size.

htinstall(*key, table, data*) *Function*
Install a key and a data item into the specified hash table.

htlookup(*key, table*) *Function*

Performs a looks up using the key in the specified table.

htgetdata(*key, table*) *Function*

This gets the data item associated with the key in the specified table if present.

htmap(*table, function*) *Function*

This applies the function to all the elements in the specified hash table.

htstat() *Function*

Prints out statistics regarding hash table usage.

htdelete(*key, table*) *Function*

This deletes the association between the specified key and data item in the specified hash table.

hthash(*s, table*) *Function*

The internal hashing function. (This can be changed if the user so desires).

htclobber(*table*) *Function*

This function deletes the table and all the elements associated with it.

B.10 Buffer routines

The internal system routines use a buffer library that is available to other programmers too.

buffer_create(*no_elements, size*) *Function*
This routine creates a circular buffer.

buffer_put(*queue, object, dontwait*) *Function*
This adds an element to a circular buffer object.

buffer_get(*queue, dontwait*) *Function*
This deletes an element from a circular buffer object.

B.11 Tree library

The tree library operates on binary trees.

tree_create(*elementsize, predicate*) *Function*
This routine creates a tree that will allow comparsions to be made based on the specified predicate.

tree_free(*root*) *Function*
Frees up the storage associated with a tree.

tree_insert(*root, elt*) *Function*
Inserts an element into the tree.

tree_lookup(*root, elt*) *Function*
Looks up the specified element in the given tree.

tree_delete(*root, elt*) *Function*
This deletes an element from the tree.

tree_traverse(*root, way, function*) *Function*
This performs the requested traversal of the tree and applies the given function to all the elements that it encounters.

B.12 Small set package

The library uses bit vectors to implement small sets. (for example, such an abstraction is useful to control sets of joints since the number of joints is typically a small number).

set_emptyset() *Function*
This creates and returns a null set.

set_singleton(*i*) *Function*
This creates and returns a set that contains the specified single element.

set_add_element(*set, elt*) *Function*
Adds the specified element to the given set.

set_delete_element(*set, elt*) *Function*
This deletes the specified element from the set.

set_memberp(*set, elt*) *Function*
Tests for membership.

set_intersect(*set1, set2*) *Function*
Returns the intersection of two given sets.

set_union(*set1, set2*) *Function*
Returns the union of two given sets.

set_difference(*set1, set2*) *Function*
Computes and return the set difference of the two specified sets.

B.13 Message Passing routines

The message passing system provides the following routines, that are available on the Sun and on the slave microprocessors.

muse_init(*proc, mode*) *Function*
This initializes the message passing system. On the Sun, the mode argument specifies the mode of interaction to be used.

mbox_vector_print(*vector*) *Function*
Prints out the handler associated with a particular vector.

mbox_vectors_print() *Function*
Prints out the vector table used internally by the message passing system on a particular processor.

mbox_vector_set(*vector, routine, name*) *Function*
This sets the handler for a specified vector to be a specified C routine.

mbox_vector_delete(*vector*) *Function*

This deletes the association between a vector number and a routine.

mbox_send(*processor, vector, data*) *Function*

This sends a message to the specified processor asking it to execute the routine associated with the specified vector and the specified data.

mbox_send_with_reply(*processor, vector, data*) *Function*

This routine forms the basis for reliable message sending. It not only sends a message like the previous routine, but also waits for a reply back from the processor.

B.14 Support for Real Time tasks

More complex than the simple real time interface is the MOS or the scheduler used by the CONDOR system. This sytem is described adequately in Siegel et al. [1985], and allows the simultaneous running of different servo loops at different rates. These real time tasks can all be running along with a so-called *background* task that can be used by the user to interact with the system.

mos_init() *Function*

Initializes the system.

mos_schedule(*name,servoloop,rate*) *Function*

This schedules the servo loop specified to run at a specified rate.

mos_reschedule(*name, servoloop*) *Function*

This reschedules the specified servo loop to be runnable.

mos_enable_loop(*name*) *Function*

This enables a given servo loop.

mos_disable_loop(*name*) *Function*

This disables a specified servo loop.

mos_start() *Function*

This starts the operation of the real time scheduler.

mos_stop() *Function*

This stops the entire scheduling operation.

mos_show_status() *Function*

This routine shows the status of a servo loop.

mos_show_full_status() *Function*

This routine shows the status of the entire scheduling system.

B.15 Debugging Support

The debugging support is provided by the program *igdb* which is a modified version of GDB [1]. We have modified GDB so that it can be linked with the `ptrace` and `wait` emulation libraries. These libraries enable the debugger process to control and symbolically debug a process running on the slave microprocessor.

[1] GDB stands for the Gnu Debugger, a product from the Free Software Foundation written by Richard. M. Stallman

B.16 Conclusion

In the above sections I have tried to provide a brief glimpse into the CONDOR system, as viewed from a programmer's stand point. I have not included facilities intended for programming the hand, performing its kinematics calculations, and controlling it using the digital position and force control algorithms discussed in the rest of this thesis. The CONDOR real time system stands now as a very useful real time programming environment. Soon, with the facilities built on top of it to control the hand, I envision it will be a useful hand programming environment.