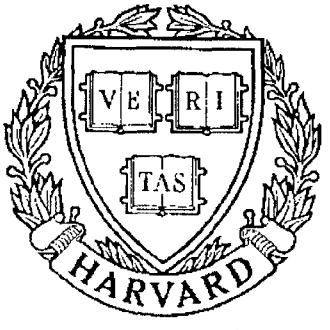
THESIS REPORT
Master's Degree

SYSTEMS
RESEARCH
CENTER



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012) and Industry

Research support for this report has been provided by Fairchild Space Company and AFOSR Research Initiative Program (AFOSR-87-0073)

Interactive Graphics and Dynamical Simulation in a Distributed Processing Environment

by R.H. Byrne
Advisor: P.S. Krishnaprasad

M.S. 90-7

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
1990		**00-00-1990 to 00-00-1990**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Interactive Graphics and Dynamic Simulation in a Distributed Processing Environment	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Maryland,The Graduate School,2123 Lee Building,College Park,MD,20742**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**134**	

Abstract

Title of Thesis:	Interactive Graphics and Dynamical Simulation in a Distributed Processing Environment
Name of Degree Candidate:	Russell H. Byrne
Degree and Year:	Master of Science, 1990
Thesis directed by:	Professor P. S. Krishnaprasad Systems Research Center Department of Electrical Engineering

An implementation of a distributed dynamical simulation is presented. Issues concerning real time simulation are discussed. Three dimensional, animated, object oriented graphics software is presented with problems and solutions. *Libipc(3)*, a library for fast, easy interprocess communications, is designed, along with a general discussion on client/server models. The equations of motion for an N body planar chain are derived in a Lagrangian setting. The solution of these equations using the Newmark technique is presented. Finally, a control system for reorienting the chain is described, and its relation to the NASREM architecture is discussed.

Interactive Graphics and Dynamical Simulation in a Distributed Processing Environment

by

Russell H. Byrne

Thesis submitted to the Faculty of The Graduate School
of The University Of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
1990

Advisory Committee:

Professor P. S. Krishnaprasad Chairman/Advisor
Assistant Professor W. Dayawansa
Assistant Professor Mark A. Austin

Dedication

To Padma

Acknowledgements

I would like to begin by thanking Fairchild Space Company. It was the Fairchild Scholars Program that brought me to the University of Maryland at College Park, and the program supported me through most of the pursuit of this degree.

Next, I would like to thank my advisor, Dr. P.S. Krishnaprasad, for all of his help. As a part-time student, I have thoroughly tested his patience over the past few years, and he has never given me a hard time. Several times I have come up against the 'brick walls' of research, and he came to the rescue with one of his mathematical pearls.

The foundation for this research was laid by Velu Sinha. He taught me nearly everything I know about doing graphics on the IRIS, and a great deal about UNIX. His help was indispensable during the first phase of this research.

I worked very closely with Amir Sela on the IPC research. He helped me figure out software interrupts, and did a lot of work that provided our lab with valuable input on the IPC topic.

Over the past two years I have had several helpful discussions with Kong Ha, of Fairchild, and Stephe Leake, of NASA/GSFC. Their many years of experience in robotics, simulation, and control provided me with several pithy suggestions.

The dynamics work was based largely on equations derived by N. Sreenath, of

Case Western University. Before leaving UMCP, he helped my understanding of these equations.

I would like to thank my immediate management at Fairchild, ST Systems Corporation, my current employer, and NASA/GSFC, where I work. All of these management teams have cheerfully put up with my unusual working hours during the pursuit of this degree.

The capabilities of MACSYMA used for this research and described here are quite new. I am grateful for the speedy response of the Technical Support Department of Symbolics, Inc. Jonathan Len, and Jeffrey P. Golden responded to my bug reports and user wishes by sending me LISP patches to fix the problem or supply a new feature within a day or two of my request. That kind of support is unparalleled.

This work was supported in part by the AFOSR University Research Initiative Program under grant AFOSR-87-0073 and by the National Science Foundation's Engineering Research Centers Program: NSFD CDR 8803012.

Finally, and most importantly, I would like to thank my better half, Padma. She has done everything imaginable to make full time employment and part time graduate school as tolerable as possible. I hope some day I can make it up to her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

In [10], Sinha envisioned a distributed simulation system comprised of at least four different machines, or at least four different processes running on a smaller number of machines. The logical tasks were identified as:

1. Graphical Output/User Interface
2. Dynamical Modeling
3. Control System
4. Planning System

With the completion of this work, we have realized three out of the above four tasks — only the planning work needs to be done. In this implementation, the control system was not separated into a process by itself. It is part of the dynamics and control process. This work is spread over four areas: Interactive Graphics, Interprocess Communication (IPC), Dynamics and Control.

1.2 Overview

Chapter 2 gives an overview of the research that was the foundation for the rest that follows: the three dimensional graphics simulation of Space Station *Freedom*, and its associated robotics.

In Chapter 3, we present the design of a library of functions being used in the Intelligent Servosystems Lab by several researchers to connect dynamical simulation programs on one workstation with graphical visualization software on another workstation. The source code for *libipc(3)* is presented in Appendix A, and the UNIX manual pages are given in Appendix B.

In Chapter 4, we define the dynamical system under study: a planar chain of rigid bodies under no external forces. We derive the equations of motion, and show how we solve them. Since MACSYMA was used extensively in this development, we give the MACSYMA source code that implements the solution for the general N body case in Appendix C. Appendix D shows how the C language source code was generated from the symbolic MACSYMA expressions, and shows an example of the output from the MACSYMA subexpression optimizer.

In Chapter 5 we define a control problem and provide a solution for the chain. The problem is to reorient the chain by actuating the arms to make closed path motions in shape space. The NASREM robot control architecture, developed by NASA and the NBS, is also discussed. In Appendix E, we give a hint for those embarking on a distributed system implementation.

Finally, in Chapter 6, we give several recommendations for future work that have arisen from the volatile nature of the graphics, networking, and UNIX fields.

We have tried to include the definition of most acronyms when first mentioned, as well as explain any nonstandard notation. In Chapter 3, we use the standard UNIX manual entry notation as in *rsh(3C)*, meaning one can find a description of the command in section 3C of the UNIX reference manual.

Chapter 2

Interactive Graphics

2.1 Introduction

We begin our study of simulation with the area of graphics. We study the problem of constructing a simulation of the space station and its associated robotics. We start by implementing a three dimensional wire frame simulation, and then discuss the problems one encounters when implementing even the simplest of shading models.

The Intelligent Servosystems Laboratory (ISL) at the University of Maryland has three Silicon Graphics IRIS workstations. The oldest is a 2400 Turbo, and the newest is a 4D/120GTXB. The fastest of the three, the 4D/120GTXB, is capable of drawing 400,000 vectors per second, or 100,000 shaded polygons per second.

However, most of the graphics research described here was done before this new workstation was procured. The older models were primarily wire frame graphics engines, as a typical Gouroud-shaded scene takes more than two seconds to render.

The best graphics performance was attained through the use of display list, or *object oriented* graphics.

2.2 Object Oriented Graphics

2.2.1 Definition

A graphics object is a list of graphical primitive operations with a name. One defines a graphics object by compiling a set of drawing commands into a display list. No drawing takes place until the list is invoked, by *calling* the object. Thus, there are basically three operations that are done in object oriented graphics:

1. Objects are built out of display lists.

2. Objects that move in the scene are rebuilt every iteration.

3. All objects in the scene are called, following a hierarchy.

2.2.2 Discussion

Constructing hierarchical relationships among many graphics objects leads to very efficient code. For example, for a robotic manipulator, each link is carried by the previous link. We can define basic building blocks, such as joints and links, and create specific instances of them by calling the generic ones for each link and joint. The hierarchical relationship is defined by arranging for defining the first link to call the first joint, which calls the second link, etcetera. In the case of an entirely revolute arm, the link objects never need to be updated. To reposition the arm, one need only redefine the joints that have changed since the last iteration, and call the base joint. The display list does the rest, navigating the hierarchy, and calling all of the dependent objects.

2.2.3 Problems

When one wants to display a shaded scene, however, things become more complicated. When objects are shaded on the screen, they occlude each other. Maintaining the proper relationships for this occlusion is tricky. The item that will appear 'on top' to the viewer is the last item rendered. If the program only contains logic to maintain a hierarchy so that the proper relative motions are handled, there will be viewing angles that appear to show an object in the back of the scene in front of objects closer to the viewer.

This leads to the need for a sorting algorithm. Many graphics workstations today have hardware that will perform Z buffering. Simply put, this hardware keeps track of the eye-Z coordinate (normal to the screen) of each polygon in the scene, and renders them from back to front, using the Painter's algorithm. The drawback is that this leads to excessive computations, and can severely hurt the performance of a real-time simulation.

Our hardware can do Z-buffering, but not while animating a scene in double buffered mode (this eliminates flicker by drawing one image while the viewer looks at a second image; the images are swapped without any clearing operation visible to the viewer).

For our purposes, real time animation is the most important thing, and we strive for as much realism as we can attain without sacrificing smooth animation. This consideration led to the following scheme.

2.3 Object Sorting

Rather than implement Z-buffering in software, which is often done when one doesn't have the hardware for it and is willing to sacrifice performance, we decided to make a compromise and just sort the *objects* in the scene. The number of items to sort is on the order of tens or a hundred, rather than tens of thousands, as in the case of polygon sorting.

Thus, we expected to have reasonable results even though the sorting was to be done in software.

2.3.1 Discussion

The method is simple: a reference point is chosen for the object, such as the center point of the bounding rectangular parallelopiped of the object, and the distance to the viewpoint is computed. A list of these distances is maintained, along with the object numbers they pertain to. At each iteration of the animation, the list is computed, and then bubble sorted. The objects are then called in far to near order.

2.3.2 Example I: Obstacles on Space Station *Freedom*

This scheme worked very well in the case shown in Figure 2.1. The objects being sorted are the three rectangular parallelopipeds that represent science experiments on the truss of the space station, and the base of the Mobile Remote Manipulator System, which navigates between the other objects. The objects are nonoverlapping, and object sorting works very well in this case. (It should be noted that squared distance can be used to determine rendering order.)

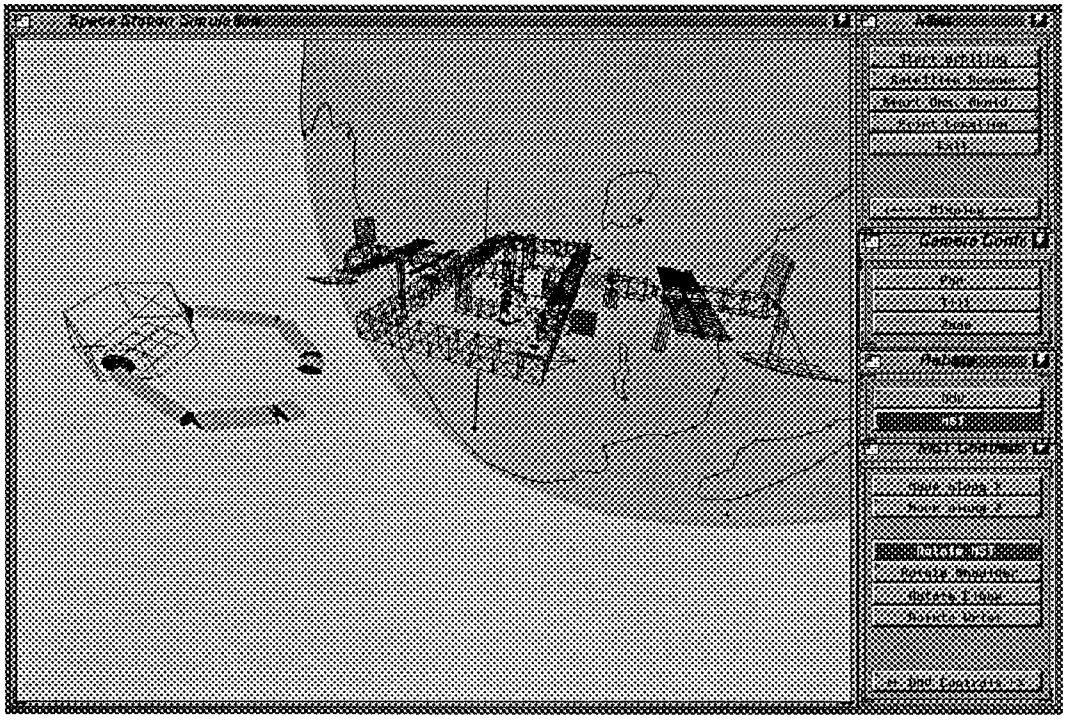


Figure 2.1: Space station over Africa with FTS/OMV in the foreground

2.3.3 Example II: The NCAR World Map

For another example, consider the problem of displaying a map of the Earth, rotating in real time below the space station, which is fixed in the scene. If one has a single graphics object for the Earth map, the back side of the Earth will be visible at all times. This is what we would like to avoid. If one had hardware Z-Buffering, one could position black disks inside the Earth model, such that they occlude the back side.

In our case, we needed to break the Earth model up into a suitable number of pieces, and display only the ones that made up the local horizon in the low Earth orbit scene. It turns out that from a low Earth orbit, one can only see about thirty degrees around the Earth. So, we divided our Earth map into thirty six longitudinal

peels, and calculate the peel closest to the sub-satellite point. This peel and four on either side are then displayed. In order to prevent the viewer from seeing the poles through the surface of the Earth, we then split the peels into three pieces each. The number of portions of the map to display should be a function of the zoom factor of the view point, approaching half the map at infinity.

2.3.4 Preserving Object Hierarchy

Going back to the manipulator example, let us try to apply object sorting. To do object sorting, we must be able to call any object, in any order. This is not directly possible using the hierarchical nature of the arm objects. The lower links are called by the upper links, and must be displayed after, even if they should appear farther away in the scene.

We need a way to preserve the hierarchy of the objects while enabling the Painter's algorithm to be used to render the scene. We propose the following solution. We set up the hierarchy based on invisible reference frames. The frame transformations call each other down the arm, just as neighboring joints and links did before. Only now, we allow each object to be the end point of *its own* hierarchy of frames.

An example: suppose the shoulder of a three joint manipulator is moved. Using plain hierarchy, the shoulder joint would be remade, it would call the upper arm link, which would call the elbow joint, and so on, down to the link after the wrist.

Now consider what we need in place in order to maintain the same hierarchical relationships between the objects, and tolerate any rendering order. Suppose we

need to render the arm from the wrist up to the shoulder. Then the wrist object must be able to be called first, and must be located as a function of all three joints. Likewise, the forearm must be able to be called and take into consideration both angles before it. Proceeding in this way, we see that by having redundant frames colocated but part of different hierarchies, we can accomplish our goal. Calling the base of any hierarchy will propagate down to its end and call one object. This can be done in any order. There is a price to pay in the number of invisible objects, but all the duplicate frame objects have the same joint angles associated with them, so the computational cost is small. For a three joint manipulator, we would be maintaining six frames rather than three. In general we would need $\frac{N}{2}(N+1)$ object frames duplicating N different joint values.

This way, each object has an invisible hierarchy of frames above it, and the objects can be called in any order. We can adopt the Painter's algorithm for realistic rendering, while preserving the hierarchical property of the display list.

2.4 The User Interface

Providing the user with a safe, convenient means of manipulating the large number of inputs to a typical robotic simulation is a challenge. The keyboard has been used less and less because it is prone to user mistakes, and it is tedious to provide exhaustive error checking for it in software.

For our space station simulation, we used popup menus, which can be conveniently programmed using the Multiple exposure (Mex) window manager that comes with the IRIS.

2.4.1 Multiple Exposure (Mex) Popup Menus, and *gl(2)* sliders

Popup menus are convenient because they don't take up any screen space when they're not wanted. When the user desires to interact, she clicks the mouse, and then the menu appears, underneath the mouse cursor.

Rolling, or nested, menus can be used to provide a large number of options in a relatively small area of the screen. Menu items can have labels that toggle each time they are selected, or can have other menus appear.

But we found that menus alone could not provide a complete interface. In the dynamical simulations described later in this work, we have as many as five parameters associated with each body of the system. It would be much too tedious (and dangerous) to have on the order of twenty parameters entered via the keyboard. We decided to use nested windows of sliders.

The sliders were implemented using the *gl(2)* library found on the IRIS, and provided a high performance input tool. Since the slider bar reaches known limits on either end, erroneous inputs are impossible. Also, since the window manager was used, collections of sliders can be grouped in a small window, and several of these windows can be placed overlapping on one corner of the screen. We found it easy to manipulate as many as twenty sliders in a single simulation.

Since standard Graphical User Interfaces (GUIs) are almost agreed upon, we didn't want to invest much effort in this area, and found that sliders and popup menus can work well enough together to provide a complete interface for input as well as display.

2.4.2 The NASA/AMES Panel Library

Shortly after the work was complete on our GUI, we were introduced to the NASA/AMES Panel Library, written by David Tristram. We found more than we had hoped for in this package. There are dozens of different *actuators* to chose from, and all have the three dimensional beveled look that is part of the HP X Windows implementation, *Motif*, and the OS/2's Presentation Manager.

By far the most important thing about the library is that it almost totally isolates one from the work involved with porting simulation code from the older line of IRISs (2000s and 3000s) to the 4D machines. The library is available for nearly every IRIS extant, and handles about 90% of the porting chore. Researchers wishing access to this free software should send mail to `dat@orville.nas.nasa.gov`.

Figure 2.2 shows the versatility of the library. At the top right is a column of buttons that perform miscellaneous actions. In particular, the first one replaces the two windows below the button window with strip chart recoders. The windows below the buttons contain *slideroids*. The middle window is a *cycle* of *cycles* of *slideroids*.

A slideroid provides mouse control over a numeric display. There are buttons for reseting the displayed value to an initial value, and for changing from coarse to fine tuning mode. One can use the mouse to adjust the value being displayed, or to adjust the rate at which it is changing. By pressing the left and right arrow buttons on the inner cycle, one chooses slideroids controlling different parameters associated with the same body of the simulation. The outer cycle selects the different bodies. In this example, using only a small portion of the screen, we have

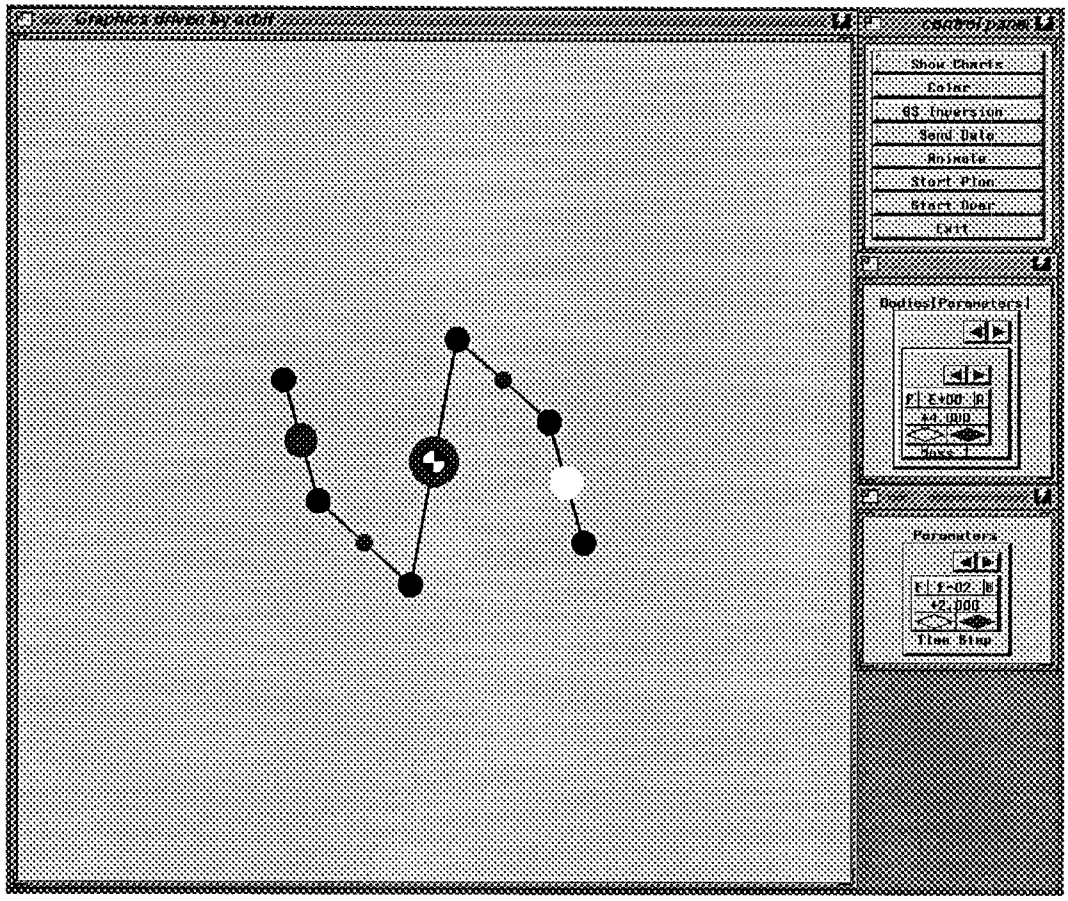


Figure 2.2: Five body chain with the NASA/AMES GUI

mouse control over numeric readout of twenty-eight variables; three strip chart recorders are available upon request, and seven buttons controlling miscellaneous functions round out the interface.

2.5 Summary

We have discussed the pros and cons involved with object oriented graphics. A solution to the rendering problem for manipulators was presented, in the context of our simulation of the robotics in the vicinity of Space Station *Freedom*.

We considered two approaches to answer the question: What GUI should one use while waiting for standards to emerge? The GUI was perfect for our research in the past years because we only used IRIS workstations. Now, however, we are starting to work with HP graphics workstations, and must ask the GUI standards question once more.

After considering attaching rigid body dynamics to our space station simulation, we decided to use a more abstract visual output, in order to concentrate on planar mechanics and control. At a later time, we can drive the three dimensional space station robotics simulation with our two dimensional models, by constraining the manipulators to move only in the plane.

We next consider how to connect the graphics portion of a simulation with the dynamics via a network.

Chapter 3

Interprocess Communication

3.1 Introduction

In this chapter, we discuss distributed computing. In the past, our simulation systems were implemented on IRIS workstations. The software running on the IRIS was responsible for solving equations of motion, and updating an animated display of the simulated objects.

After analyzing the performance of the software, it was noted that the bottleneck was in the numerical code[1]. We wanted a convenient way to split our problems in at least two pieces: a simulation server, responsible for all numeric computations, and a graphics client, responsible for the animated display. These two units would be able to run on different machines if we had a way to connect the programs via a network.

Sizing a simulation to more than one machine is not a trivial task. One would like the graphics to be simple enough so that the graphics hardware can update the screen at least 20 times per second, with 30 frames per second preferred. This gives us about 33 milliseconds to render one frame of the simulation. This time is

[1]This is especially true for the older 2000 and 3000 series machines.

broken into two basic pieces, clearing a graphics frame buffer, and drawing the next image. On the older IRIS models, the clear took as long as 12 milliseconds, while the newer ones have whittled the specification down to a little over 2 milliseconds. After subtracting the buffer clear time, we're left with about 27 milliseconds to draw the image. One must then consider two more specs: the number of vectors per second, and the number of polygons per second that the graphics workstation can draw. These figures will lead one to a count of how many polygons (for a shaded image), or how many vectors (for wire frame) can be presented in the simulation. Given that one has some idea of how complex a typical simulated scene is, one must then choose to either do shaded or wire frame animation. The more recent graphics workstations available for reasonable prices (say, under $50k), are now capable of animating Gouroud-shaded scenes of the space station.

A similar calculation must be done on the numerical side. Given about 33 milliseconds per iteration on a given hardware platform, how complicated can the dynamical model be? The programmer must decide whether or not to include effects such as various types of friction, elastic mechanics, and how many degrees of freedom are to be modeled.

The goal is for both sides to make best use of the simulation frame rate, but often this is very difficult to do. We have examples of simulations that are graphics-intensive, where scenes of the space station and the Earth-Moon system are displayed with very little shading, and the older IRIS's struggle to render 5 frames per second. On the other hand, we have dynamics-intensive simulations, where the graphics are simple two dimensional images of chains of rigid bodies. The

dynamics for the five body chain problem involve iterating several hundred lines of tightly optimized C code, and non-rigid effects haven't even been considered. The result is a simulation that runs at about 20 frames per second.

In our implementation of the network connection, we were aided considerably by what came to be known as the 'IPC Bible' in our lab. Without [6], our understanding of the complicated UNIX™ system calls involved would be quite sketchy. We will begin our IPC discussion with a description of the Client/Server paradigm.

3.2 The Client/Server Model

In the past few years, the Client/Server model has become increasingly important. This is due in part to the increased use of the X Window System, and the Network extensible Windowing System (NeWS).

A client is a program that requires assistance in order to complete its task. A server is a program that makes itself available to other programs in order to render some service. The windowing systems mentioned above include a window manager, which is the server, performing the service of maintaining the windows on a workstation screen. Client applications, such as a calculator program, or an editor, request the server to draw information in the windows, and the server informs the clients when the user manipulates the windows with a mouse. In this way, an application running in a window can request its window to be redrawn if it was recently exposed, etc. The real power of these systems is that the window manager is run on one machine, and the clients it communicates with may reside on any machine on the network. This has led to the recent introduction of a

device known as the X terminal, which only runs the window manager locally. All applications driving the windows are run on other machines, and use IPC to communicate with the window manager.

Our system fits into this model as follows. We treat the graphics program running on the IRIS as the client, which requests a service from another machine: "Please compute the next state of the system." The simulation server runs on a Sun workstation, but since it only deals with numeric data, it could run on any machine connected to the network.

A word on our interface to the network. Like most universities, the University of Maryland is tied together by several Local Area Networks (LANs) that are part of a Wide Area Network (WAN), called the Internet. The Internet is basically a network of LANs, connected together by gateway machines. All machines connected to the Internet are required to understand two network protocols: the Internet Protocol (IP), and the Transmission Control Protocol (TCP). These protocols may be implemented by a variety of operating systems, so the Internet is composed of a diverse set of machines. The most common operating system used in the research environment is some flavor of UNIX. The Berkeley version of UNIX provides a rich set of Interprocess Communications facilities. These facilities are accessed by means of operating system functions, or *system calls*. These system calls provide an entry into the network protocol stack at the TCP level. The programmer doesn't need to know anything about how the TCP protocols are implemented using the more general IP protocols, or how data is formated and transported across the physical network using the Ethernet protocol.

Still, just understanding the TCP facilities is an unpleasant task, and shouldn't need to be accomplished by everyone interested in doing distributed programming. So we wanted to create a simple set of tools that C programmers could use to easily connect software running on different machines. The tools we created are sufficiently general that programs can be distributed over machines located anywhere on the Internet. We found that the network performance was good enough to have a numerical server computing dynamics at the University of Maryland in College Park and transmitting the data to a graphical animation client running on a workstation at NASA/Goddard Space Flight Center (GSFC) in Greenbelt, several miles away.

3.3 *Libipc(3)*, a Library for Fast and Easy Interprocess Communication

A library of routines has been created, called *libipc(3)*. This library provides an easy to use interface to the IPC facilities of Berkeley UNIX. No knowledge of IPC is needed in order to make use of the library. It acts as a layer on top of the Ethernet/TCP/IP protocol stack (see Figure 3.1).

The user is responsible for implementing his own protocol to set up deterministic responses to commands from each side of the interface. The way data is represented is also up to the user, although a convenient means of sending structures of data is provided. Data Representation will be discussed further in section 3.3.3.

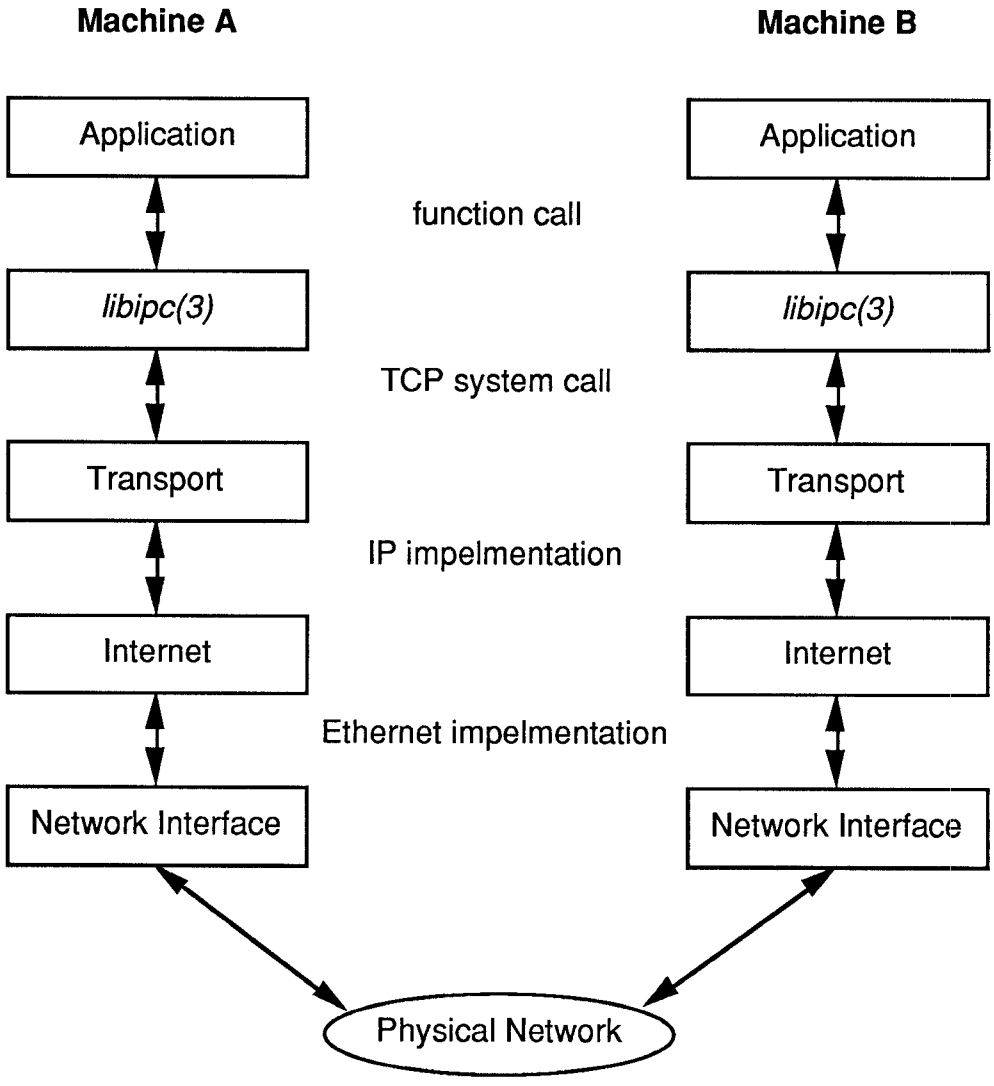


Figure 3.1: Relationship between *libipc(3)* and the standard protocols

The source code for the library is shown in Appendix A. The UNIX manual pages, shown in Appendix B, give a description of the library, examples of its use, and how to link with it. Examples are included for code written under the SunOS, IRIS UNIX, and IRIX operating systems.

In order to understand the discussions to follow, a brief introduction to the notion of a *socket* is necessary. The Berkeley socket is a software abstraction for a communications device. In UNIX terms, it can be thought of as a bidirectional, *named pipe*. The socket provides a mechanism for unrelated processes to communicate with each other on different machines, possibly running different operating systems. Since file name conventions vary from system to system, they are not a convenient means to identify services, or programs. Instead, one uses a port number and a machine address to identify the party with which one would like to communicate.

The analogy of the telephone system is helpful. When a process makes a system call to create a socket, it is equivalent to one purchasing a phone. The phone cannot be used until a number has been assigned to it. This is the point of *naming* the socket, which involves assigning it a port number, and a local machine address. Dialing a phone number is much the same as one process issuing a *connection request*. The desired machine and port must be known at this time. When one answers a ringing phone, it is very much like a process *accepting* a connection request.

This procedure can be selective. One may hang up if one doesn't want to speak to the calling party. This can be done in software too. Upon accepting a connection

request, a process can determine the address of the machine that made the request. If this is not a machine with which communication is desirable, the connection can be terminated.

Next, we will discuss the specifics of how port numbers and machine addresses are assigned and used.

3.3.1 Connection Establishment

All machines attached to the Internet have an Internet address. This is a series of four numbers in the range [0,255]. These addresses are uniquely matched with names, which often have four components. For example, the Sun 3/260 used as a simulation server in our lab has the Internet address 128.8.111.81, and the Internet name `orbit.src.umd.edu`. A central authority is responsible for assigning these addresses and names, in order to guarantee uniqueness. The central authority, the *Network Information Center (NIC)*, is located at SRI International [2].

Since the machine addresses are tightly controlled, programmers don't have to worry about them. The system manager of each machine must obtain an address and a name by pursuing proper channels. This done, the programmer needs only to find out the name and address of the machines she wishes to use.

The only other information necessary to request a connection is a port number. Port numbers smaller than `IPPORT_RESERVED` are reserved for privileged processes. Many of these ports have been assigned to 'well-known' services, and are consistently defined on all Internet machines.

The naive way to do things is to pick a port number larger than IPPORT_RESERVED, and code the constant into both programs that wish to communicate. The server uses this number to name its socket, and the client uses it when issuing the connection request. There are several problems with this approach. First, only one image of the programs may be executing at a time. A more general approach would allow several images of the programs on each machine to be run concurrently, say to show a simulation at different time instants, or with different parameter values. Another problem is that of uniqueness. How can one be assured that no one else is using a given port number?

As shown in Appendix A, the source code for our library, the solution to these problems is for the server to use the special port number 0 when naming its socket (see function *passive_socket(3)*). This is a request to the operating system to use the next available port. The port number chosen by the system is then determined with a system call (*getsockname(2)*). The server must then make this port number known to the client.

Our approach is specific to a two process system, with one server, and one client. It also assumes that the *rsh(1C)* command[2] is supported on the server machine. Given these two assumptions, the server informs the client of its port number by starting the client via the *rsh(1C)* command, passing the port number and its machine name on the command line. The client then has all the information it needs to issue the connection request, which the server then accepts. This protocol is implemented by the routines *start_server(3)*, and *connect_socket(3)*. The alert reader

[2]The *r*emote *sh*ell command is used to execute one command on another machine.

will be confused by the fact that the *start_server(3)* function performs the *server* side of the connection, and the *connect_socket(3)* function performs the *client* side. We use opposite conventions for the role of client and server depending on the context. In the context of IPC, the server is the process that makes the call to *accept(2)*, while the client calls *connect(2)*. In the context of a simulation (which is the context used by most users of the library), the server is the process providing numerical services for a graphics client.

If *rsh(1C)* is not available, a different approach must be taken. At NASA/GSFC, we have implemented an IPC system that connects a process on a VAX 8350 running VMS to a process on an IRIS 4D running IRIX. The connection is established in a more symmetric way than the method used in *libipc(3)*. Either side of the TCP/IP connection can be started first. The *connect(2)* call was put in a loop, so the client may be started before the server.

A more general approach is necessary if more than one client needs to communicate with a server. The server must either use a well known port number that all prospective clients are aware of, or it must register its port number in a database. The database approach is preferred, since constant port numbers generated by users cannot be guaranteed to be unique. The *yellow pages* which is part of the Network File System, provides this functionality.

3.3.2 Polled vs. Interrupt-driven Communication

The first implementation of the library was based on the *polled* I/O approach. Namely, once per simulation iteration, the program checks to see if there is data

waiting to be read. If so, it reads the data and acts on it. The client would normally draw the next image based on state data received, and the server would typically process a command from the client.

While this is an intuitive approach, it wastes time. The time taken to check for data availability and receive a negative response could be put to better use. A better approach is to have the incoming data *interrupt* the recipient. This way, when no data is waiting, a process continues to do it's task, whether it is animating a scene, or calculating state vectors. When communication is necessary, the process finishes the current simulation loop, and then acts on the newly arrived data.

This is implemented using interrupt-driven socket I/O. The problem is, this is not available on all machines. For instance, it is available on Sun workstations and IRIS 4Ds, but not IRIS 2000s, or 3000s. It is worth checking for the availability of interrupt-driven socket I/O, since it can speed throughput by as much as a factor of twenty [9]. One can check by looking in the file `fcntl.h` (located in `/usr/include/sys` on UNIX systems), for the definition of `FASYNC`. If it is present, then interrupt-driven socket I/O is supported.

It should be noted that it is not always necessary to have interrupt-driven socket I/O on both machines in order to improve performance. For example, if the graphics client takes twenty milliseconds to execute one simulation loop, and the simulation server takes thirty milliseconds using the polling approach, an improvement will be seen if only the server is changed to use interrupt-driven socket I/O.

Libipc(3) provides both interfaces to the user. As described in the manual pages in Appendix B, the only difference in the programming interface between the polling approach and the interrupt-driven approach is a function call at start-up (*init_isr(3)*), and a test of a flag (`NewRequest`) at the top of the main loop. The performance improvement comes from replacing a system call to check for data availability with a simple `if (NewRequest)` statement checking a global flag to see if an interrupt has occurred during the last loop.

3.3.3 Data Representation

Multiprocess systems share data in a variety of ways. If running on the same CPU, it is common for two process systems to communicate via shared memory. This is also possible when multiple CPUs share the same bus. But when the processes are on different machines separated by a network, this is not possible. The data must be sent from one process to another. In this section, we discuss the various approaches used to do this.

The most common approach is to send characters across the network. Data is converted into an ASCII representation, packed into a message, and shipped across. On the receiving side, the message must be decoded and the data converted into machine representation. These conversion processes are quite time consuming.

Our goal was to avoid the translation to and from ASCII. Fortunately, our situation was helped by the fact the machines we are using store data in very similar ways. In fact, the only difference in data representation on the IRIS 2000s, 3000s and the Sun 3/260 is that the Sun `double` type is known on the IRIS as a `long`

`float`. Thus, without any conversion, we can send structures of various types of data back and forth between two of these machines. This is not the case if a VAX is involved. The VAX stores integers as well as floating point data differently from most UNIX workstations. For the integer differences, there are standard macros to perform byte and word swapping between the local host's representation and a standard network representation[3].

Until recently, no similar standard existed for floating point and more complicated types. The Network File System, from Sun Microsystems, includes a library to handle this problem. The eXternal Data Representation (XDR) functions provide a means of encoding various different types of data into a network standard, and decoding them on the receiving side into the local host's specific formats. Since the machines in the Intelligent Servosystems Laboratory (ISL) are so similar in the way they store data, we decided not to use XDR in order to keep performance at a maximum.

The *libipc(3)* library includes two functions, *send_structure(3)* and *receive_structure(3)* that make the use of structures convenient for shipping data back and forth over the network. However, a problem was encountered when doing this with the IRIS 4D machine and the Sun 3/260. Even though each machine represents the types `float` and `short` in exactly the same way, a structure on one machine with several fields of each type was found to have a length different from the same structure on the other machine. The IRIS 4D ensures that the fields of a structure line up on boundaries determined by the largest type in the structure,

[3] *htons(3N)*, *htonl(3N)*, *ntohs(3N)*, and *ntohl(3N)* where the *h* is for host, *n* for network, *s* for short and *l* for long.

while the Sun doesn't. For example, a structure with fourteen `long float` fields (of eight bytes each) and one `int` type (of four bytes) is 116 bytes long when used on the Sun, but is 120 bytes long when used on the IRIS 4D. The IRIS pads the `int` field with four bytes so that all fields start on eight byte boundaries. So when passing raw data from machine to machine, several experiments must be conducted first, to ensure that the format is identical for each desired type, and also that the format of the structure is the same on both machines.

For the IPC system implemented at NASA/GSFC, we used a message passing approach. The data is converted into ASCII, and terminated by a newline character. After the sending process sends the message over the network, the receiving process will detect that data is available. The problem is that the receiver may not know in advance how long the message is, and thus doesn't know how many bytes to expect over the network. One wants to avoid the receiver trying to read a whole message before it has arrived in full, because the message would then have to be buffered and pieced together. Fortunately, there is a mechanism for `PEEK`ing at the data available from the socket, without actually reading it. We used this mechanism to search the newly arrived data for the newline character, in `PEEK` mode. If it is present, we know that an entire message is ready to be read, so this is done. If not, we return control to the calling routine, passing back status that there are no full messages to read.

It is up to the programmer to decide what method to use, taking into consideration performance requirements and the types of machines involved. The sockets created by the routines in *libipc(3)* can be used with the *read(2V)* and *write(2V)* func-

tions to implement a message passing system, or, when appropriate, the structure-passing routines can be used to send data directly.

3.4 Summary

We have discussed general animation considerations in designing a two process simulation system, with a graphics client, and a simulation server. After introducing the Client/Server model, and a few words on the Internet, we presented *libipc(3)*. This library was called *fast*. Using raw data transmission and interrupt-driven socket I/O, we have measured communications throughput as fast as 14 kilobytes per second [9]. The library was also called *easy*. One can see from the examples given in Appendix B that only one or two simple function calls are necessary to initiate communications, and one or two more in each loop to transport the data.

We discussed ways of establishing the connection, and different ways of handling I/O — the polled approach, and the interrupt-driven approach. We concluded with a discussion on the various ways of represending data, and their impact on performance and portability.

Several different programmers have made use of *libipc(3)* in the ISL, and have been able to construct distributed applications without any knowledge of the details of IPC. In view of this fact, our main IPC research goal has been achieved.

In Appendix E, we present a suggestion that may be helpful to programmers implementing distributed applications.

Chapter 4

Coupled Planar Rigid Body Dynamics

4.1 Introduction

In this chapter we define the equations of motion and an approach for solving them that makes use of numerical methods and symbolic computation. The equations are derived using the Lagrangian approach, which provides several physical quantities (energy, momentum, equations of motion) that can be checked to ensure that the solution is stable. We will describe how the Newmark method was applied to our problem, producing a linearized system that we solve using Newton-Raphson iteration.

4.2 Lagrangian Formulation of the Equations of Motion

The system under consideration is an arbitrary number of planar rigid bodies connected in the form of an open chain, with the center of mass of each interior body on the line connecting its two joints. We begin by describing the system as in [12], for the case of a chain.

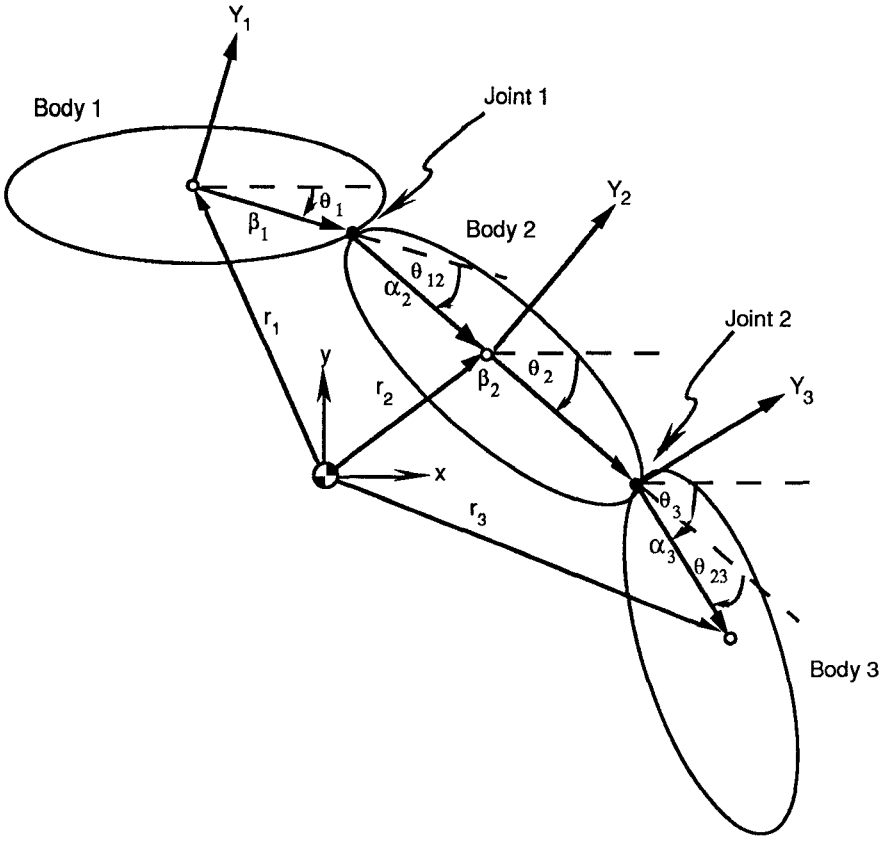


Figure 4.1: Definition of parameters for the three body problem

4.2.1 Notation

Define the origin of the inertial coordinate system to be fixed at the system center of mass with its Z axis normal to the plane of the paper. Similarly, for each body, define a local frame of reference located at the body's center of mass, with its Z axis out of the paper. Refer to Figure 4.1.

The following notation will be used:

θ_i — Angle from the inertial frame of reference to local frame i

$\theta_{i,j}$ — Angle from local frame i to local frame j

$R(\theta_i)$ — (2×2) rotation matrix associated with body i

$$R(\theta_i) = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i) \end{bmatrix}$$

We will also make use of the following:

$\tilde{\alpha}_i$ — Vector from joint $(i-1)$ to the center of mass of body i in the local frame of body i

$\tilde{\beta}_i$ — Vector from joint $(i-1)$ to joint i in the local frame of body i

κ_i — Linear parameter $\in [0,1]$ locating the center of mass of body i on the line segment from joint $(i-1)$ to joint i

$\tilde{r}_i$ — Vector from the system center of mass to the center of mass of body i

I_i — Moment of inertia of body i at its center of mass and along an axis normal to the plane of the paper

m_i — Mass of body i

The fractional masses are defined as:

$$\varepsilon_i = \frac{m_i}{\sum_{j=1}^{N} m_j}$$

The joint angles in terms of absolute body attitude are defined as:

$$\theta_{i,j} = \theta_j - \theta_i$$

We define the *mass distribution constants* $a_{i,k}$ and $b_{i,k}$, which are used to write the mass matrix of the system.

The mass distribution constants $a_{i,k}$ are:

$$a_{i,k} = \begin{cases} 0, & i = N \\ -\sum_{j=i+1}^{N} \varepsilon_j, & k \leq i \leq N-1 \\ 1 - \sum_{j=i+1}^{N} \varepsilon_j, & 1 \leq i \leq k-1 \end{cases}$$

The mass distribution constants $b_{i,k}$ are:

$$b_{i,k} = \begin{cases} 0, & i = 1 \\ -\sum_{j=i+1}^{N} \varepsilon_j, & i \neq k, 2 \leq i \leq N \\ 1 - \sum_{j=i+1}^{N} \varepsilon_j, & i = k \neq 1 \end{cases}$$

The link lengths are:

$$\widetilde{\beta}_i = \begin{cases} \begin{bmatrix} 0 \\ 0 \end{bmatrix}, & i = N \\ d_i \begin{bmatrix} 1 \\ 0 \end{bmatrix}, & \text{otherwise} \end{cases}$$

The center of mass locations are:

$$\widetilde{\alpha_i} = \begin{cases} \begin{bmatrix} 0 \\ 0 \end{bmatrix}, & i = 1 \\ d_N \begin{bmatrix} 1 \\ 0 \end{bmatrix}, & i = N \\ \kappa_i d_i \begin{bmatrix} 1 \\ 0 \end{bmatrix}, & \text{otherwise} \end{cases}$$

The kinematics descriptor is:

$$\widetilde{\delta}_{i,k} = a_{i,k}\widetilde{\beta}_i + b_{i,k}\widetilde{\alpha}_i$$

The augmented inertia is:

$$\widetilde{I}_k = I_k + \sum_{j=1}^{N} m_j \|\widetilde{\delta}_{k,j}\|^2$$

The mass matrix product terms are:

$$\widetilde{\lambda}_{j,l} = \sum_{k=1}^{N} m_k \widetilde{\delta}_{j,k} \cdot \widetilde{\delta}_{l,k} \cos(\theta_{j,l})$$

We note that we can gather all of the mass property information into a coefficient that we call $h_{j,l}$. This will prove very useful in keeping the symbolic equations at a manageable size. So we write:

$$\widetilde{\lambda}_{j,l} = h_{j,l} \cos(\theta_{j,l})$$

The mass matrix can now be expressed in terms of the parameters defined so far:

$$\mathbf{M} = \begin{bmatrix} \tilde{I}_1 & \tilde{\lambda}_{1,2} & \dots & \tilde{\lambda}_{1,N} \\ \tilde{\lambda}_{1,2} & \tilde{I}_2 & \dots & \tilde{\lambda}_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{\lambda}_{1,N} & \tilde{\lambda}_{2,N} & \dots & \tilde{I}_N \end{bmatrix}$$

As an example of how the mass matrix depends on the mass properties and the joint angles of the chain, we display below the mass matrix **M** and the mass coefficients $h_{i,j}$ for the $N = 3$ case.

The mass matrix is:

$$\mathbf{M} = \begin{bmatrix} h_{1,1} & h_{1,2}\cos(q_2 - q_1) & h_{1,3}\cos(q_3 - q_1) \\ h_{1,2}\cos(q_2 - q_1) & h_{2,2} & h_{2,3}\cos(q_3 - q_2) \\ h_{1,3}\cos(q_3 - q_1) & h_{2,3}\cos(q_3 - q_2) & h_{3,3} \end{bmatrix}$$

The mass coefficients ($h_{i,j}$) are:

$$h_{1,1} = d_1^2 (1 - \varepsilon_2 - \varepsilon_3)^2 m_3 + d_1^2 m_2 (1 - \varepsilon_2 - \varepsilon_3)^2 - d_1^2 m_1 (\varepsilon_2 + \varepsilon_3)^2 + I_1$$

$$h_{2,1} = d_1 (d_2 (1 - \varepsilon_3) - d_2 \varepsilon_2 \kappa_2) (1 - \varepsilon_2 - \varepsilon_3) m_3$$
$$+ d_1 m_1 (\varepsilon_2 + \varepsilon_3) (d_2 \varepsilon_3 + d_2 \varepsilon_2 \kappa_2) + d_1 m_2 (1 - \varepsilon_2 - \varepsilon_3) (d_2 (1 - \varepsilon_2) \kappa_2 - d_2 \varepsilon_3)$$

$$h_{2,2} = (d_2 (1 - \varepsilon_3) - d_2 \varepsilon_2 \kappa_2)^2 m_3 - m_1 (d_2 \varepsilon_3 + d_2 \varepsilon_2 \kappa_2)^2$$
$$+ m_2 (d_2 (1 - \varepsilon_2) \kappa_2 - d_2 \varepsilon_3)^2 + I_2$$

$$h_{3,1} = d_1\, d_3\, (1 - \varepsilon_3)\, (1 - \varepsilon_2 - \varepsilon_3)\, m_3 - d_1\, m_2\, d_3\, (1 - \varepsilon_2 - \varepsilon_3)\, \varepsilon_3$$
$$+ d_1\, m_1\, d_3\, (\varepsilon_2 + \varepsilon_3)\, \varepsilon_3$$

$$h_{3,2} = d_3\, (d_2\, (1 - \varepsilon_3) - d_2\, \varepsilon_2\, \kappa_2)\, (1 - \varepsilon_3)\, m_3$$
$$+ m_1\, d_3\, \varepsilon_3\, (d_2\, \varepsilon_3 + d_2\, \varepsilon_2\, \kappa_2) - m_2\, d_3\, \varepsilon_3\, (d_2\, (1 - \varepsilon_2)\, \kappa_2 - d_2\, \varepsilon_3)$$

$$h_{3,3} = d_3^2\, (1 - \varepsilon_3)^2\, m_3 + I_3 + m_2\, d_3^2\, \varepsilon_3^2 + m_1\, d_3^2\, \varepsilon_3^2$$

As derived in [12], the forward kinematics of the chain are[1]:

$$\widetilde{r}_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \sum_{l=1}^{N} R(\theta_l)\widetilde{\delta}_{l,i} \tag{4.2.1}$$

Again, as an example of how the kinematics depend on the joint angles and mass properties, we show the detailed form for the case of $N = 3$:

$$x_1 = -d_3\, \varepsilon_3\, \cos q_3 - (d_2\, \varepsilon_3 + d_2\, \varepsilon_2\, \kappa_2)\, \cos q_2 - d_1\, (\varepsilon_3 + \varepsilon_2)\, \cos q_1$$

$$y_1 = -d_3\, \varepsilon_3\, \sin q_3 - (d_2\, \varepsilon_3 + d_2\, \varepsilon_2\, \kappa_2)\, \sin q_2 - d_1\, (\varepsilon_3 + \varepsilon_2)\, \sin q_1$$

$$x_2 = -d_3\, \varepsilon_3\, \cos q_3 + (d_2\, (1 - \varepsilon_2)\, \kappa_2 - d_2\, \varepsilon_3)\, \cos q_2 + d_1\, (1 - \varepsilon_2 - \varepsilon_3)\, \cos q_1$$

$$y_2 = -d_3\, \varepsilon_3\, \sin q_3 + (d_2\, (1 - \varepsilon_2)\, \kappa_2 - d_2\, \varepsilon_3)\, \sin q_2 + d_1\, (1 - \varepsilon_2 - \varepsilon_3)\, \sin q_1$$

$$x_3 = d_3\, (1 - \varepsilon_3)\, \cos q_3 + (d_2\, (1 - \varepsilon_3) - d_2\, \varepsilon_2\, \kappa_2)\, \cos q_2 + d_1\, (1 - \varepsilon_2 - \varepsilon_3)\, \cos q_1$$

$$y_3 = d_3\, (1 - \varepsilon_3)\, \sin q_3 + (d_2\, (1 - \varepsilon_3) - d_2\, \varepsilon_2\, \kappa_2)\, \sin q_2 + d_1\, (1 - \varepsilon_2 - \varepsilon_3)\, \sin q_1$$

[1]Note that the subscripts on $\widetilde{\delta}$ are reversed from those in [12] throughout this work.

In order to study the effects of torsional springs in our system, we define a stiffness matrix **S**, where s_i is the spring constant of the spring at joint i:

$$\mathbf{S} = \begin{bmatrix} s_1 & -s_1 & 0 & \dots & 0 \\ -s_1 & s_1 + s_2 & -s_2 & \dots & 0 \\ 0 & -s_2 & s_2 + s_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & s_N \end{bmatrix}$$

The stiffness matrix for the $N = 3$ case is:

$$\mathbf{S} = \begin{bmatrix} s_1 & -s_1 & 0 \\ -s_1 & s_1 + s_2 & -s_2 \\ 0 & -s_2 & s_2 \end{bmatrix}$$

As we will derive in the following section, the vector equation of motion for the system is given by:

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{S}\mathbf{q} + \{(\nabla m_{i,j} \cdot \dot{\mathbf{q}})_{i,j}\}\dot{\mathbf{q}} - \frac{1}{2}\nabla(\dot{\mathbf{q}}^{\mathrm{T}}\mathbf{M}\dot{\mathbf{q}}) = \boldsymbol{\tau} \tag{4.2.2}$$

4.2.2 Equations of Motion Derivation

We now derive the equation of motion previewed in the last section (adapted from [13]). To write the Lagrange equation for our system, we begin by defining the kinetic energy:

$$T = \frac{1}{2}\dot{\mathbf{q}}^{\mathrm{T}}\mathbf{M}(\mathbf{q})\,\dot{\mathbf{q}}$$

The potential energy associated with the torsional springs at each joint is defined as:

$$V = \frac{1}{2}\mathbf{q}^{\mathrm{T}}\mathbf{S}\,\mathbf{q}$$

The Lagrangian function is $L = T - V$.

Now, the Lagrangian equation of motion is

$$\frac{d}{dt}\frac{\partial T}{\partial \dot{\mathbf{q}}} - \frac{\partial T}{\partial \mathbf{q}} = \mathbf{Q}, \tag{4.2.3}$$

where $\mathbf{Q}$ is the generalized force vector acting on the system and $\mathbf{q}$ is the vector of generalized coordinates. In our case, $\mathbf{q}$ is the vector of inertial body reference angles θ_i, and $\mathbf{Q}$ is the vector whose components represent the net torque acting at the center of mass of the corresponding body. This is due to motor torques at the neighboring joints, and spring torques as well. So we write:

$$\mathbf{Q} = \mathbf{Q}_m + \mathbf{Q}_s$$

We note that the spring torque term $\mathbf{Q}_s$ is the gradient of the potential energy V:

$$\mathbf{Q}_s = -\mathbf{S}\,\mathbf{q} = -\nabla V$$

Noting that T depends on both $\mathbf{q}$ and $\dot{\mathbf{q}}$, and V depends only on $\mathbf{q}$, we have:

$$\begin{aligned} \frac{d}{dt}\frac{\partial T}{\partial \dot{\mathbf{q}}} &= \frac{d}{dt}\frac{\partial L}{\partial \dot{\mathbf{q}}} \\ \frac{\partial L}{\partial \mathbf{q}} &= \frac{\partial T}{\partial \mathbf{q}} - \frac{\partial V}{\partial \mathbf{q}} = \frac{\partial T}{\partial \mathbf{q}} + \mathbf{Q}_s \end{aligned}$$

so that equation 4.2.3 becomes:

$$\frac{d}{dt}\frac{\partial T}{\partial \dot{\mathbf{q}}} - \frac{\partial T}{\partial \mathbf{q}} = \mathbf{Q}_s + \mathbf{Q}_m$$

$$\frac{d}{dt}\frac{\partial T}{\partial \dot{\mathbf{q}}} - \left(\frac{\partial T}{\partial \mathbf{q}} + \mathbf{Q}_s\right) = \mathbf{Q}_m$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\mathbf{q}}} - \frac{\partial L}{\partial \mathbf{q}} = \mathbf{Q}_m$$

We call the motor-induced torques on the bodies $\boldsymbol{\tau}$, rather than $\mathbf{Q}_m$ from here on. Computing the derivatives indicated above:

$$\frac{\partial L}{\partial \dot{\mathbf{q}}} = \mathbf{M}(\mathbf{q})\,\dot{\mathbf{q}}$$

$$\frac{\partial L}{\partial \mathbf{q}} = \frac{1}{2}\nabla \dot{\mathbf{q}}^{\mathrm{T}}\mathbf{M}(\mathbf{q})\,\dot{\mathbf{q}} + \mathbf{S}\,\mathbf{q}$$

$$\frac{d}{dt}(\mathbf{M}(\mathbf{q})\,\dot{\mathbf{q}}) = \mathbf{M}(\mathbf{q})\,\ddot{\mathbf{q}} + \{(\nabla m_{i,j} \cdot \dot{\mathbf{q}})_{i,j}\}\,\dot{\mathbf{q}}$$

Combining the last four equations yields 4.2.2.

We have the following scalar equations for the $N = 3$ case[2]:

$$-\Big(h_{1,3}\,\dot{q}_3^2\,\sin(q_3 - q_1) - h_{1,3}\,\ddot{q}_3\,\cos(q_3 - q_1) + h_{1,2}\,\dot{q}_2^2\,\sin(q_2 - q_1)$$
$$-h_{1,2}\,\ddot{q}_2\,\cos(q_2 - q_1) + s_1\,q_2 - \ddot{q}_1\,h_{1,1} - q_1\,s_1\Big) = \tau_1$$

$$-\Big(h_{2,3}\,\dot{q}_3^2\,\sin(q_3 - q_2) - h_{2,3}\,\ddot{q}_3\,\cos(q_3 - q_2) - \dot{q}_1^2\,h_{1,2}\,\sin(q_2 - q_1)$$
$$-\ddot{q}_1\,h_{1,2}\,\cos(q_2 - q_1) + s_2\,q_3 - \ddot{q}_2\,h_{2,2} - q_2\,s_2 - s_1\,q_2 + q_1\,s_1\Big) = \tau_2$$

$$\dot{q}_2^2\,h_{2,3}\,\sin(q_3 - q_2) + \ddot{q}_2\,h_{2,3}\,\cos(q_3 - q_2) + \dot{q}_1^2\,h_{1,3}\,\sin(q_3 - q_1)$$
$$+\ddot{q}_1\,h_{1,3}\,\cos(q_3 - q_1) + \ddot{q}_3\,h_{3,3} + s_2\,q_3 - q_2\,s_2 = \tau_3$$

4.3 Solution of the Equations

Next, we turn to the task of solving the equations just derived. In [7], the Newmark trapezoidal method is described. We will apply this method to our problem as follows.

[2]Note that the three generalized forces on the right hand side are linear combinations of two motor torques.

Consider a set of N second order nonlinear differential equations, containing $3N$ unknowns[3]:

$$\begin{aligned} f_1(q_1,\ldots,q_N;\dot{q}_1,\ldots,\dot{q}_N;\ddot{q}_1,\ldots,\ddot{q}_N) &= 0 \\ &\vdots \\ f_N(q_1,\ldots,q_N;\dot{q}_1,\ldots,\dot{q}_N;\ddot{q}_1,\ldots,\ddot{q}_N) &= 0 \end{aligned} \tag{4.3.4}$$

The objective is to find the solution $\{q(t_{n+1}), \dot{q}(t_{n+1}), \ddot{q}(t_{n+1})\}$ given the previous solution $\{q(t_n), \dot{q}(t_n), \ddot{q}(t_n)\}$. We use a special case of the Newmark trapezoidal rule.

Assume that between time steps t_n and t_{n+1}, the acceleration is constant, and equal to the average of the values at the end points of the interval:

$$\ddot{q}(t) = \frac{1}{2}(\ddot{q}_n + \ddot{q}_{n+1}), t \in [t_n, t_{n+1}] \tag{4.3.5}$$

Definite integration over the interval yields:

$$\dot{q}_{n+1} = \dot{q}_n + \frac{h}{2}(\ddot{q}_n + \ddot{q}_{n+1}), \text{where } h = t_{n+1} - t_n$$

Definite integration twice of 4.3.5 over the interval gives:

$$q_{n+1} = q_n + \frac{t_{n+1}^2 - t_n^2}{4}(\ddot{q}_n + \ddot{q}_{n+1}) + h\dot{q}_n$$

Assuming $h \ll t_n$, we can write $t_{n+1}^2 - t_n^2 \approx (t_{n+1} - t_n)^2$. This gives us:

$$q_{n+1} = q_n + \frac{h^2}{4}(\ddot{q}_n + \ddot{q}_{n+1}) + h\dot{q}_n$$

[3]Depending on the context, we will use various notations for the variables. The subscripts n and $n+1$ indicate samples of the vector at times t_n and t_{n+1}, while the subscripts $1, 2, \ldots, i, \ldots, N$ indicate a particular component. Occasionally, we will use both.

To keep the length of the expressions at a minimum, we define the following variables:

$$\begin{aligned} a_{i,n} &= \dot{q}_{i,n} + \frac{h}{2}\ddot{q}_{i,n} \\ b_{i,n} &= q_{i,n} + h\dot{q}_{i,n} + \frac{h^2}{4}\ddot{q}_{i,n} \end{aligned}$$

The Newmark substitutions are then defined as follows[4]:

$$\begin{aligned} \dot{q}_i &= \frac{h}{2}\ddot{q}_i + a_{i,n} \\ q_i &= \frac{h^2}{4}\ddot{q}_i + b_{i,n} \end{aligned}$$

We use the Newmark substitutions to eliminate $\mathbf{q}_{n+1}$ and $\dot{\mathbf{q}}_{n+1}$ from 4.3.4 evaluated at time t_{n+1}.

This leads to a new set of N nonlinear equations in $\ddot{\mathbf{q}}_{n+1}$:

$$\begin{aligned} g_1(\ddot{q}_1, \ldots, \ddot{q}_N) &= 0 \\ &\vdots \\ g_N(\ddot{q}_1, \ldots, \ddot{q}_N) &= 0 \end{aligned} \tag{4.3.6}$$

We solve these equations using Newton-Raphson iteration. I.e., we linearize the equations about $\ddot{\mathbf{q}}_n$, and arrive at the equation:

$$\mathbf{J}\Delta\ddot{\mathbf{q}}_{n+1} = -\mathbf{g}$$

$\Delta\ddot{\mathbf{q}}_{n+1}$ is the change in the approximation to $\ddot{\mathbf{q}}_{n+1}$ over one iteration:

$$\Delta\ddot{\mathbf{q}}_{n+1} = \ddot{\mathbf{q}}^{i+1}_{n+1} - \ddot{\mathbf{q}}^{i}_{n+1}$$

[4] $q_{i,n+1}$, $\dot{q}_{i,n+1}$, and $\ddot{q}_{i,n+1}$ are written q_i, $\dot{q}_i$, and $\ddot{q}_i$ for brevity.

$\ddot{\mathbf{q}}^i_{n+1}$ is the i^{th} approximation of $\ddot{\mathbf{q}}$ at time step $n+1$, and $\mathbf{J}$ is the Jacobian of the system 4.3.6. We evaluate $\mathbf{J}$ and $\mathbf{g}$ at $\ddot{\mathbf{q}}^i_{n+1}$. This gives us a linear equation to solve, which we do using LU decomposition, followed by forward and backward substitution [8].

We start the iteration by setting $\ddot{\mathbf{q}}^1_{n+1} = \ddot{\mathbf{q}}_n$, and continue until $\Delta\ddot{\mathbf{q}}_{n+1}$ is small enough. We then set $\ddot{\mathbf{q}}_{n+1} = \ddot{\mathbf{q}}^M_{n+1}$, where M is the number of iterations necessary for the Newton-Raphson technique to converge.

Continuing our example using the three body case, after doing the Newmark substitutions, the dynamics are:

$$\left(\frac{1}{4}\,h_{1,3}\,\ddot{q}_3^2\,h^2 + h_{1,3}\,a_{3,n}\,\ddot{q}_3\,h + h_{1,3}\,a_{3,n}^2\right)\sin\left((\ddot{q}_1 - \ddot{q}_3)\,\frac{h^2}{4} - b_{3,n} + b_{1,n}\right)$$
$$+ h_{1,3}\,\ddot{q}_3\cos\left((\ddot{q}_1 - \ddot{q}_3)\,\frac{h^2}{4} - b_{3,n} + b_{1,n}\right)$$
$$+ \left(\frac{1}{4}\,h_{1,2}\,\ddot{q}_2^2\,h^2 + h_{1,2}\,a_{2,n}\,\ddot{q}_2\,h + h_{1,2}\,a_{2,n}^2\right)\sin\left((\ddot{q}_1 - \ddot{q}_2)\,\frac{h^2}{4} - b_{2,n} + b_{1,n}\right)$$
$$+ h_{1,2}\,\ddot{q}_2\cos\left((\ddot{q}_1 - \ddot{q}_2)\,\frac{h^2}{4} - b_{2,n} + b_{1,n}\right)$$
$$+ (\ddot{q}_1\,s_1 - s_1\,\ddot{q}_2)\,\frac{h^2}{4} - s_1\,b_{2,n} + \ddot{q}_1\,h_{1,1} + b_{1,n}\,s_1 = \tau_1$$

$$\left(\frac{1}{4}\,h_{2,3}\,\ddot{q}_3^2\,h^2 + h_{2,3}\,a_{3,n}\,\ddot{q}_3\,h + h_{2,3}\,a_{3,n}^2\right)\sin\left((\ddot{q}_2 - \ddot{q}_3)\,\frac{h^2}{4} - b_{3,n} + b_{2,n}\right)$$
$$+ h_{2,3}\,\ddot{q}_3\cos\left(,\ddot{q}_2 - \ddot{q}_3)\,\frac{h^2}{4} - b_{3,n} + b_{2,n}\right)$$
$$+ \left(-\frac{1}{4}\,\ddot{q}_1^2\,h_{1,2}\,h^2 - a_{1,n}\,\ddot{q}_1\,h_{1,2}\,h - a_{1,n}^2\,h_{1,2}\right)\sin\left((\ddot{q}_1 - \ddot{q}_2)\,\frac{h^2}{4} - b_{2,n} + b_{1,n}\right)$$
$$+ \ddot{q}_1\,h_{1,2}\cos\left((\ddot{q}_1 - \ddot{q}_2)\,\frac{h^2}{4} - b_{2,n} + b_{1,n}\right) - (s_2\,\ddot{q}_3 - \ddot{q}_2\,s_2 - s_1\,\ddot{q}_2 + \ddot{q}_1\,s_1)\,\frac{h^2}{4}$$
$$- s_2\,b_{3,n} + \ddot{q}_2\,h_{2,2} + b_{2,n}\,s_2 + s_1\,b_{2,n} - b_{1,n}\,s_1 = \tau_2$$

$$\left(-\frac{1}{4}\,\ddot{q}_2^2\,h_{2,3}\,h^2 - a_{2,n}\,\ddot{q}_2\,h_{2,3}\,h - a_{2,n}^2\,h_{2,3}\right)\sin\left((\ddot{q}_2 - \ddot{q}_3)\,\frac{h^2}{4} - b_{3,n} + b_{2,n}\right)$$
$$+\,\ddot{q}_2\,h_{2,3}\cos\left((\ddot{q}_2 - \ddot{q}_3)\,\frac{h^2}{4} - b_{3,n} + b_{2,n}\right)$$
$$+\left(-\frac{1}{4}\,\ddot{q}_1^2\,h_{1,3}\,h^2 - a_{1,n}\,\ddot{q}_1\,h_{1,3}\,h - a_{1,n}^2\,h_{1,3}\right)\sin\left((\ddot{q}_1 - \ddot{q}_3)\,\frac{h^2}{4} - b_{3,n} + b_{1,n}\right)$$
$$+\,\ddot{q}_1\,h_{1,3}\cos\left((\ddot{q}_1 - \ddot{q}_3)\,\frac{h^2}{4} - b_{3,n} + b_{1,n}\right)$$
$$+\,(s_2\,\ddot{q}_3 - \ddot{q}_2\,s_2)\,\frac{h^2}{4} + \ddot{q}_3\,h_{3,3} + s_2\,b_{3,n} - b_{2,n}\,s_2 = \tau_3$$

4.4 Implementation

As mentioned in the introduction, MACSYMA was very useful in implementing the solution of the equations of motion for the chain. Equation 4.2.2 was coded in terms of N using customized versions of the `DOT` and `GRAD` operations for vectors; matrix multiplication, and the `SUM()` function.

As will be described more fully in the next chapter, the desired output from MACSYMA was a C language module (a collection of functions) responsible for the dynamics, inverse dynamics and forward kinematics of the chain for a specific value of N. Appendix C contains the MACSYMA source code that achieves this goal.

One of the first things done in the MACSYMA code is setting the value of N. The simulation running in our laboratory uses the output for the $N = 5$ case. Next, the definitions of $a_{i,k}, b_{i,k}, \widetilde{\beta}_i, \widetilde{\alpha}_i, \widetilde{\delta}_{i,k}, \widetilde{I}_k$, and $h_{j,l}$ are formed. The mass matrix **M** is then formed based on these definitions. Next, Equation 4.2.2 is defined.

The Newmark substitutions are then performed, and the system is manipulated into the form of Equation 4.3.6. The Jacobian of this system is then found, which completes the dynamical manipulations. The forward kinematics are implemented using Equation 4.2.1.

Until recently, MACSYMA was no help in translating symbolic equations into computer languages other than FORTRAN. A new function, `GENTRAN()` is now available that can be used to generate FORTRAN, RATFOR, and C language output. Given a matrix `jac`, the MACSYMA fragment:

```
GENTRANLANG : C $

gentran(jac : eval(jac)) ;
```

produces syntactically correct C language statements assigning every element of `jac`.

Another thing that used to be lacking in MACSYMA's treatment of language translation involved repetition of common subexpressions. Simple things like `sin(x)`[5] were repeated over and over in a long trigonometric expression. For years programmers had to declare temporary variables, assign them, and insert them where the common subexpressions appeared. MACSYMA now does all of this, through the use of the `OPTIMIZE()` function directly, or indirectly by setting `GENTRANOPT : TRUE`.

To make things more convenient for the programmer, there is also a `GENTRANIN()` function, that will read a composite file containing both MACSYMA statements (generally `GENTRAN()` commands) and static C language statements, and

[5]A patch was received from Symbolics to make all C language output appear in lower case.

produce a file by leaving the C language statements intact, and replacing the MACSYMA code fragments with their output. Thus, a compilable file can easily be generated without human intervention.

The first part of Appendix D shows a portion of the composite file `JAC.MAC_C` for the three body chain problem. Sections delimited with << and >> are processed by MACSYMA, and replaced by their output. The other sections are passed to the output without change. The second part of Appendix D shows the corresponding output generated by sending this file through the `GENTRANIN` function.

If one needs output in a language other than FORTRAN, RATFOR, or C, one can use the `LITERAL()` function in conjunction with `GENTRAN`, and produce any syntax desired. The symbolic expressions that result from manipulations within MACSYMA can be picked apart using the `PART()` function, and surrounded with the syntax of the desired language. At NASA/Goddard Space Flight Center, this method is being used to generate Ada code for forward kinematics and manipulator Jacobians.

There is one more new feature in MACSYMA that will enable it to compete favorably with Mathematica in the near future: TEX output. All of the three body example equations given earlier in this chapter are slightly edited TEX output generated by MACSYMA. Many researchers have made the switch from MACSYMA to Mathematica simply because they wanted C and TEX language output. Now that MACSYMA has these features, it is not necessary for people in the scientific computing community to switch for these reasons.

4.5 Summary

We have discussed both symbolic and numerical issues involved with writing and solving the equations of motion for a chain of N rigid bodies connected by revolute joints free to move about in the plain, without any applied external force or torque. The symbolic issues included writing the equations using definitions from [12], and using the Lagrangian approach. We also discussed how MACSYMA can be used to do all of the symbolic manipulations and translate the results into optimized C code.

The numerical issues concerned the solution of the equations. We discussed how the Newmark method was applied to our problem, linearizing it in an iterative process of finding the solution at the current time instant, given the solution at the previous instant. In the next chapter, we will see how the C module discussed here fits into the simulated control system.

Chapter 5

Attitude Control for a Floating Chain

5.1 Introduction

In this chapter, we discuss the design of the control system for a robot whose equations of motion were derived in the last chapter. The fact that the base of the robot is not fixed with respect to the inertial reference frame (as with terrestrial robots) is important here. We will be careful to note when quantities are in *joint space* or *configuration space.*

The problem we solve is as follows. With the system initially at rest, we desire to rotate the entire chain, to attain a prescribed new attitude. Most spacecraft do this using reaction wheels and/or torquer bars. We will show that a floating robot (such as the Flight Telerobotic Servicer (FTS,[11]), to be flight-tested in 1993) doesn't need one — its manipulators can be used instead. As we will detail in the next section, by moving the joints of the manipulators along some closed path, an attitude change can be accomplished.

In the sections that follow, we will discuss the design of the system. We will see that our architecture is similar to that described in [1].

5.2 Attitude Change via Shape Space Loops

In [5], a formula is given for the phase shift of the central body in a chain, given the loop traversed in the joint angle (or shape) space. Using our notation, the formula is:

$$\Delta\theta_1 = -\int_\Gamma \frac{\mathbf{e}\cdot\mathbf{M}\mathbf{L}\,d\phi}{\mathbf{e}\cdot\mathbf{M}\mathbf{e}}, \tag{5.2.1}$$

where

$\Delta\theta_1$ is the change in attitude of body 1 (the center body),

$d\phi = (d\phi_1, \ldots, d\phi_{n-1})$ is the vector of joint differentials,

$\mathbf{M}$ is the $n \times n$ mass matrix,

$\mathbf{e} = [1\ 1 \cdots 1]^T$,

Γ is the loop traversed in joint space, and

$\mathbf{L}$ is the $n \times (n-1)$ matrix defined by

$$l_{ij} = \begin{cases} 1 & i > j \\ 0 & \text{otherwise} \end{cases}$$

To get an idea how this works, consider the following example. Suppose a person is seated in a swivel chair. Holding a weight (say, a briefcase) with her arm outstretched, she swings her arm from her side to her front. She then draws the weight in to her body, and swings her arm back to the side, and finally she extends her arm again.

The two joints used in this motion have completed a shape space loop, and the person has rotated in the chair. Why? Because the total angular momentum of the system is conserved. Thus, when her arm swings one way, the chair rotates the

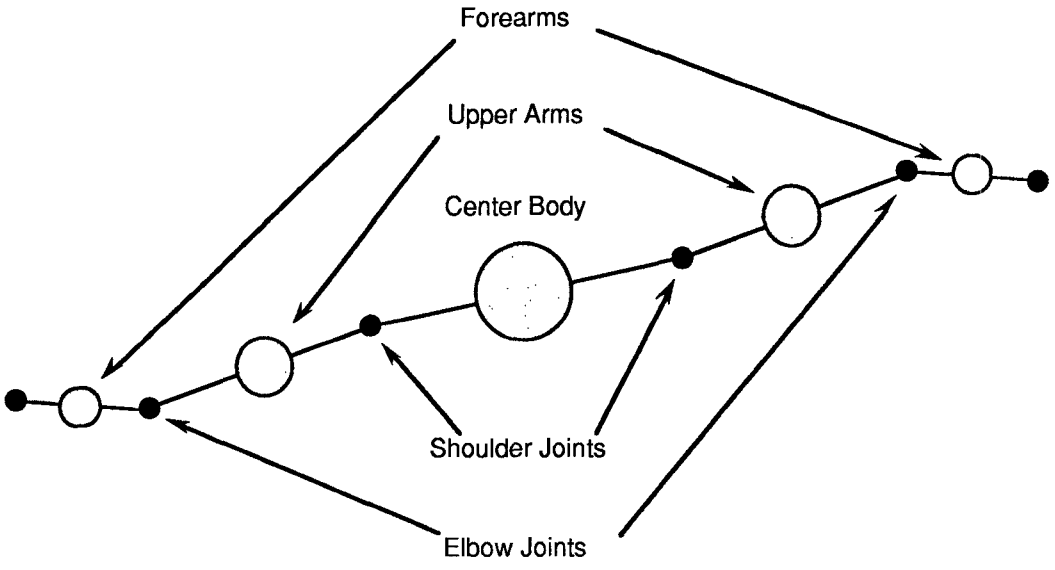


Figure 5.1: Chain terminology

other. Since the inertia of the system during the first swinging motion was larger than during the second one, the chair rotated more to compensate during the first motion. Thus, a net rotation of the system was achieved.

Some might argue that this only happens due to the friction in the example, but we will see later that it works in a frictionless system as well.

For our simulation, we took $n = 5$, which can be thought of a planar robot with a body, and two arms of two links each (see Figure 5.1). We command each arm so that it traverses a path similar to that in the above example. Our top level control problem, then, is to command the motor torques on the four joints of the chain to achieve a specified system rotation of $\Delta\theta_1$.

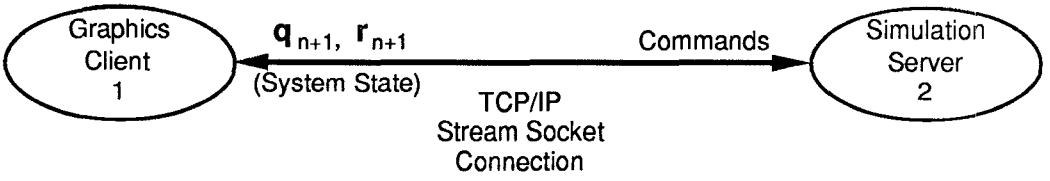


Figure 5.2: Context diagram

5.3 Simulation Server Description

The Simulation Server is the process running on a machine with good floating point performance. This process is responsible for all calculations concerning the system, and must respond to user requests received over the network. A request might be to change a certain parameter, such as the mass of a given body, and restart the simulation. The context diagram for the simulation server is given in Figure 5.2.

The server is broken into two pieces: the controller and the simulator. The Data Flow Diagram (DFD) is shown in Figure 5.3.

The controller handles the interface to the low level (servo) controller, and makes use of the trajectory generator and the inverse dynamics. The DFD for these interfaces is shown in Figure 5.4.

The simulator handles command processing, and the interfaces to the planner, the forward kinematics, and the dynamics. The DFD for these interfaces is shown in Figure 5.5.

In the sections that follow, we go into more detail in describing the various pieces of the simulation.

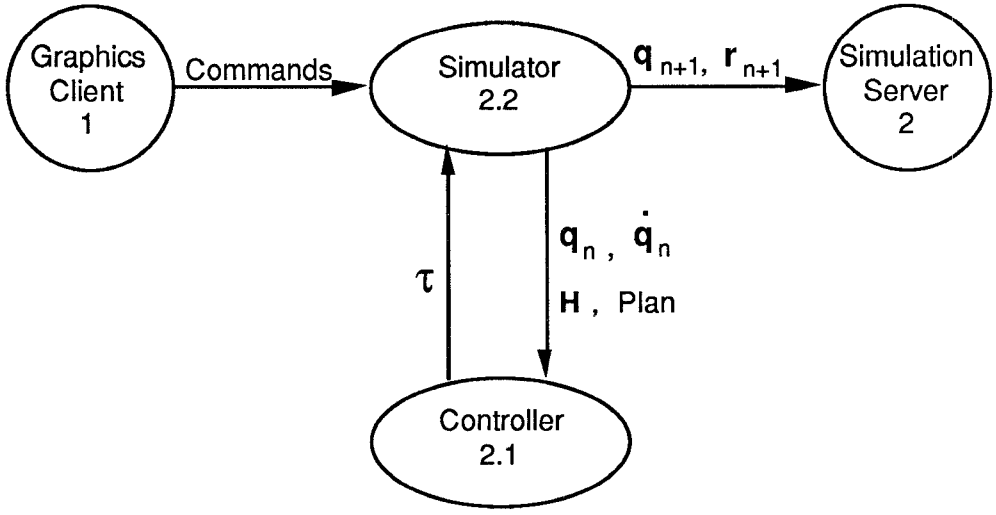


Figure 5.3: Simulation Server DFD

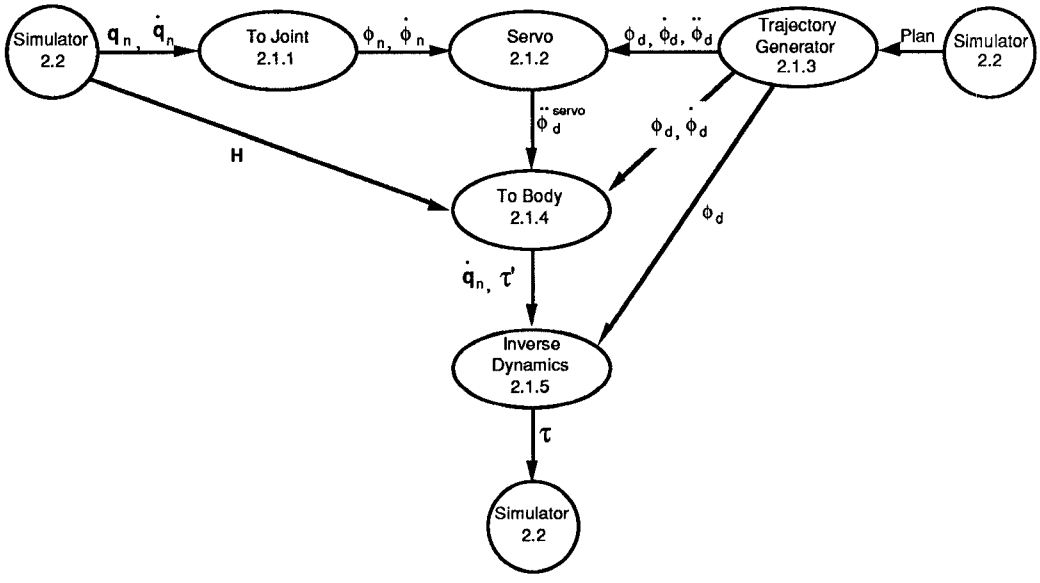


Figure 5.4: Controller DFD

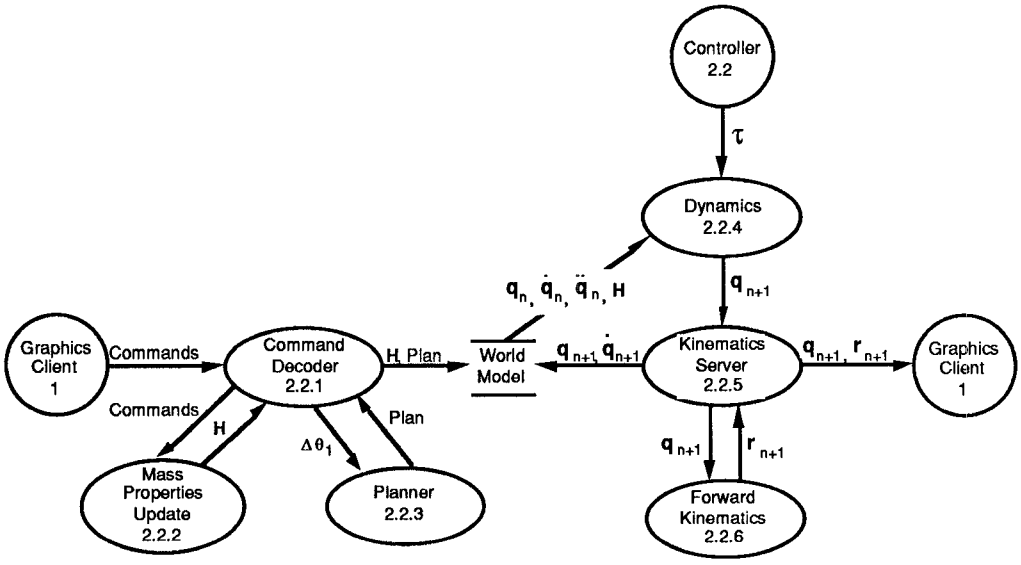


Figure 5.5: Simulator DFD

5.3.1 Dynamical Model

The Dynamical Model is used for two purposes. One is to find out what happens to the system when a certain torque is applied. Given the torque, the dynamical model gives us state information (body angles, rates and accelerations) for the next cycle of the simulation. In addition, looking at the equations of motion in another way, given the desired state of the system, the dynamical model gives us the torque that should be applied.

Dynamics

The dynamics routine is used for the case of prescribing a set of torques (generalized forces) on the system, and finding out how the system reacts. The equations of motion derived in the last chapter are solved using the Newmark technique to find

the accelerations on each body. The accelerations are then integrated to give the rates and angular positions.

Inverse Dynamics

Upon inspection of the equations of motion for the three body case shown in the last chapter, we note that the angular positions of the bodies always appear in differences. Thus, we can use joint angles[1].

This is not the case for the rates and accelerations, however. These quantities must be expressed in configuration space. This is not convenient, since the desired behavior of the system is most naturally thought of in terms of the joint space. This is because the base of the manipulator is not fixed, so when one body moves, others react, making it difficult to specify the desired motions and rates in configuration space.

This difficulty is resolved by converting desired *joint space* information into configuration space for use by the inverse dynamics routine. This is explained further in the next section.

[1]For example, $q_3 - q_1 = \phi_1 + \phi_2$, where the q_i are absolute body angles, and the ϕ_i are joint angles.

5.3.2 Controller

The controller implements the partitioned control law described in [3]. Rewriting the equation of motion (4.2.2) to emphasize the dependence on ϕ, $\dot{\mathbf{q}}$, and $\ddot{\mathbf{q}}$, we have:

$$\mathbf{M}(\phi)\,\ddot{\mathbf{q}} + \mathbf{S}_\phi \dot{\phi} + \mathbf{V}(\phi, \dot{\mathbf{q}}) = \tau$$

where $\mathbf{S}_\phi$ is an $N \times (N-1)$ matrix that satisfies $\mathbf{S}\,\mathbf{q} = \mathbf{S}_\phi \phi$.

We choose the input τ to the system based on the partitioned control law:

$$\tau = \alpha\,\tau' + \beta$$

where α and β are chosen as:

$$\begin{aligned} \alpha &= \mathbf{M}(\phi) \\ \beta &= \mathbf{V}(\phi, \dot{\mathbf{q}}) + \mathbf{S}_\phi \dot{\phi} \end{aligned}$$

Substituting the above definitions into the partitioned control law, we see that with this choice of the model-based parameters α and β, the equation becomes:

$$\tau' = \ddot{\mathbf{q}}$$

so that the system appears to be a unit mass from the τ' input. For the servo portion of the partitioned control law, we set:

$$\ddot{\phi}_d^{\text{servo}} = \ddot{\phi}_d + K_v(\dot{\phi}_d - \dot{\phi}) + K_p(\phi_d - \phi) + K_i \int (\phi_d - \phi)\, dt$$

where the subscript d indicates *desired*, and ϕ is the vector of joint angles.

We need to convert $\ddot{\phi}_d^{\text{servo}}$, which is the desired joint acceleration augmented by the servo error, into configuration space, so that it can be used by the inverse

dynamics. Leaving the details of this transformation to the next section, we write it as:

$$\boldsymbol{\tau}' = \mathbf{f}_a(\phi_d, \dot{\phi}_d, \ddot{\phi}_d^{\text{servo}})$$

$\dot{\phi}_d$ (the desired joint velocity) must also be converted into configuration space for use by the inverse dynamics routine. For now, we write this as $\mathbf{f}_v(\phi_d, \dot{\phi}_d)$. So our final control $\boldsymbol{\tau}$ is:

$$\boldsymbol{\tau} = \mathbf{M}(\phi_d)\, \mathbf{f}_a(\phi_d, \dot{\phi}_d, \ddot{\phi}_d^{\text{servo}}) + \mathbf{V}(\phi_d, \mathbf{f}_v(\phi_d, \dot{\phi}_d)) + \mathbf{S}_\phi \phi_d \tag{5.3.2}$$

We are left with the problem of finding the functions $\mathbf{f}_v$, and $\mathbf{f}_a$, which map joint angle, rate and acceleration into the body rate and acceleration.

Joint Space to Configuration Space Transformation

The inverse dynamics were written in terms of joint angle, body rate, and body acceleration. We need a function that will compute body rate given the joint state of the system, and another function to compute body acceleration given the joint state.

To derive these functions, we use conservation of momentum to supply the missing equations (we are given $N - 1$ joint rates and accelerations, and need the N body rates and accelerations). With $\mathbf{e}$ and $\mathbf{L}$ as defined previously, we write the body rates in terms of the joint rates as follows:

$$\dot{\mathbf{q}} = \dot{q}_1 \mathbf{e} + \mathbf{L}\dot{\phi} \tag{5.3.3}$$

The interpretation of the above equation is that the rate of the first body in the chain is propagated down the chain via the joint rates to give the body rates of the other bodies. Premultiplying the above equation by the mass matrix, we have:

$$\mathbf{M}(\phi)\,\dot{\mathbf{q}} = \dot{q}_1\mathbf{M}(\phi)\,\mathbf{e} + \mathbf{M}(\phi)\,\mathbf{L}\,\dot{\phi}$$

Next, we dot both sides with **e**:

$$\mathbf{e}\cdot\mathbf{M}(\phi)\,\dot{\mathbf{q}} = \dot{q}_1\,\mathbf{e}\cdot\mathbf{M}(\phi)\,\mathbf{e} + \mathbf{e}\cdot\mathbf{M}(\phi)\,\mathbf{L}\,\dot{\phi}$$

We now recognize that the quantity on the left is the angular momentum of the system, μ. In general, this is a known constant, since momentum is conserved by the system. In our case, we take $\mu = 0$, which gives us an equation for $\dot{q}_1$:

$$\dot{q}_1 = -\frac{\mathbf{e}\cdot\mathbf{M}(\phi)\,\mathbf{L}\,\dot{\phi}}{\mathbf{e}\cdot\mathbf{M}(\phi)\,\mathbf{e}} \tag{5.3.4}$$

Substituting $\dot{q}_1$ into equation 5.3.3 gives us $\mathbf{f}_v$:

$$\mathbf{f}_v(\phi,\dot{\phi}) = \dot{\mathbf{q}} = -\frac{\mathbf{e}\cdot\mathbf{M}(\phi)\,\mathbf{L}\,\dot{\phi}}{\mathbf{e}\cdot\mathbf{M}(\phi)\,\mathbf{e}}\mathbf{e} + \mathbf{L}\,\dot{\phi}$$

For the accelerations. we differentiate equation 5.3.4, obtaining the relationship:

$$\ddot{q}_1 = \frac{(\mathbf{e}\cdot\mathbf{M}(\phi)\,\mathbf{L}\,\dot{\phi})(\mathbf{e}\cdot\frac{d\mathbf{M}(\phi)}{dt}\,\mathbf{e})}{(\mathbf{e}\cdot\mathbf{M}(\phi)\,\mathbf{e})^2} - \frac{\mathbf{e}\cdot\frac{d\mathbf{M}(\phi)}{dt}\,\mathbf{L}\,\dot{\phi} + \mathbf{e}\cdot\mathbf{M}(\phi)\,\mathbf{L}\,\ddot{\phi}}{\mathbf{e}\cdot\mathbf{M}(\phi)\,\mathbf{e}}$$

The other accelerations are found from the time derivative of equation 5.3.3:

$$\mathbf{f}_a(\phi,\dot{\phi},\ddot{\phi}) = \ddot{\mathbf{q}} = \ddot{q}_1(\phi,\dot{\phi},\ddot{\phi})\,\mathbf{e} + \mathbf{L}\,\ddot{\phi}$$

Projector

There is a problem with the control law (5.3.2). In general, the law is *unphysical*, since the sum of the elements of the torque vector do not always add to zero. For a model of a physical system, controlled by motors at the joints, the torques generated are internal, and thus should add to zero.

We solve this problem using a *projector*. We derive a matrix P that will transform a nonphysical torque vector into a physical one. We also require that P leave physical torque vectors unchanged. We write the physical requirement as follows. We rewrite (5.3.2) leaving out the arguments of $\mathbf{f}_a$ and $\mathbf{f}_v$ for brevity:

$$\boldsymbol{\tau} = \mathbf{M}(\phi_d)\,\mathbf{f}_a + \mathbf{V}(\phi_d, \mathbf{f}_v) + \mathbf{S}_\phi \dot{\phi}_d$$

Then the matrix P should transform $\boldsymbol{\tau}$ into a physical torque T:

$$\mathbf{T} = \mathbf{P}\,\boldsymbol{\tau}$$

If T is physical, we have $\mathbf{e} \cdot \mathbf{T} = \mathbf{e} \cdot \mathbf{P}\,\boldsymbol{\tau} = \mathbf{0}$. This holds if the physical condition is met:

$$\mathbf{P}^{\mathrm{T}}\mathbf{e} = \mathbf{0} \tag{5.3.5}$$

We also require that $\mathbf{P}$ leave unchanged torques that are already physical. In the setting of planar N-body problems with applied internal torques due to joint motors, we can write the physical torque vector as:

$$\begin{aligned}
\mathbf{T} &= \begin{bmatrix} \widetilde{\tau}_1 \\ -\widetilde{\tau}_1 + \widetilde{\tau}_2 \\ -\widetilde{\tau}_2 + \widetilde{\tau}_3 \\ \vdots \\ -\widetilde{\tau}_{N-2} + \widetilde{\tau}_{N-1} \\ -\widetilde{\tau}_{N-1} \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ -1 & 1 & 0 & \ddots & \vdots \\ 0 & -1 & 1 & \ddots & 0 \\ 0 & 0 & -1 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \cdots & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} \widetilde{\tau}_1 \\ \widetilde{\tau}_2 \\ \widetilde{\tau}_3 \\ \vdots \\ \widetilde{\tau}_{N-1} \end{bmatrix} \\
&= \tilde{\mathbf{Q}}\,\widetilde{\boldsymbol{\tau}}
\end{aligned}$$

The requirement that $\mathbf{P}$ not change physical torques is equivalent to writing

$$\mathbf{P}\tilde{\mathbf{Q}}\,\widetilde{\boldsymbol{\tau}} = \tilde{\mathbf{Q}}\,\widetilde{\boldsymbol{\tau}}$$

for all motor torque vectors $\widetilde{\boldsymbol{\tau}}$. We can thus write the other requirement for $\mathbf{P}$ as:

$$\mathbf{P}\tilde{\mathbf{Q}} = \tilde{\mathbf{Q}} \tag{5.3.6}$$

If one writes down the $N \times (N-1)$ equations in (5.3.6) and substitutes them back into (5.3.5), one finds that (5.3.5) only provides one additional independent equation. We thus have $N^2 - (N-1)$ equations in the N^2 unknown elements of $\mathbf{P}$, which gives us a family of solutions parameterized by $N-1$ parameters.

For the $N = 5$ case, we give the general form of $\mathbf{P}$ in terms of five parameters and one constraint (equivalent to four independent parameters):

$$\mathbf{P} = \begin{bmatrix} \alpha & \alpha-1 & \alpha-1 & \alpha-1 & \alpha-1 \\ \beta-1 & \beta & \beta-1 & \beta-1 & \beta-1 \\ \gamma-1 & \gamma-1 & \gamma & \gamma-1 & \gamma-1 \\ \delta-1 & \delta-1 & \delta-1 & \delta & \delta-1 \\ \varepsilon-1 & \varepsilon-1 & \varepsilon-1 & \varepsilon-1 & \varepsilon \end{bmatrix}, \text{ where } \alpha+\beta+\gamma+\delta+\varepsilon = 4$$

A block diagram of the control system we have just completed describing is given in Figure 5.6. The block labeled $\mathbf{D}^{-1}$ represents the inverse dynamics, $\mathbf{D}$ represents the dynamics, and $\mathbf{P}$ represents the projector.

Trajectory Generator

For a joint motion from one angle to another, we use a ramp-ramp trajectory, so named for the shape of the velocity profile. See Figure 5.7. In a more general system, there would be a maximum motor velocity imposed, which results in velocity limiting. This results in a velocity profile called 'ramp-coast-ramp.'

The trajectory generator takes desired joint acceleration magnitude $\ddot{\phi}_d$, final time $\mathbf{t}_f$, initial joint angle ϕ_0, and final desired joint angle ϕ_f, and for the current simulation time t, computes the next desired joint state of the system. The equations

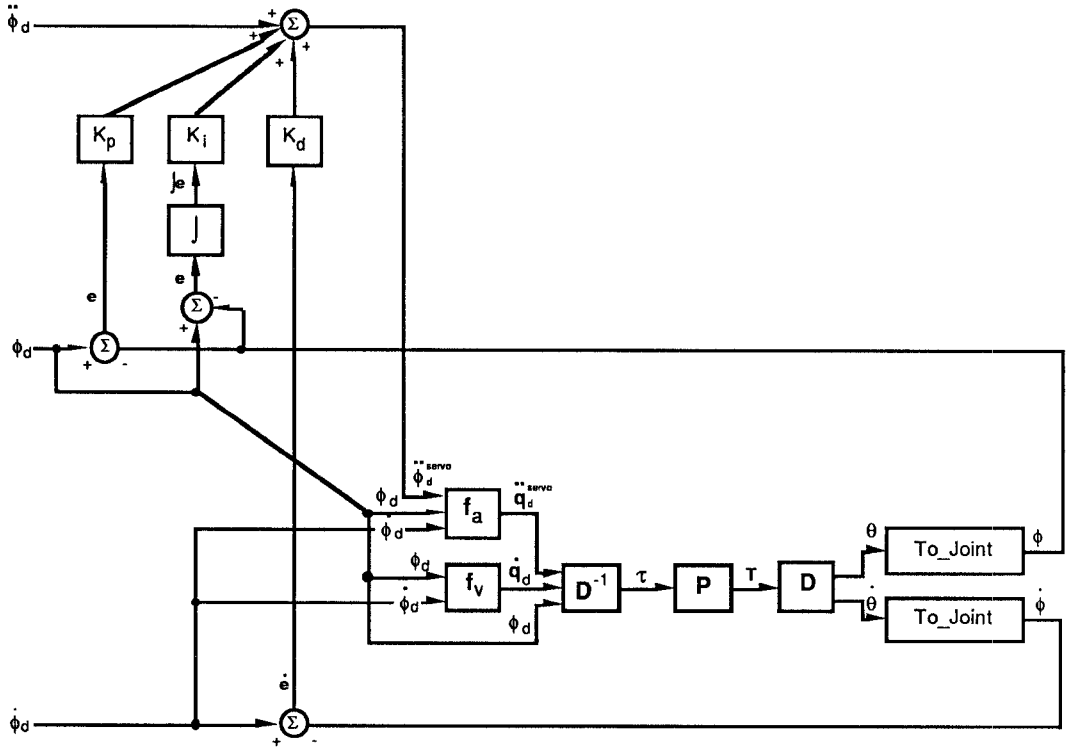


Figure 5.6: Block diagram for the joint-servo control system

of the three profiles in Figure 5.7 of a single joint are:

$$\left.\begin{aligned} \ddot{\phi}_d(t) &= +\ddot{\phi}_d \\ \dot{\phi}_d(t) &= \ddot{\phi}_d t \\ \phi_d(t) &= \tfrac{1}{2}\ddot{\phi}_d t^2 + \phi_0 \end{aligned}\right\} \quad 0 \le t \le \frac{t_f}{2}$$

$$\left.\begin{aligned} \ddot{\phi}_d(t) &= -\ddot{\phi}_d \\ \dot{\phi}_d(t) &= \ddot{\phi}_d(t_f - t) \\ \phi_d(t) &= -\tfrac{1}{2}\ddot{\phi}_d(t_f - t)^2 + \phi_f \end{aligned}\right\} \quad \frac{t_f}{2} < t \le t_f$$

The collection of angle, velocity and acceleration trajectories for all of the joints that take the system from one point in joint state space to another is called a *path segment*. The planner outputs a plan which consists of several path segments. The

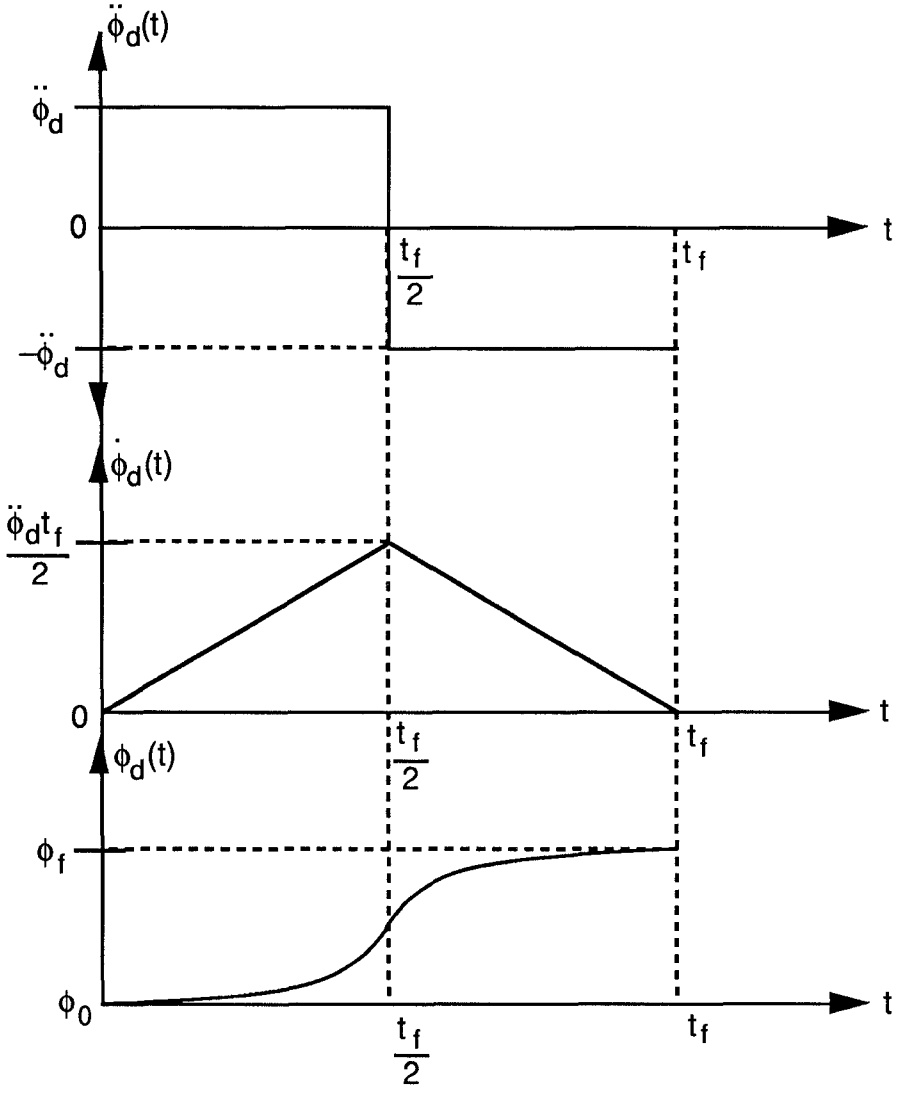


Figure 5.7: Ramp-ramp trajectories for a single joint

trajectory generator accepts the plan as input, and at each iteration, checks for a path segment switch. This happens when the simulation time t exceeds the motion duration t_d for all joints. At this point, the simulation time t is reset to zero, and the plan step counter is incremented.

The trajectories used here are such that when taken in succession, the velocity profile of each joint is piecewise linear.

We will go into more detail about the planner during the discussion of the simulator in the following section.

5.3.3 Simulator

The simulator handles the interface to the world model (the collection of the mass property information and the state of the system), the planner, the dynamics, and the kinematics. The torque calculated by the controller and the state of the system from the last iteration (from the world model) is fed into the dynamics to yield the current state of the system. This is stored in the world model, and the forward kinematics are used to find the center of mass locations for each body, given the absolute body angles. The center of mass locations are and body angles are output over the network to the graphics client.

The graphics client may have a request for the server to act upon. This is handled by the command processor.

Command Processing

There are four commands implemented so far:

`RESET` — This is used to initialize the system, and to set all system parameters to their default values.

`MATRIX` — This command toggles between two matrix inversion routines used by the linear equation solver: Gauss-Siedel, and LU decomposition.

`CONTROL` — This is used to update all system parameters to the values received from the graphics client. This includes desired system rotation, masses, lengths, inertias, and time step.

`QUIT` — This is used to perform an orderly shutdown of the system. The `QUIT` command is echoed back to the graphics client, and the process exits to the operating system.

When the `CONTROL` command is received from the client, the new mass property information is used to update the mass coefficient matrix, which contains the $h_{i,j}$ values described in the last chapter. In addition, the new value for the desired system rotation $\Delta\theta_1$ is input to the planner to generate a new plan.

Planner

We define a *plan* as a set of connected path segments. Each plan is designed to accomplish a task. In our case, the task is to reorient the chain of rigid bodies, achieving the desired rotation, $\Delta\theta_1$.

So far in this chapter, and in the previous chapter, all of the algorithms and their implementation were designed with the general N body chain in mind. When appropriate, examples were given with $N = 3$ in the interest of showing the structure of the equations for a case containing a middle body, while keeping the expressions reasonably short.

From this point forward, we will take $N = 5$, and design a plan generator specific for this chain. We will speak of the four joints of the chain as the two shoulder joints, and the two elbow joints, with a planar FTS in mind (Figure 5.1) as an example.

The general form of the plan, parameterized by an angle α, is as follows.

1. The system is initially at rest, with all joint angles zero.
2. The shoulder joints move in opposite directions, to an angle α.
3. The elbow joints move to an angle of 180 degrees, folding the arms.
4. The shoulder joints move back to zero, carrying a smaller load.
5. The elbow joints move back to zero, unfolding the arms.

This plan results in a net rotation of the system, with a magnitude that is a function of the displacement α. The goal of the planner is to determine the angle α that will yield the desired system rotation, and generate the four step plan described above that is input to the trajectory generator.

The joint angle profiles for our plan are shown in Figure 5.8. These four profiles taken together form the shape space loop Γ over which we integrate equation 5.2.1.

The motion of the chain is depicted at the bottom of the figure via snapshots at the end of each path segment.

We now give the result of working out the integral for our path:

$$\begin{aligned}
\Delta\theta_1 &= \int_0^{\alpha} \frac{h_{3,3} + \eta\cos(\phi_2)}{\sigma + 2\eta\cos(\phi_2)}\, d\phi_2 + \int_0^{\pi} \frac{A(\alpha) + B\cos(\phi_1) + C\cos(\phi_1+\alpha)}{D(\alpha) + 2B\cos(\phi_1) + 2C\cos(\phi_1+\alpha)}\, d\phi_1 \\
&+ \int_{\alpha}^{0} \frac{h_{3,3} + (H-C)\cos(\phi_2)}{L + 2(H-C)\cos(\phi_2)}\, d\phi_2 \\
&+ \int_{\pi}^{0} \frac{A(0) + (B+C)\cos(\phi_1) + C\cos(\phi_1+\alpha)}{D(0) + 2(B+C)\cos(\phi_1)}\, d\phi_1 \qquad (5.3.7)
\end{aligned}$$

where:

$$\begin{aligned}
\sigma &= h_{1,1} + h_{2,2} + h_{3,3} + h_{4,4} + h_{5,5} + 2(h_{1,2} + h_{1,4} + h_{1,5} + h_{2,4} + h_{2,5} + h_{4,5}) \\
\eta &= h_{1,3} + h_{2,3} + h_{3,4} + h_{3,5} \\
A(\alpha) &= h_{2,2} + h_{3,3} + h_{4,4} + 2h_{2,4} + 2(h_{2,3} + h_{3,4})\cos(\alpha) \\
B &= h_{1,2} + h_{1,4} + h_{2,5} + h_{4,5} \\
C &= h_{1,3} + h_{3,5} \\
D(\alpha) &= h_{1,1} + h_{2,2} + h_{3,3} + h_{4,4} + h_{5,5} + 2(h_{1,5} + h_{2,4}) + 2(h_{2,3} + h_{3,4})\cos(\alpha) \\
H &= h_{2,3} + h_{3,4} \\
L &= h_{1,1} + h_{2,2} + h_{3,3} + h_{4,4} + h_{5,5} + 2(h_{1,5} + h_{2,4} - h_{1,2} - h_{1,4} - h_{2,5} - h_{4,5})
\end{aligned}$$

Now that we have written down the relationship between α and $\Delta\theta_1$ in terms of the mass properties of our system, it is fairly straightforward to invert (5.3.7) numerically. The following sequence of actions is performed when the system initializes, and whenever the mass properties of the system are changed.

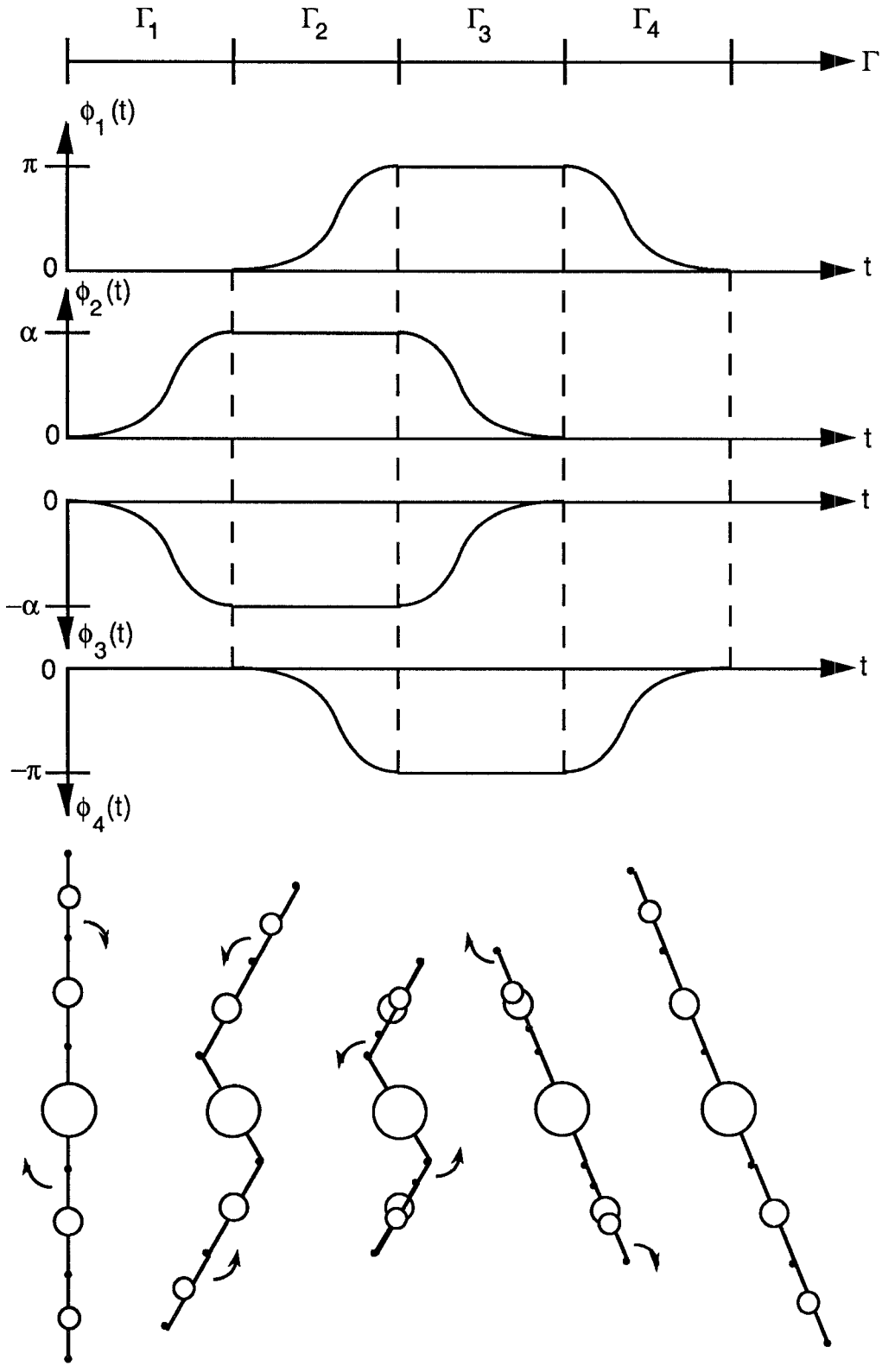


Figure 5.8: Definition of shape space loop

First, using the Rhomberg integration routine provided in [8], we evaluate equation 5.3.7 for twenty values of α evenly spaced over the region of interest, which we take to be $(0, \pi)$. This gives us the function in tabular form.

Next, we fit a five term polynomial to the tabular data, making use of the Least Squares routine in [8]. This gives us a close approximation to the relationship, using a well behaved function. Finally, we find the root of the polynomial in the region of interest to give us our answer, α, given $\Delta\theta_1$. An alternative approach would be to use linear interpolation on the tabular data.

Once α is known, it is a simple matter for the planner to fill in the plan structure needed by the trajectory generator. The plan can be thought of as four two dimensional arrays. The indices for all of the arrays are the joint (1 – 4), and the plan step (1 – 4). The four pieces of information for each of the sixteen joint/step pairs are the initial angle ϕ_0, the final angle ϕ_f, the desired acceleration $\ddot{\phi}_d$, and the final time t_f. The first three of these are shown in Table 5.1. The same acceleration a is used for all motions, since it doesn't have any effect on the system rotation.

The last task for the planner is to calculate the final times for each of the sixteen joint/step pairs using the formula:

$$t_f = 2\sqrt{\frac{|\phi_f - \phi_0|}{\ddot{\phi}_d}}$$

	Joint	Plan Step			
		1	2	3	4
ϕ_0	1	0	0	π	π
	2	0	α	α	0
	3	0	$-\alpha$	$-\alpha$	0
	4	0	0	$-\pi$	$-\pi$
ϕ_f	1	0	π	π	0
	2	α	α	0	0
	3	$-\alpha$	$-\alpha$	0	0
	4	0	$-\pi$	$-\pi$	0
$\ddot{\phi}_d$	1	0	a	0	$-a$
	2	a	0	$-a$	0
	3	$-a$	0	a	0
	4	0	$-a$	0	a

Table 5.1: Definition of the plan

5.4 Client/Server Interface

Until now, we have deferred the discussion of the data that is passed back and forth between the client and server, and the protocol they use. The physical connection between the two processes is Ethernet.

Layered on top of the Ethernet protocols is the Internet Protocol (IP), and the Transmission Control Protocol (TCP). As discussed in chapter 3, UNIX provides the programming interface to the TCP/IP socket abstraction via system calls. The *libipc(3)* library, described in chapter 3, is used to open the connection for bidirectional data flow between the client and the server. However, all of the data and protocol used by the two simulation partners is left entirely up to the designer.

We begin by discussing initialization and shutdown. The system is started by a user seated at the graphics workstation. The graphics client program is invoked at the shell prompt, with the name of the server machine passed on the command

line. The server program is invoked by the client program using a system call and the `rsh` mechanism. Once the socket connection is established, the client sends the server a `RESET` command, in order to receive the initial locations of the objects to animate on the screen.

At the end of the simulation, upon user request, the graphics client program sends the server a `QUIT` command, waits for an acknowledgement, and then terminates.

At any time during the simulation, the user can request changes in the mass properties of the system. When done specifying the new data using the graphical user interface, the user hits the `Send Data` button. The client program assigns this command a new sequence number, and sends a packet of data to the server. It then ignores all data packets received from the server until it receives one containing the new sequence number. The client then begins animating the new simulation.

The data formats for client to server and server to client have much in common, but are not the same. Table 5.2 shows the data sent from the client to the server, and Table 5.3 shows the data sent from the server to the client. (The initial angle and velocity data is used to study the free dynamics of the system.)

5.5 NASREM

The NASA/NBS Standard Reference Model for Telrobot Control System Architecture (NASREM), is helpful for laying out distributed robot control systems. In this section, we begin by giving a brief overview of NASREM, and then show how our control system fits into the general NASREM structure.

Item	Description
tag	The command sequence number
command	The command in integer form
TimeStep	The simulation time step
InitAng[]	The initial body angles
InitVel[]	The initial body rates
Mass[]	The masses of the bodies
Inertia[]	The inertia about the body center of mass
Length[]	The joint to joint body lengths
iter	The number of iterations for Gauss-Siedel inversion
gauss	The LU decomposition or Gauss-Siedel matrix inversion toggle flag
phase	The desired system rotation

Table 5.2: Data sent from client to server

5.5.1 NASREM Overview

The NASREM architecture, described in [1], specifies an approach for telerobot control based on a hierarchy for each of three system components: SENSORY PROCESSING, WORLD MODELING, and TASK DECOMPOSITION. There are six levels described for each: SERVICE MISSION, SERVICE BAY, TASK, E-MOVE, PRIMITIVE, and COORDINATE TRANSFORM SERVO (see Figure 5.9).

The TASK DECOMPOSITION modules are responsible for decomposing a goal into its components, from a high level manipulative goal down to low level motor torque commands. These modules are also responsible for monitoring the status information returned from the lower level modules.

Item	Description
tag	The command sequence number
command	The command in integer form
TimeStep	The simulation time step
InitAng[]	The initial body angles
InitVel[]	The initial body rates
Mass[]	The masses of the bodies
Inertia[]	The inertia about the body center of mass
Length[]	The joint to joint body lengths
iter	The number of iterations for Gauss-Siedel inversion
gauss	The LU decomposition or Gauss-Siedel matrix inversion toggle flag
phase	The desired system rotation
x[]	The array of x coordinates of the bodies
y[]	The array of y coordinates of the bodies
ang[]	The array of body attitudes
eqmo	The sum of the absolute values of the LHS of the equations of motion (should be zero)
momentum	The momentum of the system (should be constant)
lagrangian	The difference between kinetic and potential energies in the system

Table 5.3: Data sent from server to client

Each TASK DECOMPOSITION module consists of three submodules:

- Job assignment manager
- Planners
- Executors

The job assignment manager decomposes a task into jobs to be performed by a planner/executor pair. Jobs are executed concurrently, with a planner composing

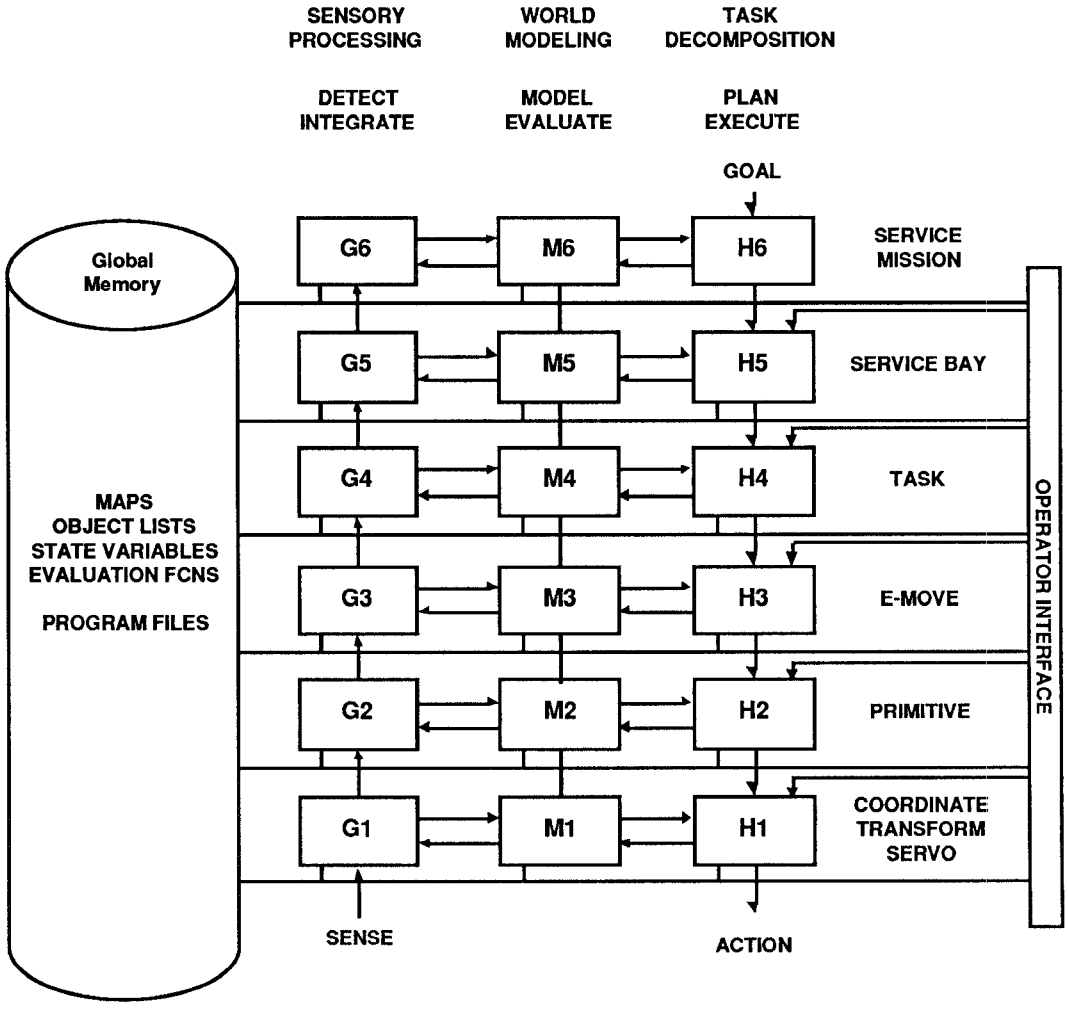


Figure 5.9: NASREM hierarchy definition

a plan consisting of a sequence of actions to perform, and the executor executing the plan.

The SENSORY PROCESSING modules are responsible for providing all levels of the world model with current input. Higher level responsibilities include obstacle detection and object recognition, and lower levels include feature extraction and edge detection.

The WORLD MODELING modules are responsible for servicing information requests from other modules and maintaining all of the data describing the system under control and its environment. This data must be broken down into the different levels and representations required by all of the other modules.

All of the data in the world model is stored in global memory. This memory need not reside on the same machine or even the same bus, but all modules must be able to address all of the data. The flow of information among horizontal modules is such that the data rate on a given level is approximately an order of magnitude slower than the rate on the level below. Data passing back and forth vertically along the hierarchy is usually such that commands are passed down the hierarchy and periodic status updates are passed up the hierarchy.

5.5.2 NASREM Breakdown of Chain System

Below, we will show how the system described here fits into the NASREM approach.

Since our system controls a simulation, not real hardware, we are obviously lacking in Sensory Processing support. As such, we won't bother discussing sensor-related issues as they pertain to NASREM.

Our system lives in a very simple two dimensional universe, free from obstacles and other manipulators, so the World Modeling software consists of routines that maintain only data describing our chain. Still, we will make note of the fact that this data is separated into two categories: relatively static (constant until the user wishes to change it), and dynamic (data that changes at nearly every simulation

iteration). The static information includes the mass properties of the system, the time step, and the desired system rotation. The joint and body angles, rates, and accelerations make up the dynamic information.

It makes sense to place control of the dynamic information in the world model at the lowest level of the hierarchy, and control the static information from higher levels. At the highest level, access to static information is done via the graphical user interface running on the graphics machine. At lower levels, state information is updated by the simulation loop as a result of running the dynamics and kinematics at each iteration.

The Task Decomposition portion of our system is broken down into three NASREM levels as follows (see Figure 5.10). At the lowest level we have the servo controller, which implements a partitioned PID control law to determine the control torque for the system. The control software one level above this is the trajectory generator, which takes a plan as input, and outputs piecewise linear velocity trajectories, and corresponding joint angles and accelerations. One level above this is the planner, which takes desired system rotation as input, and calculates a plan consisting of four path segments that will achieve the desired rotation.

While our planner is very specific, it does represent a `PRIM` level module in the NASREM hierarchy for Task Decomposition. Our trajectory generator represents an `E-MOVE` module, and our servo controller is at the `SERVO` level.

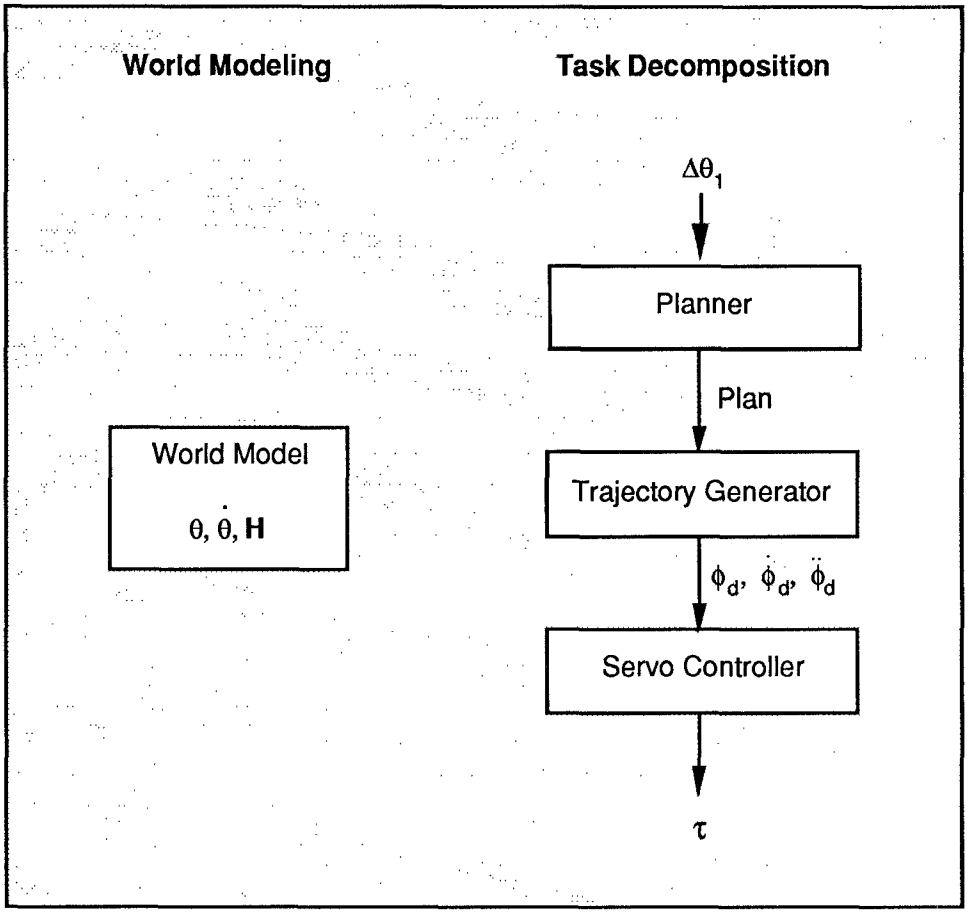


Figure 5.10: NASREM levels of chain attitude control system

We note that our system only barely spans three of the six levels described by NASREM. This illustrates how general the NASREM architecture is. At the highest level, the input is a command such as `SERVICE SATELLITE`, and at the lowest level, it is a joint motor torque.

5.6 Summary

We began our description of the control system for our simulation with a discussion of how a floating robot can reorient itself in a zero gravity environment by

'swimming in space,' or moving its manipulators through shape space loops. We then discussed how the equation for the attitude change was implemented.

We described each component of the control system, and how it interfaces with the others. We defined the protocol used by the client and server to communicate, and what data they pass back and forth to each other.

We completed the control system description with a discussion of how it relates to the NASREM architecture.

Chapter 6

Conclusion

6.1 Summary

This work covers a broad range of topics relating to real-time simulation of dynamical systems. In order to improve the performance of the simulation, we have shown how to employ the Client/Server model to break the problem into two pieces.

We discussed the design issues involved with making the best use of the simulation iteration time period for numerical and graphical calculation. We reviewed object oriented graphics software design, and presented some of its merits and drawbacks. We found that implementing object sorting in software was a reasonable compromise between display realism and performance.

We discussed the user interface in some detail, presenting two different graphical approaches. We found that graphical user interfaces are a convenient means for making erroneous user input impossible.

The connection between the two parts of the simulation over the network was discussed. The Berkeley UNIX Interprocess Communications facilities were presented, along with the design of *libipc(3)*, a collection of IPC programming tools utilized by researchers in the Intelligent Servosystems Laboratory.

We studied the dynamics of a chain of rigid bodies moving in the plane. The mass properties were written as in [12], and the equations of motion were derived in a Lagrangian setting. We then showed how to apply the Newmark technique to solve the equations. We showed how the MACSYMA system was used to facilitate the symbolic calculations, and produce C source code ready to incorporate into the simulation.

Finally, we studied the attitude control problem for the chain. We found that the lack of a fixed point in the system led to complications in the control law design. This was due to the fact that the equations of motion were written in configuration space, and servoing is most naturally done in joint space. We presented a solution to this difficulty derived from momentum conservation. The control software was found to have many similarities with the NASREM architecture, developed at the National Institute of Standards and Technology, although this was not an original goal.

During the course of this work, the idea of distributed computing has become increasingly popular. We have several suggestions for continuing research in this area, taking advantage of recent industrial advancements.

6.2 Recommendations for Future Work

In the following sections, we give ideas and suggestions for work that we feel should be done in the areas we have explored in this research.

6.2.1 Animation

In order to take advantage of the new graphics workstations that are capable of animating a fairly complex scene using Gouroud-shading, a small library of primitive solid (or surface) shapes should be constructed. Gouroud-shaded models could then be built out of these primitives.

Often, as in the case of our space station simulation, data for an object in the simulation comes from a Computer Aided Design (CAD) package, in the form of an Initial Graphics Exchange Specification (IGES) file. This data usually contains only wireframe information. It is possible to take wireframe data and synthesize a surface that bounds it, for use in the animation. An algorithm for surface synthesis should be implemented.

6.2.2 NeWS, X, *Open Look* and *Motif*

There are many standards, formal and defacto, emerging in the areas of distributed computing, portable and open operating systems, and graphical user interfaces. The battle between UNIX International (UI, which includes AT&T and Sun) and the Open Software Foundation (OSF, which includes IBM, DEC, and HP-Apollo) will produce several powerful tools for research in distributed simulation.

In chapter 2 we presented two approaches to the problem of creating a Graphical User Interface (GUI) to the simulation. Neither of these was based on a standard of any kind. This is because at the time of implementation, there was no clear choice as to what would be the standard graphical user interface. Today, we know that there are two main contenders: *Motif*, from the OSF, and *Open Look*, from UI.

Motif derives from the X Window System (X), and *Open Look* is a descendent of the Network extensible Window System (NeWS). Both offer similar capabilities, with the main advantages being portability and look-and-feel.

The NASA/AMES Panel Library, introduced in chapter 2, has an outstanding look-and-feel, but only runs on Silicon Graphics workstations — a serious drawback.

Porting the user interface of our simulation to one of the two emerging standards would make many more Client/Server hardware combinations possible, and would allow our lab to share software with other research institutions more easily.

6.2.3 RPC and XDR

Sun Microsystems created one of the first defacto standards in the workstation market with the Network File System (NFS). Two of the peripheral parts of NFS are the eXternal Data Representation library and specification, and the Remote Procedure Call (RPC).

In chapter 3 we discussed the difficulties involved with transferring data from one machine to another in a heterogeneous network. ASCII message passing is often used, but is exceedingly slow. We decided to take advantage of the fact

that the ISL's computers all have nearly identical data storage formats, by passing structures of data back and forth between them.

While this method is fast, it is clearly not portable, say, to VAX computers, since they store data differently from most workstations.

To make the data transfer more robust, eXternal Data Representation routines should be used to provide a means of encoding and decoding local structures of data into a network standard formant. This would allow simulations to be distributed over IRISs, Suns, HPs and VAX workstations, among others.

If the routines prove to be fast enough, the Remote Procedure Call provided with NFS should also be used. Using this method of interprocess communication hides the data format problem within a higher level interface between programs on different machines. Eventually, *libipc(3)* could be replaced with a supported product.

6.2.4 The Network Computing System

What makes matters difficult is that the two UNIX factions both have solutions to distributed computing problems [4]. The Network Computing System, originated by Apollo, is perhaps the most impressive. It includes an RPC specification incompatible with Sun's, and a data representation scheme known as the Network Data Representation model (NDR). Instead of always transforming data from a local representation to a network standard, as in XDR, the NDR method tags the data, identifying the type of machine from which it was sent. If the destination machine has the same format, the conversion process is bypassed.

Perhaps the most interesting part of NCS is the Location Broker (LB). Using the LB, one can set up a network of machines that cooperate to provide computation services for each other. A client process is described by an estimate of the resources it will require, and is submitted to the LB. The LB then polls the available machines on the network for bids on the task, that are weighted based on the required resource mixture. The task is given to the machine giving the highest bid.

Sun offers a system called Open Network Computing (ONC) that encorporates NFS, XDR, and the *yellow pages* service to accomplish similar results.

Both of these systems should be evaluated for applicability to the research plans of the ISL.

6.2.5 Dynamical Modeling and Numerical Solution

A motor model should be added to the dynamical simulation, including bearing friction. Corresponding to this, a maximum motor speed (and hence joint rate) would be imposed, so the trajectory generator should be modified to use a 'ramp-coast-ramp' velocity profile when necessary.

The zero-gravity three dimensional dynamics of simple manipulators with revolute joints should be studied. The results of these simulations would be of use to the long term FTS project.

As the models become more complicated, more sophisticated numerical methods will be required. Other forms of the Newmark technique may be more suitable to these problems.

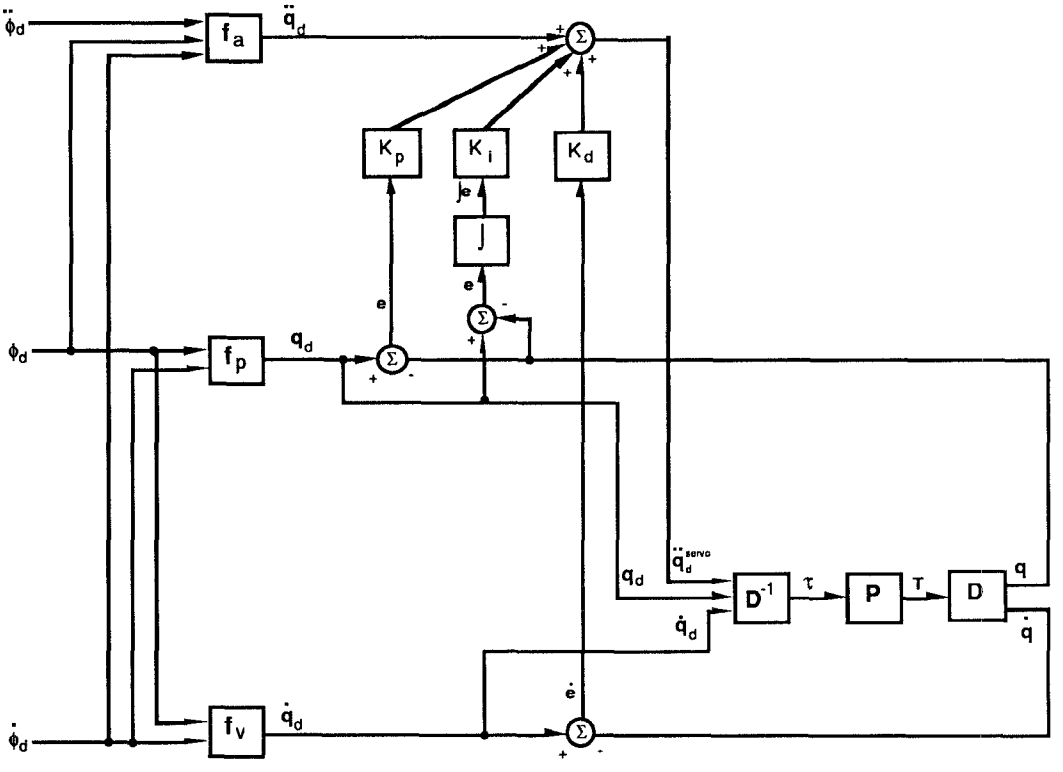


Figure 6.1: Block diagram for the body-servo control system

6.2.6 Control System

The servo portion of the partitioned control law discussed in Chapter 5 was computed in joint space. It is possible to do this in configuration space. This would require the existence of a transformation from desired joint state information to desired absolute body angles. Integrating the $\mathbf{f}_v$ transformation derived in Chapter 5 would produce the desired relationship, $\mathbf{f}_p$. The feedback loop would then be free from transformations between joint space and configuration space — everything would be done in configuration space. The block diagram for this system is shown in Figure 6.1. The configuration space servo law should be compared with the joint space servo law.

For either of these control systems, the PID gains should be generalized from their present scalar form to diagonal matrices. Gain scheduling, based on the system's configuration, may be helpful.

The trajectory generator used in the present system makes a transition from one path segment to the next based on the motion duration for all of the joints. A further check might be appropriate. Before executing a path segment switch (switching to the next 'ramp-ramp' trajectory in the plan), all of the joints should be given several iterations to settle to within some tolerance of their desired joint angles.

Finally, the planner should be considerably more general. Different plans should be made available, with at least one plan defined for each of the numbers of degrees of freedom (DOF) possible for a given system. For example, in the five body system, we have implemented a four DOF plan. The same system is capable of executing two and three DOF plans as well, so these should be made available as options.

Appendix A

Libipc(3) Source Code

```
/* "ipc.c" */

#ifndef lint
static char SccsIdIpc[] = "%W% %H% %T%";
#endif

#include <stdio.h>
#include "islipc.h"
#include "debug.h"

extern int NewRequest;
/*
  (Must be declared by the user.
  Used for signaling receipt of SIGIO interrupt.)
*/


start_server(remote_host, service, sock)
char *remote_host, *service;
int *sock;
{
  char command[1024], local_host[25], cLocalPort[10];
  int s, iLocalPort, namelen = 24;

  struct hostent *hp, *gethostbyname();
/*
  Get local host's name
*/
  gethostname(local_host, namelen);
```

```
/*
  Get local host's hostent struct to find local host aliases
  and for use by passive_socket()
*/
  if ((hp = gethostbyname(local_host)) == 0) {
    perror("ipc: gethostbyname()");
    exit(EGETHOSTBYNAME);
  }
/*
  Create a passive socket on which to listen
*/
  passive_socket(&s, &iLocalPort, hp);
  itoa(iLocalPort, cLocalPort);
/*
  System() fork()'s a child and exec()'s /bin/sh to execute
  the rsh command, firing up the remote server.  The local
  host's alias is used because the VAX gethostbyname(3N)
  only accepts the full host name. This way, VAXs and Suns
  (et al) can be used as servers.
*/
#ifdef IRIS
  sprintf(command, "rsh %s '%s %s %s' &",
    remote_host, service, hp->h_aliases[0], cLocalPort);
# else
  sprintf(command, "rsh %s '%s %s %s' &",
    remote_host, service, local_host, cLocalPort);
#endif

  if (system(command) == ERROR) {
    perror("ipc: system() (fork() or exec())");
    exit(ESYSTEM);
  }
/*
  Accept connection requests
*/
  if ((*sock = accept(s, 0, 0)) <= 0) {
    perror("ipc: accept()");
    exit(EACCEPT);
  }
}
```

```
passive_socket(sock, local_port, hp)
int *sock, *local_port;
struct hostent *hp;
{
  int length, i;
  struct sockaddr_in sin, system_sock;
/*
  Create a socket
*/
  if ((*sock = socket (AF_INET,SOCK_STREAM,0)) < 0) {
    perror("ipc: socket()");
    exit(ESOCKET);
  }
/*
  Initialize socket data structure
*/
  sin.sin_family = AF_INET;
  bcopy(hp->h_addr, (char *) &(sin.sin_addr.s_addr),
    hp->h_length);
/*
  Request system to assign a port
*/
  sin.sin_port = htons(0);
  for (i = 0; i < 8; i++)
    sin.sin_zero[i] = '\0';
/*
  Bind socket data structure to this socket
*/
  if (bind(*sock, &sin, sizeof(sin))) {
    perror("ipc: bind()");
    exit(EBIND);
  }
/*
  Get the port that the system assigned
*/
  length = sizeof(struct sockaddr_in);
  if (getsockname (*sock, &system_sock, &length)) {
    perror("ipc: getsockname()");
    exit(EGETSOCKNAME);
  }
/*
  Return socket port assigned by the system
*/
  *local_port = htons(system_sock.sin_port);
```

```
/*
  Prepare socket queue for connection requests
*/
  listen(*sock, MAXREQUESTS);
}


connect_socket(sock, host, charPort)
int *sock;
char *host, *charPort;
{
  int i;
  struct sockaddr_in sin;
  struct hostent *hp, *gethostbyname();
/*
  Create the socket
*/
  if ((*sock = socket (AF_INET,SOCK_STREAM,0)) < 0) {
    perror("ipc: socket()");
    exit(ESOCKET);
  }
/*
  Initialize socket data structure
*/
  if ((hp = gethostbyname(host)) == 0) {
    perror("ipc: gethostbyname()");
    exit(EGETHOSTBYNAME);
  }

  sin.sin_family = AF_INET;
  bcopy(hp->h_addr, (char *) &(sin.sin_addr.s_addr),
    hp->h_length);
  sin.sin_port = htons(atoi(charPort));
  for (i = 0; i < 8; i++)
    sin.sin_zero[i] = '\0';
/*
  Connect to remote host
*/
  if (connect(*sock, &sin, sizeof(sin)) < 0) {
    close(*sock);
    perror("ipc: connect()");
    exit(ECONNECT);
  }
}
```

```
data_present(sock, time_out)
int sock;
long time_out;
{
  fd_set fds;
  struct timeval timeout;
  short result;

  FD_ZERO(&fds);
  FD_SET(sock, &fds);

  timeout.tv_sec = time_out;
  timeout.tv_usec = 0;

  if ((result = select(FD_SETSIZE, &fds, NOFDS, NOFDS,
      &timeout)) == ERROR) {
    perror("ipc: select()");
    exit(ESELECT);
  }

  return(FD_ISSET(sock, &fds));
}


send_structure(sock, ptr, size)
int sock;
char *ptr;
int size;
{
  int sent, acc=0, RETRY=FALSE;
  char log_buf[132];
  static int count=0, retries=0;

  while (acc < size) {
    if ((sent = write(sock, (char *) (ptr+acc),
        size-acc)) < 0) {
      perror("ipc.c: send_structure()");
      exit(EWRITE);
    } else {
      acc += sent;
      if (acc < size) {
        RETRY = TRUE;
        retries++;
      }
    }
  }
```

```
#ifdef DEBUG
  count++;
  if (RETRY) {
    sprintf(log_buf, "count: %d, retries: %d\n", count,
      retries);
    log_message(log_buf, __FILE__, __LINE__);
  }
#endif
}


receive_structure(sock, ptr, size)
int sock;
char *ptr;
int size;
{
  int got=0, acc=0;

  while (got < size) {
    if ((got = read(sock, (char *) (ptr+acc), size-acc)) < 0) {
      perror("ipc.c: receive_structure()");
      exit(EREAD);
    } else {
      acc += got;
    }
  }
}


#ifndef IRIS
init_isr(sock)
int sock;
{
  int sigio_isr();
/*
  Cause sigio_isr() to be invoked upon receipt of SIGIO signal
*/
  signal(SIGIO, sigio_isr);
/*
  Set the process group receiving SIGIO/SIGURG signals
  to this process group. Note the minus sign.
*/
  if (fcntl(sock, F_SETOWN, -getpid()) < 0) {
    perror("fcntl F_SETOWN");
    exit(1);
  }
```

```
/*
  Allow receipt of asynchronous i/o signals
*/
  if (fcntl(sock, F_SETFL, FASYNC) < 0) {
    perror("fcntl F_SETFL, FASYNC");
    exit(1);
  }
}


sigio_isr()
{
/*
  Cause sigio_isr() to be invoked upon receipt of SIGIO signal
*/
  signal(SIGIO, sigio_isr);
/*
  Set flag for server process indicating presence of new client
  request
*/
  NewRequest = TRUE;
}

#endif


/*
  Close_sock() isolates the close()'s, should something more
  elaborate be required later.
*/

close_sock(sock)
int sock;
{
  close(sock);
}
```

```
itoa(n,s)
int n;
char *s;
{
  int i = 0, sign;

  if ((sign = n) < 0)
    n = -n;

  do
    s[i++] = n % 10 + '0';
  while ((n /= 10) > 0);

  if (sign < 0)
    s[i++] = '-';
  s[i] = '\0';
  reverse(s);
}


reverse(s)
char *s;
{
  int c, i, j;

  for (i = 0, j = strlen(s) - 1; i < j; i++, j--) {
    c = s[i];
    s[i] = s[j];
    s[j] = c;
  }
}
```

Appendix B

Libipc(3) Manual Pages

NAME

libipc(3) — Interprocess Communications Library

SYNOPSIS

```
start_server(remote_host, service, sock)
char *remote_host, *service;
int *sock;

passive_socket(sock, local_port, hp)
int *sock, *local_port;
struct hostent *hp;

connect_socket(sock, host, charPort)
int *sock;
char *host, *charPort;

data_present(sock, time_out)
int sock;
long time_out;

send_structure(sock, ptr, size)
int sock;
char *ptr;
int size;

receive_structure(sock, ptr, size)
int sock;
char *ptr;
int size;

init_isr(sock)
int sock;

close_sock(sock)
int sock;

int NewRequest;
```

DESCRIPTION

Libipc(3) is a set of functions designed to facilitate the development of distributed applications using the Interprocess Communications facilities of UNIX 4.3BSD. The user is isolated from the rather nasty IPC system calls, and needs not understand the use of such functions as *socket(2)*, *bind(2)*, *connect(2)*, *accept(2)*, *select(2)*, *listen(2)*, *getsockname(2)*, *gethostname(2)*, and *gethostbyname(3N)*.

Most users will not need to use the *passive_socket(3)* function — it is called by *start_server(3)* to establish a simple bidirectional communications stream (a stream socket) between two processes that aren't necessarily running on the same machine.

The most efficient use of this library is for users to declare the external integer `NewRequest`, and check this flag to see if new data has arrived on the socket. `NewRequest` is set to `TRUE` by an interrupt service routine initially enabled by calling *init_isr(3)*.

ERRORS

All system calls are checked for erroneous return values. In such cases, an error message is printed on the standard error file using *perror(3C)*, and an exit status corresponding to the error is returned with an *exit(2)*. The exit codes are defined in `.../local/include/islipc.h`.

AUTHORS

Russell Byrne, Amir Sela

FILES

(Present on each machine.)
`...local/include/islipc.h`
`...local/lib/libipc.a`

NOTE

The communication initialization sequence is as follows:

1. The client process uses *start_server(3)* to invoke the specified program on the specified machine. This call blocks until:
2. The server process uses *connect_socket(3)* to connect to the client.
3. Both sides return, with the pass-by-reference integer parameter set to a non-negative integer which is a descriptor for the stream socket connection. This socket descriptor my then be used freely with the *read(2)*, *write(2)*, *send(2)*, and *receive(2)* functions.
4. If asynchronouse file I/O is available on a machine[1], then the process on that machine should declare the global variable `NewRequest`, and call *init_isr(3)* before entering the simulation loop. When new data is

[1]Such as any Sun workstation; to check, see if the constant `FASYNC` is defined in `/usr/include/sys/fcntl.h`

available on the socket, this flag will be set to TRUE by an interrupt service routine.

EXAMPLES

The following code and makefile fragments should explain the details of how to use this library. In the example, data is written from the server to the client, the other direction is left out for brevity.

Here is a code fragment from the client (IRIS workstation graphics program):

```
/* "main.c" */

#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
/*
 argv[1] is the name of the machine to use for the server
 process.
*/
 int sock;
 char errmsg[80];
 define data_struct as desired

 if (argc != 2) {
  sprintf(errmsg, "\nUsage:  %s:  <remote_host>", argv[0]);
  perror(errmsg);
  exit(EUSAGE);
 }
/*
 Initialize the stream socket
*/
 start_server(argv[1], " byrne/chain/chain_server", &sock);
/*
 sock is now ready for reading and writing
*/
```

```
 while (TRUE) {
  if (user wants to quit) {
   notify server
   close_comm(sock);
   exit(0);
  }

  if (data_present(sock, (long) 0)) {
    receive_structure(sock, data_struct, sizeof data_struct);
  }

  draw next graphics image
  process graphical user interface inputs
 }
}
```

Here is a piece of the makefile used to compile the above code on a 2000 or 3000 series IRIS:

```
OBJ = main.o
LIB = -lipc -lbsd -ldbm -Zg
CFLAGS = -c -I/usr/include/bsd -I/usr/local/include

chainsim:  $(OBJ)
 cc -o chainsim $(OBJ) $(LIB)
```

Here is a piece of the makefile used to compile the above code on an IRIS 4D:

```
OBJ = main.o
LIB = -L/usr/local/lib -lipc -lbsd -lc_s -Zg
CFLAGS = -c -I/usr/local/include

chainsim:  $(OBJ)
 cc -o chainsim $(OBJ) $(LIB)
```

Here is a fragment from the server code (simulation program running on a Sun workstation):

```
/* "chain.c" */

#include <stdio.h>

int NewRequest=FALSE;

main(argc, argv)
int argc;
char *argv[];
{
 int sock;

/*
 Create socket and connect to client, using the port number
 given by the client.  argv[1] is the client machine name,
 and argv[2] is the port the client is using to listen for
 connection requests.  Both of these arguments are set by
 start_server(3).
*/
 connect_socket(&sock, argv[1], argv[2]);

 init_isr(sock); /* initialize interrupt service routine */
 do_data(sock); /* execute simulation loop */
}
do_data(sock)
int sock;
{
 while (TRUE) {
  if (NewRequest) {
   NewRequest = FALSE;
   if (data_present(sock, (long) 0)) {
    read data from client using sock
    if (client wants to quit) {
     close_sock(sock);
     exit(0);
    }
   } /* data_present() */

   do calculations, send result using sock
  }
 }
}
```

Finally, here is a fragment from the above program's makefile:

```
OBJ = chain.o
LIB = -L/usr/local/lib -lipc -ldbm
CFLAGS = -c -I/space/local/include

chain_server:  $(OBJ)
 cc -o chain_server $(OBJ) $(LIB)
```

BUGS

Currently, the only IPC model easily implemented is the single client/server pair. More elaborate schemes can be developed using *passive_socket(3)*, *connect_socket(3)*, and *system(3)*, or *fork(2)* and one of the *exec(3)* functions.

The error handling should be redone, with the *exit(2)* calls replaced by a chain of `return` statements, so exception handling can be done by the user.

Problem reports should be mailed to `byrne@ra.src.umd.edu`.

Appendix C

MACSYMA Symbolic Manipulations

```
load("util_root:[macsyma]tools.mac") ;

write_tex_file("defs.tex");
SHOW_MACSYMA_SOURCE_WITH_TEX_CODE : FALSE $
NUMER : TRUE $
KEEPFLOAT : TRUE $
RATPRINT : FALSE $

/* TeX macros */
qput(mcm, "{\\it h}", TEX_NAME) $
qput(qd, "{\\dot q}", TEX_NAME) $
qput(qdd, "{\\ddot q}", TEX_NAME) $
qput(epsilon, "\\varepsilon", TEX_NAME) $
qput(i, "I", TEX_FUNNAME) $

qput(displaylines, [matchfix, "\\displaylines{\\quad ",
  " \\quad\\cr}"], tex_op) $
qput(display_flush_left, [matchfix, "\\displaylines{\\quad ",
  " \\hfill\\cr}"], tex_op) $
n : 3 $
q : genvector(q, n) $
qd : genvector(qd, n) $
qdj : genvector(qdj, n-1) $
qdd : genvector(qdd, n) $
qddj : genvector(qddj, n-1) $
tau : genvector(tau, n) $
/*
Joint angle.  Probablly should change theta(i,j) to
theta[i,j].
*/
theta(i,j) := q[j, 1] - q[i, 1] $
theta_dot(i,j) := qd[j, 1] - qd[i, 1] $
```

```
/* 2D rotation.  Z axis implied. */
rot2(ang) := matrix([cos(ang), -sin(ang)],
                    [sin(ang), cos(ang)]) $
sq_norm2(vec) := vec[1][1]**2 + vec[2][1]**2 $
dot2(a, b) := a[1][1]*b[1][1] + a[2][1]*b[2][1] $
dot(a, b) := sum(a[i,1] * b[i,1], i, 1, n) $

grad(f, x) :=
  block (
    [temp, i] ,
    for i thru n do (
      temp[i] : diff(f, x[i, 1])
    ) ,
    genvector(temp, n)
  ) $

jacobian(funcs, vars) :=
  block (
   [jac_temp],
   for i : 1 thru n do (
     for j : 1 thru n do (
      jac_temp[i,j] : diff(funcs[i,1], vars[j,1])
     )
   ) ,
   return(genmatrix(jac_temp, n, n))
  ) $

/* Mass distribution coefficients */
a(i, k) :=
  if k = 1 then
   if i = n then
     0.0
   else
     -sum(epsilon[j], j, i+1, n)
  else
   if i = n then
     0.0
   else
     if 1 <= i and i <= k - 1 then
      1.0 - sum(epsilon[j], j, i+1, n)
     else
      if k <= i and i <= n-1 then
        -sum(epsilon[j], j, i+1, n)
      else
        error("a(): Arguments out of range") $
```

```
/* More mass distribution coefficients */
b(i, k) :=
  if k = 1 then
   if i = 1 then
     0.0
   else
     -epsilon[i]
  else
   if i = 1 then
     0.0
   else
     if i # k and 2 <= i and i <= n then
      -epsilon[i]
     else
      if k = i then
        1.0 - epsilon[k]
      else
        error("b(): Arguments out of range") $

/* Link lengths */
beta_tilde(i) :=
  if i = n then
   matrix([0.0], [0.0])
  else
   d[i] * matrix([1.0], [0.0]) $

/* CM locations */
alpha_tilde(i) :=
  if i = 1 then
   matrix([0.0], [0.0])
  else
   if i = n then
     d[i] * matrix([1.0], [0.0])
   else
     kappa[i] * d[i] * matrix([1.0], [0.0]) $

/* Kinematics descriptor */
delta_tilde(i, k) := a(i,k) * beta_tilde(i) + b(i,k) *
  alpha_tilde(i) $

/* Augmented inertia */
i_tilde(k) := i[k] + sum(m[j] * sq_norm2(delta_tilde(k,j)), j, 1, n) $
```

```
/*
Off-diagonal mass matrix elements (mcm is for Mass
Coefficient Matrix)
*/
lambda_tilde(j, l) := mcm[j,l] * cos(theta(j,l)) $

/*
 * mcf is for Mass Coefficient Function (portion of lambda_tilde
 * not dependent on system configuration)
 */
mcf(j, l) := sum(m[k] * dot2(delta_tilde(j,k),
  delta_tilde(l,k)), k, 1, n) $

/*
 * Generate the mass matrix as a function of the system
 * configuration (through lambda_tilde which depends on
 * cos(theta(j,l))) and constants independent of system
 * configuration.  Also generate a similar matrix which
 * depends only on joint values, rather than absolute body
 * positions.
 *       mass    = mcm
 *           i,i      i,i
 *
 *       mass    = mcm    cos(theta   ), i <> j
 *           i,j      i,j          i,j
 */
for i_ind : 1 thru n do (
  for j_ind : 1 thru n do (
    if i_ind = j_ind then (
     temp_mass[i_ind, i_ind] : mcm[i_ind, i_ind],
     temp_mass_joint[i_ind, i_ind] : 'mcm[i_ind, i_ind]
   ) else (
     if i_ind > j_ind then (
        temp_mass[i_ind, j_ind] : temp_mass[j_ind, i_ind],
        temp_mass_joint[i_ind, j_ind] : temp_mass_joint[j_ind,
          i_ind]
     ) else (
      temp_mass[i_ind, j_ind] : lambda_tilde(i_ind, j_ind),
      temp_mass_joint[i_ind, j_ind] : subst(qj[j_ind-1] +
        sum(qj[k],k,i_ind,j_ind-2), q[j_ind][1]-q[i_ind][1],
     temp_mass[i_ind, j_ind])
     )
   )
  )
) $
mass : genmatrix(temp_mass, n, n) $
```

```
tex("The mass matrix is:", "\\bf H" = mass, " ") $
mass_joint : genmatrix(temp_mass_joint, n, n) $

/*
 * Generate the time derivative of the mass matrix as a
 * function of the system configuration and constants
 * independent of system configuration:
 *      d
 *     -- mass    = 0
 *     dt     i,i
 *
 *     d
 *    -- mass    = -mcm    sin(theta   ) theta_dot    , i <> j
 *    dt     i,j       i,j          i,j           i,j
 */
for i_ind : 1 thru n do (
  for j_ind : 1 thru n do (
    if i_ind = j_ind then
      temp_mass_dot[i_ind, i_ind] : 0.0
    else (
      if i_ind > j_ind then
        temp_mass_dot[i_ind, j_ind] :
          temp_mass_dot[j_ind, i_ind]
      else (
        temp_mass_dot[i_ind, j_ind] :
          -mcm[i_ind, j_ind] * sin(theta(i_ind, j_ind)) *
            theta_dot(i_ind, j_ind),
        temp_mass_dot[i_ind, j_ind] :
          subst (
            qj[j_ind-1] + sum(qj[k],k,i_ind,j_ind-2),
            q[j_ind][1]-q[i_ind][1],
            temp_mass_dot[i_ind, j_ind]
          ),
        temp_mass_dot[i_ind, j_ind] :
          subst (
            qdj[j_ind-1][1] + sum(qdj[k][1],k,i_ind,j_ind-2),
            qd[j_ind][1]-qd[i_ind][1],
            temp_mass_dot[i_ind, j_ind]
          )
      )
    )
  )
) $
mass_dot : genmatrix(temp_mass_dot, n, n) $
tex("The mass matrix time derivative is:", "\\dot{\\bf H}"
   = mass_dot, " ") $
```

```
/* Define the Mass Coefficient Matrix */
for i_ind : 1 thru n do (
  for j_ind : 1 thru n do (
    if i_ind = j_ind then
     temp_mcm[i_ind, i_ind] : i_tilde(i_ind)
   else
     if i_ind > j_ind then
      temp_mcm[i_ind, j_ind] : temp_mcm[j_ind, i_ind]
     else
       temp_mcm[i_ind, j_ind] : mcf(i_ind, j_ind),
   print("Done with temp_mcm[",i_ind,"][",j_ind,"]")
  )
) $
mcm : genmatrix(temp_mcm, n, n) $
close_tex_file() ;
write_tex_file("mass_coeff.tex") ;
tex("The mass coefficients ($h_{i,j}$) are:") $
for i : 1 thru n do (
  for j : 1 thru i do (
    tex(displaylines('mcm[i,j] = mcm[i,j]))
  )
) $
close_tex_file() ;

/* Forward kinematics */
for i_ind : 1 thru n do (
  temp_cm[i_ind, 1] : sum(rot2(q[l, 1]) . delta_tilde(l, i_ind),
    l, 1, n)[1][1],
  temp_cm[i_ind, 2] : sum(rot2(q[l, 1]) . delta_tilde(l, i_ind),
    l, 1, n)[2][1]
) $
cm : genmatrix(temp_cm, n, 2) $
write_tex_file("fwd_kin.tex") ;
tex(" ", "The forward kinematics of the chain are:") $
for i : 1 thru n do (
  tex(display_flush_left(x[i] = cm[i,1])) ,
  tex(display_flush_left(y[i] = cm[i,2]))
) $
close_tex_file() ;
```

```
/* Dynamics begin here */

/* Stiffness matrix */
for i_ind : 1 thru n do (
  for j_ind : 1 thru n do (
    temp_stiff[i_ind, j_ind] :
      if i_ind = j_ind then
        if i_ind = 1 then
          s[1]
        else
          if i_ind = n then
            s[n-1]
          else
            s[i_ind] + s[i_ind-1]
      else
        if abs(i_ind - j_ind) = 1 then
          -s[(i_ind+j_ind+1)/2 - 1]
          /* e.g, indices 3,4 and 4,3 give -s[3] */
        else
          0.0
  )
) $
stiff : genmatrix(temp_stiff, n, n) $
write_tex_file("dynamics.tex") ;
tex(" ", "The stiffness matrix is:", "\\bf S" = stiff) $

/* calculate the term due to dL/dqd: */
for i  thru n do (
  for j thru n do (
    dldqd_temp[i, j] : dot(grad(mass[i, j], q), qd)
  )
) $
dldqd_term : genmatrix(dldqd_temp, n, n) $

/* Joint to body conversion matrix */
for i_ind : 1 thru n do (
  for j_ind : 1 thru n-1 do (
    temp_m[i_ind, j_ind] :
      if i_ind > j_ind then
        1.0
      else
         0.0
  )
) $
m : genmatrix(temp_m, n, n-1) $
```

```
tex("The following matrix is used in the joint to body ",
  "relationship: \\bf M" = m, " ") $

for i thru n do (
  temp_e[i] : 1.0
) $
e : genvector(temp_e, n) $

e_dot_J_M_thetad : dot(e, mass_joint . m . qdj) $
e_dot_J_e : dot(e, mass_joint . e) $

omega[1] : - e_dot_J_M_thetad / e_dot_J_e $
for i : 2 thru n do (
  omega[i] : 'omega[1] + sum(qdj[k][1], k, 1, i-1)
) $
omega : genvector(omega, n) $

e_dot_dJdt_e : dot(e, mass_dot . e) $
e_dot_dJdt_M_thetad : dot(e, mass_dot . m . qdj) $
e_dot_J_M_thetadd : dot(e, mass_joint . m . qddj) $

omega_dot[1] : 'e_dot_J_M_thetad * 'e_dot_dJdt_e /
  ('e_dot_J_e * 'e_dot_J_e) -
    ('e_dot_dJdt_M_thetad + 'e_dot_J_M_thetadd) / 'e_dot_J_e $
for i : 2 thru n do (
  omega_dot[i] : 'omega_dot[1] + sum(qddj[k][1], k, 1, i-1)
) $
omega_dot : genvector(omega_dot, n) $

/* Dynamic equations. */
eq_lhs : mass . qdd + stiff . q + dldqd_term . qd -
    0.5 * grad(transpose(qd) . mass . qd, q) $
eq_lhs : fullratsimp(eq_lhs) $

/* eq_rhs : j2b_tau . tau $ */
eq_rhs : tau $
/* eq_rhs : fullratsimp(eq_rhs) $ */
close_tex_file() ;

write_tex_file("scalar_dyn.tex") ;
tex("Substituting for the matrices defined above, we have the",
  "following scalar equations:") $
for i thru n do (
  tex(displaylines(factorsum(expand(eq_lhs[i,1] = eq_rhs[i]))))
) $
close_tex_file() ;
```

```
/*
Save a copy of the dynamic equations to use for inverse dynamics
*/
eq_dyn_lhs : copymatrix(eq_lhs) $

factorout_arg : [q[1][1]] $
for i : 2 thru n do (
  factorout_arg : cons(q[i][1], factorout_arg)
) $

for i thru n do (
  eq_dyn_lhs[i][1] : apply('factorout, cons(eq_dyn_lhs[i][1],
    factorout_arg))
) $

for i : 1 thru n-1 do (
  for l : i+1 thru n do (
    for m thru n do (
      eq_dyn_lhs[m][1] : subst(qj[l-1] + sum(qj[k],k,i,l-2),
        q[l][1]-q[i][1], eq_dyn_lhs[m][1]),
      eq_dyn_lhs[m][1] : subst(-qj[l-1] - sum(qj[k],k,i,l-2),
        q[i][1]-q[l][1], eq_dyn_lhs[m][1])
    )
  )
)$

/*
 * Here are the Newmark substitutions.
 * q[1], qd[1], qdd[1], ..., q[N], qd[N], qdd[N] define the
 * trajectory at time n+1.  This is the unknown.  All else
 * is known. Constants an[i] and bn[i] are given by:
 *
 *                  .      h ..
 *        an   =   q    + - q
 *          i       i,n   2  i,n
 *
 *                                2
 *                           .   h  ..
 *        bn   = q    + h q    + -- q
 *          i     i,n      i,n   4   i,n
 */
write_tex_file("newmark.tex") ;
```

```
for i_ind : 1 thru n do (
  for j_ind : 1 thru n do (
    def_qd[j_ind] : qd[j_ind,1] = h/2.0 * qdd[j_ind,1] +
      an[j_ind] ,
    def_q[j_ind] : q[j_ind,1] = h**2.0/4.0 * qdd[j_ind,1] +
      bn[j_ind] ,
    eq_lhs[i_ind] : ratsubst(rhs(def_qd[j_ind]),
      lhs(def_qd[j_ind]), eq_lhs[i_ind]) ,
    eq_lhs[i_ind] : ratsubst(rhs(def_q[j_ind]),
      lhs(def_q[j_ind]), eq_lhs[i_ind]) ,

    eq_zero[i_ind] : factorsum(expand(eq_lhs[i_ind,1] -
      eq_rhs[i_ind])) = 0 ,
    print("Done with eq_zero[",i_ind,"][",j_ind,"]")
  )
) $

tex("After doing the Newmark substitutions, the dynamics ",
  "are:") $
for i thru n do (
  tex(displaylines(factorsum(expand(eq_lhs[i,1] = eq_rhs[i]))))
) $

/*
Pick out the LHS of the eq_zero equations, which all
have a RHS = 0:
*/
for i : 1 thru n do (
  temp_eq_zero_lhs[i] : lhs(eq_zero[i])[1]
) $
eq_zero_lhs : genvector(temp_eq_zero_lhs, n) $

jacob : jacobian(eq_zero_lhs, qdd) $

GENTRANLANG : C $
CLINELEN : 65 $
GENTRANOPT : TRUE $
TEMPVARTYPE : double $
OPTIMPREFIX : O $
GENFLOAT : TRUE $
NUMER : FALSE $

gentranin("jac.mac_c", ["jac.c"]) ;

close_tex_file();
```

Appendix D

Translation to C language

D.1 MACSYMA/C Template File

```
/* opt_jac3.c */

#ifndef lint
 static char SccsIdOpt_Jac<<gentran(eval(n))$>>[] =
   "%W% %H% %T%";
#endif

#include <<gentran(literal("<math.h>"))$>>

#define power(x,y) ((y) == 2 ? \
  ((double) (x)) * ((double) (x)) : \
  pow ((double) (x), (double) (y)))
#define NUM_BODIES <<gentran(eval(n)) $>>
<<NUMER : TRUE $>>
compute_jac(mcm, h, an, bn, qdd, s, j)
double mcm[][NUM_BODIES+1], h, an[], bn[], qdd[], s[],
j[][NUM_BODIES+1];
{
<<gentran (j : eval(jacob)) $>>
}
```

```
compute_g(mcm, h, an, bn, qdd, tau, s, g)
double mcm[][NUM_BODIES+1], h, an[], bn[], qdd[], tau[],
  s[], g[];
{
<<
/*
  For this function, we call OPTIMIZE() ourselves, rather
  than letting GENTRAN() do it by setting GENTRANOPT to
  TRUE.  This is because the desired output is a single
  dimensional array, which was treated as an nx1
  matrix for MACSYMA's matrix multiply.  Thus, GENTRAN()
  outputs an nx1 array as the C language output.
  We override this as follows.
*/
eq_zero_lhs_opt : optimize(eq_zero_lhs) $
opt_length : length(eq_zero_lhs_opt) $

for i thru opt_length - 2 do (
  gentran (
    literal (
      "double O", eval(i), ";", cr
    )
  )
) $

gentran(literal(cr)) $

for i thru opt_length - 2 do (
  gentran (
    literal (
      "O", eval(i), " = ",
      eval (
        /* Get RHS of ':' assignment */
       part (
          /* Get current temp var. assignment */
          part (eq_zero_lhs_opt, i+1),
          2
        )
      ), ";", cr
    )
  )
) $

gentran(literal(cr)) $
```

```
for i thru n do (
  gentran (
    literal (
      "g[", eval(i), "] = ", eval(part(eq_zero_lhs_opt,
      opt_length)[i,1]), ";", cr
    )
  )
) $
>>
}
```

D.2 Optimized C Language File

The output shown below is the C language translation of the output of the `OPTIMIZE()` function of MACSYMA. This function uses a recursive algorithm to search expressions for common subexpressions. When found, they are replaced by a temporary variable. The output is a list of temporary variable assignments, followed by the optimized expression using the temporary variables.

The function `compute_jac()` shown below is the output of `GENTRAN()`, with the `GENTRANOPT` flag set to `TRUE`. This causes the `OPTIMIZE()` function to be called automatically, and the temporary variables are declared as `double` because the `TEMPVARTYPE` variable was set to `DOUBLE`.

For the function `compute_g()` we wanted more control over the translation, since the default translation for an N element vector is an $N \times 2$ array. So, we called `OPTIMIZE()` directly, and used the `PART()` function to isolate the elements of the block returned by `OPTIMIZE()` (see the previous section).

```
/* "opt_jac3.c" */

#ifndef lint
 static char SccsId0pt_Jac3[] =
   "%W% %H% %T%";
#endif
#include <math.h>

#define power(x,y) ((y) == 2 ? \
  ((double) (x)) * ((double) (x)) : \
  pow ((double) (x), (double) (y)))
#define NUM_BODIES 3

compute_jac(mcm, h, an, bn, qdd, s, j)
double mcm[][NUM_BODIES+1], h, an[], bn[], qdd[], s[],
j[][NUM_BODIES+1];
{
double t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,t13,t14,t15
 ,t16,t17,t18,t19,t20,t21,t22,t23,t24,t25;
{
    t0 = power(h,2);
    t1 = -(s[1]*t0);
    t2 = power(an[2],2);
    t3 = -(4.0*bn[1]);
    t4 = -qdd[1];
    t5 = 0.25*((qdd[2]+t4)*t0+4.0*bn[2]+t3);
    t6 = cos(t5);
    t7 = power(h,3);
    t8 = power(qdd[2],2);
    t9 = power(h,4);
    t10 = sin(t5);
    t11 = power(an[3],2);
    t12 = 4.0*bn[3];
    t13 = 0.25*((qdd[3]+t4)*t0+t12+t3);
    t14 = cos(t13);
    t15 = power(qdd[3],2);
    t16 = sin(t13);
    t17 = s[1]*t0;
    t18 = -(4.0*mcm[1][2]*t6);
    t19 = power(an[1],2);
    t20 = power(qdd[1],2);
    t21 = -(s[2]*t0);
    t22 = 0.25*((qdd[3]-qdd[2])*t0+t12-(4.0*bn[2]));
    t23 = cos(t22);
    t24 = sin(t22);
    t25 = s[2]*t0;
```

```
    {
        j[1][1] = -(0.25*(-(mcm[1][3]*qdd[3]*t0*t16)-(0.25*
         mcm[1][3]*t15*t9*t14)-(mcm[1][3]*an[3]*qdd[3]*t7*
         t14)-(mcm[1][3]*t11*t0*t14)-(mcm[1][2]*qdd[2]*t0*
         t10)-(0.25*mcm[1][2]*t8*t9*t6)-(mcm[1][2]*an[2]*qdd
         [2]*t7*t6)-(mcm[1][2]*t2*t0*t6)+t1-(4.0*mcm[1][1]))
         );
        j[1][2] = -(0.25*(3.0*mcm[1][2]*qdd[2]*t0*t10+4.0*
         mcm[1][2]*an[2]*h*t10+0.25*mcm[1][2]*t8*t9*t6+mcm[1
         ][2]*an[2]*qdd[2]*t7*t6+mcm[1][2]*t2*t0*t6+t18+t17)
         );
        j[1][3] = -(0.25*(3.0*mcm[1][3]*qdd[3]*t0*t16+4.0*
         mcm[1][3]*an[3]*h*t16+0.25*mcm[1][3]*t15*t9*t14+mcm
         [1][3]*an[3]*qdd[3]*t7*t14+mcm[1][3]*t11*t0*t14-(
         4.0*mcm[1][3]*t14)));
        j[2][1] = -(0.25*(-(3.0*qdd[1]*mcm[1][2]*t0*t10)-(
         4.0*an[1]*mcm[1][2]*h*t10)+0.25*t20*mcm[1][2]*t9*t6
         +an[1]*qdd[1]*mcm[1][2]*t7*t6+t19*mcm[1][2]*t0*t6+
         t18+t17));
        j[2][2] = -(0.25*(-(mcm[2][3]*qdd[3]*t0*t24)-(0.25*
         mcm[2][3]*t15*t9*t23)-(mcm[2][3]*an[3]*qdd[3]*t7*
         t23)-(mcm[2][3]*t11*t0*t23)+qdd[1]*mcm[1][2]*t0*t10
         -(0.25*t20*mcm[1][2]*t9*t6)-(an[1]*qdd[1]*mcm[1][2]
         *t7*t6)-(t19*mcm[1][2]*t0*t6)+t21+t1-(4.0*mcm[2][2]
         )));
        j[2][3] = -(0.25*(3.0*mcm[2][3]*qdd[3]*t0*t24+4.0*
         mcm[2][3]*an[3]*h*t24+0.25*mcm[2][3]*t15*t9*t23+mcm
         [2][3]*an[3]*qdd[3]*t7*t23+mcm[2][3]*t11*t0*t23-(
         4.0*mcm[2][3]*t23)+t25));
        j[3][1] = 0.25*(3.0*qdd[1]*mcm[1][3]*t0*t16+4.0*an[1
         ]*mcm[1][3]*h*t16-(0.25*t20*mcm[1][3]*t9*t14)-(an[1
         ]*qdd[1]*mcm[1][3]*t7*t14)-(t19*mcm[1][3]*t0*t14)+
         4.0*mcm[1][3]*t14);
        j[3][2] = 0.25*(3.0*qdd[2]*mcm[2][3]*t0*t24+4.0*an[2
         ]*mcm[2][3]*h*t24-(0.25*t8*mcm[2][3]*t9*t23)-(an[2]
         *qdd[2]*mcm[2][3]*t7*t23)-(t2*mcm[2][3]*t0*t23)+4.0
         *mcm[2][3]*t23+t21);
        j[3][3] = 0.25*(-(qdd[2]*mcm[2][3]*t0*t24)+0.25*t8*
         mcm[2][3]*t9*t23+an[2]*qdd[2]*mcm[2][3]*t7*t23+t2*
         mcm[2][3]*t0*t23-(qdd[1]*mcm[1][3]*t0*t16)+0.25*t20
         *mcm[1][3]*t9*t14+an[1]*qdd[1]*mcm[1][3]*t7*t14+t19
         *mcm[1][3]*t0*t14+t25+4.0*mcm[3][3]);
    }
}

}
```

```
compute_g(mcm, h, an, bn, qdd, tau, s, g)
double mcm[][NUM_BODIES+1], h, an[], bn[], qdd[], tau[],
  s[], g[];
{
double o1;
double o2;
double o3;
double o4;
double o5;
double o6;
double o7;
double o8;
double o9;
double o10;
double o11;
double o12;
double o13;
double o14;
double o15;
double o16;
double o17;
double o18;
double o19;
double o20;
double o21;
double o22;
double o23;

o1 = power(h,2);
o2 = -(4.0*bn[1]);
o3 = -qdd[1];
o4 = 0.25*((qdd[2]+o3)*o1+4.0*bn[2]+o2);
o5 = cos(o4);
o6 = power(an[2],2);
o7 = sin(o4);
o8 = power(qdd[2],2);
o9 = 4.0*bn[3];
o10 = 0.25*((qdd[3]+o3)*o1+o9+o2);
o11 = cos(o10);
o12 = power(an[3],2);
o13 = sin(o10);
o14 = power(qdd[3],2);
o15 = -(4.0*bn[2]*s[2]);
o16 = 4.0*s[2]*bn[3];
```

```
o17 = -(qdd[2]*s[2]*o1);
o18 = s[2]*qdd[3]*o1;
o19 = power(an[1],2);
o20 = power(qdd[1],2);
o21 = 0.25*((qdd[3]-qdd[2])*o1+o9-(4.0*bn[2]));
o22 = cos(o21);
o23 = sin(o21);

g[1] = -(0.25*(mcm[1][3]*o14*o1*o13+4.0*mcm[1][3]*an[3]*qdd[
 3]*h*o13+4.0*mcm[1][3]*o12*o13-(4.0*mcm[1][3]*qdd[3]*o11)+
 mcm[1][2]*o8*o1*o7+4.0*mcm[1][2]*an[2]*qdd[2]*h*o7+4.0*mcm[
 1][2]*o6*o7-(4.0*mcm[1][2]*qdd[2]*o5)+s[1]*qdd[2]*o1-(qdd[1
 ]*s[1]*o1)+4.0*s[1]*bn[2]-(4.0*qdd[1]*mcm[1][1])+4.0*tau[1]
 -(4.0*bn[1]*s[1])));
g[2] = -(0.25*(mcm[2][3]*o14*o1*o23+4.0*mcm[2][3]*an[3]*qdd[
 3]*h*o23+4.0*mcm[2][3]*o12*o23-(4.0*mcm[2][3]*qdd[3]*o22)-(
 o20*mcm[1][2]*o1*o7)-(4.0*an[1]*qdd[1]*mcm[1][2]*h*o7)-(4.0
 *o19*mcm[1][2]*o7)-(4.0*qdd[1]*mcm[1][2]*o5)+o18+o17-(s[1]*
 qdd[2]*o1)+qdd[1]*s[1]*o1+o16-(4.0*qdd[2]*mcm[2][2])+4.0*
 tau[2]+o15-(4.0*s[1]*bn[2])+4.0*bn[1]*s[1]));
g[3] = 0.25*(o8*mcm[2][3]*o1*o23+4.0*an[2]*qdd[2]*mcm[2][3]*
 h*o23+4.0*o6*mcm[2][3]*o23+4.0*qdd[2]*mcm[2][3]*o22+o20*mcm
 [1][3]*o1*o13+4.0*an[1]*qdd[1]*mcm[1][3]*h*o13+4.0*o19*mcm[
 1][3]*o13+4.0*qdd[1]*mcm[1][3]*o11+o18+o17+4.0*qdd[3]*mcm[3
 ][3]-(4.0*tau[3])+o16+o15);

}
```

Appendix E

Implementation Issues

When implementing a distributed system that passes actual data back and forth across the network (rather than messages), the type definitions for the structures used are very important.

It is crucial that both sides of the communications interface agree on the definition of the structure used. In our case, the structure depends on a defined constant, NUM_BODIES the number of bodies in the simulation. If the definition of this constant should change, certain modules of code on both machines must be recompiled.

It is easy to set up compilation dependencies for code resident on a single machine using *make(1)*. However, it takes some doing to specify dependencies involving files on different filesystems (i.e., not NFSed together, but on the same TCP/IP network).

Shown below is a `Makefile` fragment that solves this problem. Assuming *rcp(1C)*, and *rsh(1C)* exist on the machine, one can then specify compilation actions to take place on another machine under certain circumstances, as follows:

```
RemoteChainIPC.h: ChainIPC.h
 rcp ChainIPC.h orbit:/space/byrne/chain
 rsh orbit 'cd chain; make'
 touch RemoteChainIPC.h
```

The file `RemoteChainIPC.h` is used to remember when the compilation on `orbit` last took place. The first line indicates that files on `orbit` depend on `ChainIPC.h`. If `ChainIPC.h` has changed since the files on `orbit` were last compiled, then `ChainIPC.h` is copied over the network to the proper directory on `orbit`, and the *make(1)* utility is invoked to build the code in that directory, using its `Makefile`. Finally, the time of this action is saved in the modification time of `RemoteChainIPC.h`.

Bibliography

[1] Albus, J.S., McCain, H.G., and Lumia, R., "NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)," U.S. Department of Commerce, National Institute of Standards and Technology, NIST Technical Note 1235, 1989 Edition.

[2] Comer, D., *Internetworking with TCP/IP*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[3] Craig, J.J., *Introduction to Robotics Mechanics & Control*, Addison-Wesley, Reading, MA, 1986.

[4] Harrison, B.T., "Harvesting Processor Power with NCS," DEC Professional, July, 1990, pp. 102–108.

[5] Krishnaprasad, P.S., "Eulerian Many-Body Problems," Contemporary Mathematics, Volume 97, 1989, pp. 187–208.

[6] Leffler, S., Fabry, R., and Joy, W., "A 4.3BSD Interprocess Communications Primer," Computer Systems Research Group, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1985.

[7] Newmark, N. M., "A Method of Computation for Structural Dynamics," Proceedings of the ASCE 85 EM3 (1959) 67–94.

[8] Press, W. H., Flannery, B. P, et al, *Numerical Recipes in C*, Cambridge University Press, Cambridge, 1988.

[9] Sela, A., "Client/Server Model for Distributed Computing: An Implementation," Electrical Engineering Department and Systems Research Center technical memorandum, University of Maryland, College Park, 1989.

[10] Sinha, V., "The Mobile Remote Manipulator System Simulator," Technical Report SRC TR 87-24, Systems Research Center, University of Maryland, December, 1986.

[11] Space Station Flight Telerobotic Servicer (FTS) DTF-1 Contract End Item Specification (CEIS) Preliminary (February 12, 1990), SEP-12-01.

[12] Sreenath, N., "Modeling and Control of Multibody Systems," Intelligent Servosystems Laboratory, University of Maryland, Systems Research Center Technical Report, SRC TR 87-67, December, 1987.

[13] Symon, K.R., *Mechanics*, Third Ed., Addison-Wesley, Menlo Park, CA, 1971.