1998

Interactive synthetic environments with force feedback

James Christopher Edwards
Iowa State University

Interactive synthetic environments with force feedback

by

James Christopher Edwards

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Mechanical Engineering

Major Professor: Greg R. Luecke

Iowa State University

Ames, Iowa

1998

Graduate College
Iowa State University

This is to certify that the Doctoral dissertation of

James Christopher Edwards

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Major Professor

Signature was redacted for privacy.

For the Major Program

Signature was redacted for privacy.

For the Graduate College

To my loving wife, Trisha

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to begin by thanking Greg Luecke for his continued support of this work. As I think back, it is doubtful that I would have pursued an advanced degree if I had not met Dr. Luecke. I would also like to express my appreciation to the members of my program of study committee, Judy Vance, Jim Oliver, Alison Flatau and Julie Dickerson. I would especially like to thank Dr. Oliver and Dr. Flatau for serving double duty on both my doctor of philosophy and master of science committees. In addition, I would like to thank the staff of the Department of Mechanical Engineering and the Iowa Center for Emerging Manufacturing Technology, especially Rosalie Enfield and Jill Shannon, for answering my numerous questions and handling the paper work associated with my research and studies.

Role of my parents, Rob and Susan, and my in-laws, Pat and Colette, have been extremely important. They all have supported my studies both emotionally and financially. Finally I would have to thank by wife, Trisha, for supporting me through out my education. I hope that I have been as supportive of her education as she has mine.

ABSTRACT

The evolution of the visual display technology used in synthetic environments is fueling the development of numerous applications. The results of these initial expeditions into virtual worlds have been promising. However, these initial investigations have also highlighted the need for force feedback in synthetic environments to make the virtual experience more immersive and easier for the traveler to interact with the objects that populate the synthetic environment. In addition the inclusion of force feedback in a synthetic environment will provide another input channel that can provide information to the traveler beyond the typical visual and audio input modes. Research in the area of force feedback for synthetic environments thus far has focused on the design and construction of specialized interface devices. These new haptic devices can be used to provide force interaction, however because these devices are unique prototypes it is difficult if not impossible to reproduce and extend results obtained at different facilities. This work proposes a new approach to force interaction in synthetic environments, virtual manipulators. The virtual manipulator control concept can be applied to any available six degree of freedom robot manipulator. Therefore experimental results obtained using the virtual manipulator control law can be reproduced at any research facility with a six degree of freedom robot. This work will develop the virtual manipulator control approach as well as investigate the stability characteristics of the control law operating on a general six degree of freedom robot. Experimental results will be presented for various virtual manipulators including the time varying extension of the virtual manipulator concept. In addition to the virtual manipulator concept this work will also develop a physically-based modeling technique that can be used to assimilate a force feedback device into a synthetic environment. This modeling approach uses finite element analysis techniques but uses the NURBS basis functions instead of the typical interpolation basis functions. As a result the dynamics of the model can be represented using the same characteristic parameters as the geometric model. Results of this modeling approach will be presented for one and two dimensional dynamic models.

CHAPTER 1. INTRODUCTION

The rapid development of synthetic environments is based on the idea of providing high quality, realistic images to the traveler in the environment. However, the main motivation is finding new ways of interacting with information. Visualization is at the root of synthetic environments and is an important application but if all that is needed to increase the utility of synthetic environments is a better quality image then there is little need for the research being done in this area. Improvements in displayed image quality will naturally follow advances in state of the art graphics hardware.

The current research is extending the boundaries of what can be done in a synthetic environment. This is simply a question of how the traveler in a synthetic environment can interact with the computer data that describes the world. The data can be viewed, the visualization aspect of the synthetic environment. The data can also be manipulated, moved, scaled and deformed. However, these operations can be performed and have been performed traditionally using a keyboard and mouse. Once the data has been modified it can again be put into the synthetic environment for a visual inspection. The modification process can be streamlined if the traveler is able to reach with her hand modify the data directly without having to transition between the synthetic environment and a traditional computer workstation.

The previous illustration clearly shows the direction of research in synthetic environments. Developing the interactive capabilities of synthetic environments is what will unlock the potential of the technology, simply having a faster computer will not. However, interacting with an synthetic environment is not a simply task. A traveler cannot reach out and grab data when all that is there is a projection, no matter how convincingly the mind of the traveler has been fooled.

Interaction devices such as a wand or glove can be used to communicate with the computer controlling the synthetic environment. However, this requires some mechanism for manipulation, such as touch the object and push a button and then move the object. Interaction in a synthetic environment is not the same as interacting with real object but these contrived mechanism for interaction may be sufficient or even preferred in some situations.

Another way to view interaction with a synthetic environment is what modalities are available to supply information to the traveler. When a wand or glove is used as the interaction device the input modalities are the same as a visualization synthetic environment. So a wand or glove interaction device has extended the interaction abilities of the synthetic environment but it has not extended the ability of the environment to supply the traveler with information. This is because these interaction devices are "passive", they can only observe what the traveler does and allow the synthetic environment to change in response. However, by using a haptic interaction device the traveler can interact with the synthetic environment in the same way as with a wand or glove but the haptic device also represents a new source of information that is available to the traveler, force information.

This force information can be used in a tradition setting such as to display the surface characteristics of an object or display to the traveler the weight or inertia of an object that is being carried. But forces can be used to represent other types of information as well, a force applied to the traveler need not represent a "real" force. This is the same idea as using a color map to represent the stress results obtained from a finite element analysis. Consider for example allowing a traveler design a part prototype in a virtual design studio. A quantity of design material is presented to the traveler. The design material could behave like clay that is used in the automobile industry. However, the real material used may not behave exactly the way the designers would want, in the synthetic environment the dynamic characteristics of the material can be customized.

One option is to allow the dynamic characteristics of the material should change during the design process. Consider that the job of the traveler in the synthetic environment design studio is to lay out a stiffing web. The traveler lays out the modeling material in the general shape and then sculpts it into the final form. During the design process a finite element analysis of the part could be running simultaneously, the results of this analysis could be used to modify the material properties of the design material. If the stress at some point in the web is high the modeling material could become stiff and viscous so that the designer can not further reduce the thickness of the web at that point.

The idea of modifying the properties of a design material could be used in other situations. When a large number of designers are working simultaneously on a complex product. They are generally given a working envelope in which their part or component must fit. A designer must not violate this envelope or risk having their component interfere with another part. As the designer shapes the part the modeling material could respond when the part approaches the envelope boundary. Varying modeling material properties could also be used to enforce continuity requirements on the surface of a part. Stiffing the modeling material at a point could be used to prevent a designer from introducing a curvature discontinuity into an automobile panel that would look "bad" under the show room lights.

Application of force feedback in the design process is not limited to prototype design. Design for assembly and disassembly is a major concern in the production of large complex machines. Using a haptic device coupled with common robot impedance fields would allow designers to investigate the installation or removal of a component part without construction of a physical prototype.

Although the potential of force feedback technology in synthetic environments is great, the development and assimilation of this technology has been slow. One reason for the slow development of haptic feedback technology is the accuracy and robustness of the human force sensors. Although the human visual system can be "fooled" with stereo images projected thirty times a second, the requirements for believable haptic interaction are more difficult to achieve. There is little doubt that the subject of haptic interaction in synthetic environments will be an open problem for some time to come.

The lack of usage of the haptic devices that do currently exist in synthetic environments is starting to be addressed. There has been a shift in the focus of many research groups, which in the past have concentrated on the design and construction of haptic interaction devices. These groups are now focusing on how to incorporate the current haptic devices into more complicated interactive dynamic simulations for use in synthetic environments. However, the integration of haptics into synthetic environments still presents some problems.

The first major problem resides in the different system requirements for a visual display system and a force display system. For a visual display system to work properly the stereo images must be updated at least thirty times per second, which is a least one order of magnitude slower than the update rate required for a haptic device. In addition the accuracy of the update rate for a visual display system is not critical, slight fluctuations in the update rate will most likely not be detrimental or even noticed by the traveler in the synthetic environment. However, variations in the update rate of a force display can lead to unstable system response that will at best degrade the experience of the traveler in the synthetic environment and at worst pose a physical risk to the traveler. Finally due to the relatively slow update rates of the visual system it is possible to communicate with the external hardware, such as position trackers, using conventional serial communication protocols. This is not the case for most force display systems due to the faster update rates and the amount of information that must be transferred for proper control of the hardware.

For these reasons the host computer selected for visual display systems and force display systems are different. Visual display systems typically run on unix platform computers with specialized graphics pipelines designed to increase display quality and speed. Where as force display systems are typically run on DOS based personal computers or on specialized control computers using a true real time operating system. As a result a synthetic environment containing both visual and force display will usually have two or more host computers which must communicate effectively for proper system performance.

Recent advances and changes in technology are reducing the magnitude of this problem. The demand for commercially available haptic interaction devices has required the development of interface circuitry and software to control haptic devices from traditional graphics workstations. Although this interface hardware and software is currently proprietary, it still leads to the conclusion that similar equipment could be developed to interface with a generic haptic device. In addition recent design choices made by the manufacturers of personal computers and graphics workstations are moving these two classes of host computers more towards a similar structure. Therefore as developers and users of haptic feedback devices adapt to these changes the process of including haptic devices in synthetic environments should be simplified. Finally, the wide availability of internet transfer protocols , such as UDP and TCP sockets, has made the process of communicating between dissimilar host computers substantially easier.

The second major problem associated with including a haptic device into a synthetic environment is due to the recent changes in the way three dimensional images are presented to the traveler. Most of the prototype and commercially available haptic feedback devices were designed and developed prior to the advent of projection style synthetic environments. As a result the design of these haptic devices are not compatible with this three dimensional graphics presentation technique. The presence of the device in the synthetic environment will occlude the projected images and have a detrimental effect on the quality of the experience of the traveler.

The third and final major problem associated with including a haptic device into a synthetic environment is caused by the unique nature of most haptic interfaces. There has, of course, been an increase in the number of commercially available force feedback devices but most devices used are one of a kind prototype devices. The one of a kind configuration of haptic devices prevents external verification and extension of research results. In addition the cost in money and time to design and construct a haptic device is prohibitively high for most research facilities. This coupled with the idiosyncratic nature of prototype hardware is sufficient to prevent the use of haptic devices in synthetic environments. Although commercially available feedback devices have reduced in some sense the isolation faced by haptics researches, proprietary programming libraries continue to stifle open communication between researchers in the haptic feedback arena.

This work addresses the problem developing a haptic interface that is compatible with all synthetic environment implementations, that is also commonly available to the haptic research community. It proposes using a generic six degree of freedom robot as a haptic interaction device for a synthetic environment, as well as a new control law which will allow the robot to behave like some virtual mechanism or manipulator. The proposed virtual manipulator control approach can make haptic interaction in synthetic environments more available to the research community. Although custom haptic devices are not prevalent, six degree of freedom robots are quite common. In addition interface hardware and software for joint level control of robots is commonly available for today's powerful personal computers. The virtual manipulator control approach is modular, can be easily changed, allowing any one of various six degree of freedom robots to represent the virtual manipulator.

This will allow research based on virtual manipulators to be verified and extend at different research facilities. Because the robot and interface equipment is readily available the cost in both money and time associated with acquiring and maintaining a haptic display system can be reduced. There are currently no six degree of freedom robots designed specifically as haptic devices, although some have been proposed. However, if such a device is ever manufactured the virtual manipulator control law would run on it as well.

The drawback to using the six degree of freedom robots currently available as a haptic device is that their general dynamic characteristics do not match those of an ideal haptic interface. The increased computational power of the personal computer offers a solution, as it is now possible to run complex, real time control laws that compensate for the physical short comings of a given robot. This will allow a less than ideal robot to behave more like an ideal haptic device.

It is also possible to use a six degree of freedom robot as a haptic display in a projection style synthetic environment. The robot can be equipped with a handle and positioned so that it is behind the traveler in the projection environment. Using the position sensors normally associated with a robot, the position of the handle in the synthetic environment can be calculated and the image of a virtual tool can be grown from this calculated position. This will allow the traveler to see the images of the scene and the tool without having the interface occlude any portion of the world.

Using the virtual manipulator control approach a six degree of freedom robot can be made to mimic the kinematic behavior of some other mechanism. The time varying extension of the virtual manipulator control can allow the traveler to interact with arbitrary rigid objects.

In addition the time varying characteristics of the a virtual manipulator can be tied to a dynamic model allowing the traveler to interact with a virtual object that possess dynamic characteristics.

The goal of this work is to advance the state of the art in haptic interaction in synthetic environments. To that end the virtual manipulator control approach will be developed. The control approach is designed for use on a general six degree of freedom robot in an effort to increase the availability of force display in synthetic environments. This is achieved by making it easier to acquire and maintain the haptic interaction device. In addition this approach should foster a greater sense of cooperation and collaboration in this area of research by allowing researchers to verify and extend the work performed by others at different facilities. In addition this approach to haptic interaction can be used in most if not all types of synthetic environments from head mounted displays to surround screen virtual environments.

The virtual manipulator approach also makes a contribution in the area of control theory and nonlinear systems apart from the area of haptic interaction. The system composed of the virtual manipulator control law and a six degree of freedom robot has an infinite continuum of equilibrium points defined by the end effector trajectory of the virtual manipulator. This is an unusual occurrence in control theory, where most systems are designed to have a single isolated equilibrium point. The stability of the continuum of equilibrium points will be demonstrated using a tradition Lyapunov argument.

Small two and three degree of freedom haptic devices are currently commercially available for use with today's graphic workstations. However, using the virtual manipulator control approach it will be possible for the next generation of graphics workstations to be equipped with a low cost six degree of freedom force feedback interface. Using a fish bowl style synthetic environment a virtual tool will be extended form the haptic device to allow engineers to develop prototype designs or analyze response data with the addition of force as an input channel. This advancement should provide a natural and effective human-computer interaction mechanism.

This work is divided into two major parts. Preceding Part I is a review of current literature in Chapter 2. Part I then discusses the area of haptic interaction. Specifically, Chapter 3 presents the virtual manipulator control law. The stability of this control law operating on a general six degree of freedom robot is shown in Chapter 4. The experimental hardware used to verify the virtual manipulator control scheme is presented in Chapter 5. In Chapter 6 experimental results obtained from the available hardware is exhibited. The second part of this dissertation develops a dynamic modeling approach that can be used with the B-spline geometric representation. This part begins with a description of finite element analysis in Chapter 7. Chapter 8 proposes a modification to the standard finite element analysis. A comparison of the standard and modified finite element analyses is presented in Chapter 9. Chapter 10 develops a technique for deforming dynamic models derived using the modified finite element analysis. A dynamic surface for use in a synthetic environment is developed in Chapter 11 using the modified finite element analysis. Following Part II, the final chapter, Chapter 12, discusses the results of this dissertation and examines areas of future research.

CHAPTER 2. LITERATURE REVIEW

Graphical display technology has advanced greatly in the past decade. Current graphics workstations can now render large, photo realistic scenes while still allowing real time interaction. These advanced computers are fueling the development of synthetic environment technology. A graphics workstation can generate two images of a particular scene, one calculated for the left eye of the observer and the second for the right eye of the observer. When the images are presented to the appropriate eye, the mind of the observer is able to fuse the images together to form a believable three dimensional picture of a scene that exists only in a computer database, a synthetic environment [41].

There are several ways to present these stereographic scenes to the traveler in a synthetic environment. The least immersive, in the sense that the traveler is least likely to believe that she is present in the conceptual world, is to simply display the images on a conventional computer monitor. A pair of liquid crystal display shutter glasses is used in this implementation approach to occlude the right eye image from the left eye and conversely to occlude the left eye image from the right eye. The result of this approach is that a three dimensional synthetic environment is created that occupies the space in front of and behind the computer monitor [20]. This synthetic environment technique is commonly referred to as a "fish bowl" because the traveler's view of the synthetic environment is similar to a person's view of fish swimming in an aquarium. Although this stereographic display technique is not highly immersive, it is a good starting point in the development of a synthetic environment.

Another approach to presenting stereographic scenes to a traveler is the use of a head mounted display (HMD). This technique uses two display monitors instead of one, as used in the fish bowl approach. Each eye is allowed to see one display and is occluded from seeing the other by means of some physical barrier [99]. This is generally achieved by using small display monitors that are mounted in a visor, that is placed over the face of the traveler. This approach is more immersive than the fish bowl technique because the traveler sees only what is displayed on the monitors, anything that is physically present around the traveler is occluded.

Although a HMD provides an immersive synthetic environment it also raises some problems that must be overcome by the designer of the synthetic environment. Because the traveler can not see the real world around her safety is an issue that must be addressed. However, a more important issue is registration. If the traveler is allowed to interact with the synthetic environments using her hands, a graphical representation of the hands must be placed in the synthetic environment. The graphical representation of the traveler's hands must be placed in the same position and orientation as her real hands, if there is error in the placement of the computer generated hands the traveler may become disoriented thus lowing the sense of immersion.

A head coupled display (HCD) is a stereographic presentation technique that is a generalization of the HMD. Because the display monitors used in a HMD must be worn on the head of the observer, they must be low weight so they do not cause fatigue or injury to the traveler. This weight limitation typically results in the use of low resolution display monitors in a HMD [14]. The HCD display technique was conceived to overcome the weight issue associated with a HMD. In a HCD heavier display monitors are used because the weight of the display hardware is counterbalanced by a mechanical linkage [69]. As a result the weight of the HCD display is of less importance, however the inertia of the display monitors and of the counterbalance mechanism are still experienced by the traveler and can impede on the sense of immersion.

The final stereographic presentation technique is a generalization of the fish bowl approach. Instead of allowing the traveler to view the synthetic environment from the outside, the stereographic images are projected onto the walls and floor of a room. This projection approach allows the traveler to step into the synthetic environment and interact with the objects that populate the environment [25], in the same fashion as the HMD. This projection technique removes some of the registration problems encountered with a HMD because there is no need to display any of the traveler's body, the traveler will be able to see her real hand. The main problem encountered with projection based systems is occlusion. If a real object or person stands between the traveler and the projection surface that portion of the synthetic environment will not be visible to the traveler.

The previous discussion was not presented to determine what is the "best" approach for presenting realistic three dimensional scenes to a traveler in a synthetic environment. The selection of a particular system is most often made by what resources are available to a project and what the ultimate goals of that project are. The main point of the discussion was to show the progression of stereographic presentation techniques and that the research and advancement of synthetic environment technology is viewed as an important and enabling technology by the world community.

The presentation of realistic three dimensional scenes focuses on the visualization capabilities of a synthetic environment. Even if synthetic environments were limited only to visualization, there is little doubt that this technology will still have a positive impact on science [26], engineering [78], architecture [12] and medicine [32]. However, synthetic environments are not limited to visualization. In fact even in synthetic environments developed solely for visualization purposes the traveler interacts with the environment by means of view point tracking [27], that is, the scene changes in response to the traveler's change in position or gaze. Much of the current research in synthetic environments is focusing on extending the potential of this technology by developing new interaction paradigms.

In order for the traveler in a synthetic environment to interact with the environment, the traveler must have an interface for communicating with the computer that is controlling the simulation of the environment. Traditionally, users interact with computers using a mouse and keyboard. A standard mouse is not an effective interface for a three dimensional synthetic environment because it is a two dimensional device. Although there are hand held keyboards, such as the Twiddler [39], keyboard interaction is also unnatural for interaction with a synthetic environment.

Two different types of devices have been developed as interface devices for travelers in a synthetic environment. The first type of device, a wand, is a three dimensional generalization of a conventional mouse. A wand is a hand held device that is instrumented so that the position and orientation of the device can be measured and utilized as input to the computer controlling the synthetic environment. In addition, a wand can be equipped with buttons that serve as additional digital inputs to the controlling computer. The second type of device, a glove [107], is used to provide more refined input to the computer controlling the synthetic environment. A glove is a device worn over the traveler's hand that is instrumented to track the motion of the traveler's hand as well as measure the motion of each digit of the hand. The data from a glove device can be used to allow dexterous manipulation of a virtual object in a synthetic environment by the traveler.

As the traveler in a synthetic environment is allowed to interact with the objects that populate the conceptual world, the database associated with the environment must extended. The database for a visualization synthetic environment must include a complete graphical description of the environment, such as object geometry, colors, textures and a lighting model. However, for an interactive synthetic environment the database must include a complete graphical description of the environment as well as a description of the interaction methodologies.

Therefore given a device, either a wand or a glove, an interaction protocol can be developed to facilitate picking up objects, moving them around and stacking them up. These types of interactions could be grouped as interacting with rigid objects, objects with constant and unchangeable geometry. Perhaps the objects that populate the synthetic environment should not be rigid, the interaction device can be used to allow the traveler to maneuver the object as well as deform the shape of the object. Once the objects that populate the synthetic environment are allowed to be deformed the size of the synthetic environment database again increases. The database must now include protocols that describe how the objects deform when the traveler acts on them.

It is easy to envision a synthetic environment developed as a design studio for an automobile stylist. The stylist can shape the contour of next years automobile not with clay but with a computer database. However, to the stylist the interaction metaphor is the same, using her hands to arrive at the desired form. The major obstacle that stands in the way of realizing this goal is the method of interaction [33]. The current interaction devices, both wands and gloves, are passive they only sense the traveler's motion and allow the synthetic environment to react based on the measurements. However, a new class of interaction devices are currently being investigated. These haptic devices [55] can not only sense the motion of the traveler in a synthetic environment but also react by applying force feedback to the traveler.

The technology used to develop haptic interaction devices is based on tele-operation equipment. Tele-operation is the control of a remote robotic slave manipulator by an operator using a master manipulator [103]. The use of tele-operation technology in synthetic environments is easy to imagine, the master manipulator interface is used to control the synthetic environment instead of the remote located slave manipulator [16]. In fact interaction techniques for synthetic environments have developed in much the same way as tele-operation equipment.

Early tele-operation systems allowed the operator to interact with the remote slave manipulator using a passive master manipulator. However, it was soon realized that these passive interaction approaches were difficult for the operator to use and the research into active force feedback master manipulators provided an effective solution to this problem. This parallel development of tele-operation systems and synthetic environments provides the designer of a synthetic environment system with a wealth of information to draw upon. Most importantly, the previous tele-operation research provides a description of a "good" force reflecting master, this information can be used to design a haptic interface for a synthetic environment.

Tele-operation research has described a "good" haptic interface as a low inertia, low friction manipulator that is back driveable [40]. This description of the ideal haptic interface manipulator is significantly different then the description of most commercially available robot manipulators. As a result most of the early research into force feedback for synthetic environments has focused on the design and construction of interface manipulators; which behave more like the ideal haptic interface than traditional robots [60].

Although all of the devices that will be described below were specially designed and fabricated, with only one prototype being built they can be classified into three groups. The first group contains highly specialized devices with only one intended use or function. Devices in this category are characterized by high fidelity reproduction of the desired sensation. However, the cost of this high fidelity is a loss of generality for the device. A good example of these types of devices is the piano action simulator [36]. Another example is a non-invasive surgical simulator developed for medical training [95].

The second group contains devices where the human-machine interface is accomplished by means of a stylist or grip. This category contains devices with both large and small working volumes. Large workspace devices have been developed with a single degree of freedom [74], four degrees of freedom [73], but most possess six degrees of freedom [1] [77] [96] [105]. These interfaces allow the human to feel contact on the palm of the hand by grasping a bar. Using these systems the traveler in a synthetic environment can sense boundaries, shapes, interact with virtual objects by moving them or deforming their geometry or even play tennis. Smaller pen-based interface arrangements have been used to deform free form surfaces [50], simulate surgical tools [5] [15] or allow standard computer interaction [56].

The third group contains devices that were developed to apply forces to the fingers of the traveler. Research has shown that for dexterous manipulation of an object it is sufficient to apply forces to the finger tips and sagital planes of the finger of the human [63]. Using various mechanical arrangements devices have been developed to apply finger tip forces to a single digit [68], two digits [49], three digits [13], and five digits [9]. In addition one device has been developed to apply finger tip forces as well as sagital plane forces to a single digit [65].

The construction of all these unique devices has effectively isolated most of the researchers in the area of haptic interaction for synthetic environments. Collaboration and verification of experimental results is difficult if not impossible when only one prototype of a haptic force feedback device exists. Some haptic devices are currently being sold to the public, making uniform hardware available to the haptics research community.

CH Products [80] and Microsoft [94] are both marketing a low end force feedback joystick aimed at PC gaming enthusiasts. Immersion Corporation [47] offers high end force feedback joysticks developed for commercial gaming use. Sensable Corporation has taken the three degree of freedom PHANToM prototype [67] into a commercially available haptic interaction device [93]. CyberGrasp [84] is offering a tendon driven, hand worn, exoskeleton force feedback device. Response to these commercially available devices has been positive. Because these devices are available to the general public, researchers have been able to establish support "communities" [91] to foster and enhance the development of haptics.

Although these commercially available devices are reducing the isolation faced by researchers in the field of haptic interaction. The manufacturers of the devices are still isolating themselves by using proprietary application programmer interfaces (API), such as the *GHOST* API [92] developed for the PHANToM device. Even in situations where the force feedback devices are for all intents and purposes the same, such as the low end PC force joystick market, the manufacturers have established proprietary API's [46] [72]. This lack of cooperation by manufacturers is of course expected but only adds to the difficulties faced by researchers and developers of haptic simulations.

However, recent research efforts in haptic interaction have focused on developing control approaches for implementing force feedback in a synthetic environment with commonly available robot manipulators. Although commonly available robots are generally not classified as ideal haptic devices, control implementations, more sophisticated then earlier tele-operation implementations, are now possible due to the increased computational power of personal computers. More sophisticated control implementations allow a less than ideal interface robot to respond as well as an ideal haptic device. In addition, because these control laws are developed for a general six degree of freedom robot manipulator, they can be applied to a common industrial robot or a high performance haptic interface [54] if this type of manipulator becomes available. These control approaches allow a general six degree of freedom robot to mimic the dynamic or kinematic behavior of some virtual manipulator.

The first attempts at controlling the dynamic behavior of robot interfaces to allow it to mimic another dynamic mechanism or object [43] utilized impedance control theory [42]. The natural converse to impedance control, admittance control has also been applied to the problem of modifying the dynamic characteristics of the interface robot by having the robot behave like another dynamic system [19] [106]. However, care must be exercised when selecting the desired dynamic behavior for an interface robot. There is without doubt a limit on the dynamic characteristics which can be mapped onto a given interface robot. Researchers are currently investigating techniques, such as Z-width for stiff walls [22], for determining how “transparent” a given interface robot can be made with respect to a desired set of dynamic characteristics [59].

The idea of having a robot behave kinematically like another virtual mechanism appears in the area of kinematically redundant robotic systems [28]. In this context the control approach was not used to generate an human - machine interface but was used to handle system redundancies. It has also been shown that the kinematic constraints imposed by a virtual manipulator in the context of a kinematically redundant system are the same as the kinematic constraints imposed in a hybrid position/force control problem [88]. The kinematic constraints of the virtual manipulator establish easily the directions of force control and position control in the hybrid formulation [52]. Although there are documented concerns associated with hybrid position/force control [29], most notably dimensionally inconsistent products, these concerns have been addressed in the area of cooperating manipulators and have resulted in the theory of generalized inverses [10].

The concept of virtual manipulators has been applied to tele-robot systems in order to simplify interaction with the master manipulator by kinematically constraining the master to follow the desired path defined by the end effector of the virtual manipulator [51]. This implementation assumes that the virtual manipulator is ideal and therefore no power can be transmitted to it or taken from it. This assumption allows the velocity of the virtual manipulator to be determined based on the interaction forces from the master and slave robots. The velocity of the virtual manipulator can then be integrated to determine the position of the virtual manipulator as a function of time during the course of operation of the system.

An alternative approach has been developed which uses the null space of the transpose of the virtual manipulator Jacobian to impose the virtual manipulator kinematic constraints onto a general six degree of freedom robot for use as an interface to a synthetic environment. Virtual manipulators have been developed to allow interaction with one dimensional [61], two dimensional [64] and three dimensional [62] constraint mechanisms. The virtual manipulator control law used in the three examples above has since been rederived in the context of a decoupling control [71] and is shown to have a performance advantage over the earlier idea virtual manipulator control law formulation [51].

In addition to the description of a good haptic device previous tele-operation research also addresses other critic control issues such as force scaling and time delays, that are of importance in synthetic environments. Force scaling is relevant in synthetic environments because the conceptual, computer generated world can have arbitrary size. Synthetic environments have been developed to allow investigation of molecular dynamics [79]. If force feedback devices are included in these microscopic synthetic environments, micro-macro force scaling will be required [53]. The issue of time delay is important in the area of distributed synthetic environments [75]. When several travelers are interacting with haptic devices remotely in the same synthetic environment the communication delays present in the system will have an impact on the performance of the haptic devices [3].

The main reason for using force feedback techniques in synthetic environments is to increase the level of immersion experienced by the traveler in attempt to increase the utility of the synthetic environment. These types of devices should increase the feeling that the traveler in a synthetic environment is present or immersed in the conceptual world. For example, when a traveler encounters a rigid object in a synthetic environment instead of the graphical representation of her hand passing through the object, resistance is felt. Rigid objects are prevalent in haptics research because they represent one of the most challenging tasks faced by a haptic display [21]. However, these devices are also capable of representing deformable objects and perhaps application of haptic devices to deformable objects will allow greater advances in the interactive abilities of synthetic environments than rigid objects.

As would be expected when a haptic device is included in a synthetic environment the database required for operation again increases. It must now include a complete graphical description of the world, an interactive description of the world and objects that populate it, and a force description [66] [102]. This force description describes the force - motion relationships between the conceptual world and the objects contained in it. The requirement of knowing the force - motion relationships immediately brings physically based modeling to interest.

Physics based modeling uses the laws of physics to obtain realistic simulations of objects defined by a computer graphics database. Because physics based models are developed using a force - motion relationship, they will make the assimilation of haptic devices into synthetic environments easier than tradition kinematic based motion and deformation techniques. Physically based models can be derived for rigid objects [7] to allow virtual objects to be positioned using force instead of moving the object by constraining it to move with the traveler's hand position. Deformable objects can also be modeled using physics based techniques thus allowing force input to produce deformation as opposed to free form deformation techniques [45] [90].

Physically based models for deformable objects have been developed using finite element analysis (FEA) techniques [38] [101]. Material behaviors other than elasticity, such as plasticity and fracture dynamics, can also be incorporated into these FEA models in a consistent framework [100]. The geometry of a FEA model can be defined by an implicit surface [70] that has been deformed either globally or locally [8] to represent the virtual object. However, it may not be possible to represent an arbitrary virtual object using an implicit function. As a result the geometry of a FEA model is typically represented with a grid of points which are interpolated using a set of shape functions to obtain the shape of the object [11]. The selection of an appropriate set of interpolation shape functions can be a complicated [17] and computer intensive [37] operation, which results in an obscure set of interpolation functions.

In order to allow the use of a common set of interpolation shape functions, some researchers have developed physically based models that have two geometric descriptions, one for display and the second for dynamic modeling [18] [81]. The results from the dynamic model are transformed by point inversion [82] or modal analysis [48]. However, the use of two geometric descriptions does not easily allow the shape of the virtual object to be represented using a traditional basis such as the B-spline or NURBS [2].

Recent work has developed physically based models using finite element techniques but using the B-spline basis in place of the traditional interpolation shape functions [30]. The use of the B-spline basis has been explored by the finite element community for modeling one-dimensional and two-dimensional time dependent problems [104], vibration of thick circular cylindrical panels [76], undular bore [35] and the non-linear Schrödinger equation [34]. This work has shown that the B-spline basis can be used effectively as a set of shape functions in FEA. However, this work used physically based dynamic models as an analysis tool instead of a mechanism for interaction.

In fact the rational extension of the B-spline basis, the NURBS basis, has been used to develop physically based models using FEA. This approach was used to associate dynamic characteristics with traditional NURBS curves and surfaces [86], NURBS swung surfaces [85] and triangular NURBS [87]. These NURBS based physically based dynamic models were used to facilitate interaction with the virtual object by the user, but the conceptual interaction forces were applied using a mouse driven, traditional, graphical user interface without a haptic force feedback device.

In addition the NURBS based dynamic models describe above were developed in the parametric space of the curve or surface. Although developing the dynamic model in parametric space simplifies model development, this results in a mismatch of the dynamic characteristic defined in parametric space and external forces defined in Cartesian space. This mismatch may result in an unexpected response of the model when the external forces are applied by a graphical user interface. However, if the external forces are applied by a haptic interaction device the mismatch will manifest as a form of force scaling, which in the worst case could result in unstable operation of the haptic interface. To prevent this mismatch it essential that the model be developed with respect to the same Cartesian coordinate system as the external forces [31]. In order to develop the model in Cartesian space the Jacobian for the curve or surface in required. This Jacobian matrix contains information describing the parameterization of the curve or surface.

This work will develop and present examples of a kinematic based virtual manipulator control law that can be used as a human-machine interaction mechanism in a synthetic environment. In addition, a framework will be developed to allow the shape of the object, that is defined by the virtual manipulator, to deform dynamically when external forces are applied. The dynamic model of the object will be developed using a B-spline based FEA approach. The combination of the haptic interaction device in concert with the physically based dynamic model will produce a synthetic environment that is more immersive that a simple graphic feedback synthetic environment.

PART I. HAPTIC INTERACTION

CHAPTER 3. CONCEPT AND CONTROL

Virtual manipulators are a new concept in the area of force feedback for synthetic environments. They allow a person traveling in a synthetic environment to interact physically with the objects that populate the computer generated world. The presentation of contact forces to the traveler will add to the realism needed to make the journey an immersive experience. In addition, the interaction force experienced by the traveler will provide more life-like and natural control over virtual objects. This chapter will begin by describing the motivation for the virtual manipulator interface approach and will conclude by deriving the appropriate robot control law to fulfill the concept requirements.

Virtual Manipulator Concept

The virtual manipulator concept is based on the idea that a general six degree of freedom robot can be used to mimic the kinematic behavior of some virtual manipulator with five or less degrees of freedom. The virtual manipulator is selected so that it provides the necessary contact forces associated with the object that is being explored. The robot interface is constrained to follow the virtual manipulator by enforcing a closed kinematic chain relationship between the robot end effector and the end effector of the virtual manipulator.

Consider, for example, constructing a virtual manipulator that will allow a traveler in a synthetic environment to manipulate a one degree of freedom crank [61], such as a gear shift mechanism. In this example the virtual manipulator is simply a one degree of freedom revolute manipulator whose position and orientation match the gear shift mechanism as shown in Figure 3.1. Figure 3.2 shows the closed kinematic chain relationship that must be enforce to ensure that robot interface behaves like the gear shift mechanism. This type of haptic display coupled with a synthetic environment containing an automobile interior could be utilized to examine the ergonomic characteristics such as position, orientation and range of motion of the gear shift mechanism.

It is important to note at this time that the virtual manipulator control approach is not limited to reproducing the kinematic characteristics of mechanisms. Time-varying virtual

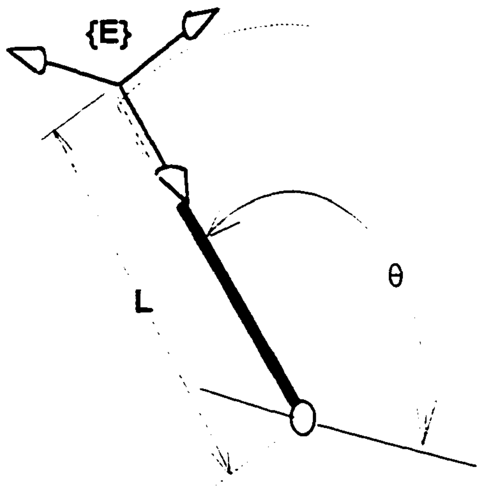


Figure 3.1. Virtual mechanism for constraint development.

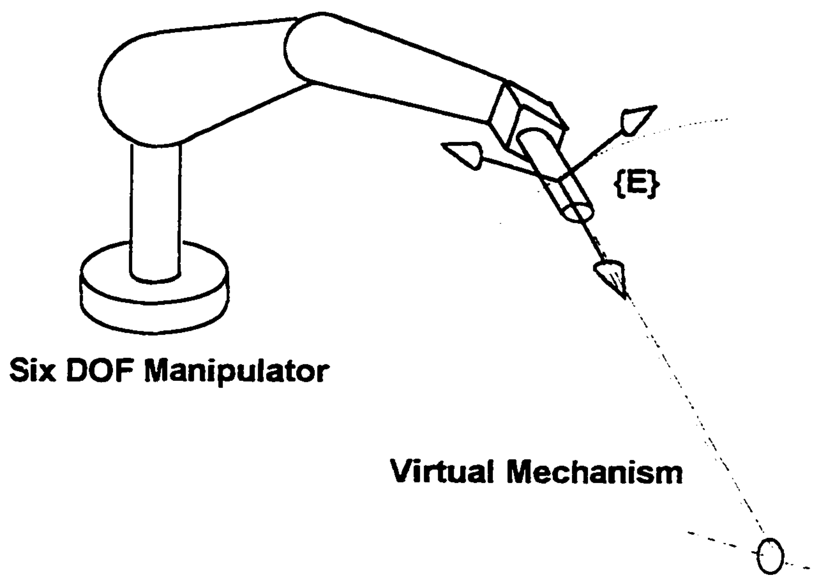


Figure 3.2. Closed kinematic chain relationship.

manipulators can be formulated to represent abstract synthetic objects such as general curves and surfaces [64]. However, the complexity of the virtual manipulator will be a function of the complexity of the object that is being represented. This subject will be addressed in more detail in Chapter 6; which describes experimental results. Using the overall concept of the virtual mechanism interface, the question of how to control the robot will be addressed.

Virtual Manipulator Control Law

In order to develop a robot control law that will enforce the closed kinematic chain requirement of virtual manipulator approach, some standard robotic analysis techniques will be used. A reader with experience in robotic analysis should have little difficulty following the derivation that will be presented, and numerous texts are available on the subject of robotic analysis that can be used to provide background information for this section [24] [89].

Consider for the moment that a virtual manipulator exists physically. If an operator grasps the virtual manipulator and applies an external force, $\mathbf{F}_e$, to the end effector a set of joint torques, τ_v, for the virtual manipulator can be calculated using the relationship in equation (3.1) that will prevent the virtual manipulator from moving. That is the vector of joint torques, τ_v, will allow the virtual manipulator under static equilibrium conditions to resist the external force, $\mathbf{F}_e$, and behave as if it were a fixed structure.

$$\tau_v = \mathbf{J}_v^t \mathbf{F}_e \tag{3.1}$$

The matrix $\mathbf{J}_v^t$ is the transpose of the Jacobian for the virtual manipulator. Note that the Jacobian, $\mathbf{J}_v$, and the external force, $\mathbf{F}_e$, must be represented in the same coordinate reference frame. For convenience this coordinate reference frame is taken to be the end effector frame of the virtual manipulator, although this selection is arbitrary.

However, a virtual manipulator is allowed to have at most five degrees of freedom. Therefore the transpose of the Jacobian for the virtual manipulator, $\mathbf{J}_v^t$, is an non-square matrix, in fact $\mathbf{J}_v^t$ will be a n by six matrix. Where n is the number of degrees of freedom for the virtual manipulator. As a result there is no unique mapping between the joint torque, τ_v, and the external end effector force, $\mathbf{F}_e$. There are an infinite number of external end effector forces, $\mathbf{F}_e$, that will yield the same vector of joint torques, τ_v.

If however, a set of artificial constraints is chosen, a unique map can be found between the joint torques, τ_v, and the external end effector forces, $\mathbf{F}_e$. One such constraint is to select the vector of external end effector forces, $\mathbf{F}_e$ to have the smallest norm in the least square sense. This result can be achieved by using the Moore and Penrose pseudo-inverse as shown in equation (3.2).

$$\mathbf{F}_e^* = \mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{J}_v\right)^{-1}\tau_v \tag{3.2}$$

The superscript (*) notation is used to indicate a least squares solution. It is important to note that the entries in the Jacobian matrix have physical units associated with them. In addition the inner product combination of these units in the term $\mathbf{J}_v^t\mathbf{J}_v$ results in the combination of physically dissimilar units [29]. This issue has been addressed by several researchers and one solution to this problem is to add a heuristic weight matrix to form a weighted pseudo-inverse [10] as shown in equation (3.3).

$$\mathbf{F}_e^* = \mathbf{A}\mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{A}\mathbf{J}_v\right)^{-1}\tau_v \tag{3.3}$$

The matrix $\mathbf{A}$ is a positive definite weighting matrix. The exact form of $\mathbf{A}$ will not be selected at this point.

Equations (3.1) and (3.3) can be combined to determine the least squares end effector force, $\mathbf{F}_e^*$,associated with a given end effector force, $\mathbf{F}_e$, as shown in equation (3.4) below.

$$\mathbf{F}_e^* = \mathbf{A}\mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{A}\mathbf{J}_v\right)^{-1}\mathbf{J}_v^t\mathbf{F}_e \tag{3.4}$$

The term $\mathbf{A}\mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{A}\mathbf{J}_v\right)^{-1}\mathbf{J}_v^t$ in equation (3.4) is commonly referred to as the range space filter, $\mathbf{R}$, for the transpose of the virtual manipulator Jacobian, $\mathbf{J}_v^t$. The range space filter, $\mathbf{R}$, for the transpose of the virtual manipulator Jacobian, $\mathbf{J}_v^t$, removes any components of the end effector force, $\mathbf{F}_e$, that don't influence the virtual manipulator's joint torques, τ_v, in the weighted least squares sense. That is the weighted least squares end effector force, $\mathbf{F}_e^*$ represents the components of the original end effector force, $\mathbf{F}_e$, that the virtual manipulator's joint actuators must resist in order to prevent motion.

The vector of forces removed by the range space filter, $\mathbf{F}_m$, can be calculated using equation (3.5).

$$\mathbf{F}_m = \mathbf{F}_e - \mathbf{F}_e^* = \left\{\mathbf{I} - \mathbf{A}\mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{A}\mathbf{J}_v\right)^{-1}\mathbf{J}_v^t\right\}\mathbf{F}_e \tag{3.5}$$

The term $\mathbf{I} - \mathbf{A}\mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{A}\mathbf{J}_v\right)^{-1}\mathbf{J}_v^t$ in equation (3.5) is commonly referred to as the null space filter, $\mathbf{S}$, for the transpose of the virtual manipulator Jacobian, $\mathbf{J}_v^t$. The null space filter, $\mathbf{S}$, for the transpose of the virtual manipulator Jacobian, $\mathbf{J}_v^t$, removes any components of the end effector force, $\mathbf{F}_e$, that influence the virtual manipulator's joint torques, τ_v, in the weighted least squares sense. That is the components of the end effector force that are in the null space of the transpose of the virtual manipulator Jacobian, $\mathbf{F}_m$, are resisted by the structure of the virtual manipulator, such as by the bearings, and do not have to be resisted by the actuators of the virtual manipulator as shown below in equation (3.6).

$$\begin{aligned}\tau_v &= \mathbf{J}_v^t \mathbf{F}_m \\ &= \mathbf{J}_v^t \left\{ \mathbf{I} - \mathbf{A}\mathbf{J}_v \left(\mathbf{J}_v^t \mathbf{A}\mathbf{J}_v \right)^{-1} \mathbf{J}_v^t \right\} \mathbf{F}_e \\ &= 0\end{aligned} \tag{3.6}$$

Now assume that a six degree of freedom robot is maneuvered so that it's end effector has the same position and orientation as the virtual manipulator with the goal that the robot will behave kinematically the same as the virtual manipulator. If the operator again grasps the end effector of the robot and applies the same external force, $\mathbf{F}_e$, that was applied previously to the virtual manipulator, a set of joint torques, τ_r, can be calculated using the relationship shown in equation (3.7) that will prevent the robot from moving.

$$\tau_r = \mathbf{J}_r^t \mathbf{F}_e \tag{3.7}$$

However the goal of the virtual manipulator control law is to constrain the robot end effector to follow the end effector of the virtual manipulator. Therefore the robot does not need to resist all components of the end effector force, $\mathbf{F}_e$, only those components that would have been resisted by the structure of the virtual manipulator, $\mathbf{F}_m$. Therefore, if the joint torques, τ_r, applied to the robot are changed to those shown in equation (3.8) the robot will be free to move along the end effector trajectory defined by the virtual manipulator but will resist motion in all other directions.

$$\tau_r = \mathbf{J}_r^t \left\{ \mathbf{I} - \mathbf{A}\mathbf{J}_v \left(\mathbf{J}_v^t \mathbf{A}\mathbf{J}_v \right)^{-1} \mathbf{J}_v^t \right\} \mathbf{F}_e \tag{3.8}$$

Equation (3.8) can be used as a control law for the robot interface to satisfy the virtual manipulator interface goal. The control law for the robot interface is independent of the control force generation scheme. That is there is no requirement on the origin of the end effector force, $\mathbf{F}_e$, this topic will be addressed in the next section.

Force Generation Scheme

The virtual manipulator control law is independent of the force generation scheme. As a result there are numerous ways to implement this control approach. One technique is to mount a six-axis force/torque transducer on the end effector of the robot and simply measure the forces being applied. However, there are difficulties associated with this approach. First, force transducers typically have poor noise characteristics. The presence of noise in force measurements can lead to unpredictable system behavior without proper compensation. Attempts to remove the noise by means of a filter introduce a phase lag into the force signal which can result in unstable user excited oscillations. Second, the high feedback gains needed for proper system response when the force loop is closed generally requires knowledge of the rate of change of the force signal for stable performance. The noise present in the original force signal prohibits obtaining force rate of change information. For these reasons measuring the end effector force for the virtual manipulator control law was not pursued. Instead the theory of operational space control [58] was selected as a viable candidate.

The operational space control formulation uses springs and dampers defined in Cartesian space to manipulate the end effector position and orientation of a robot. Figure 3.3 illustrates this control approach by showing the three linear springs used to control the end effector position, superimposed on the image of the robot. To control all six degrees of freedom of a general interface robot, three torsional springs, to control the orientation, are required in addition to the three linear springs shown in Figure 3.3.

The control law used for the operational space formulation is shown in equation (3.9).

$$\tau_r = \mathbf{J}_r^t \mathbf{F}_e \tag{3.9}$$

The terms τ_r, $\mathbf{J}_r^t$ and $\mathbf{F}_e$ are the vector of robot control torques, the transpose of the robot Jacobian and the external control force applied by the Cartesian space springs and dampers respectively. The external control force applied by the Cartesian space springs and dampers

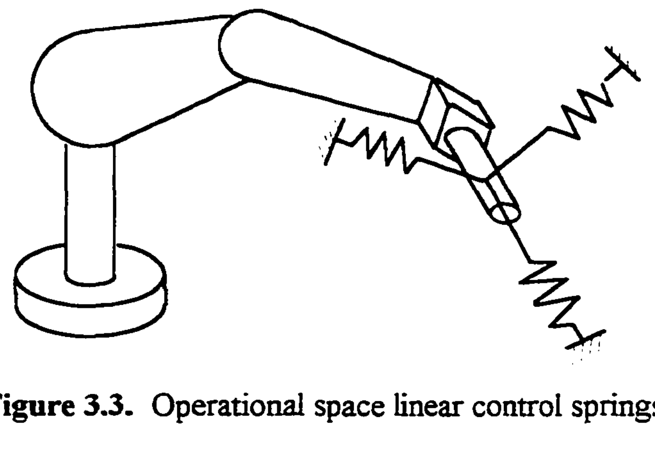


Figure 3.3. Operational space linear control springs.

can be evaluated using equation (3.10).

$$\mathbf{F}_e = \mathbf{K}_p\mathbf{e} + \mathbf{K}_v\dot{\mathbf{e}} \tag{3.10}$$

The matrices $\mathbf{K}_p$ and $\mathbf{K}_v$ are positive definite, symmetric gain matrices. In addition $\mathbf{e}$ is an error vector, containing both position and orientation errors, that describes the displacement of the robot end effector relative to the desired end effector position and orientation and $\dot{\mathbf{e}}$ is the rate of change of this error vector.

When the external force applied to the robot is developed using the operational space formulation, the virtual manipulator control law can be expressed in the form shown in equation (3.11).

$$\tau_r = \mathbf{J}_r^t\left\{\mathbf{I} - \mathbf{A}\mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{A}\mathbf{J}_v\right)^{-1}\mathbf{J}_v^t\right\}\left(\mathbf{K}_p\mathbf{e} + \mathbf{K}_v\dot{\mathbf{e}}\right) \tag{3.11}$$

With proper selection of the gain matrices $\mathbf{A}$, $\mathbf{K}_p$ and $\mathbf{K}_v$ the control law shown in equation (3.11) will constrain a general six degree of freedom robot to behave kinematically like the virtual manipulator with the null space filter, $\mathbf{S}$. The structure of the gain matrices $\mathbf{A}$, $\mathbf{K}_p$ and $\mathbf{K}_v$ will be selected based on the equilibrium point and stability analyses of the dynamic system; which will be addressed in the next chapter.

CHAPTER 4. STABILITY OF INTERACTION

The subject of safety for a haptic device is of paramount importance. Most large force feedback manipulators could without doubt deliver a fatal injury to its operator and even the smaller units posses sufficient force delivery to injury fingers and hands. The subject of robotic safety at the university or corporate level is generally handled by means of stringent safety requirements that are imposed onto a system designer. These safety requirements are essentially restate the most common sense rule of robotics, do not allow the operator or observer to stand within the reach of the robot manipulator. The safety requirements also specify various hardware and software protocols to immediately shut down an operating robot in the event that a person enters the workspace of the robot, whether intentionally or accidentally.

However, these legislated safety requirements are in direct conflict with the goals of the man-machine interface sought for this or any other haptic display. The Occupational Safety and Health Administration (OSHA) has little flexibility in the execution of their regulator responsibilities, if it walks like a duck and quacks like a duck, it is a duck. OSHA regulations apply specifically to autonomous devices. Powered robots under the control of human operators, such as an excavator, are unregulated. Experience in extending the level of autonomous action of the interface robot is needed to allow development of industry standards for this type of system.

As a result, for these types of devices to take a place our everyday life will require a change in societal opinions concerning computer controlled hardware . Most people would probably conclude that it is safer to navigate through congested holiday traffic on the interstate, than interact with a robotic interface. Is this the correct conclusion? Can society learn to "trust" computer controlled hardware? The current onslaught of internet hip is bringing computer hardware into more homes than ever before. This also includes haptic devices such as the Force FX joystick [80]. This increased familiarity with computers and their associated peripherals should allow the assimilation of an anthropomorphic interface devices such as the system considered in this work.

However, the assimilation process would be severely hampered by any accidents occurring at this early stage in development of the technology. Clearly the legislated safety requirements currently available were not intended or developed for haptic interaction devices. Until this technology has matured to the point where a concise set of safety requirements specifically developed for a general class of haptic interaction device can be written, the system designers will have to follow a more ambiguous set of moral guidelines such as Asimov's Laws of Robotics [4].

Asimov's Laws of Robotics

0. A robot may not injure humanity or, through inaction, allow humanity to come to harm.

1. A robot may not injure a human being, or through inaction, allow a human being to come to harm, except where that would conflict with the Zeroth Law.

2. A robo⁺ must obey the orders given to it by a human being, except where that would conflict with the Zeroth or First Law.

3. A robot must protect it's own existence, except where that would conflict with the Zeroth, First or Second Law.

Although it may appear that these laws are some what grandiose when applied to typical haptic display, the main point is clear. The safety of the operator must be the first concern of the robot. Because most if not all commonly encountered robots lack the sentience typically associated with fictional robotic systems the job of safety falls to the system designer. The first step in ensuring the safety of the operator interacting with a six degree of freedom robot running a virtual manipulator control law is to ensure that the dynamic system has appropriate stability characteristics.

The remainder of this chapter is devoted to the presentation of background material which will collimate in the proof of stability of the virtual manipulator control law when applied to a general six degree of freedom robotic manipulator. The second section in this chapter will present the system of dynamic equations for a six degree of freedom robot, as well as describe some of the mathematical properties of the elements of this dynamic model. The third section of this chapter will analyze the dynamic system of equations in order to determine the equilibrium points of the system. The fourth and final section will present a proof of stability to show that the virtual manipulator control law, as a general class, has acceptable stability characteristics for application as a haptic interface.

Dynamic System: Structure and Properties

This section will introduce and discuss several important mathematical properties of the dynamic system composed of a six degree of freedom robot that is controlled by the virtual manipulator control scheme. Specifically this section will discuss the matrices that make up the dynamic equations of motion of a general six degree of freedom robot as well as the gain matrices, $\mathbf{K}_p$ and $\mathbf{K}_d$, the null space filter matrix, $\mathbf{S}$, associated with the virtual manipulator control approach.

General Robot Dynamics

In order to begin the stability analysis of the haptic interface, the dynamic equations of motion for a general six degree of freedom robot with be introduced. For a general six degree of freedom robot, the joint space dynamic equations are shown in equation (4.1).

$$\mathbf{M}(\Theta)\ddot{\Theta} + \mathbf{V}(\Theta,\dot{\Theta}) + \mathbf{G}(\Theta) = \hat{\tau} - \mathbf{J}_r^t\mathbf{F}_e \qquad (4.1)$$

The matrix $\mathbf{M}(\Theta)$ is a six by six, positive definite and symmetric inertia matrix for the robot which is a function of the position of all of the joints of the robot, $\mathbf{V}(\Theta,\dot{\Theta})$ is a six by one vector of Coriolis and centrifugal terms which is a function of the joint positions as well as joint rates, $\mathbf{G}(\Theta)$ is a six by one vector forces arising from acceleration due to gravity which like the mass matrix is a function of only the configuration of the manipulator , finally $\hat{\tau}$ and

$\mathbf{F}_e$ are six by one vectors of joint actuator forces and external Cartesian forces and moments respectively.

The dynamics of robot can be determined analytically using a number of techniques such as Lagrangian or Newton-Euler. However, even for devices with two or three degrees of freedom these equations become extremely cumbersome and difficult to derive. Many researchers are turning to symbolic approaches to determine the equations of motion for six degree of freedom robots [19]. Symbolic processing packages such as Maple possess the ability to derive, optimize and code the dynamic equations of motion for a given manipulator. However, this process with not be undertaken in this work. The proof of stability presented here will only deal with the robot dynamics in the matrix form presented above. There will, however, be times during the proof where certain properties of the matrices and vectors contained in the equations of motion will be needed. The special properties used will be presented and proved for a general manipulator thereby allowing the proof of stability to apply to any six degree of freedom manipulator.

The first assumption that will be made in this proof of stability is that accurate gravity compensation can be obtained. The gravity compensation can take the form of an accurate analytic model of the gravitation forces that act on the manipulator or could be based on experimental data. The presence of gravity compensation will modify the system dynamics.

$$\mathbf{M}(\Theta)\ddot{\Theta} + \mathbf{V}(\Theta,\dot{\Theta}) + \mathbf{G}(\Theta) = \tau + \hat{\mathbf{G}}(\Theta) - \mathbf{J}_r^t\mathbf{F}_e \tag{4.2}$$

The vector $\hat{\mathbf{G}}(\Theta)$ is a six by one vector of gravity compensating joint forces. If the gravity compensation is reasonably accurate, then equation (4.3) below is true.

$$\mathbf{G}(\Theta) \approx \hat{\mathbf{G}}(\Theta) \tag{4.3}$$

As a result the system dynamics reduce to the form show in the equation below.

$$\mathbf{M}(\Theta)\ddot{\Theta} + \mathbf{V}\left(\Theta, \dot{\Theta}\right) = \tau - \mathbf{J}_r^t \mathbf{F}_e \tag{4.4}$$

In addition it is common to see the vector of Coriolis and centrifugal terms, $\mathbf{V}\left(\Theta, \dot{\Theta}\right)$, partitioned into a matrix form as shown in equation (4.5).

$$\mathbf{V}\left(\Theta, \dot{\Theta}\right) = \mathbf{V}_m\left(\Theta, \dot{\Theta}\right)\dot{\Theta} \tag{4.5}$$

The matrix $\mathbf{V}_m\left(\Theta, \dot{\Theta}\right)$ is a six by six matrix. Careful examination of the vector, $\mathbf{V}\left(\Theta, \dot{\Theta}\right)$, will reveal that there are several different ways to partition the vector into a matrix such that equation (4.5) is satisfied. However, it is assumed that the matrix, $\mathbf{V}_m\left(\Theta, \dot{\Theta}\right)$, is the result of a special type of partitioning of the original vector, $\mathbf{V}\left(\Theta, \dot{\Theta}\right)$. Due to the nature of the partitioning the matrix, $\mathbf{V}_m\left(\Theta, \dot{\Theta}\right)$, is endowed with two properties of interest. First the matrix, $\mathbf{V}_m(\Theta,\dot{\Theta})$, is a symmetric matrix. Second the matrix, $\mathbf{V}_m(\Theta,\dot{\Theta})$, will make the matrix, $\mathbf{Q}$, defined below a skew-symmetric matrix.

$$\mathbf{Q} = \dot{\mathbf{M}}(\Theta) - 2\mathbf{V}_m(\Theta,\dot{\Theta}) \tag{4.6}$$

The matrix, $\dot{\mathbf{M}}(\Theta)$, is the time derivative of the inertia matrix, $\mathbf{M}(\Theta)$, of the robot. These two properties of the matrix of Coriolis and centrifugal elements, $\mathbf{V}_m(\Theta,\dot{\Theta})$, will be used in the proof of stability for the dynamic system.

The joint space dynamics of a robot shown in equation (4.4) can be transformed into Cartesian space using the Jacobian relationship for the robot. The Jacobian of a robot provides a linear relationship between the joint space rates and Cartesian space rates as shown in equation (4.7).

$$^{b}\dot{\mathbf{x}} = {}^{b}\mathbf{J}\dot{\Theta} \tag{4.7}$$

The matrix, $^{b}\mathbf{J}$, is Jacobian of the robot and the vectors, $^{b}\dot{\mathbf{x}}$ and $\dot{\Theta}$, are the Cartesian space velocities and the joint space velocities, respectively. The leading superscript, *b*, on the Jacobian matrix and on the vector of Cartesian space velocities is used to represent the coordinate frame with which the Jacobian and the Cartesian space velocities are expressed in. As a result there are any number of Jacobian matrices for a given robot that can be used in equation (4.7). The selection of a particular Jacobian matrix is arbitrary and is generally made in order to simplify calculations. In addition to relating joint space and Cartesian space velocities the Jacobian of a robot can also be used to relate Cartesian space forces to joint space forces.

$$\tau={}^{b}\mathbf{J}^{T}\,{}^{b}\mathbf{F} \tag{4.8}$$

The vectors, τ and ${}^{b}\mathbf{F}$, are a vector of joint space forces and a vector of Cartesian space forces represented in the coordinate frame labeled *b*, respectively.

The Jacobian of a six degree of freedom robot will be a six by six matrix. For a given manipulator there are certain joint configurations where the Jacobian of the robot will lose column rank. These singular configurations typically form the boundaries of the usable workspace of the robot. If it is assumed that the robot is moving in an area of the workspace free of singularities, then the Jacobian of the robot can be inverted to obtain the inverse relationships of equations (4.7) and (4.9).

$$\dot{\Theta}={}^{b}\mathbf{J}^{-1}\,{}^{b}\dot{\mathbf{x}} \tag{4.9}$$

$${}^{b}\mathbf{F}={}^{b}\mathbf{J}^{-T}\tau \tag{4.10}$$

Equation (4.9) and its derivative and equation (4.10) can be used to transform the joint space robot dynamics into the Cartesian space. The derivative of equation (4.9) is shown below.

$$\ddot{\Theta}={}^{b}\mathbf{J}^{-1}\,{}^{b}\ddot{\mathbf{x}}+{}^{b}\dot{\mathbf{J}}^{-1}\,{}^{b}\dot{\mathbf{x}} \tag{4.11}$$

Substituting equations (4.9) and (4.11) into the joint space dynamic equation, equation (4.4) and pre-multiplying by the transpose of the inverse of the robot Jacobian yields the Cartesian space dynamic equations for the robot.

$$\hat{\mathbf{M}}\ddot{\mathbf{x}} + \hat{\mathbf{V}}\dot{\mathbf{x}} = \mathbf{J}^{-T}\tau - \mathbf{F}_e \tag{4.12}$$

Note the leading superscript on the Jacobian matrix has been removed to show that the dynamics can be obtained with respect to any coordinate frame. The matrices, $\hat{\mathbf{M}}$ and $\hat{\mathbf{V}}$, are the transformed inertia matrix and Coriolis and centrifugal matrix, respectively. The expression for these transformed matrices are shown below.

$$\hat{\mathbf{M}} = \mathbf{J}^{-T}\mathbf{M}\mathbf{J}^{-1} \tag{4.13}$$

$$\hat{\mathbf{V}} = \mathbf{J}^{-T}\mathbf{V}\mathbf{J}^{-1} + \mathbf{J}^{-T}\mathbf{M}\dot{\mathbf{J}}^{-1} \tag{4.14}$$

The proof of stability presented in the final section of this chapter will be performed in Cartesian space therefore the dynamic system model shown in equations (4.12), (4.13) and (4.14) will be used.

Gain Matrices

Two gain matrices are used in the virtual manipulator control scheme, $\mathbf{K}_p$ and $\mathbf{K}_d$. The matrix $\mathbf{K}_p$ is the position gain matrix and $\mathbf{K}_d$ is the damping gain matrix. The position gain matrix $\mathbf{K}_p$ will be discussed first followed the by damping gain matrix $\mathbf{K}_d$.

The position gain matrix, $\mathbf{K}_p$, is used to transform the end effector position and orientation errors into control forces [58]. The form of the position gain matrix, $\mathbf{K}_p$, is shown in the following equation.

$$\mathbf{K}_p = \begin{bmatrix} k_p^l & 0 & 0 & 0 & 0 & 0 \\ 0 & k_p^l & 0 & 0 & 0 & 0 \\ 0 & 0 & k_p^l & 0 & 0 & 0 \\ 0 & 0 & 0 & k_p^r & 0 & 0 \\ 0 & 0 & 0 & 0 & k_p^r & 0 \\ 0 & 0 & 0 & 0 & 0 & k_p^r \end{bmatrix} \tag{4.15}$$

The coefficients k_p^l and k_p^r are a linear control spring rate and a rotational control spring rate respectively. Both k_p^l and k_p^r are positive constants. The gain matrix, $\mathbf{K}_p$, is diagonal therefore it is also symmetric. The eigenvalues are all positive which allows the conclusion that, $\mathbf{K}_p$, is also a positive definite matrix.

The block style of the position gain matrix is used so that the same linear or rotational error produces the same force regardless of the direction of the error. In addition the block style of the gain matrix accounts for the dissimilar nature of the two types of error, linear and rotational by allowing the selection of different control spring rates.

The selection of the constants k_p^l and k_p^r are left to the system designer with the only restriction that they be greater than zero. As these constants are increased the "stiffness" of the system is increased. Increasing the stiffness of the system will reduce steady state errors and provide greater disturbance rejection, there is however a limit on how high these gains can be made. This performance limit is a function of the robot's physical characteristics and will vary from manipulator to manipulator. This performance limit is based on the amount of damping present in the system [22]. So although any six degree of freedom robot can be used as the interface device using the virtual manipulator control approach some manipulators such as the proposed high performance six degree of freedom haptic interface [54] which has been designed with a large amount of damping may offer a performance benefit.

The damping matrix, $\mathbf{K}_d$, has a structure that is similar to the position gain matrix, $\mathbf{K}_p$, but it's function is quite different. The damping gain matrix, $\mathbf{K}_d$, is used to transform the velocity of the end effector both linear and angular into viscous damping forces [58]. The form of the position gain matrix, $\mathbf{K}_d$, is shown in the following equation.

$$\mathbf{K}_d = \begin{bmatrix} k_d^l & 0 & 0 & 0 & 0 & 0 \\ 0 & k_d^l & 0 & 0 & 0 & 0 \\ 0 & 0 & k_d^l & 0 & 0 & 0 \\ 0 & 0 & 0 & k_d^r & 0 & 0 \\ 0 & 0 & 0 & 0 & k_d^r & 0 \\ 0 & 0 & 0 & 0 & 0 & k_d^r \end{bmatrix} \tag{4.16}$$

The coefficients k_d^l and k_d^r are a linear viscous damping rate and a rotational viscous damping rate respectively. Both k_d^l and k_d^r are positive constants. The gain matrix, $\mathbf{K}_d$, is diagonal therefore it is also symmetric. The eigenvalues are all positive which allows the conclusion that, $\mathbf{K}_d$, is also a positive definite matrix.

The block style of the damping gain matrix was selected for the same reasons described in the discussion of the position gain matrix. In addition as alluded to in the position gain matrix discussion, there is a correlation between the damping gain matrix and the position gain matrix. On common technique for determining the optimal values for the damping and position gain matrices involves increasing the damping gain matrix until excessive noise is transmitted through the actuators of the robot.

Once these maximum values for the damping gain matrix have been determined the elements of the matrix will be analyzed term by term as specified below. The value of the damping gain is obtained by using approximately eighty percent of this maximum value. The value of the damping matrix is then taken to have the form shown below.

$$k_d = 2\zeta\omega_n \tag{4.17}$$

The terms ζ and ω_n are a dimensionless damping ratio and a nature frequency respectively. The value of damping ratio, ζ, is selected by the system designer, a range between 0.7 and

1.0 is common. Based on the selected value for the damping ratio, ζ, the value of the nature frequency, ω_n, is determined using the equation (4.18).

$$\omega_n = \frac{2\zeta}{k_d} \tag{4.18}$$

The value of the position gain can be determined using the natural frequency, ω_n, found, above as shown below.

$$k_p = \omega_n^2 \tag{4.19}$$

The individual elements found using the procedure above can then be combined to form the position and damping gain matrices.

The heuristic procedure that was outlined above to determine the position and damping gain matrices is just one technique that can be applied. The application of the virtual manipulator control scheme does not in any way specify a procedure to select the position and damping gain matrices. This clearly shows the flexibility of the control strategy. The force generation scheme can be based on the position and damping gain matrices as described in this work using any appropriate technique for determination of the values in the gain matrices.

Null Space Filter Matrix

The null space filter matrix, $\mathbf{S}$, this matrix is constructed using a weighted Moore-Penrose pseudo-inverse of the transpose of the Jacobian of the virtual manipulator. The equation for the null space filter matrix is shown in the equation below.

$$\mathbf{S} = \mathbf{I} - \mathbf{A}\mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{A}\mathbf{J}_v\right)^{-1}\mathbf{J}_v^t \tag{4.20}$$

The null space filter matrix, $\mathbf{S}$, is idempotent, that is $\mathbf{S}$ times itself is equal to $\mathbf{S}$. This property is shown mathematically in the equation below.

$$\mathbf{SS} = \mathbf{S} \tag{4.21}$$

The proof that $\mathbf{S}$ is idempotent is shown below.

$$\begin{aligned}
\mathbf{SS} &= \left\{\mathbf{I} - \mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{J}_v\right)^{-1}\mathbf{J}_v^t\right\}\left\{\mathbf{I} - \mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{J}_v\right)^{-1}\mathbf{J}_v^t\right\} \\
&= \mathbf{I} - 2\mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{J}_v\right)^{-1}\mathbf{J}_v^t + \mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{J}_v\right)^{-1}\mathbf{J}_v^t\mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{J}_v\right)^{-1}\mathbf{J}_v^t \\
&= \mathbf{I} - 2\mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{J}_v\right)^{-1}\mathbf{J}_v^t + \mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{J}_v\right)^{-1}\mathbf{J}_v^t \\
&= \mathbf{I} - \mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{J}_v\right)^{-1}\mathbf{J}_v^t \\
&= \mathbf{S}
\end{aligned}$$

It is known that the Jacobian of the virtual manipulator does not have full rank, for this reason it is intuitive that the null space filter matrix, $\mathbf{S}$, will not have full rank. However, the rank of the null space filter matrix, $\mathbf{S}$, is not known and will be critical in the next section concerning the equilibrium analysis of the dynamic system. Therefore, the following paragraphs will perform a detailed analysis of the null space filter matrix aimed at determining the rank of the matrix.

The process of determining the rank of the null space filter, $\mathbf{S}$, will be begin by determining the rank of the range space filter, $\mathbf{R}$.

$$\mathbf{R} = \mathbf{A}\mathbf{J}_v\left(\mathbf{J}_v^t\mathbf{A}\mathbf{J}_v\right)^{-1}\mathbf{J}_v^t \tag{4.22}$$

The matrix, $\mathbf{J}_v$, is the m by n Jacobian of the virtual manipulator, which is assumed to have full column rank.

$$\operatorname{rank}\left(\mathbf{J}_v\right) = n \tag{4.23}$$

The integer n is the number of degree of freedom of the virtual manipulator. In addition m is the number of degrees of freedom of the interface robot. As a result m is generally six, however, results are shown for the case of planar motion, in this situation m will be three. The matrix, $\mathbf{A}$, in equation (4.22) is an m by m positive definite and symmetric weighting matrix. The fact that the weighting matrix, $\mathbf{A}$, is positive definite and symmetric ensures that the matrix has full rank.

$$\operatorname{rank}(\mathbf{A}) = m \tag{4.24}$$

The product of the weighting matrix and the Jacobian of the virtual manipulator, $\mathbf{AJ}_v$, will be an m by n matrix. The following rank inequality [44] will allow the rank of the matrix, $\mathbf{AJ}_v$, to be determined.

$$\operatorname{rank}(\mathbf{C}) + \operatorname{rank}(\mathbf{D}) - q \le \operatorname{rank}(\mathbf{CD}) \le \min\{\operatorname{rank}(\mathbf{C}), \operatorname{rank}(\mathbf{D})\} \tag{4.25}$$

The matrix, $\mathbf{C}$, has size p by q and the matrix, $\mathbf{D}$, has size q by r. Using the inequality in equation (4.25) it can be concluded that the rank of the product, $\mathbf{AJ}_v$, is the same as the rank of the virtual manipulator Jacobian, $\mathbf{J}_v$.

$$\operatorname{rank}(\mathbf{AJ}_v) = \operatorname{rank}(\mathbf{J}_v) = n \tag{4.26}$$

The information in equation (4.26) can now be used with the rank inequality shown in equation (4.25) to investigate the rank of the n by n matrix, $\mathbf{J}_v^T\mathbf{AJ}_v$.

$$2n - m \le \operatorname{rank}(\mathbf{J}_v^T\mathbf{AJ}_v) \le n \tag{4.27}$$

As shown in equation (4.27) the rank of the matrix, $\mathbf{J}_v^T\mathbf{A}\mathbf{J}_v$, can not be determined, an upper and lower bound can only be specified. However, for the range space filter, $\mathbf{R}$, to exist the inverse of the matrix, $\mathbf{J}_v^T\mathbf{A}\mathbf{J}_v$, must exist. This requires that the matrix, $\mathbf{J}_v^T\mathbf{A}\mathbf{J}_v$, has full rank.

$$\operatorname{rank}\left(\mathbf{J}_v^T\mathbf{A}\mathbf{J}_v\right) = n \tag{4.28}$$

If equation (4.28) is not true then the range space filter, $\mathbf{R}$, will not exist and consequently the null space filter, $\mathbf{S}$, will not exist. If the null space filter matrix, $\mathbf{S}$, does not exist then the virtual manipulator control law can not be implemented. Therefore it will be assumed that equation (4.28) is true for all virtual manipulator control laws of interest.

If the rank of the n by n matrix, $\mathbf{J}_v^T\mathbf{A}\mathbf{J}_v$, is n then the inverse of the matrix exists and also has full rank.

$$\operatorname{rank}\left(\mathbf{J}_v^T\mathbf{A}\mathbf{J}_v\right) = \operatorname{rank}\left(\left[\mathbf{J}_v^T\mathbf{A}\mathbf{J}_v\right]^{-1}\right) = n \tag{4.29}$$

Knowing that the m by n matrix, $\mathbf{A}\mathbf{J}_v$, has rank equal to n, equation (4.26), and the n by n matrix, $\left[\mathbf{J}_v^T\mathbf{A}\mathbf{J}_v\right]^{-1}$, has rank equal to n, the rank inequality shown in equation (4.25) can be used to verify equation (4.29).

$$\operatorname{rank}\left(\mathbf{A}\mathbf{J}_v\left[\mathbf{J}_v^T\mathbf{A}\mathbf{J}_v\right]^{-1}\right) = n \tag{4.30}$$

Finally, application of the rank inequality shown in equation (4.25) to the *m* by *n* matrix, $\mathbf{AJ}_v\left[\mathbf{J}_v^T\mathbf{AJ}_v\right]^{-1}$, with rank equal to *n* and the *n* by *m* matrix, $\mathbf{J}_v^T$, with rank equal to *n* allows the conclusion that the *m* by *m* range space filter, $\mathbf{R}$, has rank equal to the rank of the virtual manipulator Jacobian, $\mathbf{J}_v$.

$$\operatorname{rank}\left(\mathbf{AJ}_v\left[\mathbf{J}_v^T\mathbf{AJ}_v\right]^{-1}\mathbf{J}_v^T\right) = \operatorname{rank}(\mathbf{R}) = n \tag{4.31}$$

Now having established the rank of the range space filter, $\mathbf{R}$, as shown in equation (4.31), the rank of the null space filter, $\mathbf{S}$, can be addressed.

$$\mathbf{S} = \mathbf{I} - \mathbf{R} \tag{4.32}$$

The matrix, $\mathbf{I}$, is an *m* by *m* identity matrix. Equation (4.32) can be rewritten as shown in equation (4.33) below.

$$\mathbf{I} = \mathbf{R} + \mathbf{S} \tag{4.33}$$

The rank inequality [44] will help establish the rank of the null space filter matrix, $\mathbf{S}$.

$$\operatorname{rank}(\mathbf{E} + \mathbf{F}) \leq \operatorname{rank}(\mathbf{E}) + \operatorname{rank}(\mathbf{F}) \tag{4.34}$$

For the case of the null space filter matrix, $\mathbf{S}$, and the range space filter matrix, $\mathbf{R}$, equation (4.34) can be expressed as shown in equation (4.35) below.

$$\operatorname{rank}(\mathbf{R}+\mathbf{S}) \leq \operatorname{rank}(\mathbf{R}) + \operatorname{rank}(\mathbf{S}) \tag{4.35}$$

Equation (4.26) can be simplified equation (4.31) and (4.33) and knowing that an identity matrix always has full rank.

$$m - n \leq \operatorname{rank}(\mathbf{S}) \tag{4.36}$$

Equation (4.35) establishes a lower limit on the rank of the null space filter matrix, $\mathbf{S}$. In order to determine the upper limit for the rank of the null space filter matrix, $\mathbf{S}$, the product of the null space filter matrix, $\mathbf{S}$, and the range space filter matrix, $\mathbf{R}$, will be used in conjunction with the rank inequality shown in equation (4.25).

$$\operatorname{rank}(\mathbf{R}) + \operatorname{rank}(\mathbf{S}) - m \leq \operatorname{rank}(\mathbf{RS}) \tag{4.37}$$

The product of the range space filter matrix, $\mathbf{R}$, and the null space filter matrix, $\mathbf{S}$, is shown in equation (4.38) below.

$$\mathbf{RS} = (\mathbf{I} - \mathbf{S})\mathbf{S} = \mathbf{S} - \mathbf{SS} = \mathbf{S} - \mathbf{S} = \mathbf{0} \tag{4.38}$$

Equation (4.38) allows the conclusion that the rank of the product, $\mathbf{RS}$, is zero. Therefore equation (4.37) can be simplified into the form shown in equation (4.39).

$$\operatorname{rank}(\mathbf{S}) \leq m - n \tag{4.39}$$

Now that the upper and lower bounds on the rank of the null space filter matrix, $\mathbf{S}$, have been established in equations (4.36) and (4.39), the rank of the null space filter, $\mathbf{S}$, can be determined.

$$\operatorname{rank}(\mathbf{S}) = m - n \tag{4.40}$$

This section has presented many important facts about the matrices that compose the dynamic system. This information will be used in the next section of this chapter which will perform any equilibrium analysis of the dynamic system as well as in the last section of this chapter which will provide a proof of stability for the virtual manipulator control law.

Equilibrium Point Analysis

Before discussing the stability of a system it is first essential to analyze the system to determine the number and location of the system's equilibrium points. An investigation of the equilibrium points of a system is the correct place to begin a discussion of the stability characteristics of a system. Although it is common to hear the expression "the system is stable" this statement is ambiguous, misleading and should be avoided. The correct result from a stability discussion is whether or not a particular equilibrium point is stable or not stable.

An equilibrium point is an invariant point, y^*, in the system's state space which has the property that if the system starts at the equilibrium point, y^*, it will remain at the equilibrium point for all time [57]. If we consider a general nonlinear autonomous system, this is a class that includes the dynamic equations of motion for a robot manipulator utilizing the virtual manipulator control law, as shown below.

$$\dot{y} = f(y) \tag{4.41}$$

Then the equilibrium points of the system are the roots to the equation shown below.

$$f(y) = 0 \tag{4.42}$$

This therefore outlines a procedure that can be followed in order to determine number and location of the equilibrium points associated with a system.

To begin an equilibrium point analysis for a general six degree of freedom robot using the virtual manipulator control law the dynamic equations of motion are required. The dynamics of a six degree of freedom robot were shown in equation (4.12) in the first section of this chapter, repeated here for convenience.

$$\hat{\mathbf{M}}\ddot{\mathbf{x}} + \hat{\mathbf{V}}\dot{\mathbf{x}} = \mathbf{J}^{-T}\tau - \mathbf{F}_e \tag{4.43}$$

The vector, τ, is specified by the virtual manipulator control developed in Chapter 3, equation (3.11), repeated here for convenience.

$$\tau = \mathbf{J}^T\mathbf{S}\mathbf{K}_p\mathbf{e} + \mathbf{J}^T\mathbf{S}\mathbf{K}_v\dot{\mathbf{e}} \tag{4.44}$$

The vector, $\mathbf{e}$, is the error between the position of the virtual manipulator end effector and the end effector of the interface robot and the vector, $\dot{\mathbf{e}}$, is the rate of change of the error vector.

Although this equation is not written in state space format like equation (4.41) an equivalent set of equilibrium condition can be derived. If the dynamic system of equations (4.43) were placed into state space form, the equivalent set of equilibrium conditions would have the form shown below.

$$\begin{aligned}\dot{\mathbf{x}} &= 0\\ \ddot{\mathbf{x}} &= 0\end{aligned}$$

Substituting the equivalent equilibrium conditions above into the dynamic system of equations yields that following equilibrium equation.

$$\mathbf{S}\mathbf{K}_p\mathbf{e} + \mathbf{S}\mathbf{K}_v\dot{\mathbf{e}} = 0 \tag{4.45}$$

It is important to note that the external force, $\mathbf{F}_e$, in equation (4.43) was also set to zero to arrive at equation (4.45). The external force, $\mathbf{F}_e$, will be applied to the system by the human operator, as a result it is not possible to model this force. Therefore the external force will be considered as a disturbance to the system. The stability of the system in the absence of disturbances should be considered first. After the stability of the system has been verified, the control system can be analyzed to determine how robust the system will be when faced with unknown disturbances. In addition, the external force $\mathbf{F}_e$ represents the coupling between robot interface and human machine. The issue of coupled stability is typically addressed using a passivity argument which is also possible in this situation but will not be discussed in this work.

Equation (4.45) can be expanded by substituting in the expressions for the error, $\mathbf{e}$, and the rate of change of the error, $\dot{\mathbf{e}}$.

$$\begin{aligned}\mathbf{e} &= \mathbf{x}_v - \mathbf{x}\\ \dot{\mathbf{e}} &= \dot{\mathbf{x}}_v - \dot{\mathbf{x}}\end{aligned}$$

$$\mathbf{SK}_p\mathbf{x}_v - \mathbf{SK}_p\mathbf{x} + \mathbf{SK}_v\dot{\mathbf{x}}_v - \mathbf{SK}_v\dot{\mathbf{x}} = 0 \tag{4.46}$$

The desired trajectory, the path of the end effector of the virtual manipulator, is not a function of time but is a function of the configuration of the robot interface. This fact comes from the existence of the desired closed kinematic chain relationship between the robot interface and virtual manipulator. The position of the virtual manipulator is related to the position of the robot by some nonlinear relationship as shown in equation (4.47).

$$\theta_v = g(\theta) \tag{4.47}$$

The vector, θ_v, contains the positions of the joints of the virtual manipulator and the vector, θ, contains the positions of the joints of the robot. The function, $g(\circ)$, in equation (4.47) is a nonlinear vector function containing *n* independent equations. The integer *n* is the number of degrees of freedom of the virtual manipulator. Equation (4.47) can be modified using the forward kinematics of the virtual manipulator and the inverse kinematics of the robot interface as shown in equation (4.48).

$$\mathbf{x}_v = g'(\mathbf{x}) \tag{4.48}$$

The function, $g'(\circ)$, is a nonlinear vector function containing *m* equations of which *n* are independent. The integer *m* is the number of degrees of freedom of the Cartesian space in which the robot interface is maneuvering.

Equation (4.48) can be differentiated to obtain the linear rate relationship between the velocity of the virtual manipulator and the velocity of the robot as shown below.

$$\dot{\mathbf{x}}_v = \mathbf{J}_{g'}\dot{\mathbf{x}} \tag{4.49}$$

The matrix, $\mathbf{J}_{g'}$, is the Jacobian matrix associated with the nonlinear vector function, $g'(\circ)$. Equations (4.48) and (4.49) can be substituted into equation (4.46) to obtain the equilibrium condition shown in equation (4.50).

$$\mathbf{SK}_p(g'(\mathbf{x}) - \mathbf{x}) + \mathbf{SK}_v(\mathbf{J}_{g'} - \mathbf{I})\dot{\mathbf{x}} = 0 \tag{4.50}$$

However, $\dot{\mathbf{x}} = 0$, for all equilibrium points therefore equation (4.50) can be further simplified.

$$\mathbf{SK}_p(g'(\mathbf{x}) - \mathbf{x}) = 0 \tag{4.51}$$

Equation (4.51) can be solved to determine the equilibrium points associated with the dynamic system. The goal of the virtual manipulator control law was to derive a control law which had an infinite number of equilibrium points and these would be defined by the trajectory of the end effector of the virtual manipulator. This desired infinite continuum of equilibrium points all satisfy equation (4.52) below.

$$\mathbf{e} = g'(\mathbf{x}) - \mathbf{x} = 0 \tag{4.52}$$

However the rank analysis of the null space filter, $\mathbf{S}$, from the previous section has shown that this matrix does not have full rank as shown in equation (4.40), repeated here for convenience.

$$\operatorname{rank}(\mathbf{S}) = m - n \tag{4.53}$$

In addition the product of the null space filter matrix, $\mathbf{S}$, and the position gain matrix, $\mathbf{K}_p$, can be shown to have the same rank as the matrix, $\mathbf{S}$. Therefore, in addition to the desired continuum of equilibrium points defined by equation (4.52) there is a second n - dimensional continuum of equilibrium points which arise from the fact that matrix, $\mathbf{SK}_p$, does not have full rank.

This second continuum of equilibrium points can be removed by using the closed kinematic chain constraint equations contained in the vector function $g(\circ)$ or $g'(\circ)$. This process of removing the undesired equilibrium points is most easily understood by viewing the position vectors, $\mathbf{x}$ and $\mathbf{x}_v$, with respect to the end effector coordinate frame of the virtual manipulator. When the displacements of the robot and the virtual manipulator are viewed from the end effector coordinate frame of the virtual manipulator the desired continuum of equilibrium points collapse into a single point, the origin of the coordinate frame. This fact will also be used in the next section concerning stability. This collapse is caused because the position of the end effector of the virtual manipulator in the end effector frame of the virtual manipulator is defined to be zero.

$${}^{E}\mathbf{x}_{v} \equiv 0 \tag{4.54}$$

Here the leading superscript, E, is used to denote that the vector, $\mathbf{x}_v$, is written with respect to the end effector coordinate frame of the virtual manipulator. Therefore equation (4.52) can be rewritten as shown below.

$${}^{E}\mathbf{e} = {}^{E}\mathbf{g}'(\mathbf{x}) - {}^{E}\mathbf{x} = -{}^{E}\mathbf{x} = 0 \tag{4.55}$$

However, in equation (4.55) the position of the robot interface, ${}^{E}\mathbf{x}$, is not only a function of the robot joint variables, θ, but also a function of the n virtual manipulator joint variables, θ_v. The closed kinematic chain constraint equations, equation (4.47), are determined by defining n elements of the displacement vector, ${}^{E}\mathbf{x}$, equal to zero. The selection of the elements in the displacement vector that are set to zero is some what arbitrary but the goal of this process is to establish n equations of the form shown in equation (4.56), which are independent of the $m-n$ equations of equation (4.51).

$${}^{E}\mathbf{x}_{i} = 0 \tag{4.56}$$

The variable, ${}^{E}\mathbf{x}_i$, is the *ith* element in the vector, ${}^{E}\mathbf{x}$.

When this process is used the only equilibrium point, when viewed from the end effector space of the virtual manipulator, is the origin. The infinite continuum of the equilibrium points that were obtained from the fact that the matrix, $\mathbf{SK}_p$, did not have full rank have been removed by appropriate selection of the closed kinematic chain constraint equations. When the system is viewed from the world coordinate system, there is an infinite continuum of equilibrium points that are defined by equation (4.52), which was the goal of the virtual manipulator control law.

Stability Analysis

The previous chapter has developed the virtual manipulator constraint controller. This control structure generates control forces based on standard Cartesian space control techniques. However, after the control forces are generated the portion of the control force that lies in the range space of the transpose of the Jacobian of the virtual manipulator is removed by multiplication by the null space filter matrix. Earlier work has shown that the Cartesian space proportional plus derivative control approach is globally uniformly asymptotically stable. However, the inclusion of the null space filter matrix and the kinematic constraints into the control law requires an analysis of stability to verify that the new control law has acceptable stability characteristics to warrant use in a haptic display application.

This stability analysis will examine the stability of the infinite continuum of equilibrium points found in the previous section. The stability analysis of a control system that contains multiple equilibrium points is a subject typically not addressed in control literature [57]. When the stability of a continuum of equilibrium points is addressed it is mainly for mathematical interest [6] and not to show the stability of a particular system. In addition the stability of these equilibrium points will be analyzed with respect to the end effector coordinate frame of the virtual manipulator. This coordinate frame is the obvious choice for stability analysis because the continuum of equilibrium points collapses to a single point located at the origin in this reference frame. As a result the stability of this equilibrium point can be determined using standard techniques such as Lyapunov theory and La Salle's theorem.

This section will present two proofs of stability, the first will be for a slightly modified virtual manipulator control law. The second proof will show the conditions necessary for stability of the original virtual manipulator control law developed in Chapter 3. The reason for the modifications to virtual manipulator control law made in the first proof will be discussed later. In addition both proofs presented in this section will be performed with respect to the end effector space of the virtual manipulator. This fact was represented in the last section by placing a leading superscript, E, on necessary variables. This section assumes that all displacements, velocities, accelerations and Jacobians are written with respect to the end effector coordinate frame of the virtual manipulator. Therefore the leading superscript notation will only be used in situations where confusion might result.

Modified Control Law

This subsection will show the stability of a slightly modified virtual manipulator control law. The modifications to the control law are shown in equation (4.57).

$$\tau = -\mathbf{J}^T\mathbf{S}\mathbf{K}_p\mathbf{x} - \mathbf{J}^T\mathbf{K}_v\dot{\mathbf{x}} \tag{4.57}$$

The form of the damping term of the virtual manipulator control has been modified in equation (4.57). Specifically, the proportional plus derivative damping term has been replaced with a minor loop velocity feedback term. This term adds some rate damping to all motion directions, including the direction of motion along the virtual constraint. In addition none of the minor loop velocity feedback term is removed by the null space filter matrix.

The dynamic system model for a general robot running the modified virtual manipulator control law in equation (4.57) is shown in equation (4.58) below.

$$\hat{\mathbf{M}}\ddot{\mathbf{x}} + \hat{\mathbf{V}}\dot{\mathbf{x}} + \mathbf{K}_v\dot{\mathbf{x}} + \mathbf{S}\mathbf{K}_p\mathbf{x} = 0 \tag{4.58}$$

The stability of the origin of this dynamic system will be investigated using Lyapunov's direct method. In order to carry out a Lyapunov stability analysis a Lyapunov function, $V(\mathbf{x},\dot{\mathbf{x}})$, must be found that is positive definite in some domain that includes the origin, the first derivative of this Lyapunov function with respect to time must be continuous in the domain and which is negative semi-definite in the domain. The proposed Lyapunov function candidate selected for this stability analysis is shown in the equation below.

$$V(\mathbf{x},\dot{\mathbf{x}}) = \tfrac{1}{2}\dot{\mathbf{x}}^t\hat{\mathbf{M}}\dot{\mathbf{x}} + \tfrac{1}{2}\mathbf{x}^t\mathbf{S}\mathbf{K}_p\mathbf{x} \tag{4.59}$$

This function was selected based on the energy of the system, which is a traditional starting point for a candidate function. The first term represents the kinetic energy associated with the haptic interface and the second term represents the potential energy. However, the function shown in equation (4.59) may not be a valid Lyapunov function when the domain, Ω, is $\Re^{2m}$. The integer, *m*, is the number of degrees of freedom of the interface robot. The product of the null space filter matrix, $\mathbf{S}$, and the position gain matrix, $\mathbf{K}_p$, does not have full rank and as a result will not be positive definite.

This product can be shown to be positive semi-definite. In order to shown that the matrix, $\mathbf{SK}_p$, is positive semi-definite the matrix must be symmetric. This restriction determines the form of the weighting matrix, $\mathbf{A}$, in the null space filter matrix. In order for the matrix, $\mathbf{SK}_p$, to be symmetric the weighting matrix must have the form shown below.

$$\mathbf{A} = \alpha\mathbf{K}_p \tag{4.60}$$

The variable, α, is a positive scalar number that is nonzero.

This coupled with the fact that the closed kinematic chain constraint equations reduces the size of the state space by the number of degrees of freedom of the virtual manipulator, *n*, allows the conclusion that the proposed candidate function is positive definite in the domain, $\Omega = \Re^{2m-n}$. The closed kinematic chain constraint equations reduce the size of the state space by *n*, by defining *n* elements of the end effector position vector of the interface robot to zero by appropriate selection of the joint variables associated with the virtual manipulator. These *n* elements in the end effector position vector have not been removed from the proposed candidate function because their associated velocities are not in general zero. This may at first appear to be a contradiction that an element of a position vector is defined to be zero but the velocity of the associated with the position element is nonzero. However, this situation is the result of having the positions and velocities defined with respect to the end effector space of the virtual manipulator. Because the reference frame is moving it is possible for a position element to remain zero while the interface robot has a nonzero velocity in the direction associated with the position element.

Now that the function candidate has been shown to be positive definite in the domain, Ω, the continuity of the first derivative of this function will be investigated. The time derivative of the Lyapunov function candidate is shown in equation (4.61).

$$\dot{V}(\mathbf{x},\dot{\mathbf{x}}) = \tfrac{1}{2}\dot{\mathbf{x}}^t\dot{\hat{\mathbf{M}}}\dot{\mathbf{x}} + \dot{\mathbf{x}}^t\hat{\mathbf{M}}\ddot{\mathbf{x}} + \dot{\mathbf{x}}^t\mathbf{S}\mathbf{K}_p\mathbf{x} \tag{4.61}$$

Note the position gain matrix, $\mathbf{K}_p$, is a constant matrix by selection and this proof assumes that in the end effector coordinate frame of the virtual manipulator the null space filter matrix, $\mathbf{S}$, is also a constant matrix. This assumption of a constant null space filter is true for all virtual manipulators presented in this work, however, a virtual manipulator may exist that would not satisfy this assumption.

The system dynamics in the end effector space of the virtual manipulator, equation (4.58), can be substituted into equation (4.61) to remove the vector of Cartesian accelerations, $\ddot{\mathbf{x}}$.

$$\dot{V}(\mathbf{x},\dot{\mathbf{x}}) = \tfrac{1}{2}\dot{\mathbf{x}}^t\dot{\hat{\mathbf{M}}}\dot{\mathbf{x}} - \dot{\mathbf{x}}^t\hat{\mathbf{V}}\dot{\mathbf{x}} - \dot{\mathbf{x}}^t\mathbf{S}\mathbf{K}_p\mathbf{x} - \dot{\mathbf{x}}^t\mathbf{K}_d\dot{\mathbf{x}} + \dot{\mathbf{x}}^t\mathbf{S}\mathbf{K}_p\mathbf{x} \tag{4.62}$$

The term, $\dot{\hat{\mathbf{M}}} - 2\hat{\mathbf{V}}$, is a skew symmetric matrix, so when put into the quadratic form reduces to zero. Therefore the time rate of change of the proposed function candidate, $\dot{V}(\mathbf{x}, \dot{\mathbf{x}})$, can be reduced to the form shown in equation (4.63).

$$\dot{V}(\mathbf{x}, \dot{\mathbf{x}}) = -\dot{\mathbf{x}}^t \mathbf{K}_d \dot{\mathbf{x}} \tag{4.63}$$

The proposed Lyapunov function candidate is continuously differentiable. In addition in the domain, Ω, that contains the origin the function is positive definite therefore the function is a valid function candidate and can be used to investigate the stability of the nonlinear system. The Lyapunov function candidate, equation (4.59), is also radially unbounded and decrescent by inspection. These two additional properties will be used to strengthen the stability conclusions for this system.

The Lyapunov function candidate will be a Lyapunov function on the domain, Ω, if the expression in equation (4.63) is negative semi-definite. The damping gain matrix, $\mathbf{K}_d$, is positive definite by selection, therefore the time derivative of the candidate function is negative semi-definite. As a result the function in equation (4.59) is a Lyapunov function and it can be concluded that the modified virtual manipulator control law running on a general six degree of freedom robot will be stable in the sense of Lyapunov subject to the state assumptions. This however, is a rather weak stability conclusion, therefore La Salle's invariance theorem will be used to determine if a stronger stability conclusion can be made.

In order to use La Salle's theorem the set, E, as shown in equation (4.64) below must be defined.

$$E = \left\{\mathbf{x}, \dot{\mathbf{x}} \in \Omega : \dot{V}(\mathbf{x}, \dot{\mathbf{x}}) = 0\right\} = \{\dot{\mathbf{x}} = 0\} \tag{4.64}$$

Now if it can be shown that the only solution that can stay identically in the set, E, is the trivial solution, then the origin of the end effector space of the virtual manipulator will be asymptotically stable. In the set, E, the velocity of the robot is zero, $\dot{\mathbf{x}} = 0$, which implies that the acceleration of the robot is zero, $\ddot{\mathbf{x}} = 0$. Substituting these conditions into the dynamic system model shown in equation (4.58) results in the following expression.

$$\mathbf{SK}_p\mathbf{x} = 0 \tag{4.65}$$

Equation (4.65) is the same as equation (4.51) in the equilibrium point analysis section of this chapter. It was shown in the equilibrium point analysis section that when the robot interface position, $\mathbf{x}$, is contained in the domain, Ω, that equation (4.65) has only the trivial solution, zero. Therefore the largest invariant set, M, in the set, E, contains only the origin. This allows the conclusion that origin of the end effector coordinate system of the virtual manipulator is asymptotically stable. This fact implies that the end effector trajectory of the virtual manipulator is asymptotically stable when viewed in world coordinates.

The stability conclusion for the modified virtual manipulator control law running on a six degree of freedom robotic manipulator can be refined by examining the Lyapunov function shown in equation (4.59). The Lyapunov function is a function of the state variable, $\mathbf{x}$ and $\dot{\mathbf{x}}$, only, as a result the Lyapunov function, $V(\mathbf{x}, \dot{\mathbf{x}})$, is automatically decrescent. The fact that the Lyapunov function is decrescent allows the stability conclusion to be extended to uniformly asymptotically stable.

One final note on the "globalness" of this stability conclusion should be made. The Lyapunov function, $V(\mathbf{x}, \dot{\mathbf{x}})$, in equation (4.59) is radially unbounded for all robot interface positions, $\mathbf{x}$, and velocities, $\dot{\mathbf{x}}$, in the domain Ω. That is the norm of the Lyapunov function goes to infinite as the norm of the state variables goes to infinite regardless of the direction the state variables move. This concept is shown mathematically below.

$$\|\mathbf{x}, \dot{\mathbf{x}}\| \to \infty \Rightarrow \|V(\mathbf{x}, \dot{\mathbf{x}})\| \to \infty \tag{4.66}$$

The radially unbounded property of the Lyapunov function is required to allow any global stability conclusions to be made.

The domain, $\Omega = \Re^{2m-n}$, includes the entire state space and the rate of change of the Lyapunov function, $\dot{V}(\mathbf{x}, \dot{\mathbf{x}})$, is negative semi-definite over the entire domain, Ω. Therefore using the global corollary to La Salle's theorem it can be concluded that the origin of the end effector coordinate frame of the virtual manipulator is globally uniformly asymptotically stable. This implies that the continuum of equilibrium points defined by the end effector trajectory of the virtual manipulator are globally uniformly asymptotically stable, when the system is viewed in world coordinates. That is the continuum of equilibrium points is globally attractive for any set of initial conditions in the state space. However, nothing can be said about where the final position of the interface robot will be along this continuum of equilibrium points. It may be argued that La Salle's invariance theorem has no global form, however, a specialization of La Salle's theorem called the theorem of Barashin and Krasovskii, is applicable and does have a global form.

This stability analysis has shown that the modified virtual manipulator control law running on a general six degree of freedom robot has acceptable stability characteristics to be used as a haptic interaction metaphor. This proof has shown that the entire class of virtual manipulator control laws is stable subject to the assumptions made. In addition, the assumptions made were not restrictive and are listed below for inspection. First it was assumed that accurate gravity compensation for the robot is available. This assumption is common in most robot system stability analyses and is readily satisfied using analytic or experimental models of the gravitational force applied to the manipulator. The second assumption that was made was that the robot was operating in a portion of the configuration space that is free of singularities so that the joint space robot dynamics could be transformed into Cartesian space. This assumption restricts the range of motion of the robot but it is required for correct operation of the control law regardless of the proof of stability. Third it was assumed that the null space filter matrix existed. This poses no restriction because if the null space filter does not exist the control law can not be implemented. Finally, it was assumed that the null space filter matrix is a constant matrix in the end effector space of the virtual manipulator. This assumption has currently placed no restriction on the virtual manipulator implementations. However, it is conceded that this assumption could pose some restriction in future work.

Original Control Law

The stability of the original virtual manipulator control law will be addressed in this subsection. The proof for the modified control law was presented first because it illustrates the stability of the entire class of modified virtual manipulator control laws. The modified control law adds damping forces in the free directions of motion which is not desired, therefore the original virtual manipulator control law will be investigated. This proof of stability will not allow such a conclusion to be made at present. However, the conditions for stability of a specific virtual manipulator control law will be described so that each control law can be verified in a case by case manner.

The proof of stability for the original virtual manipulator control law will be demonstrated by showing that the error between the position of the end effector of the interface robot and the position of the end effector of the virtual manipulator is stable. Therefore the continuum of equilibrium points defined by the end effector trajectory of the virtual manipulator is stable. This type of error analysis is traditionally used in trajectory following applications and works well in this constraint enforce situation.

In order to begin the analysis the dynamics of the error must be determined. This will include the dynamics of the robot, equation (4.67) as well as the dynamics of the virtual manipulator.

$$\tilde{\mathbf{M}}\ddot{\mathbf{x}}_r + \hat{\mathbf{V}}\dot{\mathbf{x}}_r = \mathbf{F}_r \tag{4.67}$$

The subscript, *r*, denotes robot variables and the vector, $\mathbf{F}_r$, is the control force applied to the robot calculated using the Cartesian space proportional plus derivative control. The dynamics of the interface robot were introduced earlier in this chapter, however, there has been no discussion in this work about the dynamics of the virtual manipulator. This is because the virtual manipulator is used as a kinematic constraint not a dynamic constraint. If analysis of the motion of the virtual manipulator was performed a dynamic model could be generated for the device. However, the motion of the virtual manipulator is based on the motion of the interface robot, therefore it is natural to conclude that the dynamics of the virtual manipulator will have the same characteristics as the dynamics of the interface robot, equation (4.67). As a result the dynamic equations of motion for the virtual manipulator are shown in equation (4.68).

$$\hat{\mathbf{M}}\ddot{\mathbf{x}}_v + \hat{\mathbf{V}}\dot{\mathbf{x}}_v = \mathbf{F}_v \tag{4.68}$$

The dynamics of the error between the two devices can be determined by subtraction of equation (4.67) and (4.68).

$$\hat{\mathbf{M}}(\ddot{\mathbf{x}}_r - \ddot{\mathbf{x}}_v) + \hat{\mathbf{V}}(\dot{\mathbf{x}}_r - \dot{\mathbf{x}}_v) = \mathbf{F}_r - \mathbf{F}_v \tag{4.69}$$

The control force applied to the virtual manipulator, $\mathbf{F}_v$, is the same control force applied to the interface robot, $\mathbf{F}_r$, but the only component of this force that must be resisted by the virtual manipulator is the component that lies in the ranges space of the transpose of the virtual manipulators Jacobian. This component of the control force can be determined by multiplying the robot control force, $\mathbf{F}_r$, by the range space filter matrix, $\mathbf{R}$, as shown in equation (4.70).

$$\tilde{\mathbf{M}}(\ddot{\mathbf{x}}_r - \ddot{\mathbf{x}}_v) + \hat{\mathbf{V}}(\dot{\mathbf{x}}_r - \dot{\mathbf{x}}_v) = (\mathbf{I} - \mathbf{R})\mathbf{F}_r \tag{4.70}$$

The error in the system will be defined in the same way as in Chapter 3 so that the control force can be expressed as shown in equation (4.44), repeated here for convenience.

$$\mathbf{F}_r = \mathbf{K}_p\mathbf{e} + \mathbf{K}_v\dot{\mathbf{e}} \tag{4.71}$$

The error is defined in equation (4.72).

$$\mathbf{e} = \mathbf{x}_v - \mathbf{x}_r \tag{4.72}$$

Using equations (4.70), (4.71) and (4.72) the error dynamics for the system can be represented by the following system of differential equations.

$$\tilde{\mathbf{M}}\ddot{\mathbf{e}} + \hat{\mathbf{V}}\dot{\mathbf{e}} + \mathbf{S}\mathbf{K}_d\dot{\mathbf{e}} + \mathbf{S}\mathbf{K}_p\mathbf{e} = 0 \tag{4.73}$$

It is important to note that in the formulation of the error dynamics the coordinate frame with respect to which the position, velocities and accelerations are written is not critical. It is only required that all quantities be expressed in terms of the same coordinate frame.

The equilibrium conditions of the error dynamics are the same as the equilibrium conditions of the robot dynamic model discussed in the second section of this chapter. Therefore a separate equilibrium point analysis is not required for the error dynamics. To ensure that the only equilibrium points for the robot are those points where the error is equal to zero, n, elements in the error vector will be defined to be zero by appropriate selection of the n joint variables associated with the virtual manipulator. However, the process of defining an element of the error vector to be zero has a slightly different effect than defining an element in the robot position vector to be zero. Because an element in the error vector represents a relative displacement between the robot and the virtual manipulator when an element in the error vector is defined to be zero all of the time derivatives of that element are also zero. This allows the error state space to be reduce from $2m$ to $2(m-n)$ where as the state space of the robot dynamics could only be reduced to $2m-n$ in size.

Because of the reduction in the number of the states of the error dynamic system, the dynamic model can be expressed as shown below.

$$\hat{\mathbf{M}}'\ddot{\mathbf{e}}_n + \hat{\mathbf{V}}'\dot{\mathbf{e}}_n + (\mathbf{SK}_d)'\dot{\mathbf{e}}_n + (\mathbf{SK}_p)'\mathbf{e}_n = 0 \tag{4.74}$$

The subscript, *n*, simply indicates that the vector contains only elements not defined to be zero and the $(\ )'$ notation indicates that the dimension of the matrix has been reduced during the process of removing elements defined to be zero. The size reduction has not affected the skew symmetric property of the time rate of change of the mass matrix and the velocity matrix or the positive definite and symmetric properties of the mass matrix. In addition the reduction in the size of the product of the null space filter matrix and the position gain matrix has made this matrix a positive definite and symmetric matrix.

The only equilibrium point for the error dynamics is the origin, a moving point at the end of the virtual manipulator, which maps to an infinite continuum of equilibrium points for the end effector of the interface robot. Because the error dynamics have only one equilibrium a Lyapunov analysis similar to the one used in the proof of stability for the modified virtual manipulator control law will be used. The proposed Lyapunov function candidate is shown below.

$$V(\mathbf{e},\dot{\mathbf{e}}) = \tfrac{1}{2}\dot{\mathbf{e}}^t\hat{\mathbf{M}}'\dot{\mathbf{e}} + \tfrac{1}{2}\mathbf{e}^t\left(\mathbf{S}\mathbf{K}_p\right)'\mathbf{e} \tag{4.75}$$

This proposed Lyapunov function candidate is globally positive definite, decrescent, radially unbounded and is continuously differentiable. Therefore the function shown in equation (4.75) is a valid Lyapunov function candidate. To determine if this function is indeed a Lyapunov function the time rate of change of the function must be determined.

$$\dot{V}(\mathbf{e},\dot{\mathbf{e}}) = -\dot{\mathbf{e}}^{t}\left(\mathbf{SK}_{d}\right)^{r}\dot{\mathbf{e}} \tag{4.76}$$

However, there is no guarantee that the time rate of change of the Lyapunov function candidate is negative semi-definite. Therefore it cannot be concluded in general that the function candidate, equation (4.75), is a Lyapunov function.

For this reason the stability of the original virtual manipulator control law cannot be shown for the entire class of control laws. The conditions for stability of the equilibrium point can be determined in order to demonstrate the stability of the original virtual manipulator control law in a case by case manner. The time rate of change of the Lyapunov function candidate contains only the rate of change of the error vector. Therefore the rate of change of the function is at best negative semi-definite. In order for the rate of change of Lyapunov function candidate to be negative semi-definite, the reduced dimension matrix, $\left(\mathbf{SK}_{d}\right)^{r}$, must be positive definite.

If this condition is satisfied the function candidate is a Lyapunov function can be used to draw conclusions about the stability of the equilibrium point. In addition, because the equation (4.75) is a Lyapunov function it can be concluded that the equilibrium point is stable in the sense of Lyapunov. This conclusion can be strengthened by application of La Salle's theorem, in the same manner as in the first proof of stability presented in this chapter. The application of La Salle's theorem supports the conclusion that the equilibrium point is globally uniformly asymptotically stable without placing additional requirements on control law.

Therefore, if it can be shown that the reduced dimension matrix, $(\mathbf{SK}_d)'$, is positive definite, then the equilibrium point associated with the error dynamics will be globally uniformly asymptotically stable. The interpretation of this stability conclusion is similar to the interpretation of the first proof of stability. The end effector trajectory of the virtual manipulator is globally attractive regardless of the initial conditions of the interface robot. However, the proof of stability for the original virtual manipulator control law does not prevent the interface robot from having a nonzero velocity along the equilibrium trajectory due to reduction in the state space.

The two proofs of stability presented in this chapter have established the necessary conditions for the stability of the modified and original virtual manipulator control laws. However, in Chapter 6 experimental results of the time varying extension of the virtual manipulator concept will be presented. Time varying virtual manipulators are used to extend the potential of the control concept by allowing a virtual manipulator to represent more complex constraints. The inclusion of time varying components into the control law has not been considered in either of the two proofs presented here. Therefore the stability of time varying virtual manipulators has not been verified and remains as a area for future work.

In the beginning of this chapter the subject of safety was addressed in general terms. The proof of stability presented here is an important step in showing that the virtual manipulator control law running on a general six degree of freedom robot will be safe. However, other issues associated with the performance of the system in the presence of fault conditions still requires exploration. These issues will be discussed in the following chapter that describes the experimental hardware used as a test bed for this study.

CHAPTER 5. EXPERIMENTAL TESTBED

Now that the virtual manipulator control law has been developed and the equilibrium and stability characteristics of the control approach have been investigated. The next step is to implement the control law on an experimental test bed. The remainder of this chapter will describe the hardware used to demonstrate the virtual manipulator control law. Specifically this chapter will describe the robot manipulator, the control interface and computer, the force torque transducer and finally some safety issues associated with this system will be addressed.

Robot Manipulator

The robot manipulator used to implement and test the virtual manipulator control concept is a PUMA 560 manipulator, shown in Figure 5.1. This manipulator was selected based on it's availability for use during this study. It is important to note that any other six degree of freedom robot could have been used. However, a PUMA 560 is probably a good choice for this investigation. The PUMA 560 is a fairly common manipulator in most university robotics laboratories. Therefore this research is not isolated, it can be reproduced and evaluated in numerous other facilities around the world. As well as the fact that there is a large volume of previous PUMA 560 research that can be utilized. In addition, the size of a PUMA 560 makes it an appropriate interface for most people in a standing posture. Finally, the PUMA 560 has a workspace that is large enough to accommodate most haptic interactions that are of interest.

As was illustrated in the Chapter 3, which developed the virtual manipulator control approach a kinematic description of the PUMA 560 is needed for implementation. The PUMA 560 is a six degree of freedom robot composed solely of revolute joints. The coordinate frame assignments used in the kinematic analysis of the PUMA are shown in Figure 5.2. These frame assignments are commonly used in the kinematic analysis of the PUMA 560 although some variation due exist.

Using the coordinate the frame assignments shown in Figure 5.2, the structure of the Denavit-Hartenberg parameters for the PUMA 560 can be determined as shown in Table 5.1.

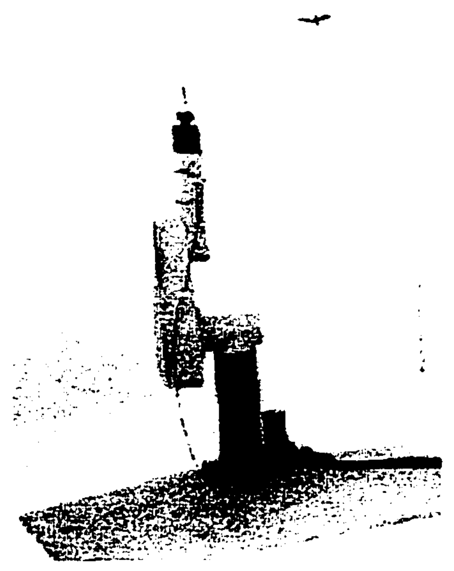


Figure 5.1. The PUMA 560 manipulator.

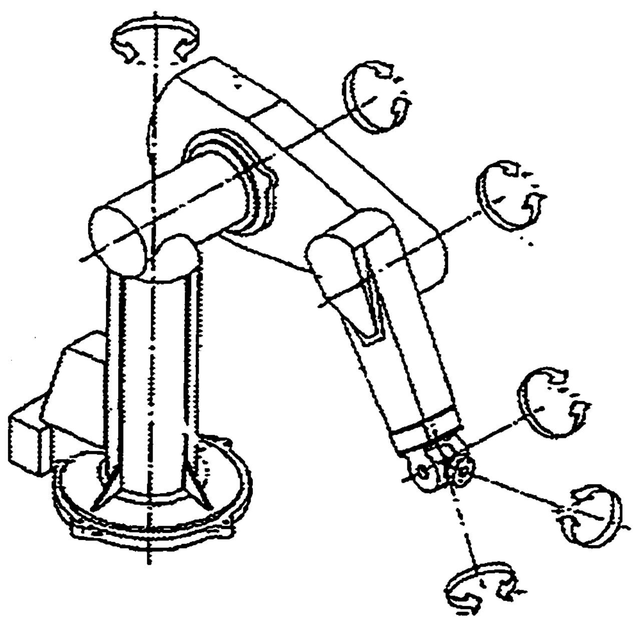


Figure 5.2. Coordinate frame assignments for the PUMA 560 manipulator.

In Table 5.1 the variables, θ_i, represent the joint variables for the robot. The variables, l_i, represent fixed offset lengths between the various revolute joints in the PUMA. The values for constants, l_i, can be obtained by direct measurement; however, this would require the disassembly of the robot. In addition the values of the constants, l_i, can also be obtained from numerous published sources [23]. Unlike some properties associated with the PUMA 560 robot, the published values for the kinematic constants are fairly uniform across the published sources. Therefore to avoid the difficulty of disassembly of the robot, the values of the kinematic constants reported in earlier work [98] will be used in this analysis. These values are presented in Table 5.2.

Using the Denavit - Hartenberg parameters shown in Table 5.1 and the transformation matrix associated with a generic set of Denavit - Hartenberg parameters, shown in equation

Table 5.1. Denavit - Hartenberg parameters for the PUMA 560 robot.

i	a_{i-1}	α_{i-1}	d_i	θ_i
1	0	0	0	θ_1
2	0	$-\frac{\pi}{2}$	0	θ_2
3	l_1	0	l_2	θ_3
4	l_3	$\frac{\pi}{2}$	l_4	θ_4
5	0	$-\frac{\pi}{2}$	0	θ_5
6	0	$\frac{\pi}{2}$	0	θ_6

(5.1) below, the transformation matrix between each coordinate frame on the PUMA 560 can be found.

$$ {}^{i-1}_{\ \ i}T = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) & 0 & a_{i-1} \\ \sin(\theta_i)\cos(\alpha_{i-1}) & \cos(\theta_i)\cos(\alpha_{i-1}) & -\sin(\alpha_{i-1}) & -d_i\sin(\alpha_{i-1}) \\ \sin(\theta_i)\sin(\alpha_{i-1}) & \cos(\theta_i)\sin(\alpha_{i-1}) & \cos(\alpha_{i-1}) & d_i\cos(\alpha_{i-1}) \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.1} $$

$$ {}^{0}_{1}T = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1) & 0 & 0 \\ \sin(\theta_1) & \cos(\theta_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.2} $$

Table 5.2. Values for kinematic constants for the PUMA 560 robot.

l_i	value (meters)
l_1	0.4318
l_2	0.1501
l_3	-0.0191
l_4	0.4311

$$ {}^{1}_{2}T = \begin{bmatrix} \cos(\theta_2) & -\sin(\theta_2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\sin(\theta_2) & -\cos(\theta_2) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.3} $$

$$ {}^{2}_{3}T = \begin{bmatrix} \cos(\theta_3) & -\sin(\theta_3) & 0 & l_1 \\ \sin(\theta_3) & \cos(\theta_3) & 0 & 0 \\ 0 & 0 & 1 & l_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.4} $$

$$ {}^{3}_{4}T = \begin{bmatrix} \cos(\theta_4) & -\sin(\theta_4) & 0 & l_3 \\ 0 & 0 & -1 & -l_4 \\ \sin(\theta_4) & \cos(\theta_4) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.5} $$

$$ {}^{4}_{5}T = \begin{bmatrix} \cos(\theta_5) & -\sin(\theta_5) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\sin(\theta_5) & -\cos(\theta_5) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.6} $$

$$ {}^{5}_{6}T = \begin{bmatrix} \cos(\theta_6) & -\sin(\theta_6) & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \sin(\theta_6) & \cos(\theta_6) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.7} $$

The sequence of transformation matrices shown in equation (5.2) - (5.7) represent a complete kinematic analysis of the PUMA 560 robot manipulator. These transformation matrices can be concatenated using standard matrix multiplication to obtain a transformation matrix that will transform a vector represented in the end effector coordinate frame of the PUMA (coordinate frame six) into a vector represented in the base or world coordinate frame space (coordinate frame zero) as shown in general in equation (5.8) and specifically in equation (5.9).

$$ {}^{0}_{6}\mathbf{T} = {}^{0}_{1}\mathbf{T}\,{}^{1}_{2}\mathbf{T}\,{}^{2}_{3}\mathbf{T}\,{}^{3}_{4}\mathbf{T}\,{}^{4}_{5}\mathbf{T}\,{}^{5}_{6}\mathbf{T} \tag{5.8}$$

$$ {}^{0}_{6}\mathbf{T} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & P_x \\ r_{21} & r_{22} & r_{23} & P_y \\ r_{31} & r_{32} & r_{33} & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.9}$$

The elements in the matrix shown in equation (5.9) are shown below.

$$ r_{11} = c_1\left\{c_{23}\left(c_4c_5c_6 - s_4s_6\right) - s_{23}s_5c_6\right\} - s_1\left\{s_4c_5c_6 + c_4s_6\right\} \tag{5.10}$$

$$ r_{21} = s_1\left\{c_{23}\left(c_4c_5c_6 - s_4s_6\right) - s_{23}s_5c_6\right\} + c_1\left\{s_4c_5c_6 + c_4s_6\right\} \tag{5.11}$$

$$ r_{31} = -s_{23}\left(c_4c_5c_6 - s_4s_6\right) - c_{23}s_5c_6 \tag{5.12}$$

$$ r_{12} = c_1\left\{-c_{23}\left(c_4c_5s_6 + s_4c_6\right) + s_{23}s_5s_6\right\} + s_1\left\{s_4c_5s_6 - c_4c_6\right\} \tag{5.13}$$

$$r_{22} = s_1\left\{-c_{23}\left(c_4c_5s_6 + s_4c_6\right) + s_{23}s_5s_6\right\} - c_1\left\{s_4c_5s_6 - c_4c_6\right\} \tag{5.14}$$

$$r_{32} = s_{23}\left(c_4c_5s_6 + s_4c_6\right) + c_{23}s_5s_6 \tag{5.15}$$

$$r_{13} = c_1\left(c_{23}c_4s_5 + s_{23}c_5\right) - s_1s_4s_5 \tag{5.16}$$

$$r_{23} = s_1\left(c_{23}c_4s_5 + s_{23}c_5\right) + c_1s_4s_5 \tag{5.17}$$

$$r_{33} = -s_{23}c_4s_5 + c_{23}c_5 \tag{5.18}$$

$$P_x = c_1\left(c_{23}l_3 + s_{23}l_4 + c_2l_1\right) - s_1l_2 \tag{5.19}$$

$$P_y = s_1\left(c_{23}l_3 + s_{23}l_4 + c_2l_1\right) + c_1l_2 \tag{5.20}$$

$$P_z = -s_{23}l_3 + c_{23}l_4 - s_2l_1 \tag{5.21}$$

In addition, to the forward kinematics of the PUMA the virtual manipulator control law also requires the Jacobian matrix for the manipulator. The Jacobian matrix for the PUMA can be found by performing a rate kinematic analysis of the mechanism. The expressions for the end effector velocity (linear and angular) contain all the information needed to generate the PUMA Jacobian. However, due to the length of the calculations in the rate kinematics analysis, only the resulting Jacobian will be presented.

$${}^6\mathbf{J} = \begin{bmatrix} {}^6\mathbf{J}^a & 0 \\ {}^6\mathbf{J}^b & {}^6\mathbf{J}^c \end{bmatrix} \tag{5.22}$$

The elements in the matrix shown in equation (5.22) are shown below.

$$
{}^{6}\mathbf{J}^{a}=\begin{bmatrix} j_{11} & j_{12} & c_4c_5c_6l_4-s_4s_6l_4+s_5c_6l_3 \\ j_{21} & j_{22} & -c_4c_5s_6l_4-s_4c_6l_4-s_5s_6l_3 \\ j_{31} & j_{32} & c_4s_5l_4+c_5l_3 \end{bmatrix} \tag{5.23}
$$

$$j_{11}=c_5c_6\left(-c_{23}c_4l_2+s_4\left(c_2l_1+c_{23}l_3+s_{23}l_4\right)\right)+s_6\left(c_{23}s_4l_2+c_4\left(c_2l_1+c_{23}l_3+s_{23}l_4\right)\right)+s_5c_6s_{23}l_2$$

$$j_{12}=c_5c_6\left(c_4\left(s_3l_1+l_4\right)\right)+s_6\left(-s_4\left(s_3l_1+l_4\right)\right)-s_5c_6\left(-c_3l_1-l_3\right)$$

$$j_{21}=-c_5s_6\left(-c_{23}c_4l_2+s_4\left(c_2l_1+c_{23}l_3+s_{23}l_4\right)\right)+c_6\left(c_{23}s_4l_2+c_4\left(c_2l_1+c_{23}l_3+s_{23}l_4\right)\right)-s_5s_6s_{23}l_2$$

$$j_{22}=-c_5s_6\left(c_4\left(s_3l_1+l_4\right)\right)+c_6\left(-s_4\left(s_3l_1+l_4\right)\right)+s_5s_6\left(-c_3l_1-l_3\right)$$

$$j_{31}=s_5\left(-c_{23}c_4l_2+s_4\left(c_2l_1+c_{23}l_3+s_{23}l_4\right)\right)-c_5s_{23}l_2$$

$$j_{32}=s_5\left(c_4\left(s_3l_1+l_4\right)\right)+c_5\left(-c_3l_1-l_3\right)$$

$$
{}^{6}\mathbf{J}^{b}=\begin{bmatrix} s_{23}\left(s_4s_6-c_4c_5c_6\right)-c_{23}s_5c_6 & c_4s_6+s_4c_5c_6 & c_4s_6+s_4c_5c_6 \\ s_{23}\left(s_4c_6+c_4c_5s_6\right)+c_{23}s_5s_6 & c_4c_6-s_4c_5s_6 & c_4c_6-s_4c_5s_6 \\ -s_{23}c_4s_5+c_{23}c_5 & s_4s_5 & s_4s_5 \end{bmatrix} \tag{5.24}
$$

$$
{}^{6}\mathbf{J}^{c}=\begin{bmatrix} -s_5c_6 & s_6 & 0 \\ s_5s_6 & c_6 & 0 \\ c_5 & 0 & 1 \end{bmatrix} \tag{5.25}
$$

The Jacobian shown above is represented in the end effector coordinate frame for the PUMA 560. However, this Jacobian can be transformed into any other coordinate frame by using a generalized rotation matrix as shown in equation (5.26) below.

$$ {}^{f}\mathbf{J} = \begin{bmatrix} {}^{f}_{6}\mathbf{R} & 0 \\ 0 & {}^{f}_{6}\mathbf{R} \end{bmatrix} \begin{bmatrix} {}^{6}\mathbf{J}^{a} & 0 \\ {}^{6}\mathbf{J}^{b} & {}^{6}\mathbf{J}^{c} \end{bmatrix} \tag{5.26} $$

The leading superscript f denotes the coordinate frame that the PUMA Jacobian has been transformed into by the generalized rotation matrix.

Control Hardware

Now that the robot manipulator has been introduced, the control hardware needed to operate the PUMA 560 will be described. When originally manufactured the PUMA 560 robot was controlled by a Unimation Val industrial computer. However, the Val computer does not provide the flexibility needed to implement the virtual manipulator control law. For this reason the Val computer used in the experimental work presented in the following chapter has been modified to provide joint level control by means of a personal computer.

To that end the majority of the control interface cards associated with the Val computer were removed and replaced with a TRC004 servo control card from Trident Robotics. The TRC004 servo control card is a general purpose card to control the operation of a robotic manipulator. It is equipped with optical encoder decoders and counters, analog inputs and analog outputs for input and output communication with the components of a robot.

In order to be used in the virtual manipulator control law the robot interface must have some instrumentation to measure the position and orientation of the end effector. This measurement instrumentation can be external device such as a magnetic tracker or an internal device such as an optical encoder. The PUMA 560 has two internal position sensors, the use of an external sensor has not been attempted but represents a viable alternative if necessary. The encoder decoder circuitry of the TRC004 card is connected to the two phase and index outputs of standard HP optical encoders located in the PUMA 560. These inputs tot he encoder circuitry determine the position of the various axes of the robot and stores this information in a set of digital counters.

In addition to the digital optical encoders as a mechanism for determining the position of the robot joints, the PUMA 560 is also equipped with potentiometer driven voltage dividers. The voltage across these analog circuits can be measured with the analog inputs of the TRC004 control card. Although analog potentiometers have been used classically as a measurement device for revolute joints, the potentiometers on the PUMA 560 do not have sufficient performance characteristics to be used in the virtual manipulator control application. Specially, the signal conditions in the potentiometer subsystem of the PUMA 560 is not sufficient to allow the position of the PUMA joints to be determined with any level of accuracy. This poor signal conditioning can most likely be attributed to the long unshielded cable which connects the Unimation, Val computer and the robot proper. Because of signal conditioning problem the digital encoders will be used to determine the position of the PUMA 560 in all control calculation.

Each axis of the PUMA 560 is also equipped with a DC servo motor to control the position of the various joints. The analog outputs of the TRC004 control card are used to apply control voltages to the DC servo motors of the robot. However, the analog outputs of the TRC004 control card are limited to plus and minus ten volts with a limited current output. Therefore the voltages from the analog outputs of the TRC004 are conditioned through the power amplifiers of the Val computer. The output voltages from the power amplifiers have a magnitude and current level capable of driving the servo motors of the PUMA.

The transmission of signals from the Val computer to the robot proper is controlled by the arm cable control card. The power amplifiers and the arm cable control card are the only original interface cards that are retained in the Val computer. The TRC004 servo control card reproduces the input - output operation of the Val computer but does not have a microprocessor to perform control calculations. Therefore, the TRC004 control card is coupled with a TRC006 interface card. The TRC006 interface card allows that input - output information maintained by the TRC004 to be accessed through the port memory of the personal computer in which the TRC006 is installed. This allows the personal computer to determine the position of the PUMA by accessing the encoder counters on the TRC004 through the TRC006 Using the position information the personal computer can perform all control calculations needed to obtain a set of voltages to apply to the servo motors of the robot. These voltages are then passed to the TRC004 for signal conditioning and application to the manipulator.

In order for the modified hardware, including the Val computer, TRC004 servo control card, TRC006 interface card and personal computer, to successfully control the PUMA 560 a control program must be written and executed on the personal computer to perform the control calculations as well as access the port memory of the computer. The first task, performing the control calculations, is easily achieved using any computer programming language. However, the second task, accessing the port memory of the computer, quickly reduces the number of acceptable programming languages. This work makes use of the C programming language. A program can be written, using C, to perform the control calculations as well as access the port memory of the computer to provide communication between the TRC006 interface card and the control program.

The C programming language provides several low level functions for performing input and output operations on the registers of the TRC006 interface card located in the port memory of the computer. However, the availability of these low level functions depends on the operating system on which the program is intended to run. For example a C program running in the DOS operating system has access to all interrupt vectors, all direct memory access channels and the entire port memory range. If a higher level operating system is used the C programming language has less if any access to these low level hardware communication channels. The Windows 95 16-bit operating system allows a C program to access the port memory range but blocks access to key interrupt vectors and direct memory access channels. But Windows 95 32-bit and Windows NT operating systems block all access to the low level C functions.

This discussion would seem to demand that a low level operating system be used for PUMA 560 control applications. However, 32-bit operating systems, Windows 95 and NT, offer advantages not available in other personal computer operating systems. First these operating systems allow access to a high performance timer, the processor oscillator. An accurate timer is required in the PUMA control applications to allow velocity calculations to be made. Second these 32-bit operating systems provide access to the standard socket functions. Socket functions are used for network communication to allow the control computer to transmit data to other computational or graphical support engines. To utilize these features of a 32-bit operating system, addition support software is needed to reproduce the low level functions associated with the C programming language that were removed by the operating system.

The most direct way of providing low level function access to the control hardware is by developing a specialized device driver for the TRC006 interface card. The creation of a custom device driver is a formidable task that has a large cost is both time and money. However, third party software, such as Driver X provided by Tetradyne, is able to provide low level interfacing with only minor expense. Although this type of software suffers from a loss in performance due to the generalized nature of the interface library, it still performs at a level acceptable to control the PUMA 560.

Force Transducer

The virtual manipulator control law developed in Chapter 3 used a Cartesian space proportional plus derivative error feedback technique for the force generation scheme. In addition this chapter also discussed the possibility of using measured end effector forces in the virtual manipulator control and concluded that this approach was not acceptable due to problems associated with the quality of the measured force signal. However, it is possible to used a low gain force control loop in conjunction with the virtual manipulator control law to provide compensation for the frictional and inertial effects of the interface robot. The addition of the force control loop is addressed in detail in the high degree of freedom virtual manipulator section of Chapter 6. The use of a force transducer in the control hardware for the virtual manipulator control law is not required but can refine the haptic interaction experience of the traveler in the synthetic environment.

As a result the interface robot, PUMA 560, has been equipped with an Assurance Technologies six axis force/torque transducer. The force transducer is mounted between the end effector of the robot and the interface handle that is manipulated by the traveler. This force transducer can measure forces up to 30 pounds and moments up to 100 inch pounds. In addition it is configured with mechanical stops to prevent damage to the strain gages in the transducer if forces or moments in excess of the rated limits are applied.

In order to minimize the noise characteristics of the force transducer, the signal from the strain gages are conditioned prior to leaving the force transducer proper for transmission to the force transducer interface box. The connection between the force transducer proper and the interface box is made with shielded cable to prevent contamination of the signals by external sources. The force transducer interface box then converts the analog strain gage signals into a consistent set of forces and moments, represented digitally, using calibration information. The digital representation of the six forces and moments are then sent to the personal computer controlling the PUMA 560 by means of a parallel interface card. The parallel interface card, like the TRC006, is located in the port memory of the personal computer. This requires that the Driver X software be used to allow the personal computer to receive and request force information.

Safety Considerations

Chapter 4 examined the equilibrium and stability characteristics of the virtual manipulator control law. This section will examine the fault tolerance and safety characteristics of the combined control law and hardware system. The discussion will begin with the interface robot, then the force transducer will be addressed and finish with comments on the control computer.

The presence of an operator in close proximity to a robot poses safety concerns. However, the nature of the virtual manipulator control law is well suited for this application. Virtual manipulator control is based on the idea of constraining the interface robot. So for low degree of freedom virtual manipulators the interface robot has only a limited range of motion and thus offers a greater level of safety for the traveler.

Even in situations where the virtual manipulator has a high number of degrees of freedom, the traveler carries a dead-man switch. This switch is hard wired to the Val computer and must be depressed in order for the robot to operate. Therefore, if at anytime during the interaction between the robot and the traveler, the traveler can immediately stop the robot by releasing the dead-man switch.

In addition due to the structure of the virtual manipulator control approach the interface robot does not move unless it is acted on by the traveler. Therefore if the traveler releases the interface robot during an interaction the motion of the robot will stop. The motion may not stop immediately due to the inertia of the mechanism and slight errors in the gravity compensation but this motion will decay. In this context the virtual manipulator control law offers real advantages over admittance based interaction control laws where the motion of the interface robot may accelerate even in the absence of contact with the traveler.

The virtual manipulator control law makes use of the Jacobian of the interface robot. In addition the proof of stability in the last chapter assumed that the robot was operating in a portion of the configuration space that is free of singularities. This is easy to ensure for low degree of freedom virtual manipulators, which are highly constrained. However, with high degree of freedom virtual manipulators it is possible to maneuver the PUMA into a singularity. Joint space impedance fields are used to prevent the robot from entering a singularity. These joint space fields limit the range of motion of the PUMA but are required to ensure stable controller operation. In addition joint space impedance fields are also used to provide joint limit protection for the robot. The joint limits of the PUMA are not protected by limit switches so the impedance fields prevent the robot or the traveler from damaging the robot by moving a joint past the mechanical limit.

The subject of mechanical limits is also important for the force transducer. If forces or moments in excess of the rated limits could damage the strain gages of the force transducer if mechanical limits are not present. Damages strain gages would lead to erroneous force measurements and possible unstable system performance. Therefore mechanical stops should be present in any force transducer used in haptic interaction.

The remaining safety considerations deal with fault detection. Both the TRC004 servo control card and the force transducer interface box have status registers to diagnose numerous fault conditions. These registers should be checked during each control cycle to ensure that no errors have been registered by the PUMA 560 or the force transducer.

The timing of the control cycles is critical in this type of application. The multiprocessing nature of the 32-bit Windows 95 / NT operating systems makes it difficult to ensure that each control cycle is a fixed time increment. However by increasing the priority of the control application it is possible to have the control cycle remain relatively constant. The slight variations in control cycle duration have caused no perceived problems but the stability of a digitally controlled system subject to varying control cycles is a complex problem. For this reason the variation in control cycle duration is recorded using the high performance system clock and this data is used to stop the operation of the robot if a control cycle becomes too long. The point at which the robot is stopped was determined heuristically by examining typical time histories of the control hardware. In addition by running the control application on a dual processor personal computer more accurate system timing can be achieved. Although multiprocessor personal computers do not offer the ability to specify which processes run on which processors, one the high priority control application has started on a processor no other lower priority process can interrupt it. These lower priority processes are simply routed to the free system processors.

Many of the safety considerations discussed above have been handled by incorporating feature into the control application. However, there is always the possibility that the control application or the computer running the application will stop functioning. This could happen for numerous reasons, most of which are not recognizable by the technician running the application or easily prevented by external hardware. This condition can however be diagnosed by a watchdog timer. The watchdog timer observes the communication between the control personal computer and the robot. If the control computer stops communicating with the robot, the operation of the robot is terminated. The time delay between the end of communication and the stopping of robot is longer than one control cycle of the robot but is only a fraction of a second. Therefore if the control computer hangs up for any reason the motion of the robot can be stopped before any erratic behavior is encountered by the traveler.

CHAPTER 6. EXPERIMENTAL RESULTS

The virtual manipulator control law developed in Chapter 3 will be demonstrated using the experimental test bed described in Chapter 5. The experimental results shown in this chapter do not represent the limit of what can be performed with virtual manipulators, they are intended to illustrate the control approach as well as reveal the potential of this concept. To that end the experiments presented here will show two basic virtual manipulator joints, revolute and prismatic, in various combinations to represent a virtual object for haptic interaction. In addition an extension to the virtual manipulator control technique, time varying virtual manipulators, with be developed in order to allow haptic interaction with complex objects such as free-form curves and surfaces.

The remainder of this chapter will be divided into four major sections. The first section will discuss a revolute virtual mechanism. The second section will show an example of a virtual mechanism constructed using prismatic joints. The third section will develop the time varying extension to the virtual manipulator concept and illustrate this extension by means of two examples. The fourth section of this chapter will develop a virtual manipulator with a varying number of degrees of freedom to allow general interaction with a virtual object.

Virtual Revolute Mechanism

This section will discuss a one degree of freedom virtual manipulator composed of a singe revolute joint. This virtual mechanism could be used to perform an ergonomic analysis of a gear shift mechanism in an automobile or to represent a flight control lever in a fight simulator. A schematic representation of this mechanism is shown in Figure 6.1. The position of the center of rotation, the orientation of the axis of rotation of this mechanism and the length of the lever arm can easily reconfigured in the control software. As a result this simple mechanism can be used to represent an infinite number of lever devices.

In order to implement the virtual manipulator control law for the one degree of freedom revolute mechanism, the forward kinematics as well as rate kinematics of the device are needed. The forward kinematics of the device are required in order to enforce the closed kinematic chain relationship that exists between the virtual manipulator and the interface

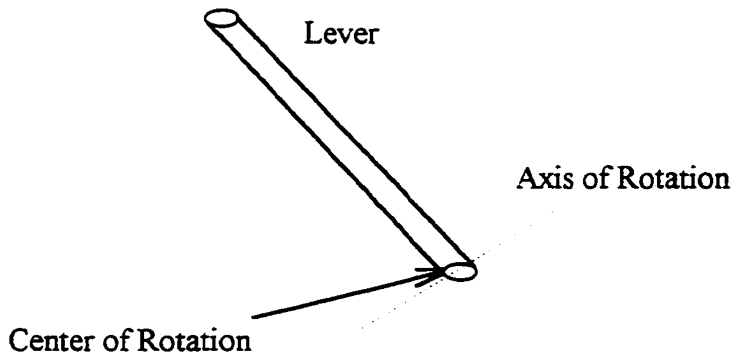


Figure 6.1. Schematic of revolute virtual mechanism.

robot. The rate kinematics of the device are required in order to determine the null space filter that is generated using the Jacobian matrix of the virtual manipulator. Both the forward kinematics and the rate kinematics of the virtual manipulator can be obtained using standard robotic analysis techniques.

The first step in this analysis is to assign coordinate frames for the mechanism. These coordinate frames will be assigned using the Denavit - Hartenberg coordinate frame assignment rules [24]. The coordinate frames used for the revolute virtual mechanism are shown in Figure 6.2, superimposed over the schematic representation of the device. The bracket notion used in Figure 6.2 is used to present the name (description) of a particular frame. The virtual manipulator has three frames associated with it, $\{0,1,2\}$, the fourth frame is the world coordinate system, $\{W\}$. Through out this work the world coordinate system for

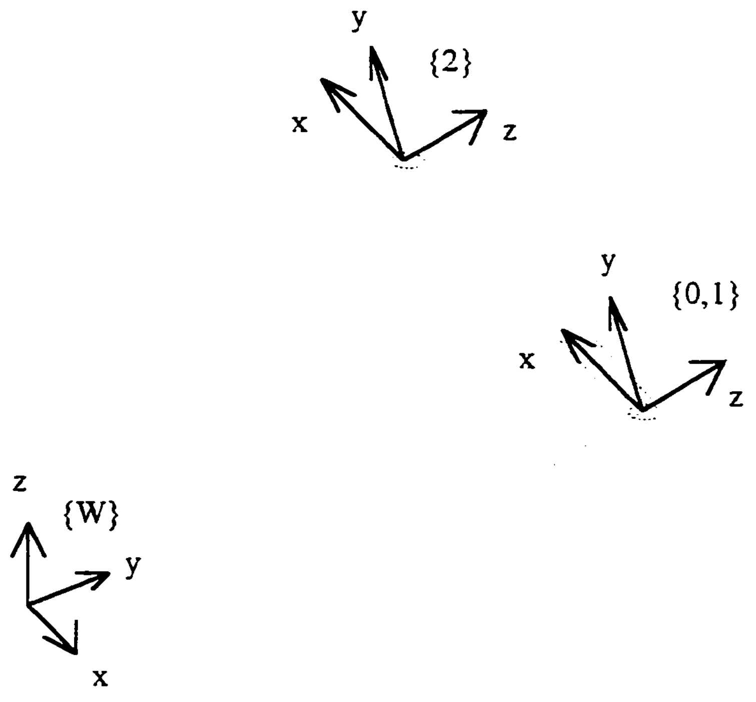


Figure 6.2. Coordinate frame assignments.

the virtual manipulator will be taken as the base coordinate system for the interface robot.

In most situations in this work the forward kinematics of the virtual manipulator will be derived independent of the position and orientation of the world coordinate system. That is Denavit - Hartenberg parameters for the coordinate frames {0, 1, 2} will be presented. These Denavit - Hartenberg parameters will allow the transformation matrix, ${}^{0}_{2}\mathbf{T}$, to be determined. The complete forward kinematic description of the virtual manipulator, ${}^{W}_{2}\mathbf{T}$, will be determined by concatenating the transformation matrix obtained from the virtual manipulators Denavit - Hartenberg parameter, ${}^{0}_{2}\mathbf{T}$, with a transformation matrix, ${}^{W}_{0}\mathbf{T}$, that describes the position of the virtual manipulators base coordinate frame with respect to the world coordinate system. The motivation for this approach is that the transformation matrix, ${}^{W}_{0}\mathbf{T}$, is used to position the center of rotation and set the axis of rotation for the virtual lever. Therefore by keeping this transformation separate from virtual manipulators transformation matrix it is easier to reconfigure the virtual manipulator to represent different types of lever mechanisms.

In addition to the general transformation frame from the world coordinate system to the base coordinate system of the virtual manipulator, some times a fourth coordinate frame will be added to the end of the virtual manipulator. The kinematic constraint being used is that the end effector coordinate frame of the interface robot will be in the same position and orientation as the end effector coordinate of the virtual manipulator. The end effector of the interface robot will be equipped with some type of handle to facilitate interaction. The fourth transformation matrix is used so that the interaction handle is oriented with respect to the virtual manipulator in an appropriate manner. As a result the fourth coordinate frame will be a constant rotation matrix which involves no translation.

The Denavit - Hartenberg parameter for the revolute virtual manipulator are shown in Table 6.1. Using the parameters from Table 6.1, the transformation matrix, ${}^{0}_{2}\mathbf{T}$, shown equation (6.1) and the Jacobian matrix, ${}^{2}\mathbf{J}_{v}$, shown in equation (6.2) can be found.

$$
{}^{0}_{2}\mathbf{T} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & L \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.1}
$$

$$
{}^{2}\mathbf{J}_{v} = \begin{bmatrix} 0 & L & 0 & 0 & 0 & 1 \end{bmatrix}^{t} \tag{6.2}
$$

The null space filter in the end effector coordinate frame of the virtual manipulator, $\mathbf{S}$, can be evaluated using the Jacobian matrix in equation (6.2).

Now that the forward kinematics of the virtual manipulator and the null space filter have been determined the virtual manipulator control, constraining the end effector of the PUMA to follow the circular arc trajectory of the virtual manipulator, can be implemented. It is important to note that heuristic tuning of the position and damping gain matrices is required

Table 6.1. Denavit - Hartenberg parameter for revolute mechanism.

i	a_{i-1}	α_{i-1}	d_i	θ_i
1	0	0	0	θ
2	L	0	0	0

to obtain proper performance for the system.

In addition there are two ways to formulate the closed kinematic constraint equations for this virtual manipulator. The first is based on the position of the interface robot, selecting the angle of the virtual manipulator by requiring that the "y" position of the robot, as shown in Figure 6.2, is zero. The second is based on the orientation of the interface robot, selecting the angle of the virtual manipulator by projecting the orientation of the robot into the plane of rotation of the virtual manipulator. Experimental results are presented for both cases and stable performance was observed for both. Experimental data for two one degree of freedom revolute virtual manipulators will be presented. In first set of data the virtual manipulator is positioned so that only the second, third and fifth axes of the PUMA are actuated.

In Figure 6.3 the end effector trajectory of the PUMA is compared with the end effector trajectory of the virtual manipulator. Due to the positioning of the virtual manipulator the motion of the robot is planar so the planar view of the data in Figure 6.3 is sufficient to capture the response of the interface robot. The data in Figure 6.3 was formulated with the positioned based closed kinematic constraint. Figure 6.4 examines the motion of the robot in Cartesian space to verify that the virtual manipulator control law enforces the constraints desired by the virtual manipulator control approach. Figures 6.5 and 6.6 reproduce the data in Figures 6.3 and 6.4, respectively. However, Figures 6.5 and 6.6 used the orientation based closed kinematic chain constraint formulation. Comparison of these four figures illustrates that regardless of the selection of the closed kinematic chain constraint equation, proper constraint enforcement is achieved.

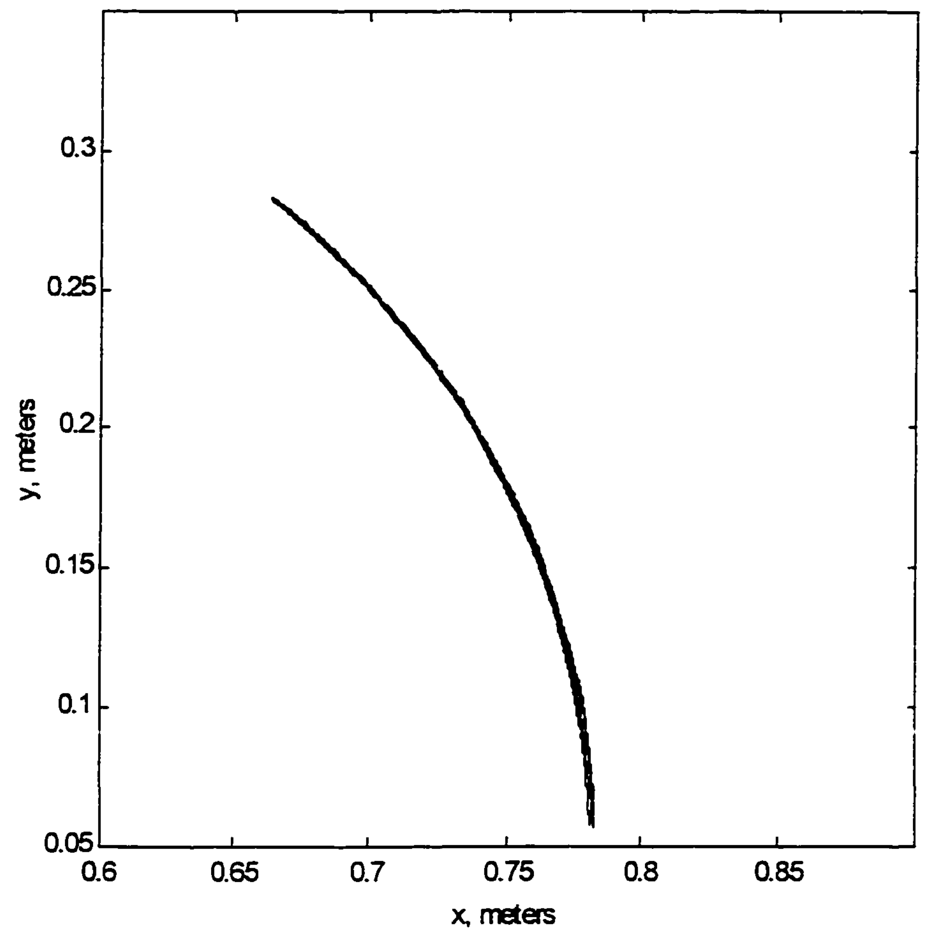


Figure 6.3. End effector trajectory, position constraint.

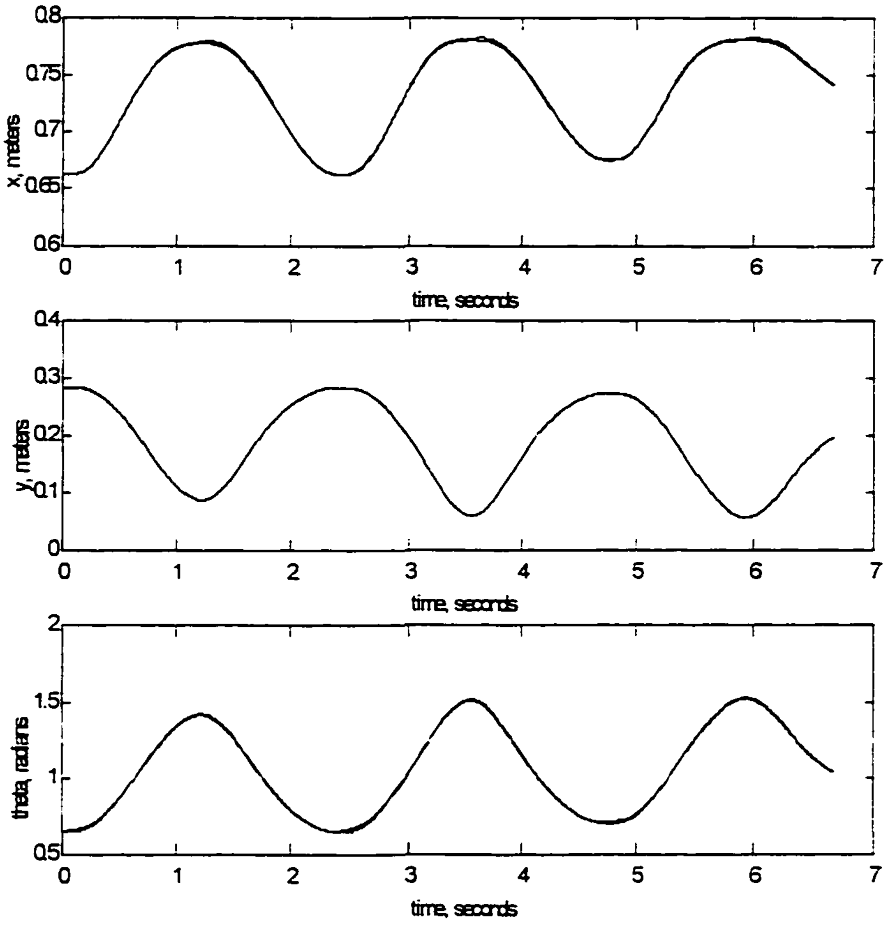


Figure 6.4. Cartesian motion, position constraint.

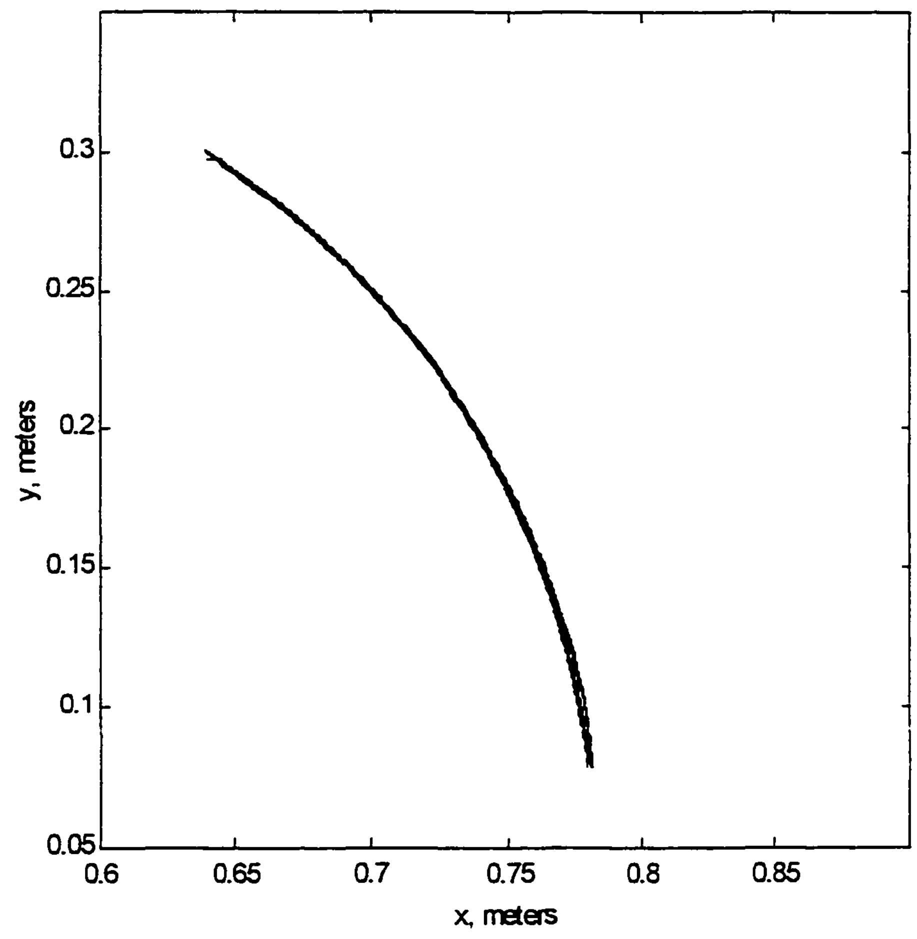


Figure 6.5. End effector trajectory, orientation constraint.

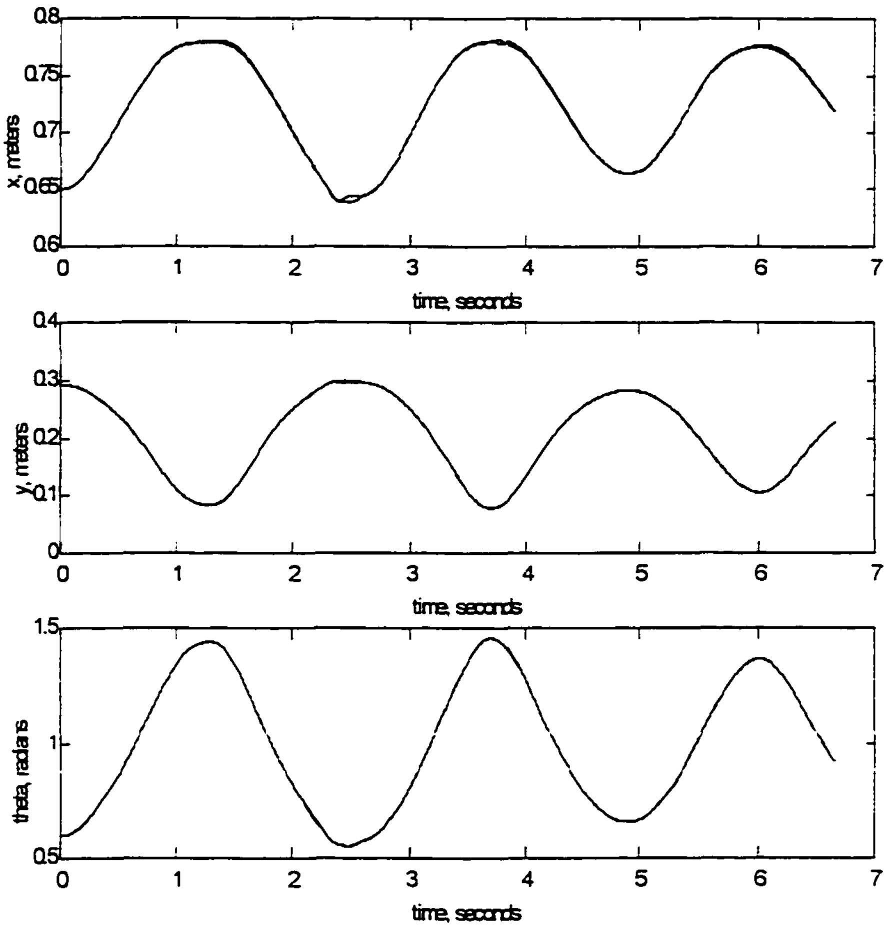


Figure 6.6. Cartesian motion, orientation constraint.

Both Figures 6.3 and 6.5 exhibit data that pulls away from the desired trajectory during the downward motion of the virtual manipulator. A close comparison of sections of Figure 6.3 and 6.5 is shown in Figure 6.7. The direction of motion of these loop effects is indicated in Figure 6.7. This loop effect is present in both Figures 6.3 and 6.5 therefore this artifact can not be associated with the selection of the closed kinematic chain constraint equations. In addition by comparing figures 6.4 and 6.6 it can be seen that the speed of the traveler's input motion is approximately equal in both cases and that the size of the loops are approximately equal. This supports the hypothesis that the loop effect is associated with the inertia of the interface robot which has not been compensated for in the virtual manipulator control law.

In order to investigate the effects of the inertia of the robot on the response of the virtual manipulator system a third set of was collected using the orientation based closed kinematic chain constraint equation. In this data, shown in Figures 6.8 and 6.9, the input speed of the traveler was approximately one third of the original data. The reduced input velocity is clearly seen in Figure 6.9. Figure 6.8 shows that the loop effect is still present but the magnitude is reduced. Therefore the inertia of the interface will affect the response of the virtual manipulator system. The inertia of the robot can be compensated for, but this requires knowledge of the mass matrix of the PUMA as well as calculation or measurement of the acceleration of the robot. Both of these quantities are difficult to obtain, with any level of accuracy, using the current configuration of the experimental hardware. Without inertial compensation, the experience of the traveler interacting with the PUMA will contain disturbance forces associated with the inertial properties of the robot. These inertial

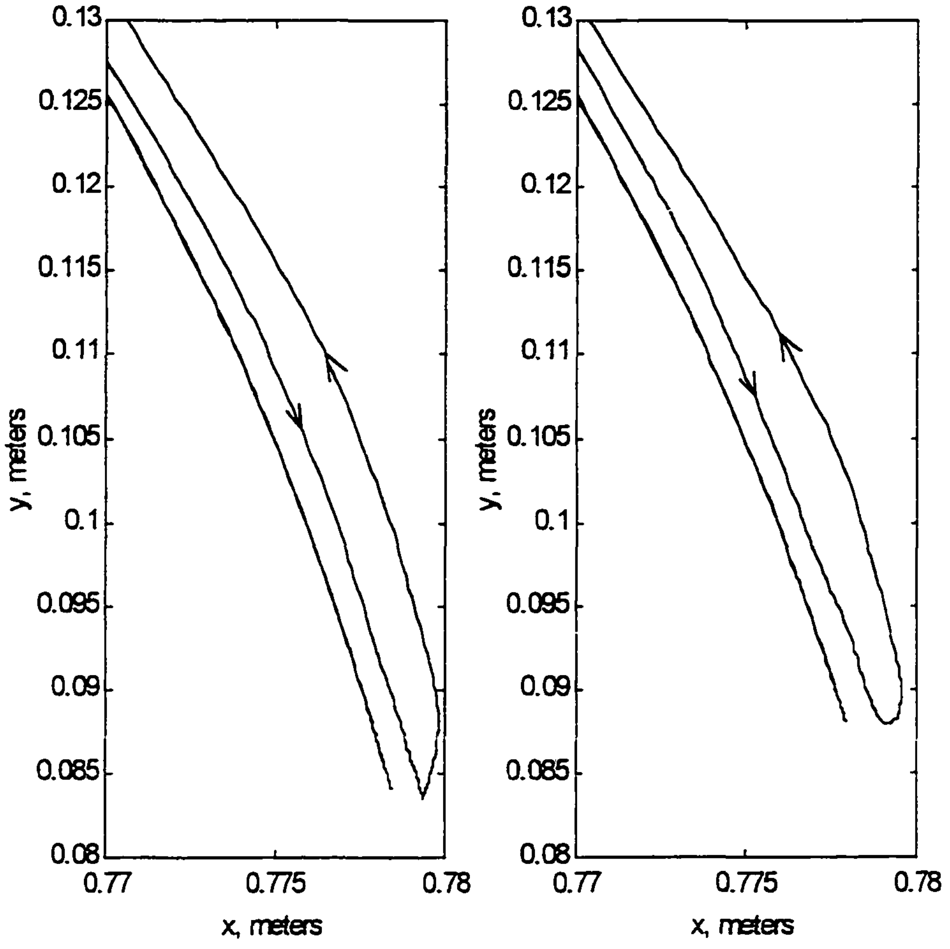


Figure 6.7. Comparison of loop effects.

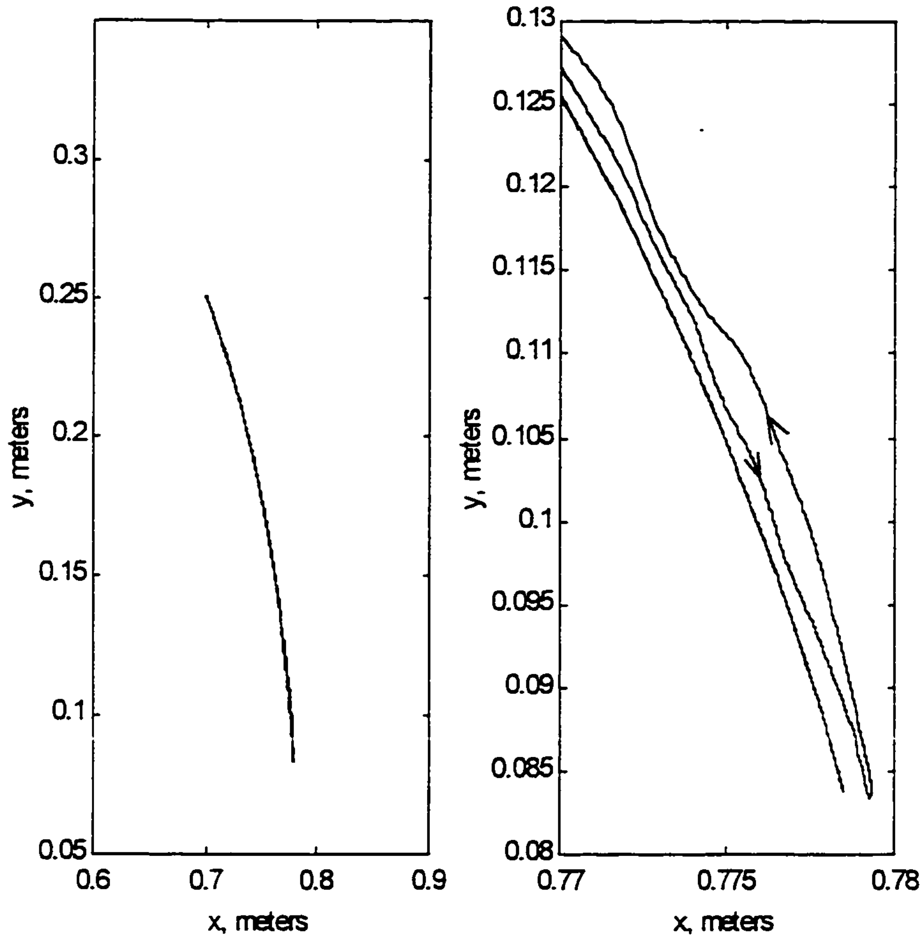


Figure 6.8. End effector trajectory, orientation constraint, slow motion.

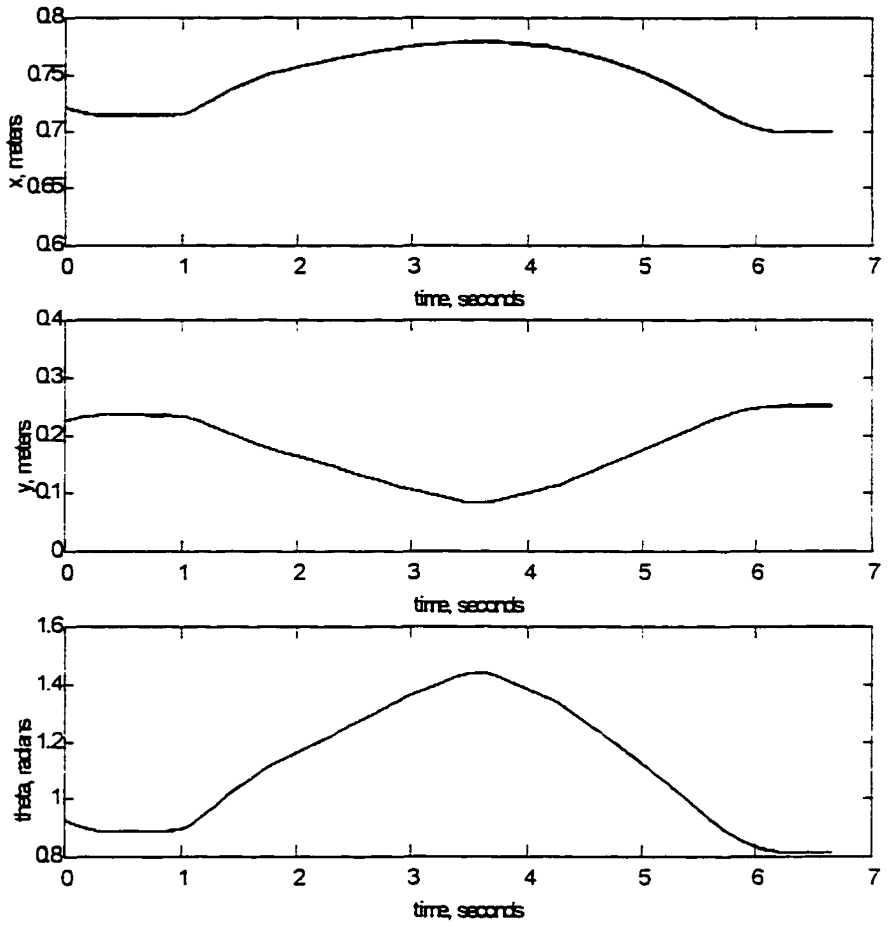


Figure 6.9. Cartesian motion, orientation constraint, slow motion.

disturbance forces may not affect the sense of immersion experienced by the traveler much. The interaction of the traveler with the haptic interface usually involves slow motion of the robot. However, if these disturbance forces become detrimental in the simulation of the synthetic environment a compensation mechanism can be developed and included into the control in a similar fashion to the inclusion of the gravity compensation.

In the second set of revolute virtual manipulator data, the one degree of freedom device is positioned so that motion of all six axes of the PUMA is required for proper constraint enforcement. The configuration of the virtual manipulator was in a horizontal plane and the axis of rotation was not collinear with the axis of rotation of the first revolute joint of the PUMA. Figure 6.10 shows a picture of the experimental hardware with the constraint trajectory superimposed. Figure 6.11 shows the desired trajectory and experimental data, viewed from the same point of view as the image shown in Figure 6.10.

The revolute joint is one of the two basic joints used to develop a virtual manipulator. The data presentation in this section has shown how a revolute constraint can be introduced into a synthetic environment using a one degree of freedom revolute manipulator. This type of virtual manipulator could be used to simulate interaction with a parking brake or gear shift lever in a vehicle or aircraft. Thus allowing evaluation of a system design without construction of a physical prototype. The second principle mechanism joint the prismatic joint will be examined in the next section.

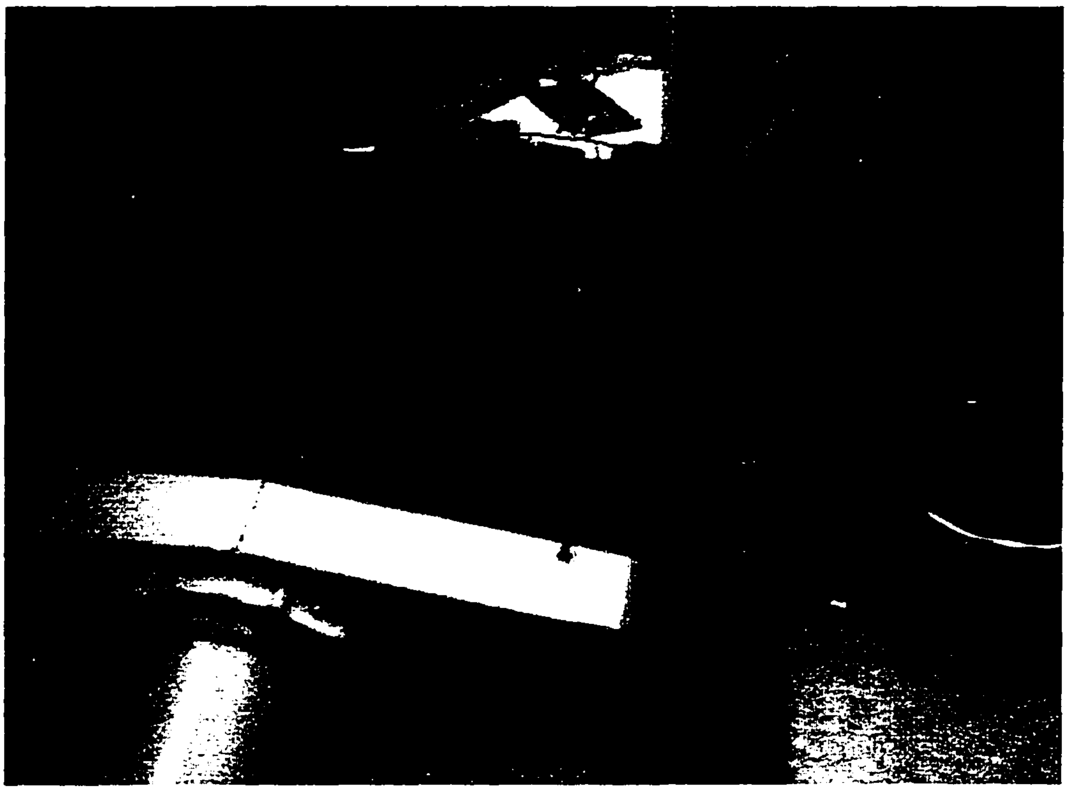


Figure 6.1. Experimental hardware with revolute constraint.

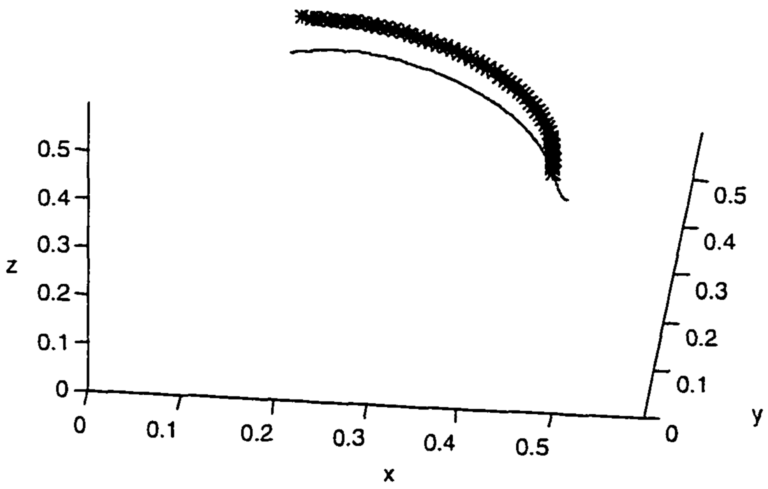


Figure 6.2. Experimental and desired trajectory.

Virtual Prismatic Mechanism

This section will present data for a virtual manipulator that contains both prismatic and revolute joints. In addition it will also introduce the first type of time varying virtual manipulator. Specifically the base mechanism used in this section will have six degrees of freedom. This will allow free motion of the interface robot, however the number of degrees of freedom will be reduced in order to simulate various point contact situations. The idea is that this virtual manipulator will allow free motion inside a cube boundary. So the traveler is free to explore inside the box but if a boundary is contacted resistance will be encountered preventing the traveler from moving outside the desired region. However, the traveler is free to orient the end effector of the interface robot in any direction.

The base virtual manipulator has six degrees of freedom and is shown schematically in Figure 6.12. The closed kinematic chain relationship is shown in Figure 6.13 using a

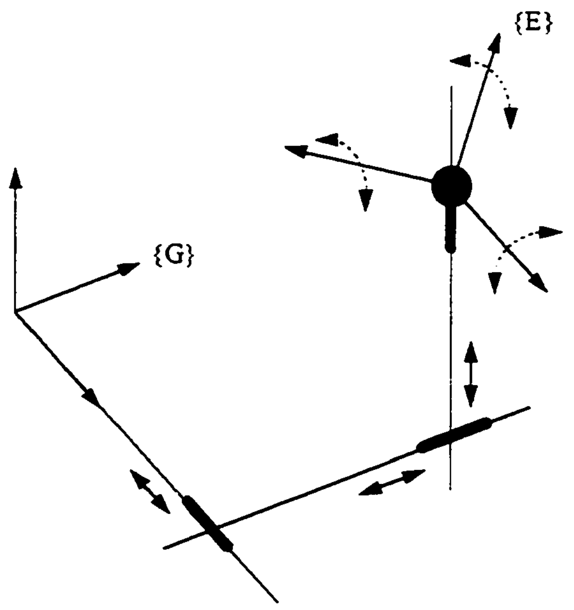


Figure 6.12. Virtual prismatic manipulator.

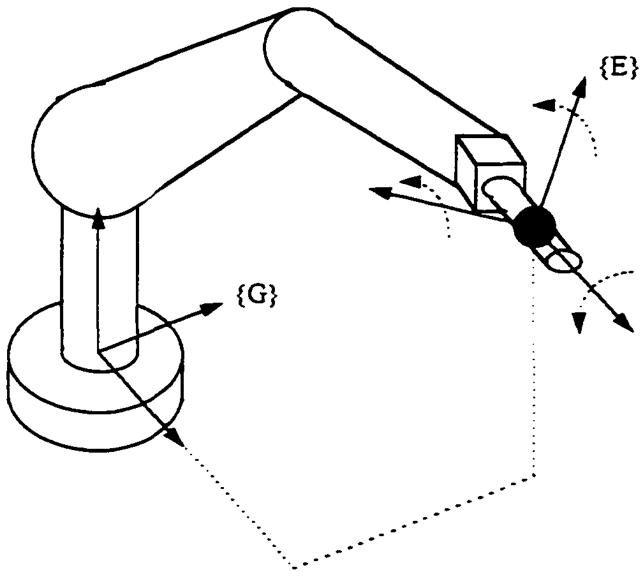


Figure 6.13. Closed kinematic chain relationship.

schematic representation of the virtual manipulator and interface robot. The forward kinematics of this base virtual manipulator are trivial to evaluate due to orthogonal prismatic joints and the decoupled nature of the end effector rotation. The Cartesian position of the end effector of the virtual manipulator is just the displacement of each of the prismatic joints and the XYZ fixed Euler angles relative tot he world coordinate system. The end effector Jacobian of the virtual manipulator is shown in equation (6.3) below.

$$
{}^{E}J_{V} = \begin{bmatrix} c_x c_y & c_x s_y s_z - s_x c_z & c_x s_y c_z + s_x s_z & 0 & 0 & 0 \\ s_x c_y & s_x s_y s_z + c_x c_z & s_x s_y c_z - c_x s_z & 0 & 0 & 0 \\ s_y & -c_y s_z & -c_y c_z & 0 & 0 & 0 \\ 0 & 0 & 0 & c_x c_y & -s_x & 0 \\ 0 & 0 & 0 & s_x c_y & c_x & 0 \\ 0 & 0 & 0 & s_y & 0 & -1 \end{bmatrix} \qquad (6.3)
$$

During contact situations the number of degrees of freedom of the virtual manipulator is reduced by introducing the idea of joint limits into the virtual manipulator concept. Due to the decoupled nature of the base manipulator the appropriate Jacobian for contact situations can be determined by removing the column of the base Jacobian associated with the axis that has reached the joint limit. There are essentially four contact situations which are shown in Figure 6.14. Part a) of Figure 6.14 shows free motion with six degrees of freedom, part b) shows plane contact with five degrees of freedom, part c) shows line contact with four degrees of freedom and part d) shows point contact with three degrees of freedom. The three

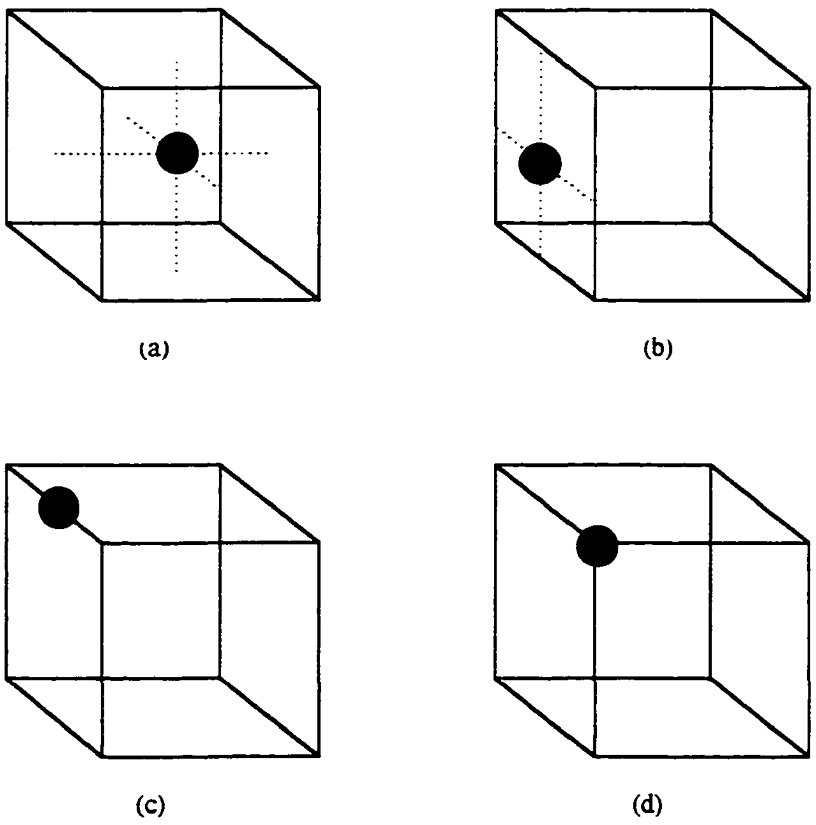


Figure 6.14. Motion constraints: a) six dof, b) five dof, c) four dof, d) three dof.

degrees of freedom during point contact are the rotational degrees of freedom because the orientation of the end effector has not been constrained.

A picture of the experimental hardware with the constraint boundary superimposed is shown in Figure 6.15. A set of experimental data is shown in Figure 6.16 in which the traveler was tracing the boundaries of the constraint box. This time varying virtual manipulator was developed to allow exploration of a synthetic environment using a point. Although the synthetic environment was quite simple, only one box, other objects could be added to the environment by modifying the joint limits imposed on the virtual manipulator. The next section will introduce another type of time varying virtual manipulators. This second class of time varying virtual manipulators will have a constant number of degrees of freedom

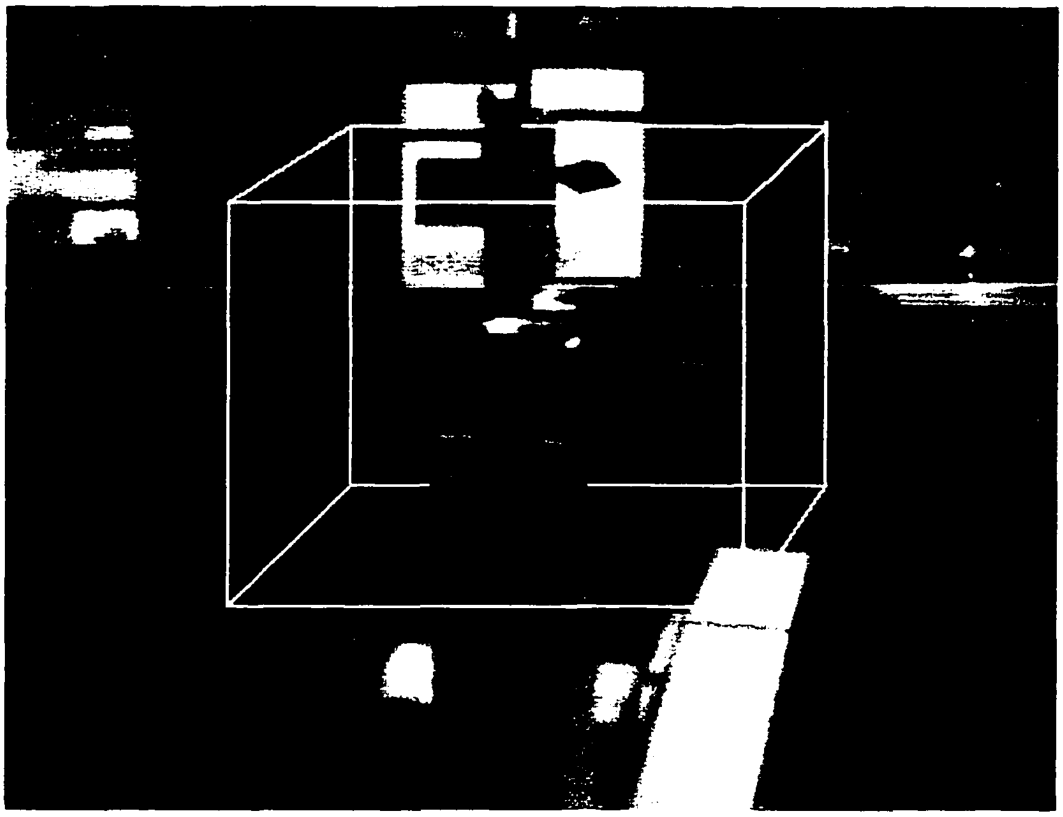


Figure 6.1. Experimental hardware with prismatic constraint.

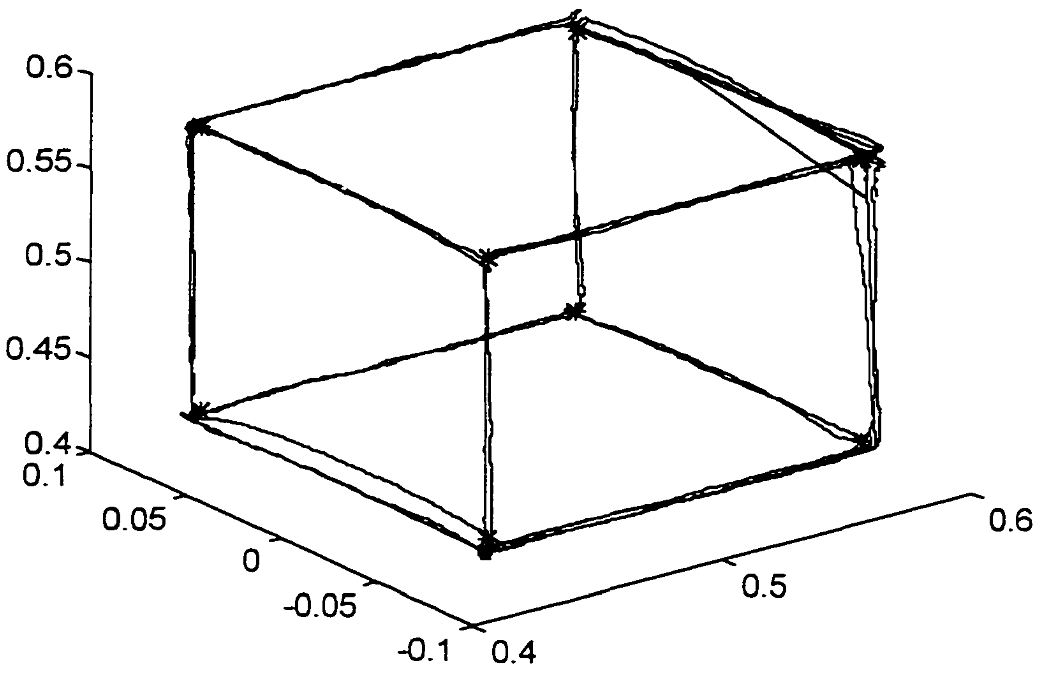


Figure 6.16. Experimental data tracing boundary.

but the configuration of the virtual manipulator will change during the course of the interaction.

Time Varying Virtual Manipulators

The last section introduced the use of joint limits to extend the virtual manipulator concept to represent various types of contact. This section will now introduce the use of a virtual manipulator with a time varying configuration to allow interaction with more complicated objects in a synthetic environment. This section will present two virtual manipulators to allow the exploration of the shape of a NURBS curve and surface.

NURBS Curve Virtual Manipulator

This development will assume that all weights in the NURBS curve are equal to one. This restriction is only added to make the presentation of this technique more tractable. Removing this restriction only increases the mathematical complexity of the equations, it does not change any of the results.

A NURBS curve, $\mathbf{C}(u)$, with all weights equal to one is defined by equation (6.4).

$$\mathbf{C}(u) = \sum_{i=0}^{N} B_{i,n}(u)\mathbf{P}_i \tag{6.4}$$

u -parameterization variable
N - number of control points
n - degree of NURBS curve
$B_{i,n}$ - NURBS basis functions
$\mathbf{P}_i$ - NURBS control points

The rate of change of the NURBS curve defined in equation (6.4) with respect to the parameterization variable, u, is shown in equation (6.5).

$$\frac{d\mathbf{C}(u)}{du} = \sum_{i=0}^{N} \frac{dB_{i,n}(u)}{du}\mathbf{P}_i \tag{6.5}$$

The tangent of the NURBS curve defined in equation (6.4) is expressed in equation (6.6).

$$\mathbf{T}(u) = \frac{d\mathbf{C}(u)}{du} \tag{6.6}$$

Equations (6.4), (6.5) and (6.6) present all the information about a NURBS curve that is needed to construct a time varying virtual manipulator that will constrain the motion of a robot to follow the curve. During a simulation in which the operator is allowed to move the end effector of a robot along a NURBS curve, the following list shows the steps performed by the virtual manipulator constraint controller.

- Perform point inversion to determine the closest point on the NURBS curve, $C(u)$, to the end effector of the robot
- Determine the configuration of the time varying virtual manipulator (this includes determining the link lengths, positions and orientations)
- Evaluate the Jacobian of the virtual manipulator
- Evaluate control law

These steps will be described for one type of virtual manipulator, a one degree of freedom prismatic manipulator.

In order to allow the end effector of a robot to trace a NURBS curve there must be way of transforming the Cartesian coordinates of the end effector (x, y, z) into the parametric coordinate of the curve, u. This is the point inversion problem and it will be discussed briefly before describing the time varying virtual manipulator. Point inversion is the process of determining the parametric coordinates associated with a given set of Cartesian coordinates

[82]. In theory the point inversion problem can be solved in closed form for curves with degree less than four. However, due to a host of numerical complications point inversion is typically performed iteratively using techniques such as a Newton search. A modified Newton search is used for point inversion in this work. After point inversion is performed the parametric coordinate u^* is determined; which represents the point on the NURBS curve that is closest to the end effector of the robot. The next step is to determine the configuration of the virtual manipulator that will approximate the shape of the NURBS curve in a region around u^*.

To approximate a NURBS curve the one degree of freedom mechanism shown schematically in Figure 6.17 will be used. This manipulator can be represented with two coordinate frames as shown in Figure 6.18. As shown in Figure 6.18 the end effector frame, {E}, of the virtual manipulator is allowed to translate along the z-axis of the base frame, {B}. A kinematic analysis of the virtual manipulator is required to formulate the virtual constraint controller. In order to determine the kinematics of this manipulator the Denavit-Hartenberg (D-H) parameters associated with it must be determined. The D-H parameters for the prismatic virtual manipulator are shown below in Table 6.2. The D-H parameters in Table 6.2 are used to construct the following kinematic transformation matrix for the virtual manipulator.

Table 6.2. D-H parameters.

i	a_{i-1}	α_{i-1}	d_i	θ_i
1	0	0	d	0

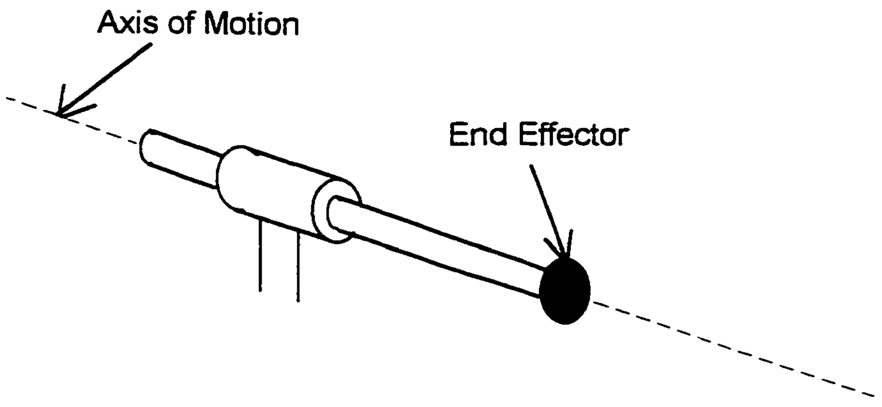


Figure 6.17. Schematic of prismatic virtual mechanism.

$$ {}^{B}_{E}\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.7} $$

This virtual manipulator will be used construct a linear approximation to the original NURBS curve. The position and orientation of the base frame, $\{B\}$ of the virtual manipulator are allowed to vary over the length of the curve. The instantaneous position and orientation of frame $\{B\}$ are determined by the characteristics of the NURBS curve at the parameter value u^* which was determined by point inversion. Specifically, the position of frame $\{B\}$ is defined by $\mathbf{C}(u^*)$. The orientation of the frame $\{B\}$ is partially determined by the tangent of the NURBS curve, $\mathbf{T}(u^*)$. Frame $\{B\}$ is oriented such that the z-axis of frame $\{B\}$ is in the direction of the $\mathbf{T}(u^*)$. However, a second vector, $\mathbf{V}_{up}$, is required to completely fix the orientation of frame $\{B\}$. The vector $\mathbf{V}_{up}$ is used to determine the orientation of frame $\{B\}$ in

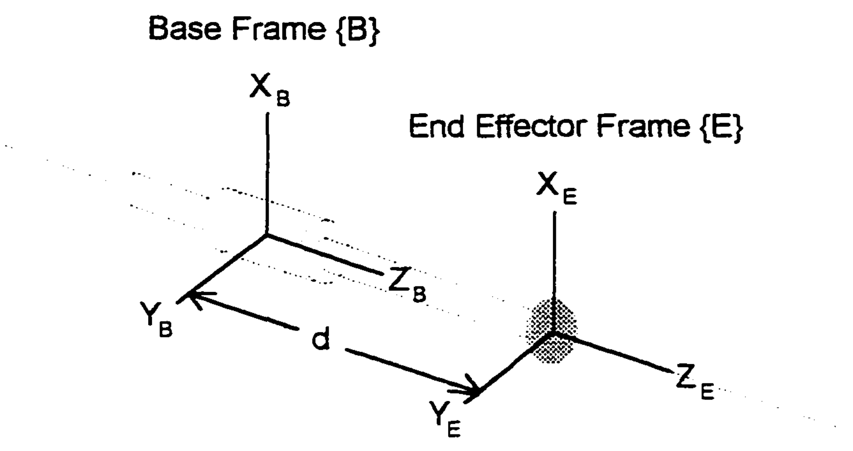


Figure 6.18. Virtual mechanism frame assignment.

same way that it is commonly used in computer graphics to determine the orientation of view reference coordinate frame [2].

$$\begin{aligned}
\mathbf{z}_B &= \frac{\mathbf{T}(u^*)}{\left|\mathbf{T}(u^*)\right|} \\
\mathbf{y}_B &= \frac{\mathbf{z}_B \times \mathbf{V}_{up}}{\left|\mathbf{z}_B \times \mathbf{V}_{up}\right|} \\
\mathbf{x}_B &= \frac{\mathbf{y}_B \times \mathbf{z}_B}{\left|\mathbf{y}_B \times \mathbf{z}_B\right|}
\end{aligned} \tag{6.8}$$

The vector $\mathbf{V}_{up}$ is arbitrary and can be select by the user. In addition, $\mathbf{V}_{up}$ can be constant or defined as a vector field over the length of the NURBS curve. After determining the configuration of the virtual manipulator its Jacobian must be found.

The Jacobian of the virtual manipulator can be determine by applying standard robot analysis techniques to the kinematic transformation matrix ${}^{B}_{E}\mathbf{T}$. The Jacobian of this one degree of freedom prismatic manipulator is shown in equation (6.9)

$$ {}^{E}\mathbf{J}_{V} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}^{T} \tag{6.9} $$

Note that the Jacobian of this virtual manipulator is quite simple and remains constant, which is a distinct advantage of this control approach.

In order to verify the effectiveness of the virtual constraint NURBS curve controller an

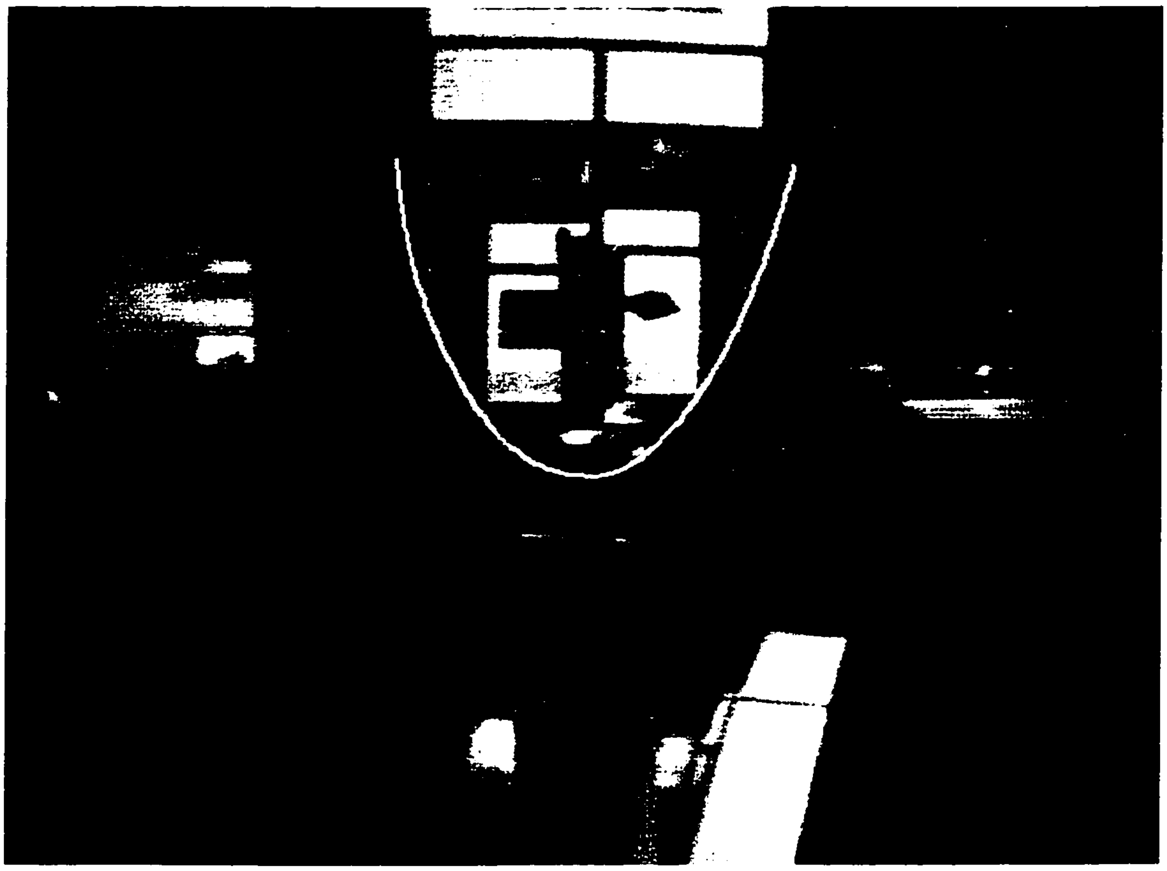


Figure 6.1. PUMA 560 manipulator with constraint NURBS curve.

experiment was performed in which a PUMA 560 manipulator was constrained to follow a quadratic NURBS curve defined by three control points. Figure 6.19 shows a picture of the PUMA with the NURBS curve constraint superimposed. Experimental data is presented for two simulations of the controller. Figure 6.20 shows experimental position data when the user is moving the end effector of the PUMA along the NURBS curve slowly. Figure 6.21 shows experimental position data when the user moves the robot more quickly.

The data presented in Figure 6.20 clearly shows that as the operator moves the PUMA along the NURBS curve slowly, good tracking performance is obtained. However, as seen in Figure 6.21 when operator moves the robot quickly, the tracking performance degrades. When the user moves the PUMA slowly the effect of the inertial properties of the robot are small thus good tracking is obtained. However, as the speed of the robot is increased the

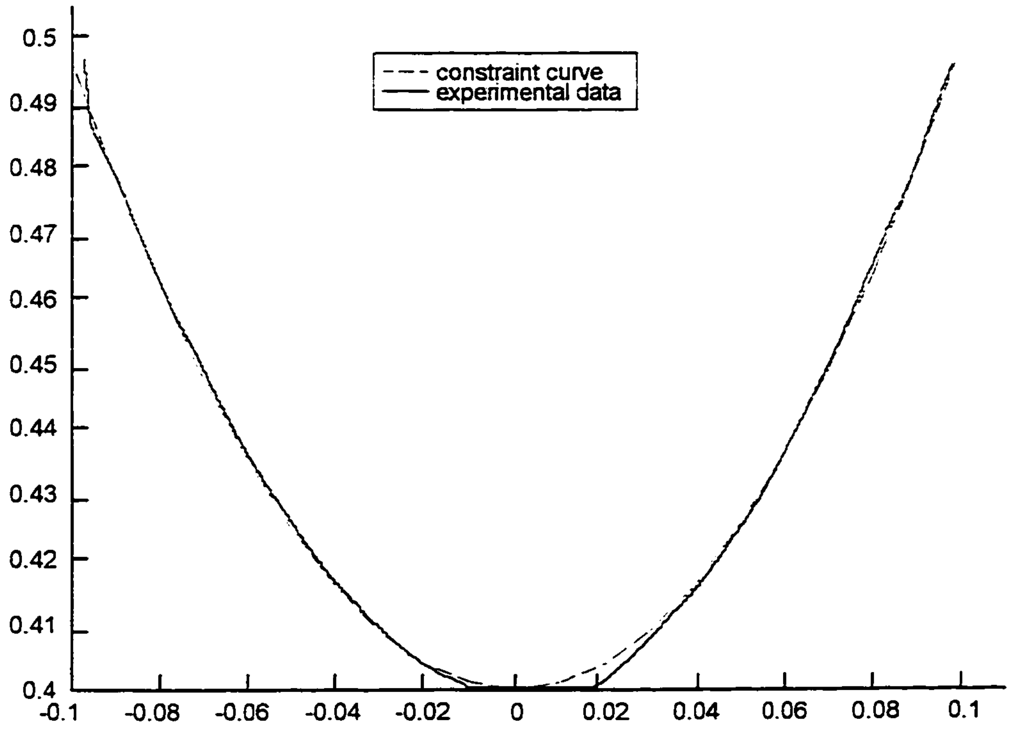


Figure 6.20. Experimental data, slow motion.

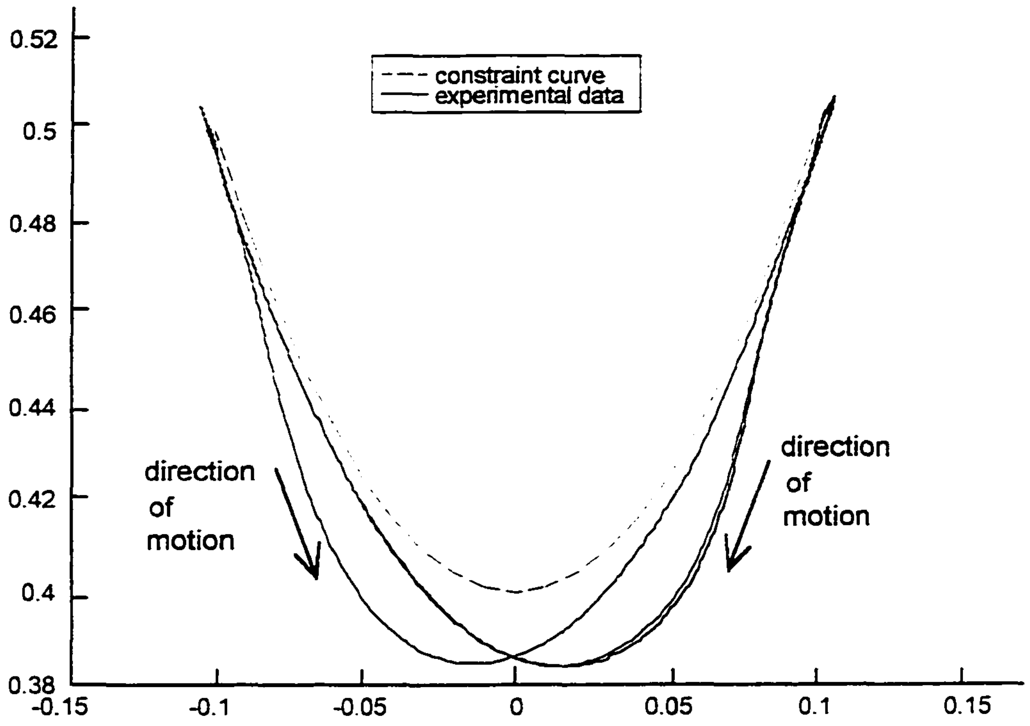


Figure 6.21. Experimental data, fast motion.

inertial effect is more dominate as seen the experimental data. The direction dependent response shown in Figure 6.21 clearly shows the effect of the PUMA inertia on the systems response.

The magnitude of the inertial disturbance can by minimized by changing the stiffness of the local error feedback control springs. However, there is a distinct limit on how high the stiffness of the control springs can be increased. If the stiffness of the control springs is increased above this limit the controller will enter a region of instability. In order to ensure the safety of operator it is essential that no controller instability occur. This upper bound is a property of the robotic manipulator and represents a functional limitation of the device.

NURBS Surface Virtual Manipulator

This paper will not present a detailed description of NURBS surfaces. Only the NURBS surfaces concepts needed to develop the time varying virtual manipulator will be presented. An interested reader is referred to Piegl and Tiller [82] for further details about NURBS surfaces. This development will assume that all weights in the NURBS surface are equal to one. This restriction is only added to make the presentation of this technique more tractable. Removing this restriction only increases the mathematical complexity of the equations, it does not change any of the results.

A NURBS surface, $\mathbf{S}(u,v)$, with all weights equal to one is defined by equation (6.10).

$$\mathbf{S}(u,v) = \sum_{i=0}^{N}\sum_{j=0}^{M} B_{i,n}(u)B_{j,m}(v)\mathbf{P}_{i,j} \tag{6.10}$$

u -parameterization variable
v - parameterization variable
N - number of control points in u-direction
M - number of control points in v-direction
n - degree of surface in u-direction
m - degree of surface in v-direction
$B_{i,n}(u)$ - basis functions in u-direction
$B_{j,m}(v)$ - basis functions in v-direction
$\mathbf{P}_{i,j}$ - matrix of control points

The rate of change of the NURBS surface defined in equation (6.10) with respect to the parameterization variable, u, is shown in equation (6.11).

$$\frac{\partial \mathbf{S}(u,v)}{\partial u} = \mathbf{S}_u(u,v) = \sum_{i=0}^{N}\sum_{j=0}^{M} \frac{dB_{i,n}(u)}{du} B_{j,m}(v)\mathbf{P}_{i,j} \tag{6.11}$$

The rate of change of the NURBS surface defined in equation (6.10) with respect to the parameterization variable, *v*, is shown in equation (6.12).

$$\frac{\partial \mathbf{S}(u,v)}{\partial v} = \mathbf{S}_v(u,v) = \sum_{i=0}^{N}\sum_{j=0}^{M} B_{i,n}(u) \frac{dB_{j,m}(v)}{dv} \mathbf{P}_{i,j} \tag{6.12}$$

The normal of the NURBS surface defined in equation (6.10) is expressed in equation (6.13).

$$\mathbf{N}(u,v) = \mathbf{S}_u(u,v) \otimes \mathbf{S}_v(u,v) \tag{6.13}$$

Equations (6.10), (6.11), (6.12) and (6.13) present all the information about a NURBS surface that is needed to construct a time varying virtual manipulator that will constrain the motion of a robot to follow the surface.

In order to allow the end effector of a robot to trace a NURBS surface there must be way of transforming the Cartesian coordinates of the end effector (*x*, *y*, *z*) into the parametric coordinates of the surface (*u*, *v*). Point inversion is the process of determining the parametric coordinates associated with a given set of Cartesian coordinates [82]. After point inversion is performed the parametric coordinates (u^*, v^*) are determined; which represent the point on the NURBS surface that is closest to the end effector of the robot. The next step is to determine the configuration of the virtual manipulator that will approximate the shape of the NURBS surface in a region around (u^*, v^*).

To approximate a NURBS surface the two degree of freedom mechanism shown schematically in Figure 6.22 will be used. This manipulator can be represented with three coordinate frames as shown in Figure 6.23, however, a fourth frame is added to produce a more "appropriate" connection between the virtual manipulator and the robot. The robot used as an experimental test-bed in this work has a handle which aligns with the z-axis of the end effector of the robot. The addition of the fixed rotation associated with the fourth coordinate frame in Figure 6.23 will allow the handle of the robot to align with the normal of the NURBS surface. The details of this handle - normal alignment will be shown in detail later; however, it should be noted that this alignment is the motivation of the addition of the extra coordinate frame.

A kinematic analysis of the virtual manipulator is required to formulate the virtual

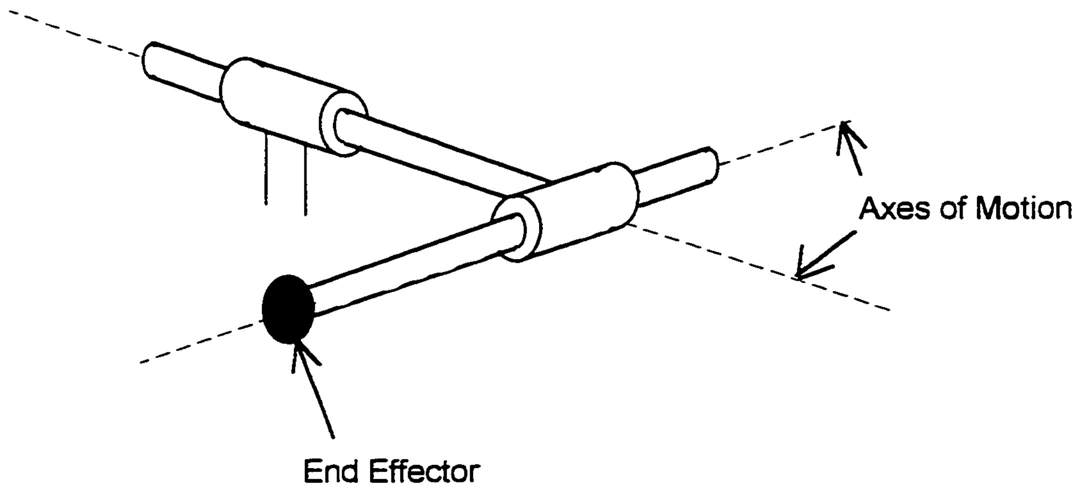


Figure 6.22. Prismatic -prismatic virtual mechanism.

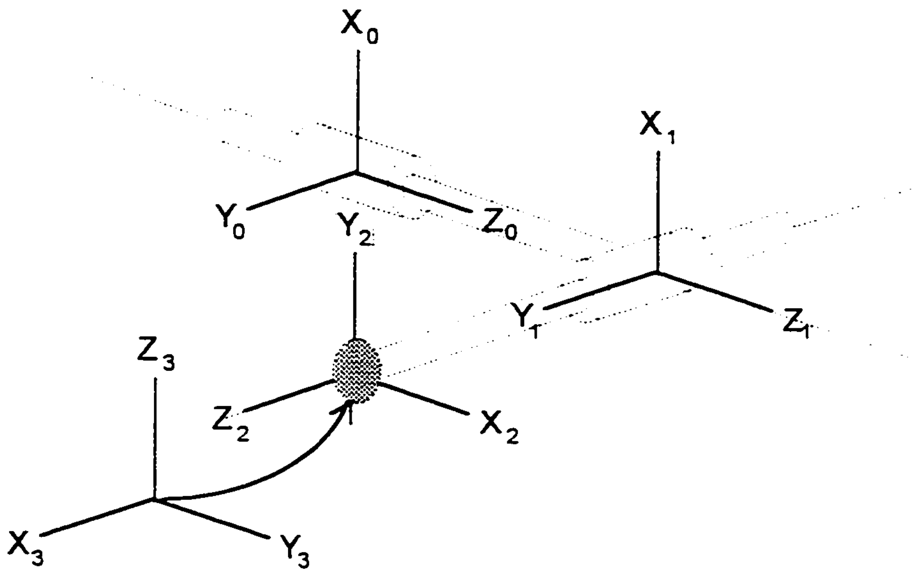


Figure 6.23. Virtual mechanism frame assignment.

constraint controller. The Denavit-Hartenberg (D-H) parameters for the prismatic - prismatic virtual manipulator are shown below in Table 6.3. The D-H parameters in Table 6.3 are used to construct the following kinematic transformation matrix for the virtual manipulator.

$$ {}^{0}_{3}\mathbf{T} = {}^{B}_{E}\mathbf{T} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & d_2 \\ 0 & 1 & 0 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (6.14)$$

This virtual manipulator will allow motion along the bilinear approximation to the

Table 6.3. D-H parameters.

i	a_{i-1}	α_{i-1}	d_i	θ_i
1	0	0	d_1	0
2	0	-90°	d_2	-90°
3	0	-90°	0	-90°

original NURBS surface. The position and orientation of the base frame, {B}, of the virtual manipulator is allowed to vary over the surface. The instantaneous position and orientation of frame {B} are determined by the characteristics of the NURBS surface at the parameter values (u^*, v^*) which were determined by point inversion.

Specifically, the position of frame {B} is defined by $\mathbf{S}(u^*, v^*)$. The orientation of the frame {B} is determined by the two tangents of the NURBS surface, $\mathbf{S}_u(u^*, v^*)$ and $\mathbf{S}_v(u^*, v^*)$. Frame {B} is oriented such that the *x*-axis of frame {B} is in the direction of the $\mathbf{N}(u^*, v^*)$, the *y*-axis is in the direction of $\mathbf{S}_u(u^*, v^*)$, and the *z*-axis is in the direction of $\mathbf{S}_v(u^*, v^*)$.

$$
\begin{aligned}
{}^{W}\hat{\mathbf{x}}_B &= \frac{\mathbf{N}(u^*, v^*)}{\left|\mathbf{N}(u^*, v^*)\right|} \\
{}^{W}\hat{\mathbf{y}}_B &= \frac{\mathbf{S}_u(u^*, v^*)}{\left|\mathbf{S}_u(u^*, v^*)\right|} \\
{}^{W}\hat{\mathbf{z}}_B &= \frac{\mathbf{S}_v(u^*, v^*)}{\left|\mathbf{S}_v(u^*, v^*)\right|}
\end{aligned}
\tag{6.15}
$$

Note that these unit vectors are written with respect to a world coordinate system which has been selected as the base frame for the robot interface for convenience. After determining the configuration of the virtual manipulator the Jacobian relating the end effector velocity with the velocity in the base coordinate frame can be computed.

The Jacobian of the virtual manipulator can be determined by applying standard robot analysis techniques to the kinematic transformation matrices. The Jacobian of this two-degree-of-freedom prismatic - prismatic manipulator is shown in equation (6.16).

$$^{E}\mathbf{J}_{V} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^{T} \tag{6.16}$$

Note that the Jacobian of this virtual manipulator is quite simple and remains constant, which will be a distinct advantage in control approach.

In order to verify the effectiveness of the virtual constraint NURBS surface controller an experiment was performed in which a PUMA 560 manipulator was constrained to a bi-quadratic NURBS surface defined by a matrix of nine control points. Figure 6.24 shows a

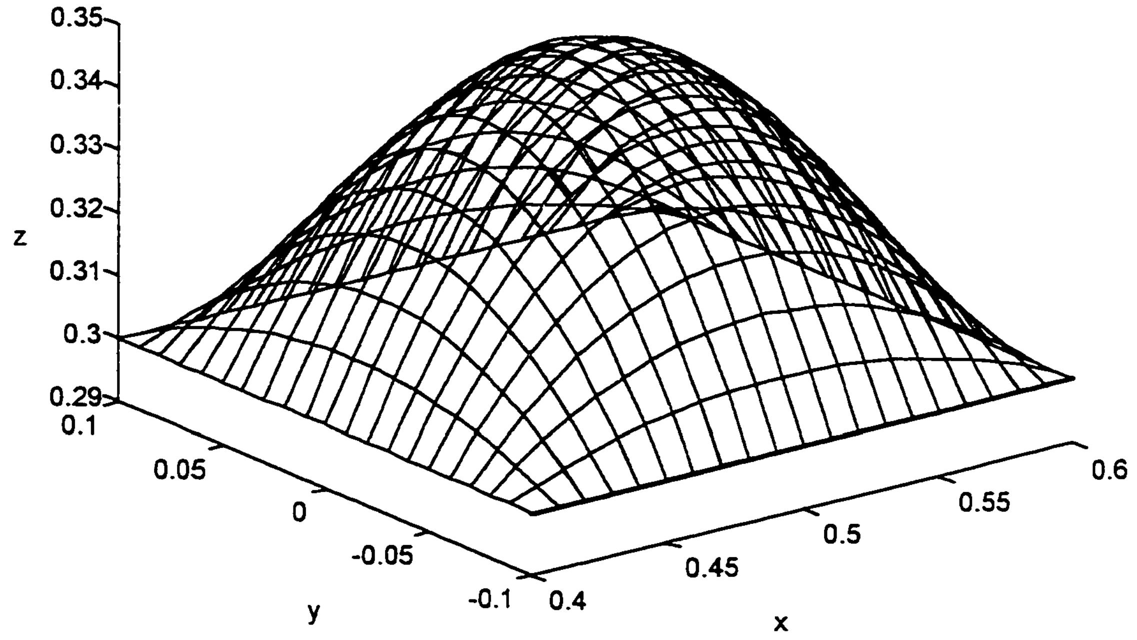


Figure 6.24. Constraint NURBS surface.

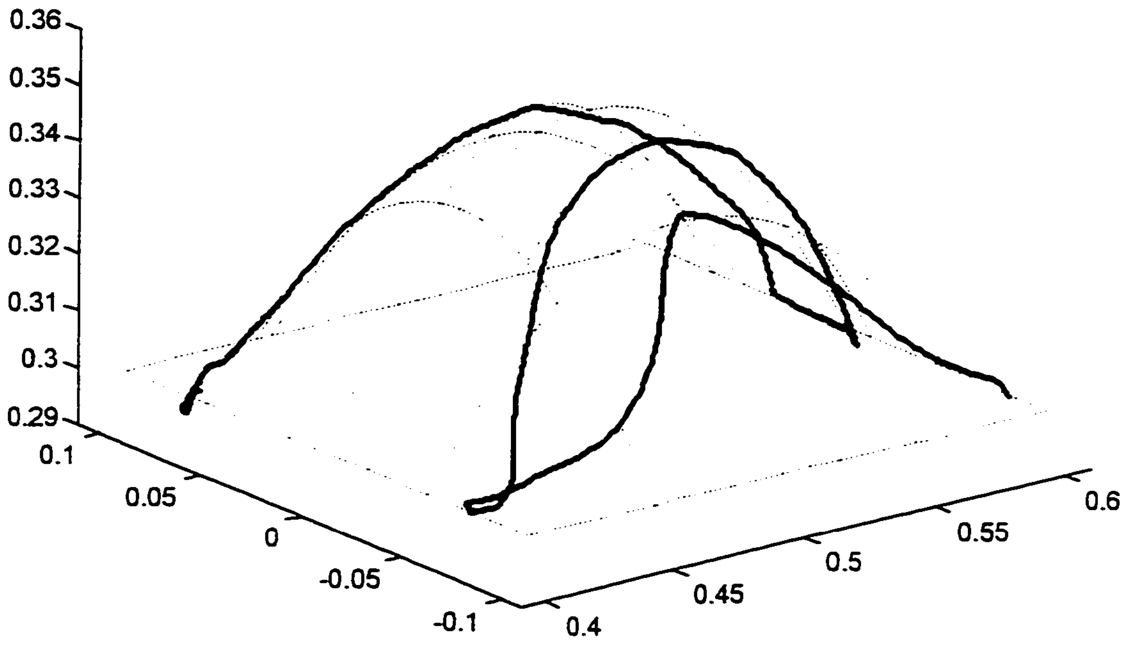


Figure 6.25. Experimental data for NURBS constraint.

picture of the NURBS surface constraint. Figure 6.25 shows experimental position data when the user is moving the end effector of the PUMA over the NURBS surface.

The data presented in Figure 6.25 clearly shows that as the operator moves the PUMA over the NURBS surface, good tracking performance is obtained. However, the constraint controller being used does not compensate for the inertial effects of the PUMA when operator moves the robot quickly, the inertia of the can clearly be felt.

The magnitude of position error caused by inertial disturbances can by minimized by changing the stiffness of the local error feedback control springs. However, there is a distinct limit on how high the stiffness of the control gains can be increased. This upper bound is a property of the robotic manipulator, sensor resolution and noise, and actuator characteristics. This performance limitation of the device is present regardless of the structure of the control scheme. Removing the constraint forces in the virtual manipulator control approach does not affect the overall stability of the underlying error controller. In order to ensure the safety of operator the feedback gains for the local control must be chosen to provide stable operation.

The motion of the end effector of the PUMA was tied to a graphical display to provide the operator with a more complete feeling of immersion. Figure 6.26 shows a typical image from the graphical display. The end of the PUMA is show as an exploration tool in the visual interface. The graphics engine is connected to PUMA control hardware by means of an

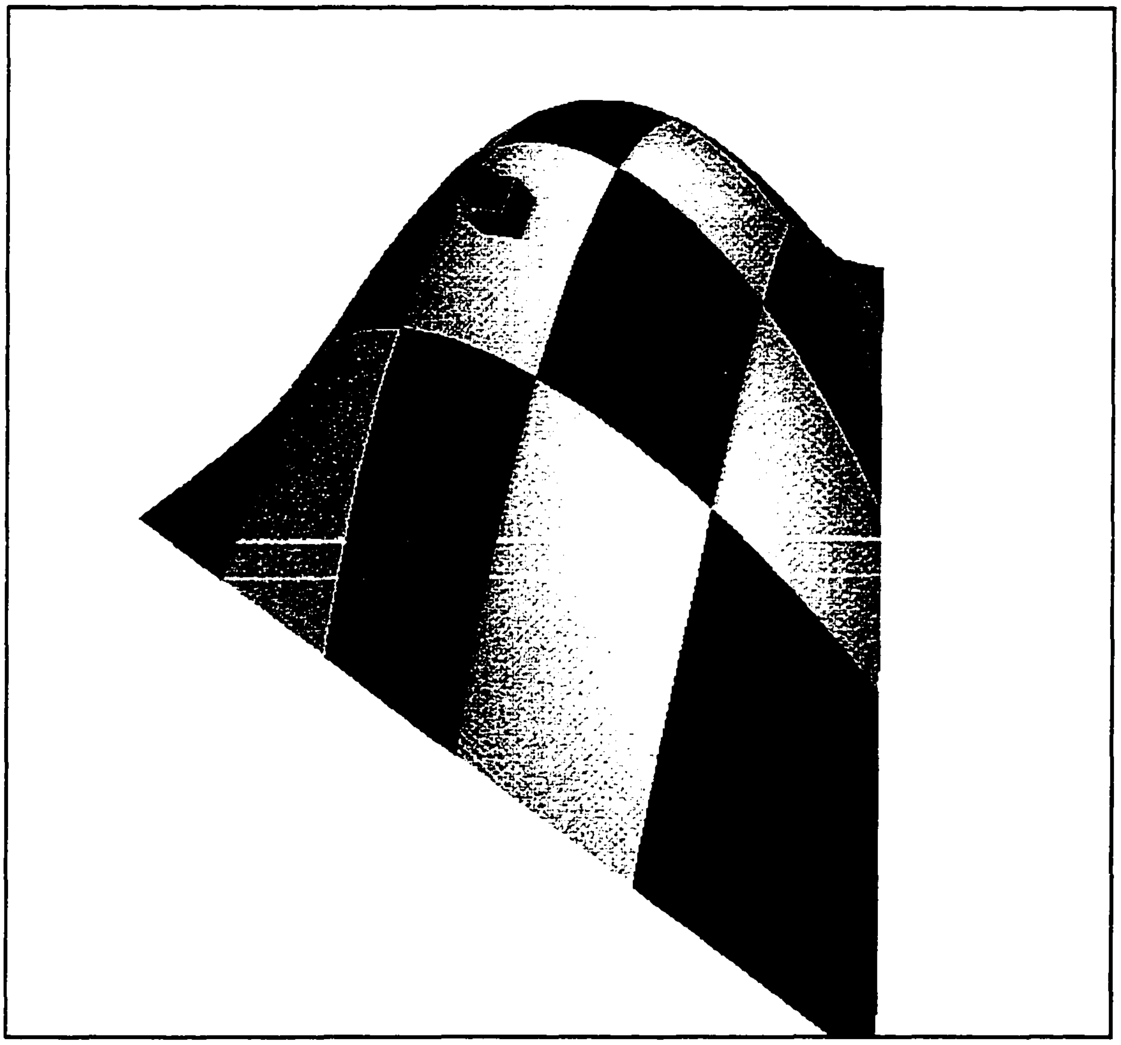


Figure 6.1. Graphical interface.

ethernet socket connect.

High Degree of Freedom Virtual Manipulator

The high degree of freedom virtual manipulator presented in this section is similar to the prismatic virtual manipulator presented earlier. The number of degrees of freedom of the virtual manipulator change to allow exploration and interaction in the synthetic environment. The interaction is achieved in this section by allowing the traveler to manipulate a virtual tool. As a result the contact between the traveler and the synthetic environment is different than with the prismatic virtual manipulator, which allowed only a single point as the tool of exploration.

A five-degree of freedom virtual manipulator will be described that provides force feedback to a young Jedi dueling with a virtual Lord Vader. This work is characteristically different from previous virtual manipulator work, which focused on highly constrained low degree of freedom virtual manipulators. This section will focus on revealing the potential of this control approach as general interaction metaphor for synthetic environments. The fanciful depiction of a light saber battle with Darth Vader is used to underscore the ability of the virtual manipulator approach. The general application of the virtual manipulator presented here will be to examine, maneuver or deform objects placed in a synthetic environment.

When the traveler in the synthetic environment is maneuvering her sword in free space the virtual manipulator will have six degrees of freedom. In this situation the configuration of the virtual manipulator is somewhat arbitrary because regardless of configuration the null space filter matrix becomes a matrix of zeros. Thus the control torques applied to the interface robot is zero and the traveler is free to move the interface robot in any manner.

If, however, the two swords make contact, the virtual manipulator losses a degree of freedom--the degree of freedom that would allow the swords to pass through one another. During contact situations the virtual manipulator has five degrees of freedom and the configuration of the virtual manipulator used is a prismatic joint, three orthogonal revolute joints and a prismatic joint. Thus during contact the traveler can maneuver the interface robot so that her sword has three axes of rotation about the point of contact and two axes of translation along Darth Vader's sword but can not pass through her opponent's sword.

Due to the nature of interaction in this synthetic environment it is difficult to present data that describes the experience. However, an experiment was performed in which the traveler's sword was positioned perpendicular to Darth Vader's sword and moved laterally until the two swords contacted. A schematic diagram of the experiment is shown in Figure 6.27. During the experiment the lateral motion of the traveler's sword was measured as well as the force applied to the end effector of the interface robot by the traveler. This data is presented in Figure 6.28.

Prior to contacting Darth Vader's sword the interface robot was moved in free space this provides an idea of the magnitude of the force necessary to manipulate the interface robot. Approximately 10 Newton's of force is required to stop and change the direction of motion of the interface. This force is most likely associated with the inertia of the device, which has not been compensated for using feed forward elements. In addition, increasing the gain of the force feedback loop can reduce the magnitude of the free space manipulation force. However, this was not done to avoid the closed loop instability sometimes associated with high gain force loops.

Once the two swords come into contact, contact a large negative force peak is seen. This force is preventing the swords from passing through each other. The initial contact gives rise to a slight recoil of the traveler's sword, which is accompanied by a positive force peak. After the contact transient the traveler continues pressing into her opponent's sword and gradually reduces the contact force.

Figure 6.29 shows a picture of the experimental hardware located in the four-wall projection environment. Figure 6.30 shows a not so young Jedi attempting to defeat the evil Lord of Sith.

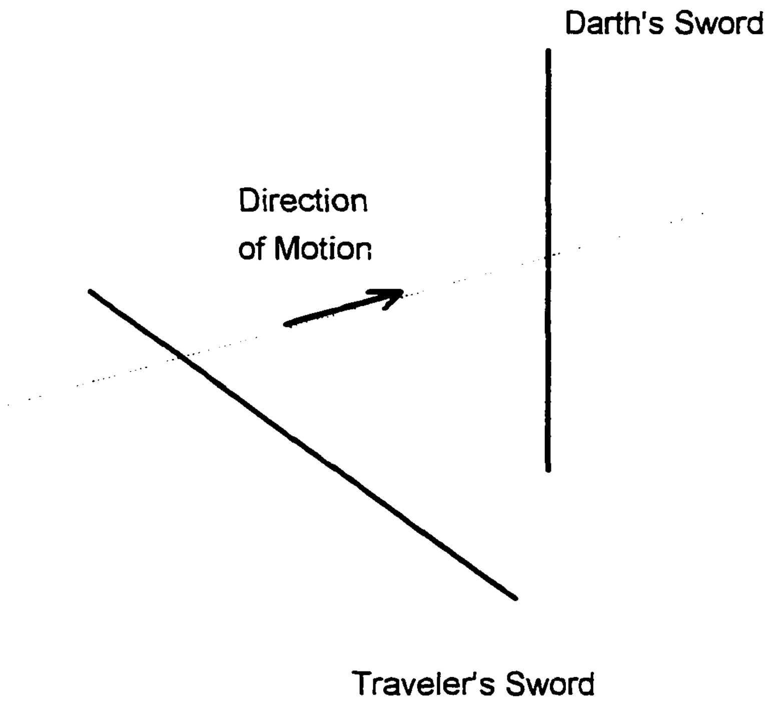


Figure 6.27. Experiment protocol.

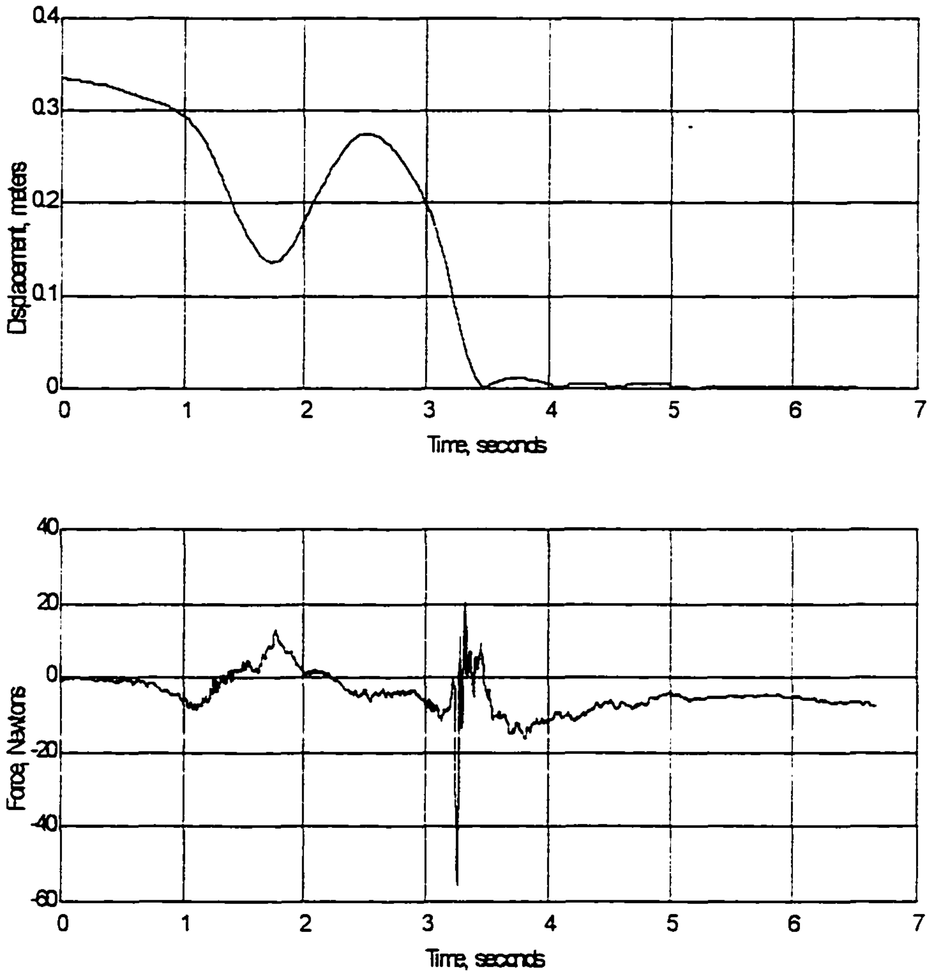


Figure 6.28. Experimental data for light saber.

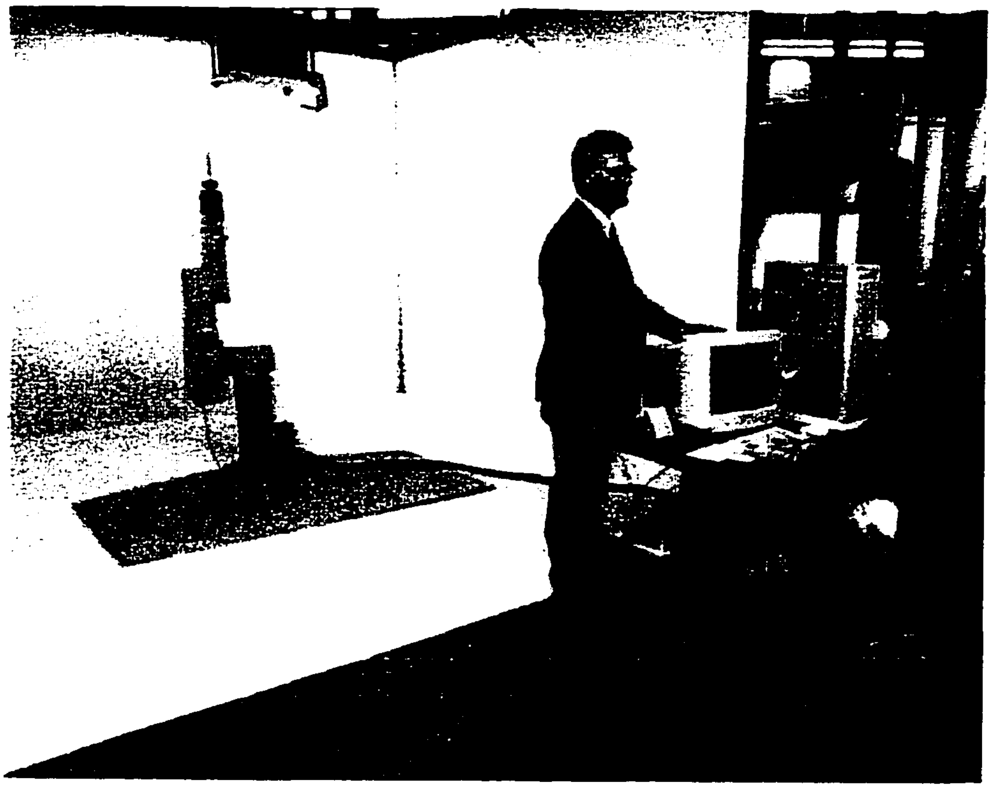


Figure 6.1. Experimental hardware.

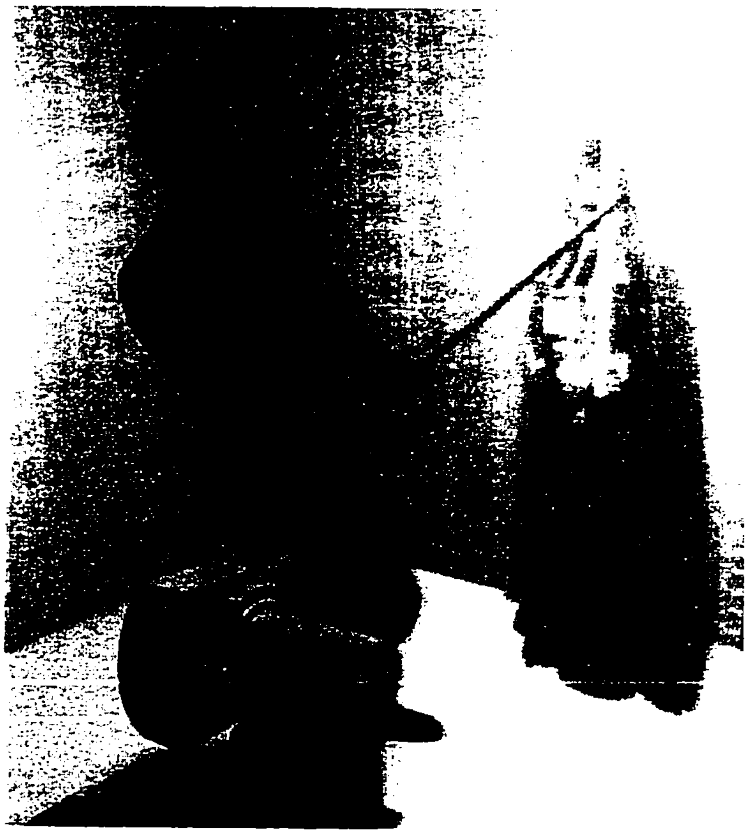


Figure 6.2. Graphical display.

PART II. DYNAMIC MODELS FOR INTERACTION

CHAPTER 7. FINITE ELEMENT ANALYSIS

This chapter will review some of the major concepts of finite element analysis (FEA). This treatment is not intended to be a complete development of FEA but will give a general overview of the technique, focusing on the topics that will be used in Chapter 8. For further information on FEA the reader is referred to [11].

FEA is a technique that reduces an infinite degree of freedom problem, to one with a finite number of degrees of freedom. This reduction in the number of degrees of freedom allows an approximate solution to be found at reduced computational expense. FEA can be divided into five steps, listed below, which will be described in the following sections.

1. Discretization
2. Interpolation
3. Elemental Description
4. Assembly
5. Solution

Discretization

Discretization is the process of dividing a continuous medium into a finite number of elements. The elements are interconnected at special points called nodes. Although the boundary of an element is defined by nodes, nodes may also be located in the interior of an element. See Figure 7.1 for a one-dimensional example of nodes and elements. The response of an element is determined by the nodal displacements associated with the element. The nodal displacements are generalized displacements, which may be translations, rotations or curvatures.

There are several types of elements that can be used; however, all elements can be classified by their dimension and interpolation scheme. Figure 7.2 shows three types of elements with linear interpolation. The selection of an element depends on the nature of the problem being solved; this will be address in more detail in the following section on interpolation.

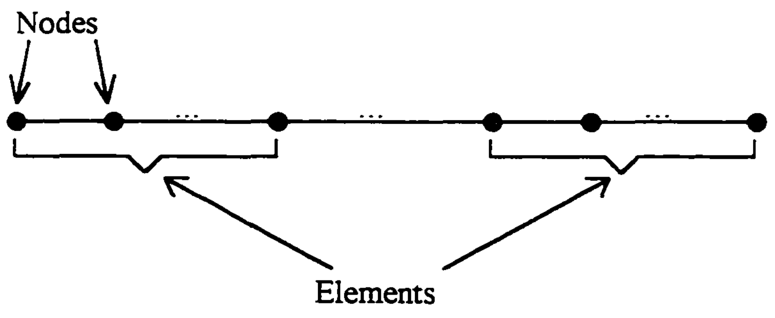


Figure 7.1. Nodes and elements.

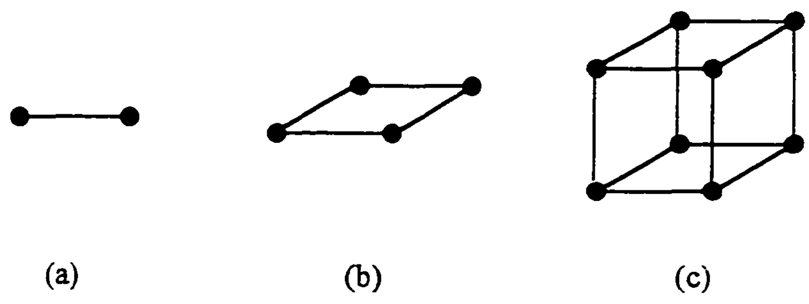


Figure 7.2. Linear elements: (a) 1-D; (b) 2-D; (c) 3-D.

Interpolation

The approximate solution obtained from a finite element model is found by interpolating the nodal displacements of an element with shape functions. The response, $\bar{u}(x,y,z,t)$, for an element is shown in equation (7.1).

$$\bar{u}(x,y,z,t) = \sum_i u_i(t) N_i(x,y,z) \tag{7.1}$$

i is the number of nodes

$u_i(t)$ are the nodal displacements

$N_i(x,y,z)$ are the shape functions

The shape functions used in equation (7.1) are a key component in FEA. The following paragraphs will discuss the requirements that shape functions must satisfy and outline the derivation of the shape functions typically used in FEA.

Shape Function Requirements

There are a number of functions that can be used as shape functions, for example: simple polynomials, Lagrange polynomials and Hermite polynomials. Regardless of type, the shape functions must interpolate the nodal displacements. That is, a shape function must have a unit value at its associated nodal coordinates and must be zero at all other nodal coordinates. This concept is expressed mathematically in equation (7.2).

$$N_i(x_j, y_j, z_j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{7.2}$$

The terms x_j, y_j and z_j are nodal coordinates. In general for a set of shape functions to be acceptable, they must satisfy the following criteria.

1. Shape functions derived from simple polynomials must be balanced with respect to all coordinate axes.
2. The shape functions must have acceptable continuity between elements.
3. The shape function must be complete with respect to the system being modeled.

The reader is referred to [11] for more details on these requirements. In addition, if the nodal displacements are limited to translational displacements the shape functions must also satisfy the following properties.

$$\begin{aligned} &\sum_i N_i = 1 \\ &\sum_i N_i x_i = x \\ &\sum_i N_i y_i = y \\ &\sum_i N_i z_i = z \end{aligned} \tag{7.3}$$

Shape functions derived from simple polynomials are capable of satisfying all of the stated requirements and are easily developed; as a result, they will be used whenever the standard FEA is used. The following section will outline the derivation of this type of shape function.

Derivation of Shape Functions

When deriving simple polynomial shape functions, the interpolation degree, *n*, and dimension of the problem are used to develop an approximate solution. For a one dimensional problem with *nth* degree interpolation the approximate solution would have the following form.

$$\bar{u}(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{7.4}$$

The coefficients a_i are constant coefficients. The approximate solution, equation (7.4), is then evaluated at the $n+1$ nodal coordinates, x_i.

$$\begin{aligned}
\bar{u}(x_1) &= u_1 = a_0 + a_1 x_1 + a_2 x_1^2 + \cdots + a_n x_1^n \\
\bar{u}(x_2) &= u_2 = a_0 + a_1 x_2 + a_2 x_2^2 + \cdots + a_n x_2^n \\
&\vdots \\
\bar{u}(x_{n-1}) &= u_{n-1} = a_0 + a_1 x_{n-1} + a_2 x_{n-1}^2 + \cdots + a_n x_{n-1}^n
\end{aligned} \tag{7.5}$$

The system of equations (7.5) can be rewritten in matrix form.

$$\begin{Bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \end{Bmatrix} = \begin{bmatrix} 1 & x_1 & \cdots & x_1^n \\ 1 & x_2 & \cdots & x_2^n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & \cdots & x_{n-1}^n \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{Bmatrix} \tag{7.6}$$

$$\mathbf{u} = \mathbf{X}\mathbf{a}$$

The coefficients, a_i, can be found in terms of the nodal coordinates, x_i, and the nodal displacements, u_i, by matrix inversion.

$$\mathbf{a} = \mathbf{X}^{-1}\mathbf{u} \tag{7.7}$$

The approximate solution can be rewritten by substituting the coefficients, $\mathbf{a}$, into equation (7.4).

$$\bar{u}(x) = \{1 \quad x \quad \cdots \quad x^n\}\begin{Bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{Bmatrix} = \{1 \quad x \quad \cdots \quad x^n\}\mathbf{X}^{-1}\mathbf{u} \tag{7.8}$$

The shape functions, $N_i(x)$, are determined by combining terms associated with each nodal displacement, u_i. This is done systematically using equation (7.9).

$$N_i(x) = \{1 \quad x \quad \cdots \quad x^n\}\begin{Bmatrix} X_1^{-1} \\ X_2^{-1} \\ \vdots \\ X_{n-1}^{-1} \end{Bmatrix}_i \tag{7.9}$$

$$\mathbf{X}^{-1} = \begin{bmatrix} X_1^{-1} & X_2^{-1} & \cdots & X_{n-1}^{-1} \end{bmatrix}.$$

Shape functions can be derived in the above fashion for all elements in a finite element model. However, the shape functions are different for each element and the process of solving for all of the different shape functions is time consuming. In order to simplify this process the coordinate transformation described in the following section is used.

Coordinate Transformation

To simplify shape function calculation, shape functions are determined for a simple parent element, which are transformed to the particular element in question. The transformation from the parent element to the actual element is achieved through a coordinate

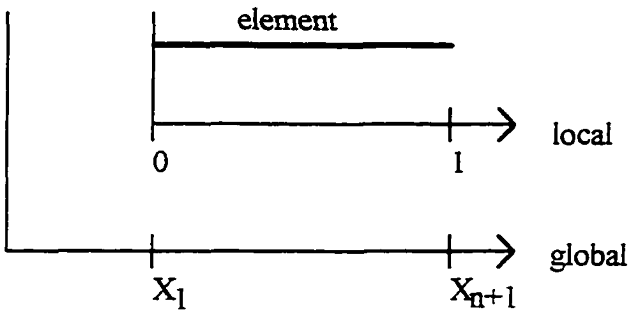


Figure 7.3. Coordinate transformation.

transformation from a local coordinate system to the global coordinate system. Figure 7.3 shows an example of this transformation for a one dimensional element of degree *n*. The use of the above coordinate transformation will be clarified in the next section, which discusses the elemental description.

As discussed in the discretization section, the selection of an element and an interpolation scheme in constructing a finite element model are not independent. The major consideration in selecting an interpolation scheme is that in the limit as the elements are refined the approximate finite element solution should converge to the exact solution.

Elemental Description

There are various ways to generate a finite element model. Direct integration of the differential equation is the most straight forward method. However, this is not always possible and in this situation there are two commonly accepted approaches to obtain the desired model. In the first approach the energy associated with the problem is determined by integration over the domain and boundary of the region. The method of Ritz is then used to obtain a set of algebraic equations [11]. The second approach uses the differential equation directly in the weak formulation. This approach is equivalent to the principle of virtual work used in mechanics. A set of algebraic equations is obtained by combining the weak formulation of the differential equation and Galerkin's method.

In order to demonstrate how Galerkin's method and the weak formulation of a differential equation are used in constructing the elemental description, the following paragraphs will develop a finite element model for the one-dimensional wave equation.

$$\frac{\partial}{\partial x}\left(k\frac{\partial u}{\partial x}\right) - p\frac{\partial^2 u}{\partial t^2} + f = 0 \quad 0 \le x \le L,\ 0 \le t \tag{7.10}$$

The coefficients k, p and f represent the restoring force, mass density and external force respectively. Two boundary conditions are required to solve equation (7.10). There are two types of acceptable boundary conditions: essential and natural.

$$\begin{aligned} &u = a \quad \text{(essential)} \\ &k\frac{\partial u}{\partial x} = b \quad \text{(natural)} \end{aligned} \tag{7.11}$$

The constants a and b are the boundary conditions. One type of boundary condition, either essential or natural, must be specified at $x = 0$ and $x = L$. In addition to the two boundary conditions, two initial conditions are also required to solve equation (7.10).

$$u(x,0) = f(x) \quad (\text{position})$$
$$\frac{\partial u(x,0)}{\partial t} = g(x) \quad (\text{velocity}) \tag{7.12}$$

The functions $f(x)$ and $g(x)$ are the initial conditions.

Galerkin's method is considered to be one of the Methods of Weighted Residuals (MWR). MWR is a general technique that finds application in minimization processes outside of FEA. In the MWR an approximate solution, $\bar{u}$, to the differential equation is selected.

$$\bar{u}(x,t) = \phi_o(x) + \sum_i c_i(t)\phi_i(x) \tag{7.13}$$

c_i are constant coefficients

ϕ_i are functions which satisfy the boundary conditions

The approximate solution, $\bar{u}$, typically used in FEA is based on the nodal displacements and shape functions as shown in equation (7.1), rewritten here in vector form.

$$\bar{u}(x,t) = \mathbf{N}^T(x)\mathbf{u}(t) = \{N_1(x) \quad N_2(x) \quad \cdots \quad N_{n-1}(x)\}\begin{Bmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ u_{n-1}(t) \end{Bmatrix} \tag{7.14}$$

The integer *n* is determined by the degree of the interpolation scheme. When equations (7.13) and (7.14) are compared it is obvious that:

$$\begin{aligned}&\phi_o = 0\\ &c_i(t) = u_i(t)\\ &\phi_i(x) = N_i(x), \quad \text{for } i = 1,2,\ldots,n+1\end{aligned} \tag{7.15}$$

The approximate solution, $\bar{u}$, is then substituted into the wave equation (7.10).

$$\frac{\partial}{\partial x}\left(k\frac{\partial \bar{u}}{\partial x}\right) - p\frac{\partial^2 \bar{u}}{\partial t^2} + f \approx 0 \tag{7.16}$$

The residual, error caused by the approximate solution, R(x) is then evaluated.

$$R(x) = \left[\frac{\partial}{\partial x}\left(k\frac{\partial \bar{u}}{\partial x}\right) - p\frac{\partial^2 \bar{u}}{\partial t^2} + f\right] - \left[\frac{\partial}{\partial x}\left(k\frac{\partial u}{\partial x}\right) - p\frac{\partial^2 u}{\partial t^2} + f\right] \tag{7.17}$$

If the approximate solution was exact then the residual would be zero. In general the residual is non-zero; therefore, the goal is to minimize the residual by requiring the weighted average of the residual to be zero. The weighted average of the residual is evaluated in equation (7.18).

$$\int_{x_1}^{x_{n+1}} R(x)w(x)dx = 0 \tag{7.18}$$

x_1 is the coordinate value of the first node of the element

x_{n-1} is the coordinate value of the last node of the element

$w(x)$ are the weight functions

There are several types of weight functions. The selection of a weight function determines the way in which the residual is minimized. In Galerkin's method the weight functions are the same as the approximating polynomials (shape functions).

$$w_i(x) = \phi_i(x) = N_i(x) \tag{7.19}$$

Before substituting the shape functions into equation (7.18) the weak form of the differential equation will be derived by integration by parts [97].

$$\int_{x_1}^{x_{n+1}} w\left[\frac{\partial}{\partial x}\left(k\frac{\partial \bar{u}}{\partial x}\right) - p\frac{\partial^2 \bar{u}}{\partial t^2} + f\right]dx = wk\frac{\partial \bar{u}}{\partial x}\bigg|_{x_1}^{x_{n+1}} - \int_{x_1}^{x_{n+1}}\left(\frac{\partial w}{\partial x}k\frac{\partial \bar{u}}{\partial x} + wp\frac{\partial^2 \bar{u}}{\partial t^2} - wf\right)dx = 0 \tag{7.20}$$

The shape functions derived in the interpolation section are then substituted into equation (7.20).

$$\int_{x_1}^{x_{n+1}} \left(p\mathbf{N}(x)\mathbf{N}^T(x)\ddot{\mathbf{u}}(t) + k\frac{d\mathbf{N}(x)}{dx}\frac{d\mathbf{N}^T(x)}{dx}\mathbf{u}(t) \right) dx = \int_{x_1}^{x_{n+1}} \mathbf{N}(x) f dx + k\mathbf{N}(x)\frac{d\mathbf{N}^T(x)}{dx}\mathbf{u}(t) \Bigg|_{x_1}^{x_{n+1}} \quad (7.21)$$

$$\frac{\partial^2 \bar{u}}{\partial t^2} = \frac{\partial^2}{\partial t^2}\left[\mathbf{N}^T(x)\mathbf{u}(t)\right] = \mathbf{N}^T(x)\frac{\partial^2}{\partial t^2}\left[\mathbf{u}(t)\right] = \mathbf{N}^T(x)\ddot{\mathbf{u}}(t)$$

$$\frac{\partial w}{\partial x} = \frac{\partial}{\partial x}\mathbf{N}(x) = \frac{d\mathbf{N}(x)}{dx}$$

$$\frac{\partial \bar{u}}{\partial x} = \frac{\partial}{\partial x}\left[\mathbf{N}^T(x)\mathbf{u}(t)\right] = \frac{d\mathbf{N}^T(x)}{dx}\mathbf{u}(t)$$

Equation (7.21) is the elemental description for the one-dimensional wave equation. Equation (7.21) is solved for the vector of nodal displacements, $\mathbf{u}(t)$. These displacements minimize the error between the actual solution, $u(x,t)$, and the approximate solution, $\bar{u}(x,t)$. The different components of the elemental model can be determined by investigating equation (7.21).

The elemental mass matrix, $\mathbf{M_e}$, is associated with the nodal accelerations, $\ddot{\mathbf{u}}(t)$, and is evaluated by equation (7.22).

$$\mathbf{M}_e\ddot{\mathbf{u}}(t) = \int_{x_1}^{x_{n+1}} p\mathbf{N}(x)\mathbf{N}^T(x)\ddot{\mathbf{u}}(t)dx$$

$$\mathbf{M}_e = \int_{x_1}^{x_{n+1}} \begin{bmatrix} pN_1(x)N_1(x) & pN_1(x)N_2(x) & \cdots & pN_1(x)N_n(x) \\ & pN_2(x)N_2(x) & \cdots & pN_2(x)N_n(x) \\ & & \ddots & \vdots \\ SYM & & & pN_n(x)N_n(x) \end{bmatrix} dx \tag{7.22}$$

The elemental stiffness matrix, $\mathbf{K_e}$, is associated with the nodal displacements, $\mathbf{u}(t)$, and is evaluated by equation (7.23).

$$\mathbf{K}_e\mathbf{u}(t) = \int_{x_1}^{x_{n+1}} k\frac{d\mathbf{N}(x)}{dx}\frac{d\mathbf{N}^T(x)}{dx}\mathbf{u}(t)dx$$

$$\mathbf{K}_e = \int_{x_1}^{x_{n+1}} \begin{bmatrix} kN_1'(x)N_1'(x) & kN_1'(x)N_2'(x) & \cdots & kN_1'(x)N_n'(x) \\ & kN_2'(x)N_2'(x) & \cdots & kN_2'(x)N_n'(x) \\ & & \ddots & \vdots \\ SYM & & & kN_n'(x)N_n'(x) \end{bmatrix} dx \tag{7.23}$$

$$N_i'(x) = \frac{dN_i(x)}{dx}$$

The elemental force vector, $\mathbf{f_e}$, is associated with the external distributed force, f, and is evaluated by equation (7.24).

$$\mathbf{f}_e = \int_{x_1}^{x_{n+1}} \mathbf{N}(x) f dx$$

$$\mathbf{f}_e = \int_{x_1}^{x_{n+1}} f \begin{Bmatrix} N_1(x) \\ N_2(x) \\ \vdots \\ N_{n+1}(x) \end{Bmatrix} dx \tag{7.24}$$

The final term in the elemental description contains information about the natural boundary conditions. The vector, $\mathbf{b_e}$, is evaluated in equation (7.25).

$$\mathbf{b}_e = k\mathbf{N}(x)\frac{d\mathbf{N}^T(x)}{dx}\mathbf{u}\Bigg|_{x_1}^{x_{n+1}}$$

$$\mathbf{b}_e = \left.\begin{bmatrix} kN_1(x)N_1'(x) & kN_1(x)N_2'(x) & \cdots & kN_1(x)N_{n+1}'(x) \\ & kN_2(x)N_2'(x) & \cdots & kN_2(x)N_{n+1}'(x) \\ & & \ddots & \vdots \\ SYM & & & kN_{n+1}(x)N_{n+1}'(x) \end{bmatrix} \begin{Bmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ u_{n+1}(t) \end{Bmatrix}\right|_{x_1}^{x_{n+1}}$$

$$N_2(x_1) = N_3(x_1) = \cdots = N_{n+1}(x_1) = 0 \quad N_1(x_1) = 1$$
$$N_1(x_{n+1}) = N_2(x_{n+1}) = \cdots = N_n(x_{n+1}) = 0 \quad N_{n+1}(x_{n+1}) = 1$$

$$\mathbf{b}_e = k\begin{Bmatrix} -\dfrac{\partial \bar{u}(x_1,t)}{\partial x} \\ 0 \\ \vdots \\ 0 \\ \dfrac{\partial \bar{u}(x_{n-1},t)}{\partial x} \end{Bmatrix} \tag{7.25}$$

The statement of the finite element model, equation (7.21), can therefore be rewritten using equations (7.22), (7.23), (7.24) and (7.25).

$$\mathbf{M}_e\ddot{\mathbf{u}}(t) + \mathbf{K}_e\mathbf{u}(t) = \mathbf{f}_e \div \mathbf{b}_e \tag{7.26}$$

The elemental model in equation (7.26) is obtained by integrating various physical parameters and the shape functions. The coordinate transformation described in the interpolation section is used to simplify these integrations. The shape functions are determined with respect to the local coordinate system, $\mathbf{N}(\psi)$, instead of the global coordinate system, $\mathbf{N}(x)$. The first term in the elemental stiffness matrix, $\mathbf{K_e}$, will be used to clarify the coordinate transformation.

$$K_{e1,1} = \int_{x_1}^{x_{n+1}} kN_1'(x)N_1'(x)dx = \int_0^1 k\frac{dN_1}{d\psi}\frac{d\psi}{dx}\frac{dN_1}{d\psi}\frac{d\psi}{dx}d\psi\frac{dx}{d\psi} \tag{7.27}$$

This coordinate transformation can be used for all parts of the elemental description, thus greatly simplifying the calculation of the elemental matrices. After the elemental matrices have been calculated they must be assembled to construct the global system model.

Assembly

The elemental description formulated in the previous section allows the elemental matrices to be determined. The elemental matrices are then assembled, in order to construct the global system model. The global system model can be developed by inspection for one-dimensional and simple multidimensional problems. For example, consider constructing the global stiffness matrix for a system with two quadratic elements as shown in Figure 7.4. There are five, one degree of freedom nodes in this model. As a result the global stiffness matrix will be a 5x5. The global stiffness matrix will contain elements from the two elemental matrices. The stiffness matrix for the first element has the form shown in equation (7.28).

$$\mathbf{K}_e^1\mathbf{u}^1 = \begin{bmatrix} k_{1,1}^1 & k_{1,2}^1 & k_{1,3}^1 \\ k_{2,1}^1 & k_{2,2}^1 & k_{2,3}^1 \\ k_{3,1}^1 & k_{3,2}^1 & k_{3,3}^1 \end{bmatrix} \begin{Bmatrix} u_1^1 \\ u_2^1 \\ u_3^1 \end{Bmatrix} \tag{7.28}$$

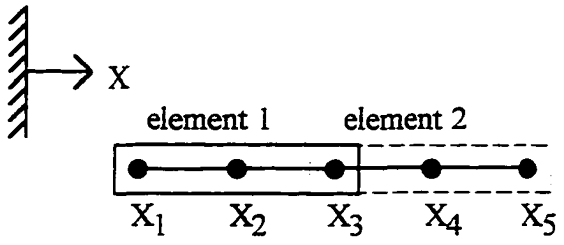


Figure 7.4. Two element model.

The stiffness matrix for the second element has the form shown in equation (7.29).

$$\mathbf{K}_e^2\mathbf{u}^2 = \begin{bmatrix} k_{1,1}^2 & k_{1,2}^2 & k_{1,3}^2 \\ k_{2,1}^2 & k_{2,2}^2 & k_{2,3}^2 \\ k_{3,1}^2 & k_{3,2}^2 & k_{3,3}^2 \end{bmatrix} \begin{Bmatrix} u_1^2 \\ u_2^2 \\ u_3^2 \end{Bmatrix} \tag{7.29}$$

The global matrix is the sum of the elemental matrices when they are expanded to the size of the global matrix. However, before the elemental matrices can be expanded to global size, the local nodal displacements must be mapped to the global nodal displacements as shown in Figure 7.5.

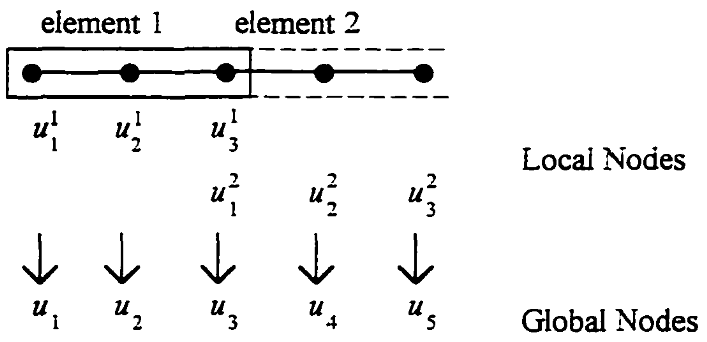


Figure 7.5. Mapping local nodal displacements to global nodal displacements.

$$\begin{aligned} u_1^1 &= u_1 \\ u_2^1 &= u_2 \\ u_3^1 &= u_3 \\ u_1^2 &= u_3 \\ u_2^2 &= u_4 \\ u_3^2 &= u_5 \end{aligned} \tag{7.30}$$

The elemental matrices can now be expanded and summed.

$$\mathbf{K}_g\mathbf{u}_g = \begin{bmatrix} k_{1,1}^1 & k_{1,2}^1 & k_{1,3}^1 & 0 & 0 \\ k_{2,1}^1 & k_{2,2}^1 & k_{2,3}^1 & 0 & 0 \\ k_{3,1}^1 & k_{3,2}^1 & k_{3,3}^1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{Bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & k_{1,1}^2 & k_{1,2}^2 & k_{1,3}^2 \\ 0 & 0 & k_{2,1}^2 & k_{2,2}^2 & k_{2,3}^2 \\ 0 & 0 & k_{3,1}^2 & k_{3,2}^2 & k_{3,3}^2 \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{Bmatrix} \tag{7.31}$$

Therefore the global stiffness matrix has the following form.

$$\mathbf{K}_g\mathbf{u}_g = \begin{bmatrix} k_{1,1}^1 & k_{1,2}^1 & k_{1,3}^1 & 0 & 0 \\ k_{2,1}^1 & k_{2,2}^1 & k_{2,3}^1 & 0 & 0 \\ k_{3,1}^1 & k_{3,2}^1 & k_{3,3}^1 + k_{1,1}^2 & k_{1,2}^2 & k_{1,3}^2 \\ 0 & 0 & k_{2,1}^2 & k_{2,2}^2 & k_{2,3}^2 \\ 0 & 0 & k_{3,1}^2 & k_{3,2}^2 & k_{3,3}^2 \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{Bmatrix} \tag{7.32}$$

The inspection method works well for one-dimensional problems; however, with higher dimension problems a more systematic approach is required.

The assembly process is easily automated by using a connectivity matrix. The connectivity matrix contains the global node numbers which map the elemental matrices into the global matrix. The connectivity matrix, **C**, for the example problem shown in Figure 7.4 is shown in Figure 7.6. Once the connectivity matrix is constructed, it can be used to map the elemental matrices into the global matrix using the following relationship.

$$k^e_{i,j} \longrightarrow K_{I,J} \qquad (7.33)$$

$$I = \mathbf{C}(e,i)$$
$$J = \mathbf{C}(e,j)$$

The terms $k^e_{i,j}$ and $K_{I,J}$ are the elements of the elemental matrices and the elements of the global matrix respectively. An element in the global matrix is the sum of all of the elemental elements mapped to the global element. This technique produces the same global matrix as the inspection method; however, the technique can be implemented in computer code to

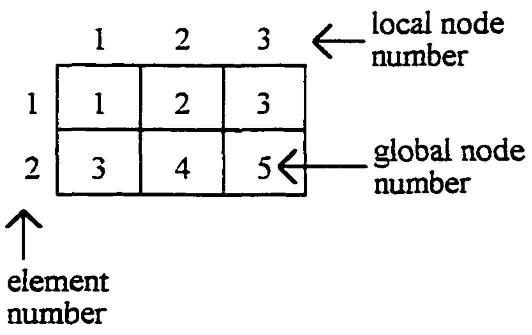


Figure 7.6. Connectivity matrix.

automate the assembly process.

Solution

After the assembly process the global system model is obtained as shown in equation (7.34).

$$\mathbf{M}_\mathbf{g}\ddot{\mathbf{u}}_\mathbf{g}(t) + \mathbf{K}_\mathbf{g}\mathbf{u}_\mathbf{g}(t) = \mathbf{f}_\mathbf{g} + \mathbf{b}_\mathbf{g} \tag{7.34}$$

The global system model is now ready to be solved for the nodal displacements. The system of differential equations will be solved using a central difference approximation, which is discussed in the following section.

Central Difference Approximation

The central difference approach reduces the system of differential equations to a system of algebraic equations by expressing the nodal accelerations, $\ddot{\mathbf{u}}_\mathbf{g}(t)$ in terms of $\mathbf{u}_\mathbf{g}(t+\Delta t)$, $\mathbf{u}_\mathbf{g}(t)$ and $\mathbf{u}_\mathbf{g}(t-\Delta t)$. The $\mathbf{u}_\mathbf{g}(t+\Delta t)$ term can be approximated using $\mathbf{u}_\mathbf{g}(t)$ and a three term Taylor series expansion [97] as shown in equation (7.35).

$$\mathbf{u}_\mathbf{g}(t+\Delta t) = \mathbf{u}_\mathbf{g}(t) + \Delta t\dot{\mathbf{u}}_\mathbf{g}(t) + \frac{(\Delta t)^2}{2}\ddot{\mathbf{u}}_\mathbf{g}(t) \tag{7.35}$$

The $\mathbf{u}_g(t - \Delta t)$ term can be approximated in the same fashion as shown in equation (7.36).

$$\mathbf{u}_g(t - \Delta t) = \mathbf{u}_g(t) - \Delta t\dot{\mathbf{u}}_g(t) + \frac{(-\Delta t)^2}{2}\ddot{\mathbf{u}}_g(t) \tag{7.36}$$

An expression for $\ddot{\mathbf{u}}_g(t)$ is found by adding equations (7.35) and (7.35).

$$\mathbf{u}_g(t + \Delta t) + \mathbf{u}_g(t - \Delta t) = 2\mathbf{u}_g(t) + (\Delta t)^2\ddot{\mathbf{u}}_g(t) \tag{7.37}$$

Solving for $\ddot{\mathbf{u}}_g(t)$:

$$\ddot{\mathbf{u}}_g(t) = \frac{\mathbf{u}_g(t + \Delta t) + \mathbf{u}_g(t - \Delta t) - 2\mathbf{u}_g(t)}{(\Delta t)^2} \tag{7.38}$$

Equation (7.38) is then substituted into the global system model, equation (7.34).

$$\mathbf{M}_g\left[\frac{\mathbf{u}_g(t + \Delta t) + \mathbf{u}_g(t - \Delta t) - 2\mathbf{u}_g(t)}{(\Delta t)^2}\right] + \mathbf{K}_g\mathbf{u}_g(t) = \mathbf{f}_g + \mathbf{b}_g \tag{7.39}$$

This substitution reduces the system of differential equations to a system of algebraic equations. Equation (7.39) can be rewritten as shown in equation (7.40).

$$\mathbf{M}_g\mathbf{u}_g(t+\Delta t) = \left[2\mathbf{M}_g - (\Delta t)^2\mathbf{K}_g\right]\mathbf{u}_g(t) - \mathbf{M}_g\mathbf{u}_g(t-\Delta t) + (\Delta t)^2\mathbf{f}_g + (\Delta t)^2\mathbf{b}_g \quad (7.40)$$

Equation (7.40) is a finite difference equation that allows $\mathbf{u}_g(t+\Delta t)$ to be determined from $\mathbf{u}_g(t)$ and $\mathbf{u}_g(t-\Delta t)$. Equation (7.40) allows an approximate solution to the original differential equation to be obtained by stepping through time. Although equation (7.40) is easy to solve there is some difficulty in starting the solution process, which will be discussed in the next section.

Start-Up

In order to solve equation (7.40) for $\mathbf{u}_g(\Delta t)$ the nodal displacements, $\mathbf{u}_g(0)$ and $\mathbf{u}_g(-\Delta t)$ must be known. However, $\mathbf{u}_g(-\Delta t)$ is not specified in the original statement of the differential equation. As a result $\mathbf{u}_g(-\Delta t)$ will be estimated using the Taylor series expansion and the given initial conditions.

$$\mathbf{u}_g(-\Delta t) = \mathbf{u}_g(0) - \Delta t\dot{\mathbf{u}}_g(0) + \frac{(-\Delta t)^2}{2}\ddot{\mathbf{u}}_g(0) \quad (7.41)$$

The constants $\mathbf{u}_g(0)$, $\dot{\mathbf{u}}_g(0)$ and $\ddot{\mathbf{u}}_g(0)$ are the initial nodal displacements, initial nodal velocities and the initial nodal accelerations respectively. The initial nodal displacements and velocities are available from the initial conditions applied to the original differential equation.

The initial nodal accelerations are not specified; however, they can be obtained by evaluating equation (7.34) at the initial conditions.

$$\ddot{\mathbf{u}}_{\mathbf{g}}(0) = \mathbf{M}_{\mathbf{g}}^{-1}\left[\mathbf{f}_{\mathbf{g}} + \mathbf{b}_{\mathbf{g}} - \mathbf{K}_{\mathbf{g}}\mathbf{u}_{\mathbf{g}}(0)\right] \tag{7.42}$$

With $\mathbf{u}_{\mathbf{g}}(-\Delta t)$ determined only one obstacle remains to be overcome before the system of equations can be solved. The system of equations must be constrained to comply with the boundary conditions.

Constraints

The system of equations (7.40) reduces to an algebraic set of equations of the form shown in equation (7.43).

$$\mathbf{M}_{\mathbf{g}}\mathbf{u}_{\mathbf{g}}(t + \Delta t) = \mathbf{rhs} \tag{7.43}$$

$$\mathbf{rhs} = \left[2\mathbf{M}_{\mathbf{g}} - (\Delta t)^2\mathbf{K}_{\mathbf{g}}\right]\mathbf{u}_{\mathbf{g}}(t) - \mathbf{M}_{\mathbf{g}}\mathbf{u}_{\mathbf{g}}(t - \Delta t) + (\Delta t)^2\mathbf{f}_{\mathbf{g}} + (\Delta t)^2\mathbf{b}_{\mathbf{g}}$$

If the boundary conditions, equation (7.11), at $x = 0$ or $x = L$ are essential boundary conditions the global system model, equation (7.43), must be augmented to enforce the boundary conditions. The boundary conditions can be enforced by rewriting the system of equations.

For example, consider the original algebraic set, equation (7.43) expanded in equation (7.44).

$$\begin{bmatrix} m_{1,1} & m_{1,2} & \cdots & m_{1,m} \\ & m_{2,2} & \cdots & m_{2,m} \\ & & \ddots & \vdots \\ SYM & & & m_{m,m} \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{Bmatrix} = \begin{Bmatrix} rhs_1 \\ rhs_2 \\ \vdots \\ rhs_m \end{Bmatrix} \tag{7.44}$$

If equation (7.44) is subject to two essential boundary conditions of the form:

$$\begin{aligned} u_1 &= A \\ u_m &= B \end{aligned} \tag{7.45}$$

The system can be constrained by rewriting the u_1 and u_m equations as shown in equation (7.46).

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ m_{2,1} & m_{2,2} & \cdots & m_{2,m-1} & m_{2,m} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ m_{m-1,1} & m_{m-1,2} & \cdots & m_{m-1,m-1} & m_{m-1,m} \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{m-1} \\ u_m \end{Bmatrix} = \begin{Bmatrix} A \\ rhs_2 \\ \vdots \\ rhs_{m-1} \\ B \end{Bmatrix} \tag{7.46}$$

After the augmented system has been constructed the system response can be determined using matrix inversion. Now that the standard FEA has been developed, Chapter

8 will modify it by using the B-spline basis functions instead of the polynomial shape functions developed in this chapter.

CHAPTER 8. MODIFIED FINITE ELEMENT ANALYSIS

Finite element analysis (FEA) was reviewed in Chapter 7, this technique can be used to obtain physically based simulations of virtual components. However, the response from the FEA is not in line with the ultimate goal of developing a physically based design system. As a result the standard FEA will be modified so that the response is more appropriate for a design system.

FEA will be modified by representing the virtual component's geometry with B-splines. The modified FEA will provide several advantages over the standard FEA. The designer will have accurate control of the virtual object's continuity. The time required for collision detection between the user's virtual hand and the virtual object can be reduced by using bounding box techniques. The response for modified FEA model can be computed at less computational expense than a standard FEA model with the same number of elements. The virtual object can be deformed using free form deformations.

This chapter will present some basic information about B-splines, the reader is referred to [2] for more details about this representation. A B-spline curve is a piece-wise continuous curve as shown in Figure 8.1. The *nth* degree curve, $\mathbf{c}(\zeta)$, is a parametric curve defined on $\zeta \in [0,1]$ which contains a given number of *nth* degree segments, $\mathbf{c}_1(\zeta)$, $\mathbf{c}_2(\zeta)$ and $\mathbf{c}_3(\zeta)$. The curve is generated by blending data points, typically called control points with a set of basis functions as shown in equation (8.1).

$$\mathbf{c}(\zeta) = \sum_{i=0}^{k-1} B_{i,n}(\zeta)\mathbf{p}_i \tag{8.1}$$

k is the number of control points

$\mathbf{c}(\zeta)$ are the coordinates of the curve at the parameter value ζ

$B_{i,n}(\zeta)$ are the basis functions at the parameter value ζ

$\mathbf{p}_i$ are the coordinates of the control points

Equation (8.1) is similar to equation (7.1), which is the standard interpolation scheme used in FEA. However, the B-spline basis functions must satisfy all of the shape function

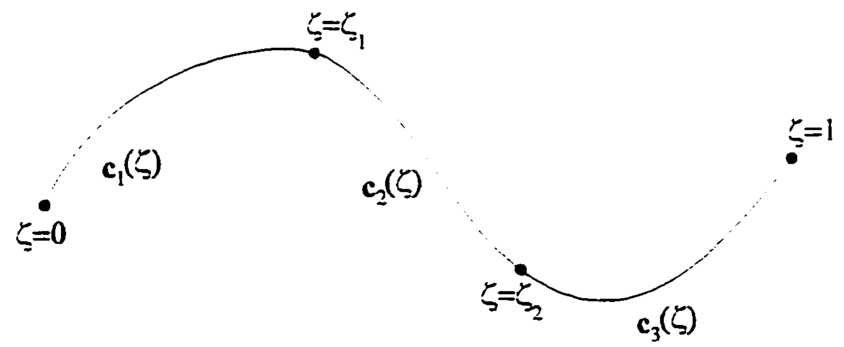


Figure 8.1. B-spline curve.

requirements specified in Chapter 7 in order to be used in FEA. The following sections will develop the B-spline basis functions and show that they are an acceptable FEA interpolation scheme.

B-spline Basis Functions

The B-spline basis functions are piece-wise continuous polynomial functions. In order to calculate the basis functions a non-decreasing sequence of real numbers called a knot vector must be specified as shown in equation (8.2).

$$Z = \{\zeta_0 \quad \zeta_1 \quad \cdots \quad \zeta_{m-1} \quad \zeta_m\} \tag{8.2}$$

$\zeta_i \leq \zeta_{i-1}$, for $i = 0, 1, \ldots, m-1$

The knot vector is essentially a list of special parameter values called breakpoints that bound the individual curve segments as shown in Figure 8.1. The knot vector is used to control the level of inter-segment continuity. The inter-segment continuity is, $n-r$. The integers n and r are the curve degree and the knot multiplicity respectively.

Non-periodic B-splines are used in this development, this means that the curve interpolates the first and last control points. This effect is obtained by repeating the first and last knots in the knot vector degree + 1 $(n+1)$ times. In addition uniform knot vectors are used, that is the interior knots are evenly spaced and have multiplicity of one. The length of the a uniform non-periodic knot vector, m, is $k+n$. The integers k and n are the number of control points and the degree of the B-spline respectively. Once the knot vector has been determined the basis functions can be evaluated.

The *ith* B-spline basis function of degree *n* is defined by the Cox-DeBor formulation as shown in equation (8.3).

$$
\begin{aligned}
B_{i,0}(\zeta) &= \begin{cases} 1 & \text{if}\left(\zeta_i \le \zeta < \zeta_{i+1}\right) \\ 0 & \text{otherwise} \end{cases} \\
B_{i,n}(\zeta) &= \frac{\zeta - \zeta_i}{\zeta_{i+n} - \zeta_i} B_{i,n-1}(\zeta) + \frac{\zeta_{i+n+1} - \zeta}{\zeta_{i+n+1} - \zeta_{i+1}} B_{i+1,n-1}(\zeta)
\end{aligned}
\tag{8.3}
$$

There may be situations where equation (8.3) leads to division by zero; this problem is eliminated by the definition shown in equation (8.4).

$$
\frac{0}{0} = 0 \tag{8.4}
$$

The basis functions are constructed iteratively starting with the step function, $B_{i,0}(\zeta)$ as shown in Figure 8.2. Now that the basis functions have been developed, they can be tested to ensure that they comply with all of the shape function requirements specified in Chapter 7.

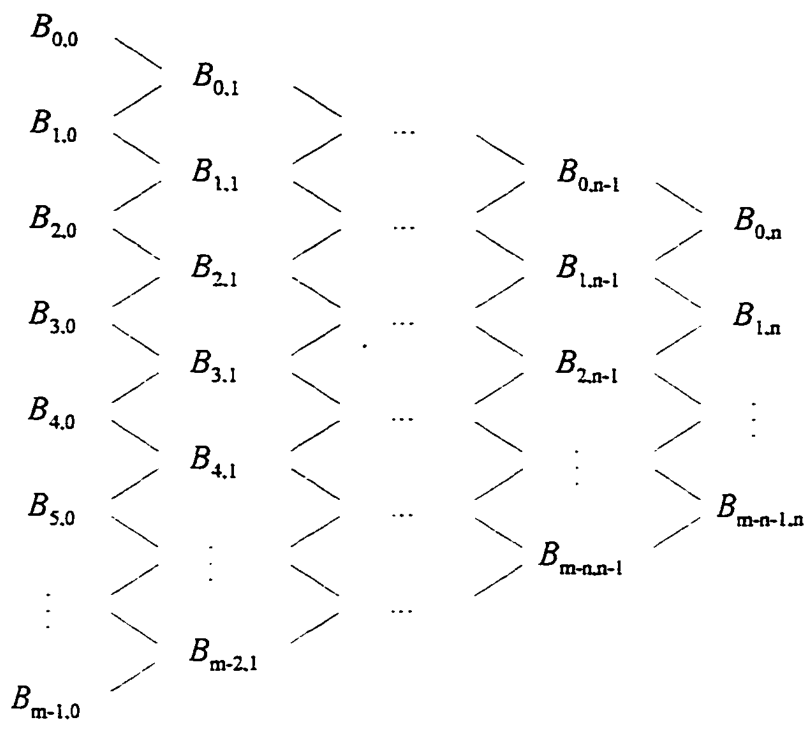


Figure 8.2. Triangular basis function table.

Shape Function Requirements

The first requirement is that the shape functions must interpolate the nodal displacements, equation (7.2). In the standard FEA the nodes are points along the surface of the virtual component; therefore, the approximate solution should pass through the nodes. However, this is not the case in the modified FEA. B-splines do not in general interpolate the control points, with the exception of the first and last control points due to non-periodic basis functions. The modified FEA does not supply nodes along the surface to be interpolated but supplies control points that are blended to yield the surface. Although the B-spline basis functions do not satisfy this requirement, it does not truly apply to them.

The primary shape function requirements from Chapter 7 are:

1. Shape functions derived from simple polynomials must be balanced with respect to all coordinate axes.
2. The shape functions must have acceptable continuity between elements.
3. The shape function must be complete with respect to the system being modeled.

The B-spline basis functions are simple polynomials so they must be balanced with respect to all coordinate axes. This requirement is automatically satisfied for B-spline curves. In addition, it can be shown that the B-spline tensor products used to represent surfaces and volumes also meet this requirement. In addition, the degree and knot vector multiplicity can be selected such that the B-spline basis functions have acceptable inter-elemental continuity and are complete with respect to the system being modeled.

The secondary shape function requirements from Chapter 7 are:

$$\begin{aligned}
&\sum_i N_i = 1\\
&\sum_i N_i x_i = x\\
&\sum_i N_i y_i = y\\
&\sum_i N_i z_i = z
\end{aligned} \tag{8.5}$$

The B-spline basis functions have a partition of unity property such that for an arbitrary knot span $[\zeta_i, \zeta_{i+1})$:

$$\sum_{j} B_{j,n}(\zeta) = 1 \text{ for all } \zeta \in [\zeta_i, \zeta_{i+1}) \tag{8.6}$$

The remaining three requirements are satisfied by expanding the vector notation of equation (8.1)

$$\begin{Bmatrix} c_x \\ c_y \\ c_z \end{Bmatrix}(\zeta) = \sum_{j} B_{j,n}(\zeta) \begin{Bmatrix} p_x \\ p_y \\ p_z \end{Bmatrix}_j \tag{8.7}$$

$c_x(\zeta)$ is the x coordinate of the curve at the parameter value ζ

$c_y(\zeta)$ is the y coordinate of the curve at the parameter value ζ

$c_z(\zeta)$ is the z coordinate of the curve at the parameter value ζ

p_x is the x coordinate of a control point

p_y is the y coordinate of a control point

p_z is the z coordinate of a control point

The B-spline basis functions fulfill the shape function requirements outlined in Chapter 7; therefore, they can be used to generate a finite element model. However, there is some ambiguity in what constitutes an element in the modified FEA, which will be addressed in the following section.

B-spline Elements

The triangular basis function table, Figure 8.2, and the knot vector, equation (8.2) are used to determine the nature of the B-spline element. To that end, consider a quadratic B-spline defined by four control points. The knot vector has a length of 6 and is shown in equation (8.8).

$$\begin{aligned} Z &= \{\zeta_0 \quad \zeta_1 \quad \zeta_2 \quad \zeta_3 \quad \zeta_4 \quad \zeta_5 \quad \zeta_6\} \\ Z &= \{0 \quad 0 \quad 0 \quad 0.5 \quad 1 \quad 1 \quad 1\} \end{aligned} \tag{8.8}$$

The B-spline curve defined by this knot vector has two segments associated with the two non-zero knot spans, $0 \le \zeta < 0.5$ and $0.5 \le \zeta < 1$. The triangle basis function table can be constructed for this knot vector as shown in Figure 8.3. For the first knot span $0 \le \zeta < 0.5$:

$$B_{i,0} = \begin{cases} 1 & \text{if } i = 2 \\ 0 & \text{if } i \ne 2 \end{cases} \tag{8.9}$$

As a result only $B_{0,2}$, $B_{1,2}$ and $B_{2,2}$ are non-zero. Therefore the first segment is only affected by the first three control points. Similarly for the second knot span $0.5 \le \zeta < 1$:

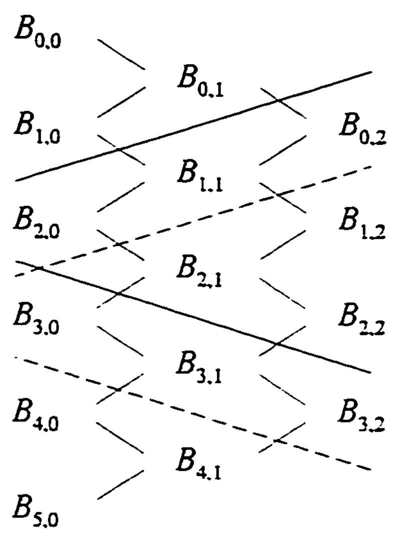


Figure 8. 3. Example triangular basis function table.

$$B_{i,0} = \begin{cases} 1 & \text{if } i = 3 \\ 0 & \text{if } i \neq 3 \end{cases} \tag{8.10}$$

As a result only $B_{1,2}$, $B_{2,2}$ and $B_{3,2}$ are non-zero. Therefore the second segment is only affected by the last three control points. The curve blend equation (8.1) can therefore be rewritten as shown in equation (8.11)

$$\mathbf{c}_j(\zeta) = \sum_{i=span-n}^{span} B_{i,n}(\zeta)\mathbf{p}_i \tag{8.11}$$

The function $\mathbf{c}_j(\zeta)$ is the curve segment associated with the knot span indicated by the integer *span*. Equation (8.11) shows that the curve segments are the B-spline elements. The number of elements in a modified FEA model is therefore determined by the degree and number of control points used to define the B-spline.

Now that the modified FEA elements have been established, the model can be developed using the elemental description derived in Chapter 7 and the B-spline basis functions described earlier. The construction of the elemental matrices will be clarified by developing a generic elemental stiffness matrix.

$$\mathbf{K}_e^{'} = \int_{\bar{u}_i}^{\bar{u}_{i+1}} \begin{bmatrix} k\frac{dB_{0,n}}{d\zeta}\frac{d\zeta}{dx}\frac{dB_{0,n}}{d\zeta}\frac{d\zeta}{dx} & k\frac{dB_{0,n}}{d\zeta}\frac{d\zeta}{dx}\frac{dB_{1,n}}{d\zeta}\frac{d\zeta}{dx} & \cdots & k\frac{dB_{0,n}}{d\zeta}\frac{d\zeta}{dx}\frac{dB_{n,n}}{d\zeta}\frac{d\zeta}{dx} \\ & k\frac{dB_{1,n}}{d\zeta}\frac{d\zeta}{dx}\frac{dB_{1,n}}{d\zeta}\frac{d\zeta}{dx} & \cdots & k\frac{dB_{1,n}}{d\zeta}\frac{d\zeta}{dx}\frac{dB_{n,n}}{d\zeta}\frac{d\zeta}{dx} \\ & & \ddots & \vdots \\ SYM & & & k\frac{dB_{n,n}}{d\zeta}\frac{d\zeta}{dx}\frac{dB_{n,n}}{d\zeta}\frac{d\zeta}{dx} \end{bmatrix} d\zeta \frac{dx}{d\zeta} \qquad (8.12)$$

The prime notation is used to indicate a modified FEA matrix, not a derivative. The other elemental matrices are constructed in the same manner. The elemental matrices are then assembled using the technique presented in Chapter 7. After assembly, a global system of the following form is obtained.

$$\mathbf{M}_g^{'}\ddot{\mathbf{p}}_g(t) + \mathbf{K}_g^{'}\mathbf{p}_g(t) = \mathbf{f}_g^{'} + \mathbf{b}_g^{'} \qquad (8.13)$$

Equation (8.13) can then be solved using the solution method developed in Chapter 7. Now that both the standard FEA and the modified FEA have been developed, the performance of the two methods can be compared.

CHAPTER 9. COMPARISON OF METHODS

The modified finite element analysis (FEA) will be compared to the standard FEA by constructing a model for the taut string shown in Figure 9.1. The unforced response of a taut string is defined by the one-dimensional wave equation shown in equation (9.1).

$$T\frac{\partial^2 u}{\partial x^2} = p\frac{\partial^2 u}{\partial t^2} \quad 0 \le x \le L,\ 0 \le t \tag{9.1}$$

The coefficients T, p and L are the tension applied to the string, the mass per unit length and the length of the string respectively. Equation (9.1) is subject to two essential boundary conditions.

$$\begin{aligned} u(0) &= 0 \\ u(L) &= 0 \end{aligned} \tag{9.2}$$

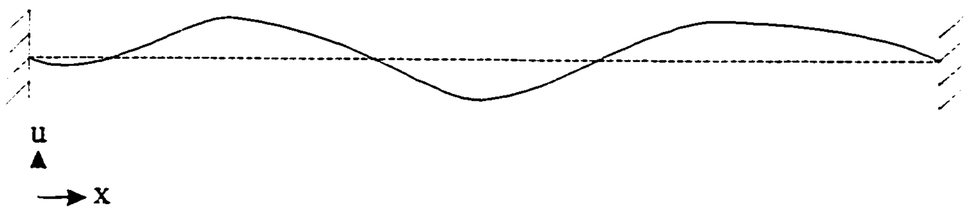


Figure 9.1. Taut string.

The next two sections will develop the string model for the standard FEA and the modified FEA.

Standard Finite Element Model

The standard finite element model will use two quadratic elements. The shape functions must be determined, using the techniques from Chapter 7, before the model is developed. The three quadratic shape functions for the parent element are shown in equations (9.3).

$$\begin{aligned} N_1 &= 2\psi^2 - 3\psi + 1 \\ N_2 &= -4\psi^2 + 4\psi \\ N_3 &= 2\psi^2 - \psi \end{aligned} \tag{9.3}$$

The variable ψ is the parameter in the local coordinate system. The shape functions are shown graphically in Figure 9.2. The shape functions can now be used to calculate the elemental matrices. The two elemental mass matrices are the same and are shown in equation (9.4).

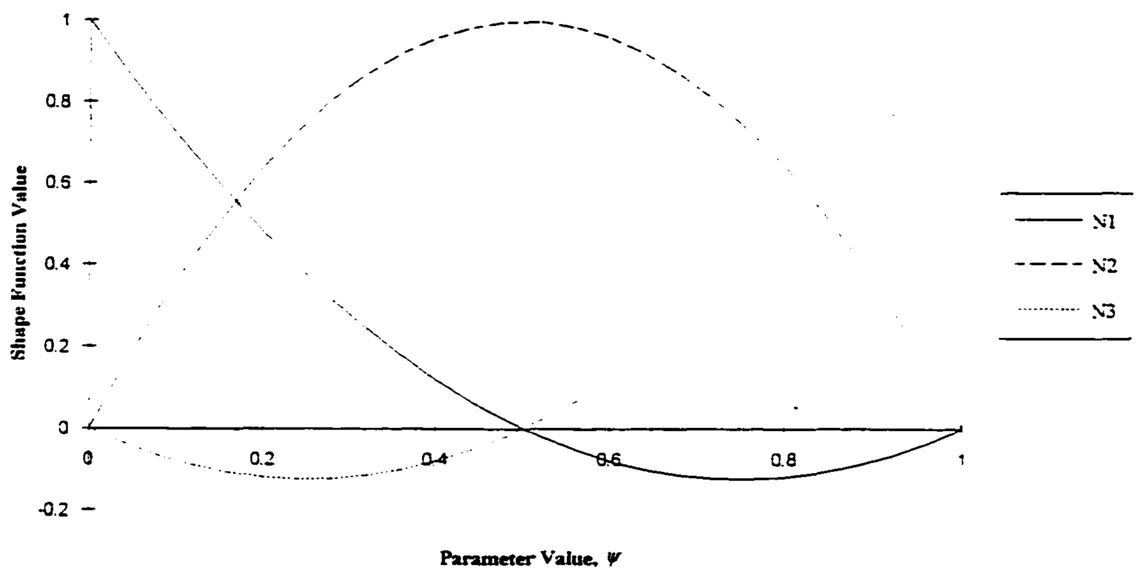


Figure 9.2. Shape functions.

$$\mathbf{M}_e^1 = \mathbf{M}_e^2 = \frac{\rho L}{60}\begin{bmatrix} 4 & 2 & -1 \\ 2 & 16 & 2 \\ -1 & 2 & 4 \end{bmatrix} \tag{9.4}$$

The two elemental stiffness matrices are also the same and are shown in equation (9.5).

$$\mathbf{K}_e^1 = \mathbf{K}_e^2 = \frac{2T}{3L}\begin{bmatrix} 7 & -8 & 1 \\ -8 & 16 & -8 \\ 1 & -8 & 7 \end{bmatrix} \tag{9.5}$$

The global system model is constructed by assembling the elemental matrices.

$$\frac{pL}{60}\begin{bmatrix} 4 & 2 & -1 & 0 & 0 \\ & 16 & 2 & 0 & 0 \\ & & 8 & 2 & -1 \\ & & & 16 & 2 \\ SYM & & & & 4 \end{bmatrix}\begin{Bmatrix} \ddot{u}_1 \\ \ddot{u}_2 \\ \ddot{u}_3 \\ \ddot{u}_4 \\ \ddot{u}_5 \end{Bmatrix}+\frac{2T}{3L}\begin{bmatrix} 7 & -8 & 1 & 0 & 0 \\ & 16 & -8 & 0 & 0 \\ & & 14 & -8 & 1 \\ & & & 16 & -8 \\ SYM & & & & 7 \end{bmatrix}\begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{Bmatrix}=\mathbf{0} \quad (9.6)$$

The global system model is now ready to be solved. The next section will develop a modified FEA model for the same string.

Modified Finite Element Model

The modified FEA model will use a quadratic B-spline defined by five control points. The knot vector for this B-spline is shown in equation (9.7).

$$Z=\begin{Bmatrix} 0 & 0 & 0 & \frac{1}{3} & \frac{2}{3} & 1 & 1 & 1 \end{Bmatrix} \quad (9.7)$$

The knot vector has three non-zero knot spans as a result the modified FEA model has three elements. Although the modified finite element model has one more element than the standard finite element model, both models are solved with the same computational expense. Once again the B-spline basis functions must be determined, using the techniques from Chapter 8, before the model is developed. The basis functions for the first non-zero knot span, $0 \le \zeta < \frac{1}{3}$, are shown in equations (9.8).

$$\begin{aligned}
B_{0,2} &= 9\zeta^2 - 6\zeta + 1 \\
B_{1,2} &= -\tfrac{27}{2}\zeta^2 + 6\zeta \\
B_{2,2} &= \tfrac{9}{2}\zeta^2 \\
B_{3,2} &= 0 \\
B_{4,2} &= 0
\end{aligned} \tag{9.8}$$

The basis functions for the second non-zero knot span, $\frac{1}{3} \leq \zeta < \frac{2}{3}$, are shown in equations (9.9).

$$\begin{aligned}
B_{0,2} &= 0 \\
B_{1,2} &= \tfrac{9}{2}\zeta^2 - 6\zeta + 2 \\
B_{2,2} &= -9\zeta^2 + 9\zeta - \tfrac{3}{2} \\
B_{3,2} &= \tfrac{9}{2}\zeta^2 - 3\zeta + \tfrac{1}{2} \\
B_{4,2} &= 0
\end{aligned} \tag{9.9}$$

The basis functions for the third non-zero knot span, $\frac{2}{3} \leq \zeta < 1$, are shown in equations (9.10).

$$\begin{aligned}
B_{0,2} &= 0 \\
B_{1,2} &= 0 \\
B_{2,2} &= \tfrac{9}{2}\zeta^2 - 9\zeta + \tfrac{9}{2} \\
B_{3,2} &= -\tfrac{27}{2}\zeta^2 + 21\zeta - \tfrac{15}{2} \\
B_{4,2} &= 9\zeta^2 - 12\zeta + 4
\end{aligned} \tag{9.10}$$

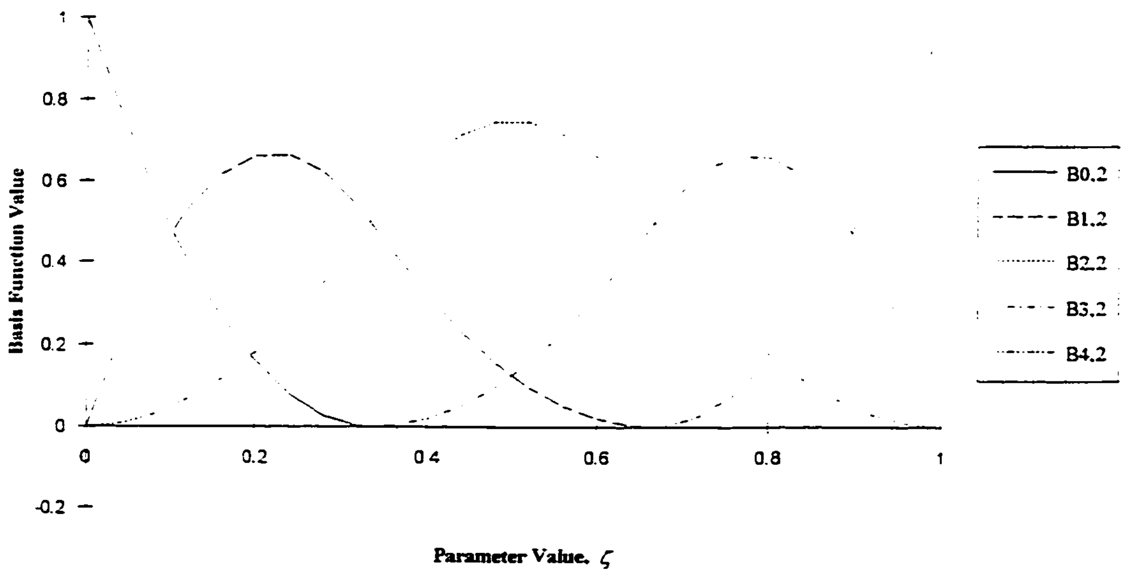


Figure 9.3. B-spline basis functions.

The basis functions are shown graphically in Figure 9.3. The basis functions can now be used to calculate the elemental matrices. The three elemental mass matrices are shown in equations (9.11), (9.12) and (9.13).

$$\mathbf{M}_e^{1'} = \frac{pL}{480}\begin{bmatrix} 44 & 23 & 3 \\ 23 & 47 & 15 \\ 3 & 15 & 7 \end{bmatrix} \tag{9.11}$$

$$\mathbf{M}_e^{2'} = \frac{pL}{480}\begin{bmatrix} 6 & 13 & 1 \\ 13 & 54 & 13 \\ 1 & 13 & 6 \end{bmatrix} \tag{9.12}$$

$$\mathbf{M}_e^{3'} = \frac{pL}{480}\begin{bmatrix} 7 & 15 & 3 \\ 15 & 47 & 23 \\ 3 & 23 & 44 \end{bmatrix} \tag{9.13}$$

The three elemental stiffness matrices are shown in equations (9.14), (9.15) and (9.16).

$$\mathbf{K}_e^{1'} = \frac{4T}{L}\begin{bmatrix} 0.7726 & -0.5452 & -0.2274 \\ -0.5452 & 0.5904 & -0.0452 \\ -0.2274 & -0.0452 & 0.2726 \end{bmatrix} \tag{9.14}$$

$$\mathbf{K}_e^{2'} = \frac{4T}{L}\begin{bmatrix} 0.3333 & -0.1667 & -0.1667 \\ -0.1667 & 0.3333 & -0.1667 \\ -0.1667 & -0.1667 & 0.3333 \end{bmatrix} \tag{9.15}$$

$$\mathbf{K}_e^{3'} = \frac{4T}{L}\begin{bmatrix} 0.2726 & -0.0452 & -0.2274 \\ -0.0452 & 0.5904 & -0.5452 \\ -0.2274 & -0.5452 & 0.7726 \end{bmatrix} \tag{9.16}$$

The global system model is constructed by assembling the elemental matrices.

$$\frac{pL}{480}\begin{bmatrix} 44 & 23 & 3 & 0 & 0 \\ & 53 & 28 & 1 & 0 \\ & & 68 & 28 & 3 \\ & & & 53 & 23 \\ SYM & & & & 44 \end{bmatrix}\begin{Bmatrix} \ddot{u}_1 \\ \ddot{u}_2 \\ \ddot{u}_3 \\ \ddot{u}_4 \\ \ddot{u}_5 \end{Bmatrix} + \frac{4T}{L}\begin{bmatrix} 0.7726 & -0.5452 & -0.2274 & 0 & 0 \\ & 0.9237 & -0.2118 & -0.1667 & 0 \\ & & 0.8785 & -0.2118 & -0.2274 \\ & & & 0.9237 & -0.5452 \\ SYM & & & & 0.7726 \end{bmatrix}\begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{Bmatrix} = 0 \tag{9.17}$$

The next two sections will compare the models modal response and dynamic response respectively.

Modal Response

The standard and modified finite element models are first analyzed to determine their modal response. The modal analysis will determine how accurate the approximate models are when compared with the exact solution [48]. Modal analysis is performed by solving the generalized eigenvalue problem with the model's mass and stiffness matrices. The following parameters are used for the string.

$$\begin{aligned} T &= 1 \\ p &= 1 \\ L &= 1 \end{aligned} \tag{9.18}$$

A comparison of system's natural frequencies is shown in Table 9.1. The percent error in each of the natural frequencies is shown in Table 9.2. The mode shapes associated with these natural frequencies are shown in Figures 9.4, 9.5 and 9.6.

The modal analysis has shown that both FEA models are accurate for the first two modes of vibration. Neither model is accurate for the third mode of vibration; although the modified FEA model clearly presents a better response. However this result is expected, the accuracy of the models decreases as the mode number increases, for a given model size.

Table 9.1. Natural frequencies, *rad/s*.

Mode	Exact	Standard FEA	Modified FEA
1	3.1416	3.1534	3.1431
2	6.2832	6.3246	6.3451
3	9.4248	11.3456	10.1011

Table 9.2. Percent error in natural frequencies.

Mode	Standard FEA	Modified FEA
1	0.38	0.05
2	0.66	0.99
3	20.38	7.18

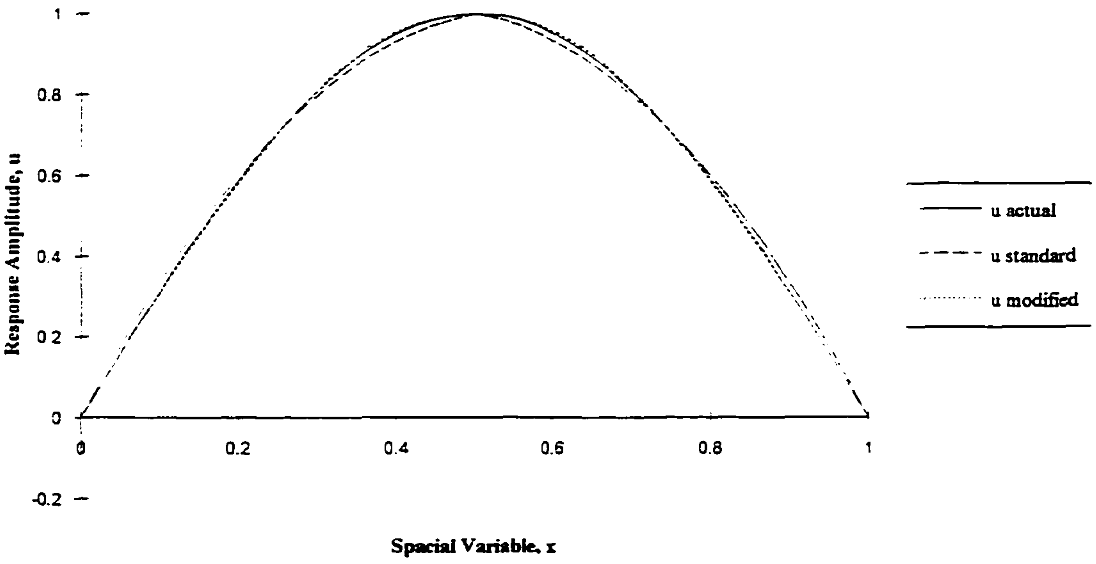


Figure 9.4. Mode 1.

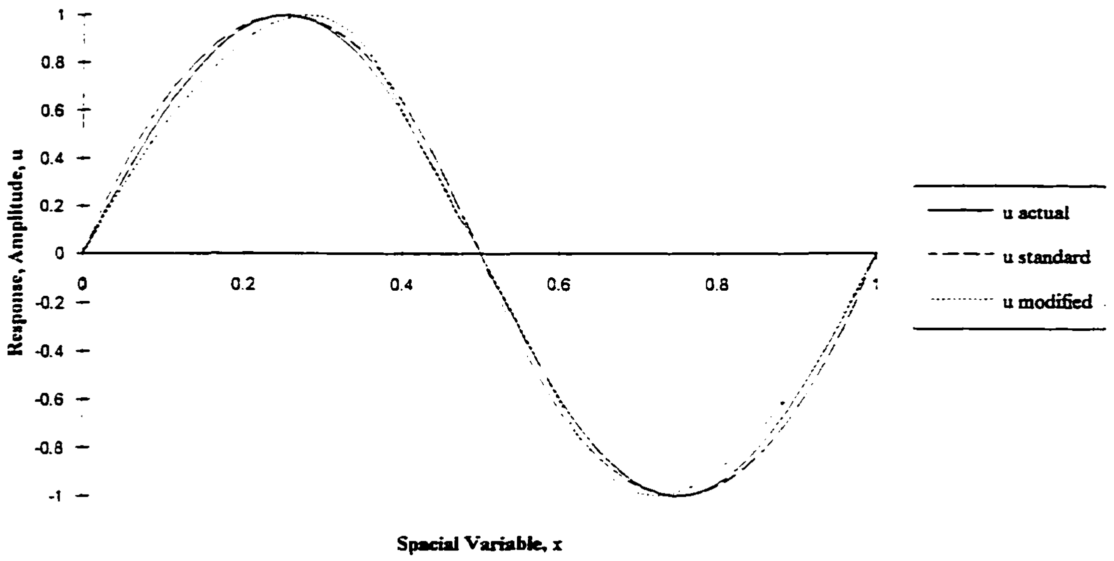


Figure 9.5. Mode 2.

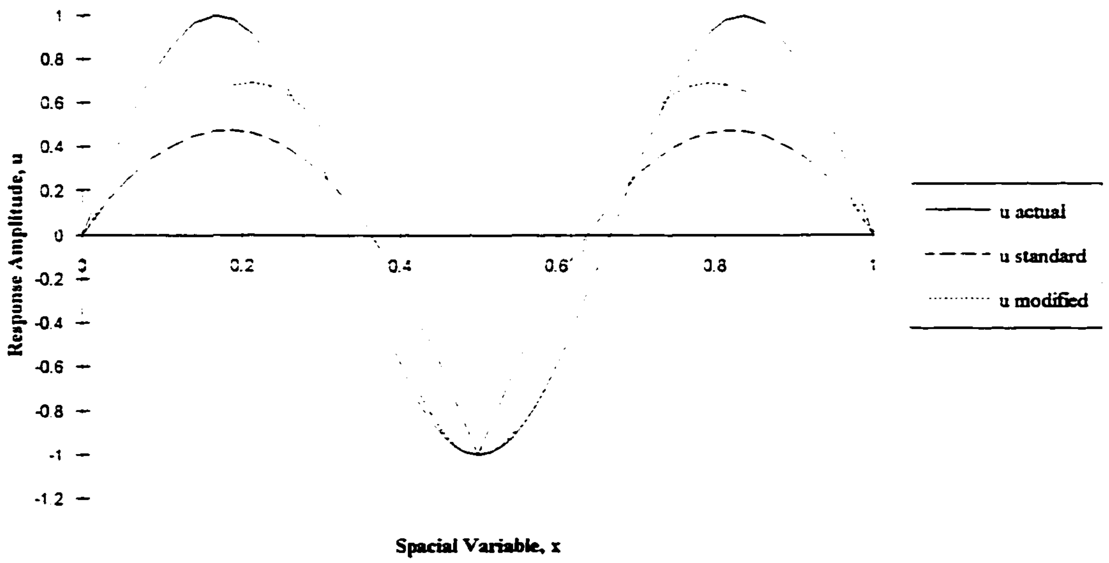


Figure 9.6. Mode 3.

Dynamic Response

The dynamic response of the string models is obtained by using the constrained system model, equation (7.44). The dynamic response of the standard and modified finite element models will be demonstrated for the first mode of vibration with $\Delta t = 0.2$ seconds. Figures 9.7 and 9.8 show the dynamic response of the standard FEA model and the modified FEA model respectively. The system was excited by placing the string into the first mode and releasing.

Figures 9.7 and 9.8 show that the dynamic response calculated by the central difference approximation accurately represents the actual dynamic response even with a coarse Δt. The accuracy of the central difference approximation increases as the Δt is refined

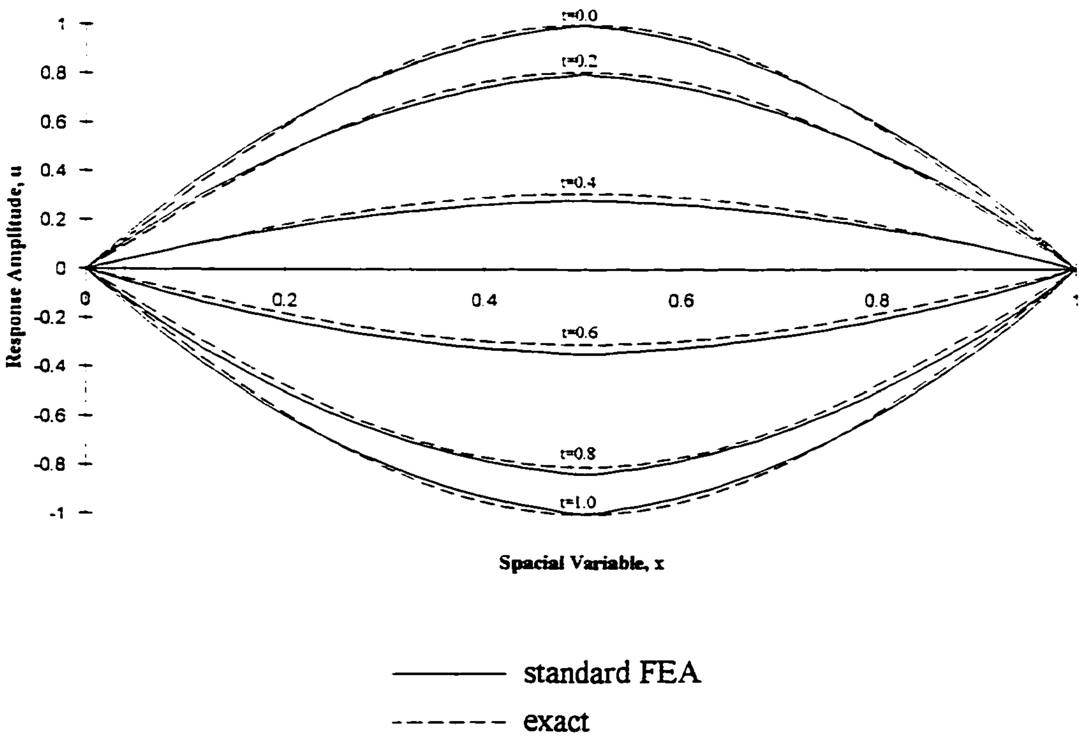
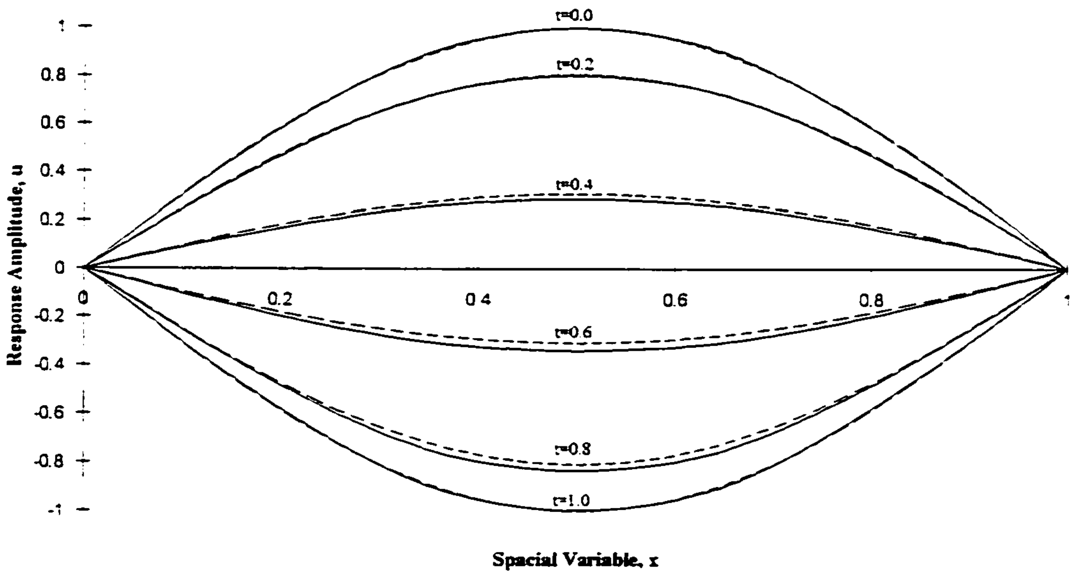


Figure 9. 7. Dynamic response of standard FEA model.



——— modified FEA

------ exact

Figure 9.8. Dynamic response of modified FEA model.

because a more accurate estimate of the system's velocity and acceleration are obtained.

Both the modal and dynamic analyses have verified that the modified FEA is an acceptable substitute for the standard FEA. Chapter 10 will describe how a modified finite element model can be deformed as well as how the force associated with the deformation can be obtained.

CHAPTER 10. MODEL DEFORMATION

Chapter 9 compared the modified finite element analysis (FEA) with the standard FEA and showed that the modified FEA is an acceptable substitute with some advantages in accuracy and continuity. However, the modified FEA offers other advantages, which makes it a more appropriate representation for use in a modeling system. These advantages arise from how the model is deformed. The following sections will describe how a modified finite element model can be deformed locally and globally.

Local Deformation

The local deformation system is used to ensure that the user's finger remains in contact with the virtual component during the deformation process. The local deformation system is a free form deformation (FFD) technique [45][90]. The problem is to determine the change in the control points required to move one point on the B-spline curve from its original location to a new final position.

The original point on the B-spline curve, $c(\zeta)$, is defined by equation (10.1).

$$c(\zeta_t) = \{B_{0,n}(\zeta_t) \quad B_{1,n}(\zeta_t) \quad \cdots \quad B_{n,n}(\zeta_t)\} \begin{Bmatrix} p_0 \\ p_1 \\ \vdots \\ p_n \end{Bmatrix} \tag{10.1}$$

The final position, $c(\zeta_t) + \Delta c(\zeta_t)$, is defined by equation (10.2).

$$c(\zeta_t) + \Delta c(\zeta_t) = \{B_{0,n}(\zeta_t) \quad B_{1,n}(\zeta_t) \quad \cdots \quad B_{n,n}(\zeta_t)\} \begin{Bmatrix} p_0 + \Delta p_0 \\ p_1 + \Delta p_1 \\ \vdots \\ p_n + \Delta p_n \end{Bmatrix} \tag{10.2}$$

When equations (10.2) and (10.1) are subtracted equation (10.3) is obtained.

$$\Delta c(\zeta_t) = \{B_{0,n}(\zeta_t) \quad B_{1,n}(\zeta_t) \quad \cdots \quad B_{n,n}(\zeta_t)\} \begin{Bmatrix} \Delta p_0 \\ \Delta p_1 \\ \vdots \\ \Delta p_n \end{Bmatrix} \tag{10.3}$$

Equation (10.3) allows the change in control points, $\Delta \mathbf{p}$, to be calculated based on the required change in the curve position, $\Delta c(\zeta_t)$. This problem is under-constrained with any number of acceptable control point configurations. Therefore the problem will be solved for the control point configuration, which minimizes the control point motion in the least squares sense. This solution is the pseudo-inverse solution to the equation (10.3) as shown in equation (10.4).

$$\begin{Bmatrix} B_{0,n}(\zeta_t) \\ B_{1,n}(\zeta_t) \\ \vdots \\ B_{n,n}(\zeta_t) \end{Bmatrix} \Delta c(\zeta_t) = \begin{Bmatrix} B_{0,n}(\zeta_t) \\ B_{1,n}(\zeta_t) \\ \vdots \\ B_{n,n}(\zeta_t) \end{Bmatrix} \begin{Bmatrix} B_{0,n}(\zeta_t) & B_{1,n}(\zeta_t) & \cdots & B_{n,n}(\zeta_t) \end{Bmatrix} \begin{Bmatrix} \Delta p_0 \\ \Delta p_1 \\ \vdots \\ \Delta p_n \end{Bmatrix} \tag{10.4}$$

$$\Delta \mathbf{p} = \left(\mathbf{B}^T\mathbf{B}\right)^{-1}\mathbf{B}^T\Delta\mathbf{c}$$

This technique allows the user to move any point on the B-spline curve to a new position. However, there is some difficulty in solving the system of equations (10.4). The matrix of basis functions, $\mathbf{B}^T\mathbf{B}$, is generally ill conditioned, if not singular. As a result, the singular value decomposition (SVD) method [83] must be used. The SVD technique tests the matrix condition as well as removes any singularities that are present. Therefore, SVD is the only acceptable method of solving equation (10.4).

Using the FFD technique the user can move one point on the B-spline at a time; however, it would be easier to obtain the desired shape if multiple points could be moved simultaneously. The FFD technique is general and does allow multiple points to be moved. The user can specify up to degree + 1 (n+1) points per B-spline element (segment) thus extending the design potential of this technique.

Any given point on a B-spline curve is only affected by a certain number of control points. Therefore the local deformation technique only specifies the positions of some of the control points. The positions of the remaining control points are determined by the global deformation system described in the following section.

Global Deformation

The global deformation system is used to established the position of any control point whose position was not specified by the local deformation system or the essential boundary conditions. The global deformation technique essentially solves a constrained version of the global system model equation (10.5).

$$\mathbf{M}_{\mathbf{g}}^{'}\mathbf{p}_{\mathbf{g}}(t+\Delta t)=\mathbf{rhs} \tag{10.5}$$

$$\mathbf{rhs}=\left[2\mathbf{M}_{\mathbf{g}}^{'}-(\Delta t)^{2}\mathbf{K}_{\mathbf{g}}^{'}\right]\mathbf{p}_{\mathbf{g}}(t)-\mathbf{M}_{\mathbf{g}}^{'}\mathbf{p}_{\mathbf{g}}(t-\Delta t)+(\Delta t)^{2}\mathbf{f}_{\mathbf{g}}^{'}+(\Delta t)^{2}\mathbf{b}_{\mathbf{g}}^{'}$$

The system is solved using the central difference approach developed in Chapter 7. However, the global system model is constrained in a different manner than discussed in Chapter 7. The global system model is constrained to enforce the essential boundary conditions as well as the control points specified by the local deformation system. The control points specified by the local deformation system are constrained in the same manner as the essential boundary conditions. For example, consider a five control point model with the first and fifth control points constrained for essential boundary conditions and the third control point constrained by the local deformation system as shown in equation (10.6).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} & m_{2,5} \\ 0 & 0 & 1 & 0 & 0 \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} & m_{4,5} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} p_0(t+\Delta t) \\ p_1(t+\Delta t) \\ p_2(t+\Delta t) \\ p_3(t+\Delta t) \\ p_4(t+\Delta t) \end{Bmatrix} = \begin{Bmatrix} A \\ rhs_2 \\ C \\ rhs_4 \\ B \end{Bmatrix} \tag{10.6}$$

The constants A, B and C are the essential boundary conditions and the constraint supplied by the local deformation system respectively. Therefore, the positions of the unspecified control points are determined by the dynamic response of the system subject to the constraints imposed by the local deformation system and the essential boundary conditions. This deformation method also allows the force required to constrain a control point to be calculated which will be discussed in the following section.

Force Determination

As seen in the previous section, the local and global deformation techniques yield a constrained system of equations, equation (10.6), from which the B-spline control points can be found. This two level deformation technique allows the control point forces to be determined. The control point forces are the forces required to keep the constrained control points in their positions. The control point forces are determined by substituting the control points found from the constrained system, equation (10.6), back into the unconstrained system of equations as shown in equation (10.7).

$$\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} & m_{1,5} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} & m_{2,5} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} & m_{3,5} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} & m_{4,5} \\ m_{5,1} & m_{5,2} & m_{5,3} & m_{5,4} & m_{5,5} \end{bmatrix} \begin{Bmatrix} p_0(t+\Delta t) \\ p_1(t+\Delta t) \\ p_2(t+\Delta t) \\ p_3(t+\Delta t) \\ p_4(t+\Delta t) \end{Bmatrix} = \begin{Bmatrix} rhs_1 \\ rhs_2 \\ rhs_3 \\ rhs_4 \\ rhs_5 \end{Bmatrix} + \Delta t^2 \begin{Bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{Bmatrix} \tag{10.7}$$

The vectors $\mathbf{p}(t+\Delta t)$ and $\mathbf{f}$ are the vector of control points determined from equation (10.6) and the vector of control point forces respectively.

The control point forces are the discrete forces that arise from deforming the modified FEA model. The control point forces are associated with a force distribution instead of a point force because the local deformation system has a finite radius of influence. However, an equivalent point force can be found for the force distribution using equation (10.8).

$$\overline{F} = \int f dx \tag{10.8}$$

The variable $\overline{F}$ is the equivalent point force for the force distribution, f.

In order to determine $\overline{F}$ the control point forces will be examined.

$$f_i = \int B_{i,n}(\zeta) f d\zeta \frac{dx}{d\zeta} \tag{10.9}$$

f_i is the *ith* control point force

$B_{i,n}(\zeta)$ is the *ith* B-spline basis function of degree n at the parameter value ζ

The sum of the control point forces is evaluated in equation (10.10).

$$\sum_i f_i = \int \left[\sum_i B_{i,n}(\zeta) \right] f d\zeta \frac{dx}{d\zeta} \tag{10.10}$$

Equation (10.10) can be simplified using the partition of unity property, $\sum_i B_{i,n}(\zeta) = 1$.

$$\sum_i f_i = \int f d\zeta \frac{dx}{d\zeta} = \int f dx \tag{10.11}$$

Therefore the equivalent point force, $\overline{F}$, can be evaluated using equation (10.12).

$$\overline{F} = \sum_i f_i \tag{10.12}$$

The point force, $\overline{F}$, has two components, one associated with the user and one associated with the constraints supplied by the boundary conditions.

$$\overline{F} = \overline{F}_u + \overline{F}_c \tag{10.13}$$

The force component associated with the user, $\overline{F}_u$, is the force that the haptic device needs to apply to the user's digit. Based on equation (10.12), $\overline{F}_u$ is the sum of the control point forces associated with the control points constrained by the local deformation system. Now that the force associated with deforming the virtual object has been determined it can be applied to the user with a haptic device.

CHAPTER 11. SURFACE MODEL

The taut string model has verified that FEA with the B-spline basis functions is an acceptable technique for obtaining physically based models of components whose shape is defined with the B-spline representation. However, because the dynamic string model has limited use, the concept was next extended to a dynamic surface. The surface model has the dynamic characteristics of a thin membrane. The thin membrane dynamic equation is similar to the dynamic model used for free-form shape design by Celniker [17]. However, this development will use the B-spline basis functions instead of the conventional FEA interpolation shape functions.

One minor difference must be addressed before constructing the physically based model for the thin membrane. The tensor product of basis functions, **N**, is a combination of basis functions for each of the two parametric directions.

$$N_{i+(n+1)j} = B(\zeta)_{i,n} B(\eta)_{j,m} \quad \text{for} \quad \begin{matrix} i = 0,1,\ldots,n \\ j = 0,1,\ldots,m \end{matrix} \tag{11.1}$$

The integers *n* and *m* are the degree of the B-spline surface in the two parametric directions. As a result the interpolation equation, equation (8.1) has a slightly different form.

$$\bar{u} = \sum_i p_i N_i \tag{11.2}$$

The physically based surface model can now be developed in the same way as the physically based curve model. The dynamics of a thin membrane are defined by the following equation [11].

$$\frac{\partial}{\partial x}\left(k_1 \frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(k_2 \frac{\partial u}{\partial y}\right) - p\frac{\partial^2 u}{\partial t^2} + f = 0 \tag{11.3}$$

The coefficients k_1 and k_2 are the stiffness coefficients in the x- and y- directions. By applying FEA techniques to equation (11.3), the components of the elemental description; $\mathbf{M}_e$, $\mathbf{K}_e$ and $\mathbf{f}_e$, can be evaluated by integrating the physical parameters and the tensor product of the B-spline basis functions.

$$\mathbf{M}_e = \iint_e p\mathbf{N}\mathbf{N}^T |\mathbf{J}| d\zeta d\eta \tag{11.4}$$

$$\mathbf{K}_e = \iint_e \mathbf{N}\begin{bmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \end{bmatrix} \mathbf{J}^{-T} \begin{bmatrix} k_1 & 0 \\ 0 & k_2 \end{bmatrix} \mathbf{J}^{-1} \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix} \mathbf{N}^T |\mathbf{J}| d\zeta d\eta \tag{11.5}$$

$$\mathbf{f}_e = \iint_e f \mathbf{N} |\mathbf{J}| d\zeta d\eta \tag{11.6}$$

The coefficients p, k and f are the mass density, restoring force and external force respectively. In addition the matrix, $\mathbf{J}$, is a Jacobian matrix that transforms the integral from Cartesian space into parametric space. Figures 11.1, 11.2 and 11.3 show a frames taken from a dynamic simulation in which the traveler was allowed to deform a surface using a wand tool. The wand can be driven with a three-dimensional mouse, magnetic tracker or a force feedback device.

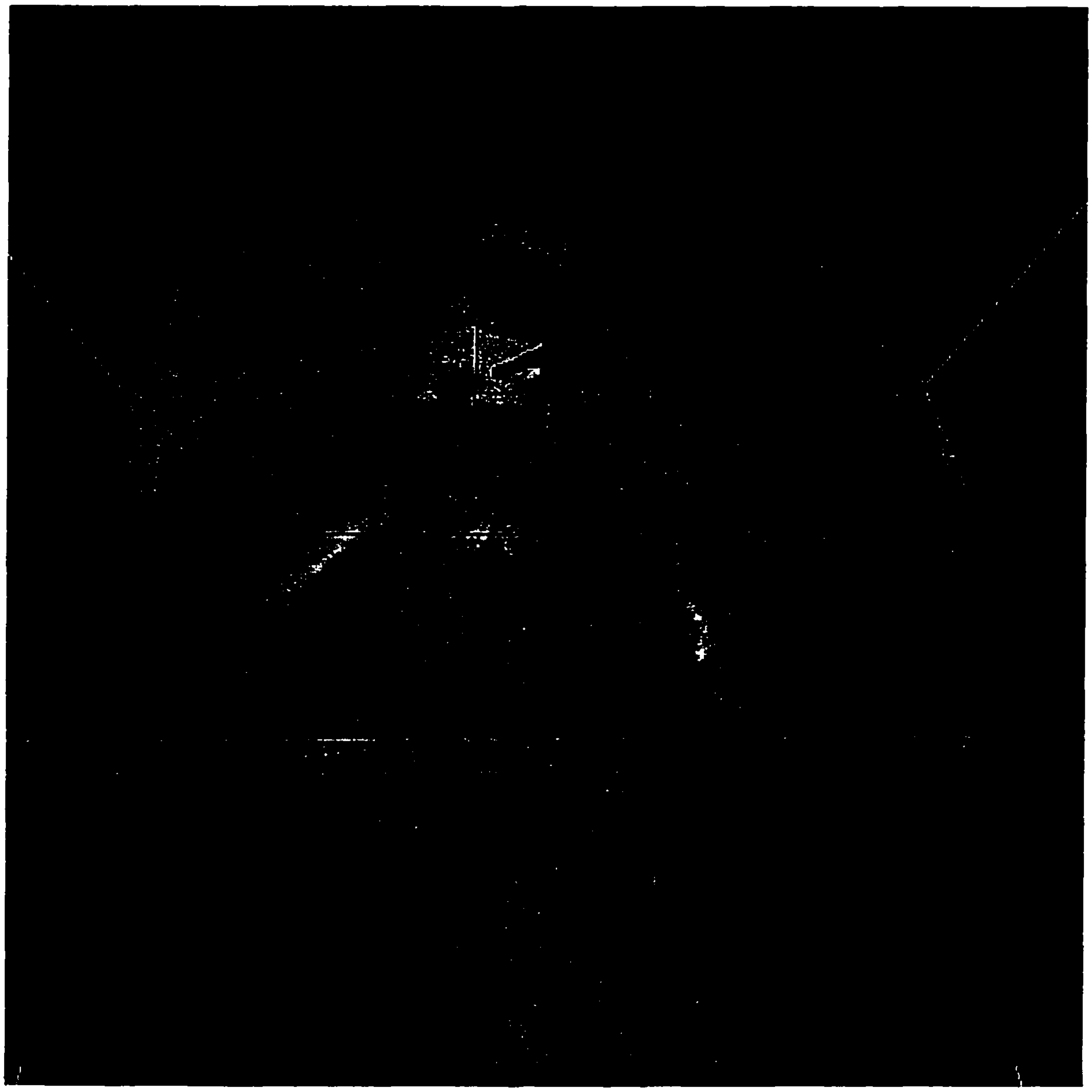


Figure 11.1. Undeformed Surface (image source: Pillsbury Corporation, "Pillsbury - Doughboy," Doughboy, June 3, 1996, www.doughboy.com/frameset.asp ?section=meet).

Figure 11.2. Initial deformation.

Figure 11.3. Resulting motion of dynamic model.

CHAPTER 12. CONCLUSIONS

The virtual manipulator control approach was developed to extend the state of the art in the area of force feedback for synthetic environments. The virtual manipulator control concept utilizes a six degree of freedom robot as the interface mechanism between the traveler and the computer running the synthetic environment. As a result the control concept reduces the isolation faced by researches in the area of force feedback. It is easier and less expensive to acquire a haptic display when a commercially available six degree of freedom robot can be used effectively. Control interface hardware and software can be obtained from numerous sources to control a general six degree of freedom robot. Finally, because any robot can be used, research results can be verified and extended at other facilities.

In addition it has been shown that a haptic display using a general six degree of freedom robot and the virtual manipulator control law can be incorporated into most if not all of the synthetic environments used today. Attention is given to projection style synthetic environments where the presence of the haptic display can occlude the images viewed by the traveler. By positioning the interface robot behind the traveler in the projection synthetic environment and extending a graphical representation of the interaction tool into the synthetic environment the haptic display will not occlude or diminish the visual display.

The virtual manipulator control law also makes contributions in the area of non-linear systems. The virtual manipulator control law operating on a general six degree of freedom robot is a highly non-linear system with an infinite continuum of equilibrium points. The design of multiple equilibrium point control laws is rarely done and the proof of stability of these systems is an important area. A proof of stability for a modified virtual manipulator control law was presented that verified the acceptable stability characteristics for the general class of modified virtual manipulator control laws. In addition the stability requirements were established for the original virtual manipulator control law. Although it was not possible to show stability for the entire class of original virtual manipulator control laws, it is easy to check the stability requirements for any given virtual manipulator.

Experimental results of several virtual manipulators have been presented. This presentation has shown how the control concept can be used to represent constraints. The time varying extension of the virtual manipulator concept was developed to represent complex general constraints shapes. In addition time varying virtual manipulators were also developed as a general interaction and exploration technique in synthetic environments. Finally, the inclusion of a virtual manipulator based haptic display into a visually immersive synthetic environment was demonstrated.

The results documented in this work have verified the ability of virtual manipulators to be used as a haptic display in synthetic environments. Although several virtual manipulators have been presented there is no limit on the number that can be developed. The development of new virtual manipulators is driven by the types of applications faced by synthetic environment developers. In addition this work has established the stability requirements for the "static" virtual manipulator control law, however an investigation of the time varying extension of the virtual manipulator concept is required. The proof of stability for the time varying virtual manipulator will probably place restrictions on how quickly the configuration of the virtual manipulator can change.

The virtual manipulator concept has potential to increase the use of force feedback in synthetic environments. The assimilation of haptic displays into synthetic environments provides an additional channel to provide information to the traveler. This information will be valuable in releasing the potential of synthetic environments for processing and interacting with computer data.

APPENDIX: COMPUTER CODE

Planar Circular Arc

```
// friction.c

#include "puma.h"

void friction(pumaFile* pumaData)
{
        int i;
        double tau=0.05305;

        if (pumaData->theta[0] > pumaData-
>theta_old[0]) pumaData->v_fric[0]=1.0;
        if (pumaData->theta[0] <= pumaData-
>theta_old[0]) pumaData->v_fric[0]=-0.9;
        pumaData->v_fric[0]=(pumaData-
>v_fric[0]*pumaData->dt+pumaData-
>v_fric_old[0]*tau)/(pumaData->dt+tau);

        if (pumaData->theta[1] > pumaData-
>theta_old[1])
        {
                if (pumaData->theta[1] > -1.57)
pumaData->v_fric[1]=-0.3;
                else pumaData->v_fric[1]=-0.9;
        }
        if (pumaData->theta[1] <= pumaData-
>theta_old[1])
        {
                if (pumaData->theta[1] > -1.57)
pumaData->v_fric[1]=0.9;
                else pumaData->v_fric[1]=0.6;
        }
        pumaData->v_fric[1]=(pumaData-
>v_fric[1]*pumaData->dt+pumaData-
>v_fric_old[1]*tau)/(pumaData->dt+tau);

        if (pumaData->theta[2] > pumaData-
>theta_old[2]) pumaData->v_fric[2]=0.47;
        if (pumaData->theta[2] <= pumaData-
>theta_old[2]) pumaData->v_fric[2]=-0.47;
        pumaData->v_fric[2]=(pumaData-
>v_fric[2]*pumaData->dt+pumaData-
>v_fric_old[2]*tau)/(pumaData->dt+tau);

        if (pumaData->theta[3] > pumaData-
>theta_old[3]) pumaData->v_fric[3]=-0.35;
        else if (pumaData->theta[3] <=
pumaData->theta_old[3]) pumaData-
>v_fric[3]=0.35;
        else pumaData->v_fric[3]=0.0;
        pumaData->v_fric[3]=(pumaData-
>v_fric[3]*pumaData->dt+pumaData-
>v_fric_old[3]*tau)/(pumaData->dt+tau);

        if (pumaData->theta[4] > pumaData-
>theta_old[4]) pumaData->v_fric[4]=-0.4;
        else if (pumaData->theta[4] < pumaData-
>theta_old[4]) pumaData->v_fric[4]=0.4;
        else pumaData->v_fric[4]=0.0;
        pumaData->v_fric[4]=(pumaData-
>v_fric[4]*pumaData->dt+pumaData-
>v_fric_old[4]*tau)/(pumaData->dt+tau);
```

```
        if (pumaData->theta[5] > pumaData-
>theta_old[5]) pumaData->v_fric[5]=-0.5;
        else if (pumaData->theta[5] < pumaData-
>theta_old[5]) pumaData->v_fric[5]=0.5;
        else  pumaData->v_fric[5]=0.0;
        pumaData->v_fric[5]=(pumaData-
>v_fric[5]*pumaData->dt+pumaData-
>v_fric_old[5]*tau)/(pumaData->dt+tau);

        for (i=0;i<6;i++)
        {
                pumaData-
>v_fric_old[i]=pumaData->v_fric[i];
        }
}
```

```
// gravity.c

#include "puma.h"

void gravity(pumaFile* pumaData)
{
        double c2,s2,c23,s23;

        c2=cos(pumaData->theta[1]);
        s2=sin(pumaData->theta[1]);

        c23=cos(pumaData->theta[1]+pumaData-
>theta[2]);
        s23=sin(pumaData->theta[1]+pumaData-
>theta[2]);

// gravity compensation
        pumaData->vg[0]=0.0;
        pumaData->vg[2]=-
1.1201*s23+0.0977*c23;
        pumaData-
>vg[1]=0.2400*s2+2.1144*c2-0.5304*pumaData-
>vg[2];
        pumaData->vg[3]=0.0;
        pumaData->vg[4]=0.0;
        pumaData->vg[5]=0.0;
}
```

```
// impedence.c

#include "puma.h"

void impedence(pumaFile* pumaData)
{
        pumaData-
>vim[0]=0.02*pow((1.0/(pumaData->theta[0]-
2.7)),3.0)+0.02*pow((1.0/(pumaData-
>theta[0]+2.7)),3.0);

        pumaData->vim[1]=-
0.02*pow((1.0/(pumaData->theta[1]-0.7)),3.0)-
0.02*pow((1.0/(pumaData->theta[1]+3.7)),3.0);

        pumaData-
>vim[2]=0.02*pow((1.0/(pumaData->theta[2]-
pumaData-
>jlimit3)),3.0)+0.02*pow((1.0/(pumaData-
>theta[2]+0.9)),3.0);

        pumaData->vim[3]=-
0.02*pow((1.0/(pumaData->theta[3]-3.2)),3.0)-
0.02*pow((1.0/(pumaData->theta[3]+1.8)),3.0);

        pumaData->vim[4]=-
0.02*pow((1.0/(pumaData->theta[4]-1.7)),3.0)-
0.02*pow((1.0/(pumaData->theta[4]+pumaData-
>jlimit5)),3.0);

        pumaData->vim[5]=-
0.02*pow((1.0/(pumaData->theta[5]-5.2)),3.0)-
0.02*pow((1.0/(pumaData->theta[5]+5.2)),3.0);
}
```

```
// main.c

#include "puma.h"

void main(void)
{
// robot stuff
        pumaFile *pumaData;
        int stop;
        int homecount;

// window's stuff
        HANDLE hprocess;
        HANDLE hthread;
        int processerror;

// timer stuff
        BOOL result;
        LARGE_INTEGER lifrequency;
        LARGE_INTEGER licount;
        LONGLONG frequency;
        double dfrequency;
        LONGLONG startcount;
        LONGLONG count;
        double currenttime;
        double dtactual;
        double dterror;
        double dtmax;

// error flags
        int timererror;
        int timeroverrun;
        int DeviceStop;
        int errorSocket;

// socket stuff
        int err;
        char szDataSend[100];
        int gcount;

// data file stuff
        double data[4][2000];
        int datalength=2000;
        int datacount;
        int datacycle;
        int datamax;
        int fileerror;
        FILE *out;

// general stuff (counter and the like)
        int i;

//////////////////////////////////////////////////////////////////////////
// Taking Care of Business
//////////////////////////////////////////////////////////////////////////

        printf("PUMA control program\n");
        printf("written by Jim Edwards for
LARCC\n");
        printf("All rights reserved\n\n\n\n");

//////////////////////////////////////////////////////////////////////////
// Code Initialization Section
//////////////////////////////////////////////////////////////////////////
// set counter error flag to pass
        timererror=1;

// set counter overrun flag to pass
        timeroverrun=1;

// start taking data at zero
        datacount=0;

// set data pass to zero
        datacycle=0;

// set process error flag to pass
        processerror=0;

// set maximum delta-t to zero
        dtmax=0.0;

// set stop to pass
        stop=1;

// set homecount to zero
        homecount=0;

// set socket error to none
        errorSocket=0;

// set graphics dump counter to zero
        gcount=0;

//////////////////////////////////////////////////////////////
////// Hardware Initialization
//////////////////////////////////////////////////////////////
// get process handle
        hprocess=GetCurrentProcess();

// set process priority
        result=SetPriorityClass(hprocess,
REALTIME_PRIORITY_CLASS);
        if (result == 0)  processerror=1;

// get thread handle
        hthread=GetCurrentThread();

// set thread priority
        result=SetThreadPriority(hthread,
THREAD_PRIORITY_TIME_CRITICAL);
        if (result == 0)  processerror=2;

// allocate memory for puma structure
        pumaData=(pumaFile
*)malloc(sizeof(pumaFile));

// connect to the puma kernel device
        DeviceStop=1;
        pumaData-
>PumaDevice=HwNewDevice(NULL);
        HwSetErrorHandler(pumaData-
>PumaDevice, MyErrorHandler);
```

```
        if (!HwConnectDevice(pumaData-
>PumaDevice, "puma"))
        {
                printf("Failed to connect to puma
device!\n");
                HwDeleteDevice(pumaData-
>PumaDevice);
                DeviceStop=0;
        }

// setup puma
        pumaInitialization(pumaData);

// open socket - useSocket = 1 use socket, = 0
don't use socket
        pumaData->useSocket=1;
        pumaData->activeSocket=0;
        openSocket(pumaData);

// test socket
        testSocket(pumaData);

// get frequency of high performance counter
        result=QueryPerformanceFrequency(&lifr
equency);
        if (result == TRUE)
        {
                frequency=lifrequency.QuadPart;
                dfrequency=((double)
frequency);
                printf("clock frequency: %f
MHz\n\n\n\n".dfrequency);
        }
        else
        {

        printf("QueryPerformanceFrequency:
failure\n");
                timererror=0;
        }

// get starting count
        printf("\n\n\nTurn Arm Power On!!!!\n");
        result=QueryPerformanceCounter(&licou
nt);
        if (result == TRUE)
        {
                startcount=licount.QuadPart;
        }
        else
        {

        printf("QueryPerformanceCounter:
failure\n");
                timererror=0;
        }

// disengage the brakes
        HwOutpw(pumaData->PumaDevice,
0x02e, 0x0001);

/////////////////////////////////////////////////////////////////
// Main Control Loop
/////////////////////////////////////////////////////////////////

        while((homecount < 2000) &&
(DeviceStop == 1) && (timererror == 1) &&
(timeroverrun == 1) && (processerror == 0))
        {
// control code
                if (kbhit()) stop=0;
                if (stop == 1)
                {
                pumaControl(pumaData);
                }
                else
                {
                homecount++;
                        pumaHome(pumaData);
                }

// increment graphics dump counter
                gcount++ ;

// send data to graphics engine
                if (gcount == 5)
                {
                gcount=0;
// but only if there is an active socket for
communication
                        if (pumaData-
>activeSocket == 1)
                {

        sprintf(szDataSend,"%4.3f %4.3f %4.3f
%4.3f %4.3f %4.3f %4.3f ",
                        pumaData->time,
                                pumaData-
>theta[0],
                                pumaData-
>theta[1],
                                pumaData-
>theta[2],
                              pumaData-
>theta[3],
                              pumaData-
>theta[4],
                              pumaData-
>theta[5]);

                    err=send(pumaData->hSock,
(LPSTR) szDataSend, 51, 0) ;
                              if
(err==SOCKET_ERROR) errorSocket=1;
                    }
                    }

// timing code
                    do
                    {
// get the current count of performance counter

          result=QueryPerformanceCounter(&licou
nt);
                         if (result == TRUE)
                         {

          count=licount.QuadPart;
// convert into time since program started

          currenttime=((double) (count-
startcount))/dfrequency;

                         }
                         else
                         {

          printf("QueryPerformanceCounter:
failure\n");
                              timererror=0;
                         }

                         dtactual=currenttime-
pumaData->time;
                    } while(dtactual < pumaData-
>dt);

// get maximum delta-t
                    if (dtactual > dtmax)
dtmax=dtactual;

// get error in delta-t
                    dterror=dtactual-pumaData->dt;
                    if (fabs(dterror) > pumaData-
>dt) timeroverrun=0;
```

```
// take some data
                    if (stop == 1)
                    {

          data[0][datacount]=pumaData->theta[1];

          data[1][datacount]=pumaData->theta[2];

          data[2][datacount]=pumaData->theta[4];

          data[3][datacount]=pumaData->vpos;

                         if (datacount == 1999)
                         {
                              datacount=0;
                              datacycle=1;
                         }
                         else  datacount++;
                    }

// update absolute time base
                    pumaData->time=pumaData-
>time+pumaData->dt;
          } // end main control loop

// engage the brakes
          HwOutpw(pumaData->PumaDevice,
0x02e, 0x0000);

/////////////////////////////////////////////////////////////////
// Hardware Clean-Up
/////////////////////////////////////////////////////////////////
// kernal device
          HwDeleteDevice(pumaData-
>PumaDevice);

// close socket
          closeSocket(pumaData);

/////////////////////////////////////////////////////////////////
// Take some data
/////////////////////////////////////////////////////////////////

// open the data file
          if ((out=fopen("out.dat","wt"))==NULL)
fileerror=0;
          else
          {
// write data
                    fileerror=1;
```

```
			fprintf(out,"max dt is
%f\n",dtmax);
			fprintf(out,"%f,
%f\n\n\n",pumaData->center[0],pumaData-
>center[1]);

			if (datacycle == 1)
datamax=datalength;
			else  datamax=datacount;

			for (i=0; i<datamax; i++)
			{
				fprintf(out,"%f, %f, %f,
%f\n",data[0][i],data[1][i],data[2][i],data[3][i]);
			}

// close file
			fclose(out);
		}

////////////////////////////////////////////////////////////////
// Final Error Messages
////////////////////////////////////////////////////////////////
	printf("\n\n\nError Messages:\n");
	if (timererror == 0)  printf("timer
malfunction\n");
	else if (timeroverrun == 0)  printf("timer
over run\n");
	else if (DeviceStop == 0)  printf("DriverX
error\n");
	else if (fileerror == 0)  printf("could not
open data file\n");
	else if (processerror == 1)  printf("could
not set process priority\n");
	else if (processerror == 2)  printf("could
not set thread priority\n");
	else if (errorSocket == 1)  printf("error
sending data over socket\n");
	else  printf("all went well\n");


	Sleep(3000);
}

/////////////////////////////////////////////////////////
////// DriverX Error Handler
/////////////////////////////////////////////////////////
void MyErrorHandler(HWDEVICE* pDevice,
DWORD nError)
{
	printf("Critical DriverX error: %d\n",
nError);
	exit(nError);
}
```

```
// puma.h

// include files
#include <windows.h>
#include <winsock.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include "DriverX.h"

// structures
typedef struct
{
// needed for all
	HWDEVICE* PumaDevice;
	double dt;
	double time;
	double encoder_scale[6];
	double encoder_offset[6];
	double theta[6];
	double voltage_out[6];

// virtual manipulator stuff
	double center[2];
	double eeold[3];

// socket stuff
	SOCKET hSock;
	int useSocket;
	int activeSocket;

// needed for me
	int first_flag;
	int last_flag;
	double kp[6];
	double kd[6];
	double error[6];
	double errorold[6];
	double errordot[6];
	double thetad[6];
	double theta_old[6];
	double thetao[6];
	double timeh;
	double vg[6];
	double v_fric[6];
	double v_fric_old[6];
	double vim[6];
```

```
        double jlimit3;
        double jlimit5;
        double vpos;
} pumaFile;

// prototypes
void main(void);
void MyErrorHandler(HWDEVICE * , DWORD);
void pumaInitialization(pumaFile *);
void pumaControl(pumaFile *);
void pumaHome(pumaFile *);
void openSocket(pumaFile *);
void closeSocket(pumaFile *);
void testSocket(pumaFile *);
void gravity(pumaFile *);
void friction(pumaFile *);
void impedence(pumaFile *);
```

```
// pumaControl.c

#include "puma.h"

void pumaControl(pumaFile* pumaData)
{
        short val[6];
        int voltage_int[6];
        int i;
        double thetaf[6];
        double tf=5.0;
        double c2, s2, c23, s23, c235, s235;
        double c35, s35, c5, s5;
        double xx[3], rr[3][3], J[3][3];
        double ew[3], ee[3], eedot[3];
        double L, xdes[3], thetav;
        double wn, z, Kp, Kv, F[3], Fm[3], T[3];
        int wall_flag=0;

// read encoders
        val[0]=HwInpw(pumaData-
>PumaDevice, 0x010);
        val[1]=HwInpw(pumaData-
>PumaDevice, 0x012);
        val[2]=HwInpw(pumaData-
>PumaDevice, 0x014);
        val[3]=HwInpw(pumaData-
>PumaDevice, 0x016);
        val[4]=HwInpw(pumaData-
>PumaDevice, 0x018);
        val[5]=HwInpw(pumaData-
>PumaDevice, 0x01a);

// convert encoders to radians
        for (i=0; i<6; i++)
        {
                pumaData->theta[i]=pumaData-
>encoder_scale[i]*(((double) val[i]) - pumaData-
>encoder_offset[i]);
        }

// gravity compensation
        gravity(pumaData);

// friction compensation
        friction(pumaData);

// impedence protection
        impedence(pumaData);

// Forward kinematics
        c2=cos(pumaData->theta[1]);
        s2=sin(pumaData->theta[1]);
        c23=cos(pumaData->theta[1]+pumaData-
>theta[2]);
        s23=sin(pumaData->theta[1]+pumaData-
>theta[2]);
        c235=cos(pumaData-
>theta[1]+pumaData->theta[2]+pumaData-
>theta[4]);
        s235=sin(pumaData-
>theta[1]+pumaData->theta[2]+pumaData-
>theta[4]);

        xx[0]=-
0.0203*c23+0.4331*s23+0.4318*c2;
        xx[1]=0.0203*s23+0.4331*c23-
0.4318*s2;
        xx[2]=pumaData->theta[1]+pumaData-
>theta[2]+pumaData->theta[4];

        rr[0][0]=c235;          rr[0][1]=0.0;
rr[0][2]=-s235;
        rr[1][0]=-s235;         rr[1][1]=0.0;
rr[1][2]=-c235;
        rr[2][0]=0.0;           rr[2][1]=1.0;
rr[2][2]=0.0;

// Evaluate the PUMA jacobian
        c35=cos(pumaData->theta[2]+pumaData-
>theta[4]);
        s35=sin(pumaData->theta[2]+pumaData-
>theta[4]);
        c5=cos(pumaData->theta[4]);
        s5=sin(pumaData->theta[4]);

        J[0][0]=0.4318*s35+0.4331*c5-
0.0203*s5;      J[0][1]=0.4331*c5-0.0203*s5;
        J[0][2]=0.0;
        J[1][0]=0.4318*c35-0.4331*s5-
0.0203*c5;      J[1][1]=-0.4331*s5-0.0203*c5;
        J[1][2]=0.0;
        J[2][0]=1.0;
                                J[2][1]=1.0;

        J[2][2]=1.0;

// first time through get current position
        if (pumaData->first_flag==1)
        {
                pumaData-
>thetao[0]=pumaData->theta[0];
                pumaData-
>thetao[1]=pumaData->theta[1];
                pumaData-
>thetao[2]=pumaData->theta[2];
                pumaData-
>thetao[3]=pumaData->theta[3];
                pumaData-
>thetao[4]=pumaData->theta[4];
                pumaData-
>thetao[5]=pumaData->theta[5];
                pumaData->first_flag=2;
        }

// final position
        thetaf[0]=0.0;
        thetaf[1]=-0.5;
        thetaf[2]=2.5;
        thetaf[3]=0.0;
        thetaf[4]=-0.4359;
        thetaf[5]=0.0;

// do cubic spline interpolation
        if (pumaData->time <= tf)
        {
                pumaData-
>thetad[0]=pumaData->thetao[0]-3.0*(pumaData-
>thetao[0]-thetaf[0])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[0]-thetaf[0])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                pumaData-
>thetad[1]=pumaData->thetao[1]-3.0*(pumaData-
>thetao[1]-thetaf[1])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[1]-thetaf[1])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                pumaData-
>thetad[2]=pumaData->thetao[2]-3.0*(pumaData-
>thetao[2]-thetaf[2])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[2]-thetaf[2])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                pumaData-
>thetad[3]=pumaData->thetao[3]-3.0*(pumaData-
>thetao[3]-thetaf[3])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[3]-thetaf[3])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                pumaData-
>thetad[4]=pumaData->thetao[4]-3.0*(pumaData-
>thetao[4]-thetaf[4])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[4]-thetaf[4])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                pumaData-
>thetad[5]=pumaData->thetao[5]-3.0*(pumaData-
>thetao[5]-thetaf[5])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[5]-thetaf[5])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
        }
// after tf stay put at final position
        else if (pumaData->time > tf)
        {
                pumaData->thetad[0]=thetaf[0];
                pumaData->thetad[1]=thetaf[1];
                pumaData->thetad[2]=thetaf[2];
                pumaData->thetad[3]=thetaf[3];
                pumaData->thetad[4]=thetaf[4];
                pumaData->thetad[5]=thetaf[5];
        }

//
// control section
//
        for (i=0;i<6;i++)
        {
// calculate error
                pumaData->error[i]=pumaData-
>thetad[i]-pumaData->theta[i];
```

```
// calculate rate of change of the error
                    pumaData-
>errordot[i]=(pumaData->error[i]-pumaData-
>errorold[i])/pumaData->dt;

// evaluate local PD control law
                    pumaData-
>voltage_out[i]=pumaData->kp[i]*pumaData-
>error[i]+pumaData->kd[i]*pumaData-
>errordot[i];
          }

// impedence based control law
          if (pumaData->time > 6.0)
          {
                    L=0.3;

                    if (pumaData->first_flag==2)
                    {
                              xdes[0]=xx[0];
                              xdes[1]=xx[1];
                              xdes[2]=xx[2];

                              pumaData-
>center[0]=xdes[0]-L;
                              pumaData-
>center[1]=xdes[1];
                              pumaData-
>first_flag=0;
                    }

// inverse kinematics of virtual manipulator
                    // position based
//                  thetav=atan2(-xx[1]+pumaData-
>center[1],xx[0]-pumaData->center[0]);

                    // orientation based
                    thetav=xx[2]-1.57;

                    pumaData->vpos=thetav;

// check joint limits of virtual manipulator
                    if (thetav >= 0.0)
                    {
                              thetav=0.0;
                              wall_flag=1;
                    }
                    else if (thetav <= -1.0)
                    {
                              thetav=-1.0;
                              wall_flag=1;
                    }

// forward kinematics of virtual manipulator

          xdes[0]=L*cos(thetav)+pumaData-
>center[0];
                    xdes[1]=-
L*sin(thetav)+pumaData->center[1];
                    xdes[2]=thetav+1.57;

// error in world coordinates
                    ew[0]=xdes[0]-xx[0];
                    ew[1]=xdes[1]-xx[1];
                    ew[2]=xdes[2]-xx[2];

// error in local coordinates

          ee[0]=rr[0][0]*ew[0]+rr[0][2]*ew[1];

          ee[1]=rr[1][0]*ew[0]+rr[1][2]*ew[1];
                    ee[2]=ew[2];

// velocity calculation
                    eedot[0]=(ee[0]-pumaData-
>eeold[0])/pumaData->dt;
                    eedot[1]=(ee[1]-pumaData-
>eeold[1])/pumaData->dt;
                    eedot[2]=(ee[2]-pumaData-
>eeold[2])/pumaData->dt;

// save some old values
                    pumaData->eeold[0]=ee[0];
                    pumaData->eeold[1]=ee[1];
                    pumaData->eeold[2]=ee[2];

// force - linear part
                    wn=210.0;
                    z=0.7;
                    Kp=wn*wn;
                    Kv=2.0*wn*z;

                    F[0]=Kp*ee[0]+Kv*eedot[0];
                    F[1]=Kp*ee[1]+Kv*eedot[1];

// force - angular part
                    wn=10.0; // 10 - angular  60 -
position
                    z=0.7;
                    Kp=wn*wn;
                    Kv=2.0*wn*z;

                    F[2]=Kp*ee[2]+Kv*eedot[2];
```

```
// null space filter
                Fm[0]=(F[0]-L*F[2])/(L*L+1.0);
                Fm[1]=F[1];
                Fm[2]=(-
L*F[0]+L*L*F[2])/(L*L+1.0);

                if (wall_flag)
                {
                        Fm[0]=F[0];
                        Fm[1]=F[1];
                        Fm[2]=F[2];
                }

// required torque

        T[0]=J[0][0]*Fm[0]+J[1][0]*Fm[1]+J[2][
0]*Fm[2];

        T[1]=J[0][1]*Fm[0]+J[1][1]*Fm[1]+J[2][
1]*Fm[2];

        T[2]=J[0][2]*Fm[0]+J[1][2]*Fm[1]+J[2][
2]*Fm[2];

// torque to voltage
                pumaData->voltage_out[1]=-
0.0515*T[0];
                pumaData-
>voltage_out[2]=0.1118*T[1];
                pumaData->voltage_out[4]=-
0.4980*T[2];
        }

// Convert voltages into integers to output to
trident board
        for (i=0;i<6;i++)
        {
                pumaData-
>voltage_out[i]=pumaData-
>voltage_out[i]+pumaData->vg[i]+pumaData-
>v_fric[i];//+pumaData->vim[i];
                if (fabs(pumaData-
>voltage_out[i]) > 9.9)
        pumaData-
>voltage_out[i]=9.9*pumaData-
>voltage_out[i]/fabs(pumaData->voltage_out[i]);
                voltage_int[i]=(int)
(4095.0*(pumaData->voltage_out[i]+10.0)/20.0);
        }

// Output voltages to trident hardware
        HwOutpw(pumaData->PumaDevice,
0x030, voltage_int[0]);
        HwOutpw(pumaData->PumaDevice,
0x032, voltage_int[1]);
        HwOutpw(pumaData->PumaDevice,
0x034, voltage_int[2]);
        HwOutpw(pumaData->PumaDevice,
0x036, voltage_int[3]);
        HwOutpw(pumaData->PumaDevice,
0x038, voltage_int[4]);
        HwOutpw(pumaData->PumaDevice,
0x03a, voltage_int[5]);

// save some old information
        for (i=0;i<6;i++)
        {
                pumaData-
>errorold[i]=pumaData->error[i];
                pumaData-
>theta_old[i]=pumaData->theta[i];
        }
}
```

```
// pumaHome.c

#include "puma.h"

void pumaHome(pumaFile* pumaData)
{
        short val[6];
        int voltage_int[6];
        int i;
        double thetaf[6];
        double localtime;
        double tf=5.0;

// read encoders
        val[0]=HwInpw(pumaData-
>PumaDevice, 0x010);
        val[1]=HwInpw(pumaData-
>PumaDevice, 0x012);
        val[2]=HwInpw(pumaData-
>PumaDevice, 0x014);
        val[3]=HwInpw(pumaData-
>PumaDevice, 0x016);
        val[4]=HwInpw(pumaData-
>PumaDevice, 0x018);
        val[5]=HwInpw(pumaData-
>PumaDevice, 0x01a);
```

```
// convert encoders to radians
        for (i=0; i<6; i++)
        {
                pumaData->theta[i]=pumaData-
>encoder_scale[i]*(((double) val[i]) - pumaData-
>encoder_offset[i]);
        }

// first time through get current position
        if (pumaData->last_flag==1)
        {
                pumaData-
>thetao[0]=pumaData->theta[0];
                pumaData-
>thetao[1]=pumaData->theta[1];
                pumaData-
>thetao[2]=pumaData->theta[2];
                pumaData-
>thetao[3]=pumaData->theta[3];
                pumaData-
>thetao[4]=pumaData->theta[4];
                pumaData-
>thetao[5]=pumaData->theta[5];
                pumaData->last_flag=0;
                pumaData->timeh=pumaData-
>time;
        }

// final position
        thetaf[0]=0.0;
        thetaf[1]=-1.57;
        thetaf[2]=1.57;
        thetaf[3]=0.0;
        thetaf[4]=0.0;
        thetaf[5]=0.0;

// time that home has been running
        localtime=pumaData->time-pumaData-
>timeh;

// do cubic spline interpolation
        if (localtime <= tf)
        {
                pumaData-
>thetad[0]=pumaData->thetao[0]-3.0*(pumaData-
>thetao[0]-
thetaf[0])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[0]-
thetaf[0])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[1]=pumaData->thetao[1]-3.0*(pumaData-
>thetao[1]-
thetaf[1])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[1]-
thetaf[1])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[2]=pumaData->thetao[2]-3.0*(pumaData-
>thetao[2]-
thetaf[2])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[2]-
thetaf[2])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[3]=pumaData->thetao[3]-3.0*(pumaData-
>thetao[3]-
thetaf[3])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[3]-
thetaf[3])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[4]=pumaData->thetao[4]-3.0*(pumaData-
>thetao[4]-
thetaf[4])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[4]-
thetaf[4])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[5]=pumaData->thetao[5]-3.0*(pumaData-
>thetao[5]-
thetaf[5])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[5]-
thetaf[5])*localtime*localtime*localtime/(tf*tf*tf);
        }
// after tf stay put in the final position
        else if (localtime > tf)
        {
                pumaData->thetad[0]=thetaf[0];
                pumaData->thetad[1]=thetaf[1];
                pumaData->thetad[2]=thetaf[2];
                pumaData->thetad[3]=thetaf[3];
                pumaData->thetad[4]=thetaf[4];
                pumaData->thetad[5]=thetaf[5];
        }


//
// control section
//
        for (i=0;i<6;i++)
        {
// calculate error
                pumaData->error[i]=pumaData-
>thetad[i]-pumaData->theta[i];

// calculate rate of change of the error
```

```
                pumaData-
>errordot[i]=(pumaData->error[i]-pumaData-
>errorold[i])/pumaData->dt;

// evaluate local PD control law
                pumaData-
>voltage_out[i]=pumaData->kp[i]*pumaData-
>error[i]+pumaData->kd[i]*pumaData-
>errordot[i];
        }

// Convert voltages into integers to output to
trident board
        for (i=0;i<6;i++)
        {
                pumaData-
>voltage_out[i]=pumaData-
>voltage_out[i];//+pumaData->vg[i]+pumaData-
>v_fric[i]+pumaData->vim[i];
                if (fabs(pumaData-
>voltage_out[i]) > 9.9)
        pumaData-
>voltage_out[i]=9.9*pumaData-
>voltage_out[i]/fabs(pumaData->voltage_out[i]);
                voltage_int[i]=(int)
(4095.0*(pumaData->voltage_out[i]+10.0)/20.0);
        }

// Output voltages to trident hardware
        HwOutpw(pumaData->PumaDevice,
0x030, voltage_int[0]) ;
        HwOutpw(pumaData->PumaDevice,
0x032, voltage_int[1]) ;
        HwOutpw(pumaData->PumaDevice,
0x034, voltage_int[2]) ;
        HwOutpw(pumaData->PumaDevice,
0x036, voltage_int[3]) ;
        HwOutpw(pumaData->PumaDevice,
0x038, voltage_int[4]) ;
        HwOutpw(pumaData->PumaDevice,
0x03a, voltage_int[5]) ;

// save some old information
        for (i=0;i<6;i++)
        {
                pumaData-
>errorold[i]=pumaData->error[i] ;
                pumaData-
>theta_old[i]=pumaData->theta[i] ;
  }
}
```

```
// pumaInitialization.c

#include "puma.h"

void pumaInitialization(pumaFile* pumaData)
{
        double frequency;
        int i;

// desired refresh rate (Hz)
        frequency=300.0;

// desired delta-t
        pumaData->dt=1.0/frequency;

// initialize absolute time base to zero
        pumaData->time=0.0;

// set some joint limits for impedence fields
        pumaData->jlimit3=4.0;
        pumaData->jlimit5=1.7;

// set flags for slow up and down
        pumaData->first_flag=1;
        pumaData->last_flag=1;

// encoder stuff
        pumaData-
>encoder_scale[0]=0.00010035;
        pumaData->encoder_scale[1]=-
0.000073156;
        pumaData->encoder_scale[2]=0.000117;
        pumaData->encoder_scale[3]=-
0.000082663;
        pumaData->encoder_scale[4]=-
0.000087376;
        pumaData->encoder_scale[5]=-
0.00016377;

        pumaData->encoder_offset[0]=0.0;
        pumaData->encoder_offset[1]=-21472.0;
        pumaData->encoder_offset[2]=-13426.0;
        pumaData->encoder_offset[3]=8000.0;
        pumaData->encoder_offset[4]=0.0;
        pumaData->encoder_offset[5]=0.0;

// initialize feedback gains
        pumaData->kp[0]=118.0;
        pumaData->kd[0]=15.4;
        pumaData->kp[1]=-288.0;
        pumaData->kd[1]=-24.0;
        pumaData->kp[2]=200.0;
        pumaData->kd[2]=20.0;
        pumaData->kp[3]=-15.0;
        pumaData->kd[3]=-2.0;
        pumaData->kp[4]=-25.2;
        pumaData->kd[4]=-2.2;
        pumaData->kp[5]=-10.0;
        pumaData->kd[5]=-2.0;

// initialize some variables
        for (i=0; i<6; i++)
        {
                pumaData->errorold[i]=0.0;
                        // error values
                pumaData->theta_old[i]=0.0;
                        // angular positions
                pumaData->v_fric_old[i]=0.0;
                // friction voltages
        }

        for (i=0; i<3; i++)
        {
                pumaData->eeold[i]=0.0 ;
        }

// calibrate encoders
        HwOutpw(pumaData->PumaDevice,
0x020, 0x0000);
        HwOutpw(pumaData->PumaDevice,
0x022, 0x0000);
        HwOutpw(pumaData->PumaDevice,
0x024, 0x0000);
        HwOutpw(pumaData->PumaDevice,
0x026, 0x1f40);
        HwOutpw(pumaData->PumaDevice,
0x028, 0x0000);
        HwOutpw(pumaData->PumaDevice,
0x02a, 0x0000);
}
```

```
// socket.c

#include "puma.h"

SOCKADDR_IN stLclName;
SOCKADDR_IN stRmtName;

void openSocket(pumaFile* pumaData)
{
        int server=0;
        int nRet;

        // ip for snow
//      char szHost[] = "129.186.232.46";
        // ip for hood
//      char szHost[] = "129.186.232.34";
        // ip for mammoth
        char szHost[] = "129.186.232.54";

        char szDataReceive[] = {0};
        unsigned long addr;
        WORD WSA_VERSION;
        WSADATA stWSAData;

        WSA_VERSION = MAKEWORD(1, 1);
  nRet=WSAStartup(WSA_VERSION,
&stWSAData);
        if (nRet==0)  printf("attached to winsock
dll\n");
        else  printf("could not attach winsock
dll\n");

        if (pumaData->useSocket == 1)
        {
                pumaData-
>hSock=socket(AF_INET, SOCK_DGRAM, 0);

                if (pumaData-
>hSock==INVALID_SOCKET)  printf("could not
get a valid socket handle\n");
                else
                {
                        if (server==1)
                        {

        stLclName.sin_family = PF_INET;

        stLclName.sin_port=htons(1026);

        stLclName.sin_addr.s_addr=INADDR_A
NY;


        nRet=bind(pumaData->hSock,
(LPSOCKADDR) &stLclName, sizeof(struct
sockaddr));
                                        if
(nRet==SOCKET_ERROR)  printf("could not
bind server socket\n");
```

```
                                        else
printf("server socket:  Open\n");


        nRet=recv(pumaData->hSock, (LPSTR)
szDataReceive, 5, 0);
                                        if
(nRet==SOCKET_ERROR)  printf("server socket
could not receive data\n");
                                        else
printf("sever socket received data\n");
                                }
                                else
                                {

        addr=inet_addr((LPSTR) szHost);
                                        if
(addr==INADDR_NONE)  printf("could not find
address of server\n");


        stRmtName.sin_family = PF_INET;

        stRmtName.sin_port=htons(1026);

        stRmtName.sin_addr.s_addr=addr;


        nRet=connect(pumaData->hSock,
(LPSOCKADDR) &stRmtName, sizeof(struct
sockaddr));
                                        if
(nRet==SOCKET_ERROR)  printf("could not
connect to server socket\n");
                                        else
                                        {

        printf("Socket Open\n");

        pumaData->activeSocket=1;
                                        }
                                }
                        }
        }
}

void closeSocket(pumaFile* pumaData)
{
        int nRet;

        if (pumaData->activeSocket == 1)
        {
                nRet=closesocket(pumaData-
>hSock);
                if (nRet==SOCKET_ERROR)
printf("error closing socket\n");
                else  printf("Socket Closed\n");
        }

        nRet=WSACleanup();

}

void testSocket(pumaFile* pumaData)
{
        int nRet;
        char szDataSend[100];
        double t0=0.0;
        double t1=1.571;
        double t2=-1.571;

        sprintf(szDataSend,"%4.3f %4.3f %4.3f
%4.3f %4.3f %4.3f %4.3f ",t0,t0,t2,t1,t0,t0,t0);

        if (pumaData->activeSocket == 1)
        {
                nRet=send(pumaData->hSock,
(LPSTR) szDataSend, 51, 0);
                if (nRet==SOCKET_ERROR)
printf("Socket test failed\n");
                else  printf("Socket test
passed\n");
        }
}
```

Spacial Circular Arc

```
// error.c

#include "puma.h"

void error(pumaFile* pumaData)
{
        double xv[3],xv_dot[3],xyz_dot[3];
        double center[3],dist;
        double theta_v;
        int i,j,k;
        double cv,sv;
        double xv_ori[3][3];
        double wn,z;
        double xyz[3];
        double ang[3];
        double fin[6];
        int f_flag;
        double fend[6], fbase[6], rv[3][3];

        center[0]=0.2;
        center[1]=0.0;
        center[2]=0.6;

// inverse kinematics of virtual manipuator
//      theta_v=atan2(x[1]-center[1],x[0]-
center[0]);

        ang[1]=atan2(-pumaData-
>r[2][0],sqrt(pumaData->r[0][0]*pumaData-
>r[0][0]+pumaData->r[1][0]*pumaData-
>r[1][0]));
        if (fabs(ang[1]-1.5708) < 0.01)
        {
                ang[2]=0.0;
                ang[0]=atan2(pumaData-
>r[0][1],pumaData->r[1][1]);
        }
        else if (fabs(ang[1]+1.5708) < 0.01)
        {
                ang[2]=0.0;
                ang[0]=-atan2(pumaData-
>r[0][1],pumaData->r[1][1]);
        }
        else
        {
                ang[2]=atan2(pumaData-
>r[1][0],pumaData->r[0][0]);
                ang[0]=atan2(pumaData-
>r[2][1],pumaData->r[2][2]);
        }

        theta_v=ang[2]-1.57;

        f_flag=0;
        if (theta_v < -0.2)
        {
                theta_v=-0.2;
                f_flag=1;
        }
        if (theta_v > 0.8)
        {
                theta_v=0.8;
                f_flag=1;
        }

        dist=0.3;

        cv=cos(theta_v);
        sv=sin(theta_v);

        rv[0][0]=-sv;
        rv[0][1]=cv;
        rv[0][2]=0.0;
        rv[1][0]=0.0;
        rv[1][1]=0.0;
        rv[1][2]=1.0;
        rv[2][0]=cv;
        rv[2][1]=sv;
        rv[2][2]=0.0;


// Determine the position of the robot in the
virtual manipulator's
// end effect space
        xv[0]=-sv*pumaData-
>x[0]+cv*pumaData->x[1]-
cv*center[1]+sv*center[0];
        xv[1]=pumaData->x[2]-center[2];
        xv[2]=cv*pumaData-
>x[0]+sv*pumaData->x[1]-dist-cv*center[0]-
sv*center[1];

        for (i=0;i<3;i++)
        {
                for (j=0;j<3;j++)
                {
                        xv_ori[i][j]=0.0;
                        for (k=0;k<3;k++)
                        {
                                xv_ori[i][j] +=
rv[i][k]*pumaData->r[k][j];
                        }
                }
        }

        xyz[1]=atan2(-
xv_ori[2][0],sqrt(xv_ori[0][0]*xv_ori[0][0]+xv_or
i[1][0]*xv_ori[1][0]));
        if (fabs(xyz[1]-1.5708) < 0.01)
        {
                xyz[2]=0.0;

        xyz[0]=atan2(xv_ori[0][1],xv_ori[1][1]);
        }
        else if (fabs(xyz[1]+1.5708) < 0.01)
        {
                xyz[2]=0.0;
                xyz[0]=-
atan2(xv_ori[0][1],xv_ori[1][1]);
```

```
        }
        else
        {

        xyz[2]=atan2(xv_ori[1][0],xv_ori[0][0]);

        xyz[0]=atan2(xv_ori[2][1],xv_ori[2][2]);
        }

        wn=60.0;
        z=0.7071;

        for (i=0;i<3;i++)
        {
                xv_dot[i]=(wn*wn*pumaData-
>dt*(xv[i]-pumaData->xv_old[i])+pumaData-
>xv_dot_old[i]*(2.0+2.0*z*wn*pumaData->dt)-
pumaData-
>xv_dot_way_old[i])/(1.0+2.0*z*wn*pumaData-
>dt+wn*wn*pumaData->dt*pumaData->dt);
                xyz_dot[i]=(wn*wn*pumaData-
>dt*(xyz[i]-pumaData->xyz_old[i])+pumaData-
>xyz_dot_old[i]*(2.0+2.0*z*wn*pumaData->dt)-
pumaData-
>xyz_dot_way_old[i])/(1.0+2.0*z*wn*pumaData-
>dt+wn*wn*pumaData->dt*pumaData->dt);
        }

        for (i=0;i<3;i++)
        {
                pumaData-
>xv_dot_way_old[i]=pumaData->xv_dot_old[i];
                pumaData-
>xv_dot_old[i]=xv_dot[i];
                pumaData->xv_old[i]=xv[i];
                pumaData-
>xyz_dot_way_old[i]=pumaData->xyz_dot_old[i];
                pumaData-
>xyz_dot_old[i]=xyz_dot[i];
                pumaData->xyz_old[i]=xyz[i];
        }

// Evaluate virtual spring force
        for (i=0;i<3;i++)
        {

        fm[i]=400.0*xv[i]+30.0*xv_dot[i];

        fm[i+3]=30.0*xyz[i]+2.0*xyz_dot[i];
        }

// Apply Null space filter
        fend[0]=0.9174*fm[0]-0.2752*fm[4];
        fend[1]=fm[1];
        fend[2]=fm[2];
        fend[3]=fm[3];
        fend[4]=-0.2752*fm[0]+0.0826*fm[4];
        fend[5]=fm[5];

        if (f_flag == 1)
        {
                fend[0]=fm[0];
                fend[1]=fm[1];
                fend[2]=fm[2];
                fend[3]=fm[3];
                fend[4]=fm[4];
                fend[5]=fm[5];
        }

// force end effector vm to base puma
        for (i=0; i<3; i++)
        {
                fbase[i]=0.0;
                for (j=0; j<3; j++)
                {

        fbase[i]=fbase[i]+rv[j][i]*fend[j];
                }
        }

        for (i=0; i<3; i++)
        {
                fbase[i+3]=0.0;
                for (j=0; j<3; j++)
                {

        fbase[i+3]=fbase[i+3]+rv[j][i]*fend[j+3];
                }
        }

// force base puma to end effector puma
        for (i=0; i<3; i++)
        {
                pumaData->fv[i]=0.0;
                for (j=0; j<3; j++)
                {
                        pumaData-
>fv[i]=pumaData->fv[i]+pumaData-
>r[j][i]*fbase[j];
                }
        }

        for (i=0; i<3; i++)
        {
```

```
			pumaData->fv[i+3]=0.0;
			for (j=0; j<3; j++)
			{
				pumaData-
>fv[i+3]=pumaData->fv[i+3]+pumaData-
>r[j][i]*fbase[j+3];
			}
		}
}
```

```
// friction.c

#include "puma.h"

void friction(pumaFile* pumaData)
{
	int i;
	double tau=0.05305;

	if (pumaData->theta[0] > pumaData-
>theta_old[0]) pumaData->v_fric[0]=1.0;
	if (pumaData->theta[0] <= pumaData-
>theta_old[0]) pumaData->v_fric[0]=-0.9;
	pumaData->v_fric[0]=(pumaData-
>v_fric[0]*pumaData->dt+pumaData-
>v_fric_old[0]*tau)/(pumaData->dt+tau);

	if (pumaData->theta[1] > pumaData-
>theta_old[1])
	{
		if (pumaData->theta[1] > -1.57)
pumaData->v_fric[1]=-0.3;
		else pumaData->v_fric[1]=-0.9;
	}
	if (pumaData->theta[1] <= pumaData-
>theta_old[1])
	{
		if (pumaData->theta[1] > -1.57)
pumaData->v_fric[1]=0.9;
		else pumaData->v_fric[1]=0.6;
	}
	pumaData->v_fric[1]=(pumaData-
>v_fric[1]*pumaData->dt+pumaData-
>v_fric_old[1]*tau)/(pumaData->dt+tau);

	if (pumaData->theta[2] > pumaData-
>theta_old[2]) pumaData->v_fric[2]=0.47;
	if (pumaData->theta[2] <= pumaData-
>theta_old[2]) pumaData->v_fric[2]=-0.47;
	pumaData->v_fric[2]=(pumaData-
>v_fric[2]*pumaData->dt+pumaData-
>v_fric_old[2]*tau)/(pumaData->dt+tau);

	if (pumaData->theta[3] > pumaData-
>theta_old[3]) pumaData->v_fric[3]=-0.35;
	else if (pumaData->theta[3] <=
pumaData->theta_old[3]) pumaData-
>v_fric[3]=0.35;
	else pumaData->v_fric[3]=0.0;
	pumaData->v_fric[3]=(pumaData-
>v_fric[3]*pumaData->dt+pumaData-
>v_fric_old[3]*tau)/(pumaData->dt+tau);

	if (pumaData->theta[4] > pumaData-
>theta_old[4]) pumaData->v_fric[4]=-0.4;
	else if (pumaData->theta[4] < pumaData-
>theta_old[4]) pumaData->v_fric[4]=0.4;
	else pumaData->v_fric[4]=0.0;
	pumaData->v_fric[4]=(pumaData-
>v_fric[4]*pumaData->dt+pumaData-
>v_fric_old[4]*tau)/(pumaData->dt+tau);

	if (pumaData->theta[5] > pumaData-
>theta_old[5]) pumaData->v_fric[5]=-0.5;
	else if (pumaData->theta[5] < pumaData-
>theta_old[5]) pumaData->v_fric[5]=0.5;
	else pumaData->v_fric[5]=0.0;
	pumaData->v_fric[5]=(pumaData-
>v_fric[5]*pumaData->dt+pumaData-
>v_fric_old[5]*tau)/(pumaData->dt+tau);

	for (i=0;i<6;i++)
	{
		pumaData-
>v_fric_old[i]=pumaData->v_fric[i];
	}
}
```

```
// gravity.c

#include "puma.h"

void gravity(pumaFile* pumaData)
{
	double c2,s2,c23,s23;

	c2=cos(pumaData->theta[1]);
	s2=sin(pumaData->theta[1]);
	c23=cos(pumaData->theta[1]+pumaData->theta[2]);
	s23=sin(pumaData->theta[1]+pumaData->theta[2]);

// gravity compensation
	pumaData->vg[0]=0.0;
	pumaData->vg[2]=-1.1201*s23+0.0977*c23;
	pumaData->vg[1]=0.2400*s2+2.1144*c2-0.5304*pumaData->vg[2];
	pumaData->vg[3]=0.0;
	pumaData->vg[4]=0.0;
	pumaData->vg[5]=0.0;
}
```

```
// impedence.c

#include "puma.h"

void impedence(pumaFile* pumaData)
{
	pumaData->vim[0]=0.02*pow((1.0/(pumaData->theta[0]-2.7)),3.0)+0.02*pow((1.0/(pumaData->theta[0]+2.7)),3.0);

	pumaData->vim[1]=-0.02*pow((1.0/(pumaData->theta[1]-0.7)),3.0)-0.02*pow((1.0/(pumaData->theta[1]+3.7)),3.0);

	pumaData->vim[2]=0.02*pow((1.0/(pumaData->theta[2]-pumaData->jlimit3)),3.0)+0.02*pow((1.0/(pumaData->theta[2]+0.9)),3.0);

	pumaData->vim[3]=-0.02*pow((1.0/(pumaData->theta[3]-3.2)),3.0)-0.02*pow((1.0/(pumaData->theta[3]+1.8)),3.0);

	pumaData->vim[4]=-0.02*pow((1.0/(pumaData->theta[4]-1.7)),3.0)-0.02*pow((1.0/(pumaData->theta[4]+pumaData->jlimit5)),3.0);

	pumaData->vim[5]=-0.02*pow((1.0/(pumaData->theta[5]-5.2)),3.0)-0.02*pow((1.0/(pumaData->theta[5]+5.2)),3.0);
}
```

```
// jacobian.c

#include "puma.h"

void jacobian(pumaFile* pumaData)
{
	double c1,s1,c2,s2,c3,s3,c23,s23,c4,s4,c5,s5,c6,s6;
	double l[5];
	l[1]=0.4318;
	l[2]=0.15005;
	l[3]=-0.0191;
	l[4]=0.4331;

	c1=cos(pumaData->theta[0]);
	s1=sin(pumaData->theta[0]);

	c2=cos(pumaData->theta[1]);
	s2=sin(pumaData->theta[1]);

	c3=cos(pumaData->theta[2]);
	s3=sin(pumaData->theta[2]);

	c23=cos(pumaData->theta[1]+pumaData->theta[2]);
	s23=sin(pumaData->theta[1]+pumaData->theta[2]);

	c4=cos(pumaData->theta[3]);
	s4=sin(pumaData->theta[3]);

	c5=cos(pumaData->theta[4]);
	s5=sin(pumaData->theta[4]);

	c6=cos(pumaData->theta[5]);
	s6=sin(pumaData->theta[5]);

// jacobian
	pumaData->eJr[0][0]=c5*c6*(-c23*c4*l[2]+s4*(c2*l[1]+c23*l[3]+s23*l[4]))+s6*(c23*s4*l[2]+c4*(c2*l[1]+c23*l[3]+s23*l[4]))+s5*c6*s23*l[2];
	pumaData->eJr[0][1]=c5*c6*(c4*(s3*l[1]+l[4]))+s6*(-s4*(s3*l[1]+l[4]))-s5*c6*(-c3*l[1]-l[3]);
	pumaData->eJr[0][2]=c5*c6*c4*l[4]-s6*s4*l[4]+s5*c6*l[3];
	pumaData->eJr[0][3]=0.0;
```

```
        pumaData->eJr[0][4]=0.0;
        pumaData->eJr[0][5]=0.0;

        pumaData->eJr[1][0]=-c5*s6*(-
c23*c4*l[2]+s4*(c2*l[1]+c23*l[3]+s23*l[4]))+c6*
(c23*s4*l[2]+c4*(c2*l[1]+c23*l[3]+s23*l[4]))-
s5*s6*s23*l[2];
        pumaData->eJr[1][1]=-
c5*s6*(c4*(s3*l[1]+l[4]))+c6*(-
s4*(s3*l[1]+l[4]))+s5*s6*(-c3*l[1]-l[3]);
        pumaData->eJr[1][2]=-c5*s6*c4*l[4]-
c6*s4*l[4]-s5*s6*l[3];
        pumaData->eJr[1][3]=0.0;
        pumaData->eJr[1][4]=0.0;
        pumaData->eJr[1][5]=0.0;

        pumaData->eJr[2][0]=s5*(-
c23*c4*l[2]+s4*(c2*l[1]+c23*l[3]+s23*l[4]))-
c5*s23*l[2];
        pumaData-
>eJr[2][1]=s5*(c4*(s3*l[1]+l[4]))+c5*(-c3*l[1]-
l[3]);
        pumaData->eJr[2][2]=s5*c4*l[4]-c5*l[3];
        pumaData->eJr[2][3]=0.0;
        pumaData->eJr[2][4]=0.0;
        pumaData->eJr[2][5]=0.0;

        pumaData->eJr[3][0]=s23*(s4*s6-
c4*c5*c6)-c23*s5*c6;
        pumaData->eJr[3][1]=s4*c5*c6+c4*s6;
        pumaData->eJr[3][2]=s4*c5*c6+c4*s6;
        pumaData->eJr[3][3]=-s5*c6;
        pumaData->eJr[3][4]=s6;
        pumaData->eJr[3][5]=0.0;

        pumaData-
>eJr[4][0]=s23*(c4*c5*s6+s4*c6)+c23*s5*s6;
        pumaData->eJr[4][1]=-s4*c5*s6+c4*c6;
        pumaData->eJr[4][2]=-s4*c5*s6+c4*c6;
        pumaData->eJr[4][3]=s5*s6;
        pumaData->eJr[4][4]=c6;
        pumaData->eJr[4][5]=0.0;

        pumaData->eJr[5][0]=-
s23*c4*s5+c23*c5;
        pumaData->eJr[5][1]=s4*s5;
        pumaData->eJr[5][2]=s4*s5;
        pumaData->eJr[5][3]=c5;
        pumaData->eJr[5][4]=0.0;
        pumaData->eJr[5][5]=1.0;
}
```

```
// kinematics.c

#include "puma.h"

void kinematics(pumaFile* pumaData)
{
        double
c1,s1,c2,s2,c3,s3,c23,s23,c4,s4,c5,s5,c6,s6;
        double l[5];
        double
v1,v2,v3,v4,v5,v6,v7,v8,v9,v10,v11;

        l[1]=0.4318;
        l[2]=0.15005;
        l[3]=-0.0191;
        l[4]=0.4331;

        c1=cos(pumaData->theta[0]);
        s1=sin(pumaData->theta[0]);

        c2=cos(pumaData->theta[1]);
        s2=sin(pumaData->theta[1]);

        c3=cos(pumaData->theta[2]);
        s3=sin(pumaData->theta[2]);

        c23=cos(pumaData->theta[1]+pumaData-
>theta[2]);
        s23=sin(pumaData->theta[1]+pumaData-
>theta[2]);

        c4=cos(pumaData->theta[3]);
        s4=sin(pumaData->theta[3]);

        c5=cos(pumaData->theta[4]);
        s5=sin(pumaData->theta[4]);

        c6=cos(pumaData->theta[5]);
        s6=sin(pumaData->theta[5]);

        pumaData-
>x[0]=c1*(c23*l[3]+s23*l[4]+c2*l[1])-s1*l[2];
        pumaData-
>x[1]=s1*(c23*l[3]+s23*l[4]+c2*l[1])+c1*l[2];
        pumaData->x[2]=-s23*l[3]+c23*l[4]-
s2*l[1];
        pumaData->x[3]=0.0;
        pumaData->x[4]=0.0;
        pumaData->x[5]=0.0;
        v1=c4*c5*c6-s4*s6;
        v2=s5*c6;
        v3=c23*v1-s23*v2;
        v4=s4*c5*c6+c4*s6;

        pumaData->r[0][0]=c1*v3-s1*v4;
        pumaData->r[1][0]=s1*v3+c1*v4;
        pumaData->r[2][0]=-s23*v1-c23*v2;

        v5=c4*c5*s6+s4*c6;
        v6=s5*s6;
        v7=-c23*v5+s23*v6;
        v8=s4*c5*s6-c4*c6;

        pumaData->r[0][1]=c1*v7+s1*v8;
        pumaData->r[1][1]=s1*v7-c1*v8;
        pumaData->r[2][1]=s23*v5+c23*v6;

        v9=c4*s5;
        v10=c23*v9+s23*c5;
        v11=s4*s5;

        pumaData->r[0][2]=c1*v10-s1*v11;
        pumaData->r[1][2]=s1*v10+c1*v11;
        pumaData->r[2][2]=-s23*v9+c23*c5;
}
```

```
// main.c

#include "puma.h"

void main(void)
{
// robot stuff
        pumaFile *pumaData;
        int stop;
        int homecount;

// window's stuff
        HANDLE hprocess;
        HANDLE hthread;
        int processerror;

// timer stuff
        BOOL result;
        LARGE_INTEGER lifrequency;
        LARGE_INTEGER licount;
        LONGLONG frequency;
        double dfrequency;
        LONGLONG startcount;
        LONGLONG count;
        double currenttime;
        double dtactual;
        double dterror;
        double dtmax;

// error flags
        int timererror;
        int timeroverrun;
        int DeviceStop;
        int errorSocket;

// socket stuff
        int err;
        char szDataSend[100];
        int gcount;

// data file stuff
        double data[3][2000];
        int datalength=2000;
        int datacount;
        int datacycle;
        int datamax;
        int fileerror;
        FILE *out;

// general stuff (counter and the like)
        int i;

/////////////////////////////////////////////////////////////////////
// Taking Care of Business
/////////////////////////////////////////////////////////////////////

        printf("PUMA control program\n");
        printf("written by Jim Edwards for
LARCC\n");
        printf("All rights reserved\n\n\n\n");

/////////////////////////////////////////////////////////////////////
// Code Initialization Section
/////////////////////////////////////////////////////////////////////
// set counter error flag to pass
        timererror=1;

// set counter overrun flag to pass
        timeroverrun=1;

// start taking data at zero
        datacount=0;

// set data pass to zero
        datacycle=0;
```

```
// set process error flag to pass
        processerror=0;

// set maximum delta-t to zero
        dtmax=0.0;

// set stop to pass
        stop=1;

// set homecount to zero
        homecount=0;

// set socket error to none
        errorSocket=0;

// set graphics dump counter to zero
        gcount=0;

////////////////////////////////////////////////////////////
////// Hardware Initialization
////////////////////////////////////////////////////////////
// get process handle
        hprocess=GetCurrentProcess();

// set process priority
        result=SetPriorityClass(hprocess,
REALTIME_PRIORITY_CLASS);
        if (result == 0)  processerror=1;

// get thread handle
        hthread=GetCurrentThread();

// set thread priority
        result=SetThreadPriority(hthread,
THREAD_PRIORITY_TIME_CRITICAL);
        if (result == 0)  processerror=2;

// allocate memory for puma structure
        pumaData=(pumaFile
*)malloc(sizeof(pumaFile));

// connect to the puma kernel device
        DeviceStop=1;
        pumaData-
>PumaDevice=HwNewDevice(NULL);
        HwSetErrorHandler(pumaData-
>PumaDevice, MyErrorHandler);
        if (!HwConnectDevice(pumaData-
>PumaDevice, "puma"))
        {
                printf("Failed to connect to puma
device!\n");
                HwDeleteDevice(pumaData-
>PumaDevice);
                DeviceStop=0;
        }

// setup puma
        pumaInitialization(pumaData);

// open socket - useSocket = 1 use socket, = 0
don't use socket
        pumaData->useSocket=1;
        pumaData->activeSocket=0;
        openSocket(pumaData);

// test socket
        testSocket(pumaData);

// get frequency of high performance counter
        result=QueryPerformanceFrequency(&lifr
equency);
        if (result == TRUE)
        {
                frequency=lifrequency.QuadPart;
                dfrequency=((double)
frequency);
                printf("clock frequency:  %f
MHz\n\n\n\n",dfrequency);
        }
        else
        {

        printf("QueryPerformanceFrequency:
failure\n");
                timererror=0;
        }

// get starting count
        printf("\n\n\nTurn Arm Power On!!!!\n");
        result=QueryPerformanceCounter(&licou
nt);
        if (result == TRUE)
        {
                startcount=licount.QuadPart;
        }
        else
        {

        printf("QueryPerformanceCounter:
failure\n");
                timererror=0;
        }

// disengage the brakes
        HwOutpw(pumaData->PumaDevice,
0x02e, 0x0001);

//////////////////////////////////////////////////////////////////
// Main Control Loop
//////////////////////////////////////////////////////////////////

        while((homecount < 2000) &&
(DeviceStop == 1) && (timererror == 1) &&
(timeroverrun == 1) && (processerror == 0))
        {
// control code
                if (kbhit()) stop=0;
                if (stop == 1)
                {
                pumaControl(pumaData);
                }
                else
                {
                homecount++;
                        pumaHome(pumaData);
                }

// increment graphics dump counter
                gcount++ ;

// send data to graphics engine
                if (gcount == 5)
                {
                gcount=0;
// but only if there is an active socket for
communication
                        if (pumaData-
>activeSocket == 1)
                {

        sprintf(szDataSend,"%4.3f %4.3f %4.3f
%4.3f %4.3f %4.3f %4.3f ",
                        pumaData->time,
                                pumaData-
>theta[0],
                                pumaData-
>theta[1],
                                pumaData-
>theta[2],
                                pumaData-
>theta[3],
                                pumaData-
>theta[4],
                                        pumaData-
>theta[5]);

                err=send(pumaData->hSock,
(LPSTR) szDataSend, 51, 0) ;
                                if
(err==SOCKET_ERROR) errorSocket=1;
                }
                }

// timing code
                do
                {
// get the current count of performance counter

        result=QueryPerformanceCounter(&licou
nt);
                        if (result == TRUE)
                        {

        count=licount.QuadPart;
// convert into time since program started

        currenttime=((double) (count-
startcount))/dfrequency;

                        }
                        else
                        {

        printf("QueryPerformanceCounter:
failure\n");
                                timererror=0;
                        }

                        dtactual=currenttime-
pumaData->time;
                } while(dtactual < pumaData-
>dt);

// get maximum delta-t
                if (dtactual > dtmax)
dtmax=dtactual;

// get error in delta-t
                dterror=dtactual-pumaData->dt;
                if (fabs(dterror) > pumaData-
>dt) timeroverrun=0;

// take some data
                if (stop == 1)
                {
                        // time
//
        data[0][datacount]=pumaData->time;
                        // fresh frequency
//
        data[1][datacount]=1.0/dtactual;
                        // voltage to axis 5
//
        data[2][datacount]=pumaData-
>voltage_out[4];


        data[0][datacount]=pumaData->x[0];

        data[1][datacount]=pumaData->x[1];

        data[2][datacount]=pumaData->x[2];


                        if (datacount == 1999)
                        {
                                datacount=0;
                                datacycle=1;
                        }
                        else  datacount++;
                }

// update absolute time base
                pumaData->time=pumaData-
>time+pumaData->dt;
        } // end main control loop

// engage the brakes
        HwOutpw(pumaData->PumaDevice,
0x02e, 0x0000);

///////////////////////////////////////////////////////////////
// Hardware Clean-Up
///////////////////////////////////////////////////////////////
// kernal device
        HwDeleteDevice(pumaData-
>PumaDevice);

// close socket
        closeSocket(pumaData);

///////////////////////////////////////////////////////////////
// Take some data
///////////////////////////////////////////////////////////////

// open the data file
        if ((out=fopen("out.dat","wt"))==NULL)
fileerror=0;
        else
        {
// write data
                fileerror=1;

                fprintf(out,"max dt is
%f\n\n\n",dtmax);

                if (datacycle == 1)
datamax=datalength;
                else  datamax=datacount;

                for (i=0; i<datamax; i++)
                {
                        fprintf(out,"%f, %f,
%f\n",data[0][i],data[1][i],data[2][i]);
                }

// close file
                fclose(out);
        }

///////////////////////////////////////////////////////////////
// Final Error Messages
///////////////////////////////////////////////////////////////
        printf("\n\n\nError Messages:\n");
        if (timererror == 0)  printf("timer
malfunction\n");
        else if (timeroverrun == 0)  printf("timer
over run\n");
        else if (DeviceStop == 0)  printf("DriverX
error\n");
        else if (fileerror == 0)  printf("could not
open data file\n");
        else if (processerror == 1)  printf("could
not set process priority\n");
        else if (processerror == 2)  printf("could
not set thread priority\n");
        else if (errorSocket == 1)  printf("error
sending data over socket\n");
        else  printf("all went well\n");


        Sleep(3000);
}

/////////////////////////////////////////////////////////
////// DriverX Error Handler
/////////////////////////////////////////////////////////
```

```
void MyErrorHandler(HWDEVICE* pDevice,
DWORD nError)
{
        printf("Critical DriverX error: %d\n",
nError);
        exit(nError);
}
```

```
// puma.h

// include files
#include <windows.h>
#include <winsock.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include "DriverX.h"

// structures
typedef struct
{
// needed for all
        HWDEVICE* PumaDevice;
        double dt;
        double time;
        double encoder_scale[6];
        double encoder_offset[6];
        double theta[6];
        double voltage_out[6];

// socket stuff
        SOCKET hSock;
        int useSocket;
        int activeSocket;

// kinematics
        double x[6];
        double r[3][3];
        double eJr[6][6];

// virtual manipulator
        double fv[6];
        double u_vm;
        double v_vm;
        double xv_old[3];
        double xv_dot_old[3];
        double xv_dot_way_old[3];
        double xyz_old[3];
        double xyz_dot_old[3];
        double xyz_dot_way_old[3];

// needed for me
        int first_flag;
        int last_flag;
        double kp[6];
        double kd[6];
        double error[6];
        double errorold[6];
        double errordot[6];
        double thetad[6];
        double theta_old[6];
        double thetao[6];
        double timeh;
        double vg[6];
        double v_fric[6];
        double v_fric_old[6];
        double vim[6];
        double jlimit3;
        double jlimit5;
} pumaFile;

// prototypes
void main(void);
void MyErrorHandler(HWDEVICE *, DWORD);
void pumaInitialization(pumaFile *);
void pumaControl(pumaFile *);
void pumaHome(pumaFile *);
void openSocket(pumaFile *);
void closeSocket(pumaFile *);
void testSocket(pumaFile *);
void gravity(pumaFile *);
void friction(pumaFile *);
void impedence(pumaFile *);
void kinematics(pumaFile *);
void jacobian(pumaFile *);
void error(pumaFile *);
```

```
// pumaControl.c

#include "puma.h"

void pumaControl(pumaFile* pumaData)
{
        short val[6];
        int voltage_int[6];
        int i, j;
        double thetaf[6];
        double tf=5.0;

// read encoders
```

```
        val[0]=HwInpw(pumaData-
>PumaDevice, 0x010);
        val[1]=HwInpw(pumaData-
>PumaDevice, 0x012);
        val[2]=HwInpw(pumaData-
>PumaDevice, 0x014);
        val[3]=HwInpw(pumaData-
>PumaDevice, 0x016);
        val[4]=HwInpw(pumaData-
>PumaDevice, 0x018);
        val[5]=HwInpw(pumaData-
>PumaDevice, 0x01a);

// convert encoders to radians
        for (i=0; i<6; i++)
        {
                pumaData->theta[i]=pumaData-
>encoder_scale[i]*(((double) val[i]) - pumaData-
>encoder_offset[i]);
        }

// gravity compensation
        gravity(pumaData);

// forward kinematics and Jacobian
        kinematics(pumaData);

// virtual manipulator control
        error(pumaData);

// evaluate jacobian
        jacobian(pumaData);

// friction compensation
        friction(pumaData);

// impedence protection
        impedence(pumaData);

// first time through get current position
        if (pumaData->first_flag==1)
        {
                pumaData-
>thetao[0]=pumaData->theta[0];
                pumaData-
>thetao[1]=pumaData->theta[1];
                pumaData-
>thetao[2]=pumaData->theta[2];
                pumaData-
>thetao[3]=pumaData->theta[3];
                pumaData-
>thetao[4]=pumaData->theta[4];
                pumaData-
>thetao[5]=pumaData->theta[5];
                pumaData->first_flag=2;
        }

// final position
        thetaf[0]=0.1428;
        thetaf[1]=-0.3966;
        thetaf[2]=0.5388;
        thetaf[3]=0.6374;
        thetaf[4]=1.4137;
        thetaf[5]=1.5168;

// do cubic spline interpolation
        if (pumaData->time <= tf)
        {
                pumaData-
>thetad[0]=pumaData->thetao[0]-3.0*(pumaData-
>thetao[0]-thetaf[0])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[0]-thetaf[0])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                pumaData-
>thetad[1]=pumaData->thetao[1]-3.0*(pumaData-
>thetao[1]-thetaf[1])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[1]-thetaf[1])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                pumaData-
>thetad[2]=pumaData->thetao[2]-3.0*(pumaData-
>thetao[2]-thetaf[2])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[2]-thetaf[2])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                pumaData-
>thetad[3]=pumaData->thetao[3]-3.0*(pumaData-
>thetao[3]-thetaf[3])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[3]-thetaf[3])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                pumaData-
>thetad[4]=pumaData->thetao[4]-3.0*(pumaData-
>thetao[4]-thetaf[4])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[4]-thetaf[4])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
```

```
                pumaData-
>thetad[5]=pumaData->thetao[5]-3.0*(pumaData-
>thetao[5]-thetaf[5])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[5]-thetaf[5])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
        }
// after tf stay put at final position .
        else if (pumaData->time > tf)
        {
                pumaData->thetad[0]=thetaf[0];
                pumaData->thetad[1]=thetaf[1];
                pumaData->thetad[2]=thetaf[2];
                pumaData->thetad[3]=thetaf[3];
                pumaData->thetad[4]=thetaf[4];
                pumaData->thetad[5]=thetaf[5];
        }

//
// control section
//
        for (i=0;i<6;i++)
        {
// calculate error
                pumaData->error[i]=pumaData-
>thetad[i]-pumaData->theta[i];

// calculate rate of change of the error
                pumaData-
>errordot[i]=(pumaData->error[i]-pumaData-
>errorold[i])/pumaData->dt;

// evaluate local PD control law
                pumaData-
>voltage_out[i]=pumaData->kp[i]*pumaData-
>error[i]+pumaData->kd[i]*pumaData-
>errordot[i];
        }

// impedence based control law
        if (pumaData->time > 6.0)
        {
                for (j=0;j<6;j++)
                {
                pumaData->voltage_out[j]=0.0;
                        for (i=0;i<6;i++)
                        {
                pumaData->voltage_out[j] +=
pumaData->eJr[i][j]*pumaData->fv[i];
                        }
                }
                pumaData-
>voltage_out[0]=pumaData->voltage_out[0]*-1.0;
                pumaData-
>voltage_out[2]=pumaData->voltage_out[2]*-1.0;
        }

// Convert voltages into integers to output to
trident board
        for (i=0;i<6;i++)
        {
                pumaData-
>voltage_out[i]=pumaData-
>voltage_out[i]+pumaData->vg[i]+pumaData-
>v_fric[i]+pumaData->vim[i];
                if (fabs(pumaData-
>voltage_out[i]) > 9.9)
        pumaData-
>voltage_out[i]=9.9*pumaData-
>voltage_out[i]/fabs(pumaData->voltage_out[i]);
                voltage_int[i]=(int)
(4095.0*(pumaData->voltage_out[i]+10.0)/20.0);
        }

// Output voltages to trident hardware
        HwOutpw(pumaData->PumaDevice,
0x030, voltage_int[0]);
        HwOutpw(pumaData->PumaDevice,
0x032, voltage_int[1]);
        HwOutpw(pumaData->PumaDevice,
0x034, voltage_int[2]);
        HwOutpw(pumaData->PumaDevice,
0x036, voltage_int[3]);
        HwOutpw(pumaData->PumaDevice,
0x038, voltage_int[4]);
        HwOutpw(pumaData->PumaDevice,
0x03a, voltage_int[5]);

// save some old information
        for (i=0;i<6;i++)
        {
                pumaData-
>errorold[i]=pumaData->error[i];
                pumaData-
>theta_old[i]=pumaData->theta[i];
        }
}
```

```
// pumaHome.c
```

```
#include "puma.h"

void pumaHome(pumaFile* pumaData)
{
        short val[6];
        int voltage_int[6];
        int i;
        double thetaf[6];
        double localtime;
        double tf=5.0;

// read encoders
        val[0]=HwInpw(pumaData-
>PumaDevice, 0x010);
        val[1]=HwInpw(pumaData-
>PumaDevice, 0x012);
        val[2]=HwInpw(pumaData-
>PumaDevice, 0x014);
        val[3]=HwInpw(pumaData-
>PumaDevice, 0x016);
        val[4]=HwInpw(pumaData-
>PumaDevice, 0x018);
        val[5]=HwInpw(pumaData-
>PumaDevice, 0x01a);

// convert encoders to radians
        for (i=0; i<6; i++)
        {
                pumaData->theta[i]=pumaData-
>encoder_scale[i]*(((double) val[i]) - pumaData-
>encoder_offset[i]);
        }

// first time through get current position
        if (pumaData->last_flag==1)
        {
                pumaData-
>thetao[0]=pumaData->theta[0];
                pumaData-
>thetao[1]=pumaData->theta[1];
                pumaData-
>thetao[2]=pumaData->theta[2];
                pumaData-
>thetao[3]=pumaData->theta[3];
                pumaData-
>thetao[4]=pumaData->theta[4];
                pumaData-
>thetao[5]=pumaData->theta[5];
                pumaData->last_flag=0;
                pumaData->timeh=pumaData-
>time;
        }

// final position
        thetaf[0]=0.0;
        thetaf[1]=-1.57;
        thetaf[2]=1.57;
        thetaf[3]=0.0;
        thetaf[4]=0.0;
        thetaf[5]=0.0;

// time that home has been running
        localtime=pumaData->time-pumaData-
>timeh;

// do cubic spline interpolation
        if (localtime <= tf)
        {
                pumaData-
>thetad[0]=pumaData->thetao[0]-3.0*(pumaData-
>thetao[0]-
thetaf[0])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[0]-
thetaf[0])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[1]=pumaData->thetao[1]-3.0*(pumaData-
>thetao[1]-
thetaf[1])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[1]-
thetaf[1])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[2]=pumaData->thetao[2]-3.0*(pumaData-
>thetao[2]-
thetaf[2])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[2]-
thetaf[2])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[3]=pumaData->thetao[3]-3.0*(pumaData-
>thetao[3]-
thetaf[3])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[3]-
thetaf[3])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[4]=pumaData->thetao[4]-3.0*(pumaData-
>thetao[4]-
thetaf[4])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[4]-
thetaf[4])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[5]=pumaData->thetao[5]-3.0*(pumaData-
>thetao[5]-
thetaf[5])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[5]-
thetaf[5])*localtime*localtime*localtime/(tf*tf*tf);
```

```
        }
// after tf stay put in the final position
        else if (localtime > tf)
        {
                pumaData->thetad[0]=thetaf[0];
                pumaData->thetad[1]=thetaf[1];
                pumaData->thetad[2]=thetaf[2];
                pumaData->thetad[3]=thetaf[3];
                pumaData->thetad[4]=thetaf[4];
                pumaData->thetad[5]=thetaf[5];
        }


//
// control section
//
        for (i=0;i<6;i++)
        {
// calculate error
                pumaData->error[i]=pumaData-
>thetad[i]-pumaData->theta[i];

// calculate rate of change of the error
                pumaData-
>errordot[i]=(pumaData->error[i]-pumaData-
>errorold[i])/pumaData->dt;

// evaluate local PD control law
                pumaData-
>voltage_out[i]=pumaData->kp[i]*pumaData-
>error[i]+pumaData->kd[i]*pumaData-
>errordot[i];
        }

// Convert voltages into integers to output to
trident board
        for (i=0;i<6;i++)
        {
                pumaData-
>voltage_out[i]=pumaData-
>voltage_out[i];//+pumaData->vg[i]+pumaData-
>v_fric[i]+pumaData->vim[i];
                if (fabs(pumaData-
>voltage_out[i]) > 9.9)
        pumaData-
>voltage_out[i]=9.9*pumaData-
>voltage_out[i]/fabs(pumaData->voltage_out[i]);
                voltage_int[i]=(int)
(4095.0*(pumaData->voltage_out[i]+10.0)/20.0);
        }

// Output voltages to trident hardware
        HwOutpw(pumaData->PumaDevice,
0x030, voltage_int[0]) ;
        HwOutpw(pumaData->PumaDevice,
0x032, voltage_int[1]) ;
        HwOutpw(pumaData->PumaDevice,
0x034, voltage_int[2]) ;
        HwOutpw(pumaData->PumaDevice,
0x036, voltage_int[3]) ;
        HwOutpw(pumaData->PumaDevice,
0x038, voltage_int[4]) ;
        HwOutpw(pumaData->PumaDevice,
0x03a, voltage_int[5]) ;

// save some old information
        for (i=0;i<6;i++)
        {
                pumaData-
>errorold[i]=pumaData->error[i] ;
                pumaData-
>theta_old[i]=pumaData->theta[i] ;
  }
}
```

```
// pumaInitialization.c

#include "puma.h"

void pumaInitialization(pumaFile* pumaData)
{
        double frequency;
        int i;

// desired refresh rate (Hz)
        frequency=300.0;

// desired delta-t
        pumaData->dt=1.0/frequency;

// initialize absolute time base to zero
        pumaData->time=0.0;

// set some joint limits for impedence fields
        pumaData->jlimit3=4.0;
        pumaData->jlimit5=1.7;

// set flags for slow up and down
        pumaData->first_flag=1;
        pumaData->last_flag=1;

// encoder stuff
```

```
        pumaData-
>encoder_scale[0]=0.00010035;
        pumaData->encoder_scale[1]=-
0.000073156;
        pumaData->encoder_scale[2]=0.000117;
        pumaData->encoder_scale[3]=-
0.000082663;
        pumaData->encoder_scale[4]=-
0.000087376;
        pumaData->encoder_scale[5]=-
0.00016377;

        pumaData->encoder_offset[0]=0.0;
        pumaData->encoder_offset[1]=-21472.0;
        pumaData->encoder_offset[2]=-13426.0;
        pumaData->encoder_offset[3]=8000.0;
        pumaData->encoder_offset[4]=0.0;
        pumaData->encoder_offset[5]=0.0;

// initialize feedback gains
        pumaData->kp[0]=118.0;
        pumaData->kd[0]=15.4;
        pumaData->kp[1]=-288.0;
        pumaData->kd[1]=-24.0;
        pumaData->kp[2]=200.0;
        pumaData->kd[2]=20.0;
        pumaData->kp[3]=-15.0;
        pumaData->kd[3]=-2.0;
        pumaData->kp[4]=-25.2;
        pumaData->kd[4]=-2.2;
        pumaData->kp[5]=-10.0;
        pumaData->kd[5]=-2.0;

        pumaData->u_vm=0.0;
        pumaData->v_vm=0.0;

// initialize some variables
        for (i=0; i<6; i++)
        {
                pumaData->errorold[i]=0.0;
                        // error values
                pumaData->theta_old[i]=0.0;
                        // angular positions
                pumaData->v_fric_old[i]=0.0;
                // friction voltages
        }

        for (i=0; i<3; i++)
        {
                pumaData->xv_old[i]=0.0;
                pumaData->xv_dot_old[i]=0.0;
                pumaData-
>xv_dot_way_old[i]=0.0;
                pumaData->xyz_old[i]=0.0;
                pumaData->xyz_dot_old[i]=0.0;
                pumaData-
>xyz_dot_way_old[i]=0.0;
        }

// calibrate encoders
        HwOutpw(pumaData->PumaDevice,
0x020, 0x0000);
        HwOutpw(pumaData->PumaDevice,
0x022, 0x0000);
        HwOutpw(pumaData->PumaDevice,
0x024, 0x0000);
        HwOutpw(pumaData->PumaDevice,
0x026, 0x1f40);
        HwOutpw(pumaData->PumaDevice,
0x028, 0x0000);
        HwOutpw(pumaData->PumaDevice,
0x02a, 0x0000);
}
```

```
// socket.c

#include "puma.h"

SOCKADDR_IN stLclName;
SOCKADDR_IN stRmtName;

void openSocket(pumaFile* pumaData)
{
        int server=0;
        int nRet;

        // ip for snow
//      char szHost[] = "129.186.232.46";
        // ip for hood
        char szHost[] = "129.186.232.34";
        // ip for mammoth
//      char szHost[] = "129.186.232.54";

        char szDataReceive[] = {0};
        unsigned long addr;
        WORD WSA_VERSION;
        WSADATA stWSAData;

        WSA_VERSION = MAKEWORD(1, 1);
  nRet=WSAStartup(WSA_VERSION,
&stWSAData);
```

```
        if (nRet==0)  printf("attached to winsock
dll\n");
        else  printf("could not attach winsock
dll\n");

        if (pumaData->useSocket == 1)
        {
                pumaData-
>hSock=socket(AF_INET, SOCK_DGRAM, 0);

                if (pumaData-
>hSock==INVALID_SOCKET)  printf("could not
get a valid socket handle\n");
                else
                {
                        if (server==1)
                        {

        stLclName.sin_family = PF_INET;

        stLclName.sin_port=htons(1026),

        stLclName.sin_addr.s_addr=INADDR_A
NY;


        nRet=bind(pumaData->hSock,
(LPSOCKADDR) &stLclName, sizeof(struct
sockaddr));
                                if
(nRet==SOCKET_ERROR)  printf("could not
bind server socket\n");
                                else
printf("server socket:  Open\n");


        nRet=recv(pumaData->hSock, (LPSTR)
szDataReceive, 5, 0);
                                if
(nRet==SOCKET_ERROR)  printf("server socket
could not receive data\n");
                                else
printf("sever socket received data\n");
                        }
                        else
                        {

        addr=inet_addr((LPSTR) szHost);
                                if
(addr==INADDR_NONE)  printf("could not find
address of server\n");

        stRmtName.sin_family = PF_INET;

        stRmtName.sin_port=htons(1026);

        stRmtName.sin_addr.s_addr=addr;


        nRet=connect(pumaData->hSock,
(LPSOCKADDR) &stRmtName, sizeof(struct
sockaddr));
                                if
(nRet==SOCKET_ERROR)  printf("could not
connect to server socket\n");
                                else
                                {

        printf("Socket Open\n");

        pumaData->activeSocket=1;
                                }
                        }
                }
        }
}

void closeSocket(pumaFile* pumaData)
{
        int nRet;

        if (pumaData->activeSocket == 1)
        {
                nRet=closesocket(pumaData-
>hSock);
                if (nRet==SOCKET_ERROR)
printf("error closing socket\n");
                else  printf("Socket Closed\n");
        }

        nRet=WSACleanup();

}

void testSocket(pumaFile* pumaData)
{
        int nRet;
        char szDataSend[100];
        double t0=0.0;
        double t1=1.571;
        double t2=-1.571;
        sprintf(szDataSend,"%4.3f %4.3f %4.3f
%4.3f %4.3f %4.3f %4.3f ",t0,t0,t2,t1,t0,t0,t0);

        if (pumaData->activeSocket == 1)
        {
                nRet=send(pumaData->hSock,
(LPSTR) szDataSend, 51, 0);
                if (nRet==SOCKET_ERROR)
printf("Socket test failed\n");
                else  printf("Socket test
passed\n");
        }
}
```

Bob The Fish

```
// calibration2.c

#include "puma.h"

void calibration2(void)
{
        loadencoder(0,0);
        loadencoder(1,0);
        loadencoder(2,0);
        loadencoder(3,8000);
        loadencoder(4,0);
        loadencoder(5,0);
}
```

```
// com_init.c

#include <stdio.h>
#include <math.h>
#include "com.h"

void com_init(long BaudRate)
{
        int status;

// turn off all interrupts on the UART
        outportb(INTERRUPT_ENABLE,NOINT
ERRUPTS);
        status=inportb(INTERRUPT_ENABLE);
        if (status == 0)  printf("Interrupts
Disabled\n");
        else  printf("Something is wrong with the
Interrupts!!!\n");

// set the baud rate
        SetBaud(BaudRate);
        printf("Baud Rate:  %ld\n",BaudRate);

// turn off modem control
        outportb(MODEM_CONTROL,NOHAN
DSHAKING);
        status=inportb(MODEM_CONTROL);
        if (status ==0)  printf("Hardware
Handshaking Disabled\n");
        else  printf("Something is wrong with
Modem Control!!!\n");

// enable 16550 FIFO
        outportb(FIFO_CONTROL,0x07);
        status=inportb(INTERRUPT_IDENT);
        if (status == 0xc1)  printf("FIFO
enabled\n");
        else
        {

        outportb(FIFO_CONTROL,0x00);
                printf("Something is wrong with
the FIFO!!!\n");
        }
}

void SetBaud(long BaudRate)
{
        long BaudRateDivisor,BaseBaud;
        int msb,lsb;

// maximum PC baud rate
        BaseBaud=115200;

// ratio of maximum baud rate and desired baud
rate
        BaudRateDivisor=BaseBaud/BaudRate;

// decompose divisor into 2 8-bit bytes
        msb=BaudRateDivisor >> 8;
        lsb=BaudRateDivisor & 0xFF;

// set divisor latch to change baud rate
        outportb(LINE_CONTROL,EIGHTDAT
ABITS | ONESTOPBITS | NOPARITY |
DIVISORLATCH);

// enter most and least significant bytes of baud
rate divisor
        outportb(RECIEVER_BUFFER,lsb);
```

```
	outportb(INTERRUPT_ENABLE,msb);

// un-set divisor latch to continue
	outportb(LINE_CONTROL,EIGHTDAT
ABITS | ONESTOPBITS | NOPARITY);
}

void com_close(void)
{
	int status;

// disable 16550 FIFO
	outportb(FIFO_CONTROL,0x00);
	status=inportb(INTERRUPT_IDENT);
	if (status == 0x01) printf("FIFO
disabled\n");
	else printf("Something is wrong with the
FIFO!!!\n");
}
```

```
// com.h

// register stuff
#define COM_1			0x3F8
#define RECIEVER_BUFFER
	COM_1 + 0x00
#define TRANSMIT_BUFFER
	COM_1 + 0x00
#define INTERRUPT_ENABLE	COM_1 +
0x01
#define INTERRUPT_IDENT
	COM_1 + 0x02
#define FIFO_CONTROL		COM_1 +
0x02
#define LINE_CONTROL		COM_1 +
0x03
#define MODEM_CONTROL
	COM_1 + 0x04
#define LINE_STATUS		COM_1 +
0x05
#define MODEM_STATUS
	COM_1 + 0x06
#define SCRATCH
	COM_1 + 0x07
#define DIVISOR_LATCH_HIGH COM_1 +
0x01
#define DIVISOR_LATCH_LOW	COM_1 +
0x00

// communication set-up stuff
#define FIVEDATABITS		0x00
#define SIXDATABITS		0x01
#define SEVENDATABITS			0x02
#define EIGHTDATABITS			0x03

#define ONESTOPBITS		0x00
#define TWOSTOPBITS		0x04

#define NOPARITY		0x00
#define ODDPARITY		0x08
#define EVENPARITY		0x18
#define MARKPARITY		0x28
#define SPACEPARITY		0x38

#define BREAKCONTROL			0x40

#define DIVISORLATCH		0x80

#define NOINTERRUPTS		0x00

#define NOHANDSHAKING			0x00

// communications prototypes
void SetBaud(long);
void com_init(long);
void com_close(void);
```

```
// control.c

#include "puma.h"
#include "com.h"
#include "puma.ext"

extern int dcount;

void control(void)
{
	int i,j;
	int int_x;
	double shit1,shit2,xabs;
	double spie[3];

// Read encoders
	for (ii=0;ii<6;ii++)
	{
		val[ii]=readencoder(ii);

	position[ii]=encoder_scale[ii]*((double)
(val[ii]) - encoder_offset[ii]);
	}
```

```
// Forward kinematics
        forkin();
        spie[0]=x[0];
        spie[1]=x[1];
        spie[2]=x[2];

// prepare stuff for serial transmission
        for (i=0;i<6;i++)
        {
                xabs=fabs(x[i]);
                int_x=((int) (10000.0*xabs));
                if (x[i] < 0.0)  int_x=int_x |
0x8000;
                serial[i][1]=int_x & 0x0FF;
                serial[i][0]=(int_x >> 8) &
0x0FF;
        }

// Evaluate the PUMA jacobian
        jacobian();

// Error calculation
        error_v();

// Position gains
        kp[0]=27.6;
        kd[0]=3.5;
        kp[1]=-71.9;
        kd[1]=-9.0;
        kp[2]=51.5;
        kd[2]=3.7;
        kp[3]=-15.0;
        kd[3]=-1.2;
        kp[4]=-15.2;
        kd[4]=-1.2;
        kp[5]=-10.0;
        kd[5]=-1.0;

        if (time < 2.0)
        {
                invkin();

// Calculate control command
                positiond[3]=0.0;
                positiond[4]=0.0;
                positiond[5]=0.0;

                for (ii=0;ii<6;ii++)
                {
                        error[ii]=positiond[ii]-
position[ii];
                        error_dot[ii]=(error[ii]-
error_old[ii])*300.0;

        voltage_out[ii]=kp[ii]*error[ii]+kd[ii]*err
or_dot[ii];
                }
        }

/*      positiond[3]=0.0;
        positiond[4]=0.0;
        positiond[5]=0.0;

        for (ii=3;ii<6;ii++)
        {
                error[ii]=positiond[ii]-
position[ii];
                error_dot[ii]=(error[ii]-
error_old[ii])*300.0;

        voltage_out[ii]=kp[ii]*error[ii]+kd[ii]*err
or_dot[ii];
        } */

// Implement impedence control to protect the
joints
        voltage_imped=0.02*pow((1.0/(position[
0]-
2.7)),3.0)+0.02*pow((1.0/(position[0]+2.7)),3.0);
        voltage_out[0] += voltage_imped+vg[0];
        if (position[0] > position_old[0])
voltage_out[0] += 1.0;
        if (position[0] < position_old[0])
voltage_out[0] -= 0.9;
        voltage_imped=-
0.02*pow((1.0/(position[1]-0.7)),3.0)-
0.02*pow((1.0/(position[1]+3.7)),3.0);
        voltage_out[1] += voltage_imped+vg[1];
        if (position[1] > position_old[1])
        {
                if (position[1] > -1.57)
voltage_out[1] -= 0.3;
                else  voltage_out[1] -= 0.9;
        }
        if (position[1] < position_old[1])
        {
                if (position[1] > -1.57)
voltage_out[1] += 0.9;
                else voltage_out[1] += 0.6;
        }
        voltage_imped=0.02*pow((1.0/(position[
2]-
4.0)),3.0)+0.02*pow((1.0/(position[2]+0.9)),3.0);
```

```
        voltage_out[2] += voltage_imped+vg[2];
        if (position[2] > position_old[2])
voltage_out[2] += 0.47;
        if (position[2] < position_old[2])
voltage_out[2] -= 0.47;
        voltage_imped=-
0.02*pow((1.0/(position[3]-3.2)),3.0)-
0.02*pow((1.0/(position[3]+1.8)),3.0);
        voltage_out[3] += voltage_imped;
        voltage_imped=-
0.02*pow((1.0/(position[4]-1.7)),3.0)-
0.02*pow((1.0/(position[4]+1.7)),3.0);
        voltage_out[4] += voltage_imped;
        voltage_imped=-
0.02*pow((1.0/(position[5]-5.2)),3.0)-
0.02*pow((1.0/(position[5]+5.2)),3.0);
        voltage_out[5] += voltage_imped;

// Convert voltages into integers to output to
trident board
        for (ii=0;ii<6;ii++)
        {
                if (fabs(voltage_out[ii]) > 9.9)
voltage_out[ii]=9.9*voltage_out[ii]/fabs(voltage_o
ut[ii]);
                voltage_int[ii]=(int)
(4095.0*(voltage_out[ii]+10.0)/20.0);
        }

// Output voltages
        for (ii=0;ii<6;ii++)
        {
                outport(BASE + 0x0030 + 2*ii,
voltage_int[ii]);
        }

// Save old position values
        position_old[0]=position[0];
        position_old[1]=position[1];
        position_old[2]=position[2];
        position_old[3]=position[3];
        position_old[4]=position[4];
        position_old[5]=position[5];

// Save old error values
        error_old[0]=error[0];
        error_old[1]=error[1];
        error_old[2]=error[2];
        error_old[3]=error[3];
        error_old[4]=error[4];
        error_old[5]=error[5];

// Increment counter
        time += 1.0/300.0;

// Take some data
        if (dcount==3)
        {
                dcount=0;
                if (data < data_max)
                {
                        data++;

        data_pts[0][data]=spie[0];

        data_pts[1][data]=spie[1];

        data_pts[2][data]=spie[2];
//                      data_pts2[data]=0.0;
                        if (data == 999) data=0;
                }
        }
        dcount++;

// Send some information over serial port
        sync++;
        if (sync == 7)
        {
                while ((inportb(LINE_STATUS)
& 0x20) == 0);

        outportb(TRANSMIT_BUFFER,serial[0][
0]);
                while ((inportb(LINE_STATUS)
& 0x20) == 0);

        outportb(TRANSMIT_BUFFER,serial[0][
1]);
                while ((inportb(LINE_STATUS)
& 0x20) == 0);

        outportb(TRANSMIT_BUFFER,serial[1][
0]);
                while ((inportb(LINE_STATUS)
& 0x20) == 0);

        outportb(TRANSMIT_BUFFER,serial[1][
1]);
                while ((inportb(LINE_STATUS)
& 0x20) == 0);

        outportb(TRANSMIT_BUFFER,serial[2][
0]);
```

```
                while ((inportb(LINE_STATUS)
& 0x20) == 0);

        outportb(TRANSMIT_BUFFER,serial[2][
1]);
                while ((inportb(LINE_STATUS)
& 0x20) == 0);

        outportb(TRANSMIT_BUFFER,serial[3][
0]);
                while ((inportb(LINE_STATUS)
& 0x20) == 0);

        outportb(TRANSMIT_BUFFER,serial[3][
1]);
                while ((inportb(LINE_STATUS)
& 0x20) == 0);

        outportb(TRANSMIT_BUFFER,serial[4][
0]);
                while ((inportb(LINE_STATUS)
& 0x20) == 0);

        outportb(TRANSMIT_BUFFER,serial[4][
1]);
                while ((inportb(LINE_STATUS)
& 0x20) == 0);

        outportb(TRANSMIT_BUFFER,serial[5][
0]);
                while ((inportb(LINE_STATUS)
& 0x20) == 0);

        outportb(TRANSMIT_BUFFER,serial[5][
1]);
                counter++;
                if (counter > 199)  counter=0;
                sync=0;
        }
}
```

```
// error.c

#include "puma.h"
#include "puma.ext"

void error_v(void)
{
        double
xc[3],pi,e[3],xdot[3],xv[3],xv_dot[3],J[6][6];
        double center[3],l1,l2,l3,spring,damp;
        int i,j;
        double kp[3],kv[3];
        double wn,z,T;

        i=0;

        center[0]=0.5;
        center[1]=0.0;
        center[2]=0.5;

// inverse kinematics of virtual manipuator
        l1=x[0]-center[0];
        if (l1 < -0.075)  l1=-0.075;
        if (l1 > 0.075)  l1=0.075;

        l2=x[1]-center[1];
        if (l2 < -0.075)  l2=-0.075;
        if (l2 > 0.075)  l2=0.075;

        l3=x[2]-center[2];
        if (l3 < -0.075)  l3=-0.075;
        if (l3 > 0.075)  l3=0.075;

// Determine the position of the robot in the
virtual manipulator's
// end effect space
        xv[0]=x[0]-center[0]-l1;
        xv[1]=x[1]-center[1]-l2;
        xv[2]=x[2]-center[2]-l3;

        wn=60.0;
        T=1.0/300.0;
        z=0.7071;

        for (i=0;i<3;i++)
        {
                xv_dot[i]=(wn*wn*T*(xv[i]-
xv_old[i])+xv_dot_old[i]*(2.0+2.0*z*wn*T)-
xv_dot_way_old[i])/(1.0+2.0*z*wn*T+wn*wn*T*
T);
        }

        for (i=0;i<3;i++)
        {

        xv_dot_way_old[i]=xv_dot_old[i];
                xv_dot_old[i]=xv_dot[i];
                xv_old[i]=xv[i];
        }

        spring=530.0;
damp=40.0;

if (l1 <= -0.05)
{
        kp[0]=spring;
        kv[0]=damp;
}
else if (l1 >= 0.05)
{
        kp[0]=spring;
        kv[0]=damp;
}
else
{
        kp[0]=0.0;
        kv[0]=0.0;
}

if (l2 <= -0.05)
{
        kp[1]=spring;
        kv[1]=damp;
}
else if (l2 >= 0.05)
{
        kp[1]=spring;
        kv[1]=damp;
}
else
{
        kp[1]=0.0;
        kv[1]=0.0;
}

if (l3 <= -0.05)
{
        kp[2]=spring;
        kv[2]=damp;
}
else if (l3 >= 0.05)
{
        kp[2]=spring;
        kv[2]=damp;
}
else
{
        kp[2]=0.0;
        kv[2]=0.0;
}

J[0][0]=oJr[0][0];
J[0][1]=oJr[0][1];
        J[0][2]=oJr[0][2];

        J[1][0]=oJr[1][0];
        J[1][1]=oJr[1][1];
        J[1][2]=oJr[1][2];

        J[2][0]=oJr[2][0];
        J[2][1]=oJr[2][1];
        J[2][2]=oJr[2][2];

        for (i=0;i<3;i++)
        {
                voltage_out[i]=0.0;
                for (j=0;j<3;j++)
                {
                        voltage_out[i] +=
J[j][i]*(kp[j]*xv[j]+kv[j]*xv_dot[j]);
                }
        }

        voltage_out[0]=voltage_out[0]*-1.0;
//      voltage_out[1]=voltage_out[1]*-1.0;
        voltage_out[2]=voltage_out[2]*-1.0;
        voltage_out[3]=0.0;
        voltage_out[4]=0.0;
        voltage_out[5]=0.0;

// Safty net
        x[0]=center[0];
        x[1]=center[1];
        x[2]=center[2];
        x[3]=0.0;
        x[4]=0.0;
        x[5]=0.0;

/*      if (time > 2.1)
        {
                x[0]=l1;
                x[1]=l2;
                x[2]=xv[2];
        }  */
}
```

```
// home.c

#include "puma.h"
#include "puma.ext"

void home(void)
{
```

```
// Read encoders
        for (ii=0;ii<6;ii++)
        {
                val[ii]=readencoder(ii);

        position[ii]=encoder_scale[ii]*((double)
(val[ii]) - encoder_offset[ii]);
        }

// Desired trajectory
        positiond[0]=0.0;
        positiond[1]=-1.57;
        positiond[2]=1.57;
        positiond[3]=0.0;
        positiond[4]=0.0;
        positiond[5]=0.0;

// Control law
        kp[0]=27.6;
        kd[0]=3.5;
        kp[1]=-71.9;
        kd[1]=-9.0;
        kp[2]=51.5;
        kd[2]=3.7;
        kp[3]=-5.0;
        kd[3]=0.0;
        kp[4]=-15.2;
        kd[4]=-1.2;
        kp[5]=-5.0;
        kd[5]=0.0;

        for (ii=0;ii<6;ii++)
        {
                error[ii]=positiond[ii]-
position[ii];
                error_dot[ii]=(error[ii]-
error_old[ii])*300.0;
//              velocity[ii]=(position[ii]-
position_old[ii])*300.0;

        voltage_out[ii]=kp[ii]*error[ii]+kd[ii]*err
or_dot[ii];
//
        voltage_imped=0.05*pow((1.0/(position[i
i]-1.85 )),3.0)+0.05*pow((1.0/(position[ii]+1.85
)),3.0);
//              if (position[ii] < 1.65)
voltage_imped=0.0;
//              if (ii==4)
voltage_out[ii]=voltage_out[ii]+voltage_imped;

                if (fabs(voltage_out[ii]) > 9.9)
voltage_out[ii]=9.9*voltage_out[ii]/fabs(voltage_o
ut[ii]);

                voltage_int[ii]=(int)
(4095.0*(voltage_out[ii]+10.0)/20.0);
        }

// Output voltages
        for (ii=0;ii<6;ii++)
        {
                outport(BASE + 0x0030 + 2*ii,
voltage_int[ii]);
        }

// Save old position values
        position_old[0]=position[0];
        position_old[1]=position[1];
        position_old[2]=position[2];
        position_old[3]=position[3];
        position_old[4]=position[4];
        position_old[5]=position[5];

// Save old error values
        error_old[0]=error[0];
        error_old[1]=error[1];
        error_old[2]=error[2];
        error_old[3]=error[3];
        error_old[4]=error[4];
        error_old[5]=error[5];
}
```

```
// init.c

#include "puma.h"
#include "puma.ext"

void init(void)
{
        DISCRETE=0x0000;

        encoder_scale[0]=0.00010035;
        encoder_scale[1]=-0.000073156;
        encoder_scale[2]=0.000117;
        encoder_scale[3]=-0.000082663;
        encoder_scale[4]=-0.000087376;
        encoder_scale[5]=-0.00016377;

        encoder_offset[0]=0.0;
        encoder_offset[1]=-21472.0;
        encoder_offset[2]=-13426.0;
        encoder_offset[3]=8000.0;
        encoder_offset[4]=0.0;
        encoder_offset[5]=0.0;

        positiond[0]=0.0;
        positiond[1]=0.0;
        positiond[2]=0.0;
        positiond[3]=0.0;
        positiond[4]=0.0;
        positiond[5]=0.0;

        error_old[0]=0.0;
        error_old[1]=0.0;
        error_old[2]=0.0;
        error_old[3]=0.0;
        error_old[4]=0.0;
        error_old[5]=0.0;

        position_old[0]=0.0;
        position_old[1]=0.0;
        position_old[2]=0.0;
        position_old[3]=0.0;
        position_old[4]=0.0;
        position_old[5]=0.0;

        xv_dot_old[0]=0.0;
        xv_dot_old[1]=0.0;
        xv_dot_old[2]=0.0;
        xv_dot_old[3]=0.0;
        xv_dot_old[4]=0.0;
        xv_dot_old[5]=0.0;

        xv_dot_way_old[0]=0.0;
        xv_dot_way_old[1]=0.0;
        xv_dot_way_old[2]=0.0;
        xv_dot_way_old[3]=0.0;
        xv_dot_way_old[4]=0.0;
        xv_dot_way_old[5]=0.0;

        xv_old[0]=0.0;
        xv_old[1]=0.0;
        xv_old[2]=0.0;
        xv_old[3]=0.0;
        xv_old[4]=0.0;
        xv_old[5]=0.0;
}
```

```
// invkin.c

#include "puma.h"
#include "puma.ext"

void invkin(void)
{
        double l[4],theta[4][6],pi,k,v1,v2,v3;
        double valid[4],limits[6][2],dist,sdist;
        double ca,sa,cb,sb,cc,sc,r[4][4];
        double c1,s1,c23,s23,c4,s4,c5,s5,c6,s6;
        double r11,r12,r21,r22,r23,r13,r33;
        int i,j,select;

        pi=3.14159;

        l[0]=0.4318;
        l[1]=0.15005;
        l[2]=-0.0191;
        l[3]=0.4331;

        valid[0]=1;
        valid[1]=1;
        valid[2]=1;
        valid[3]=1;

        select=0;

        limits[0][0]=-2.92;
limits[0][1]=2.89;
        limits[1][0]=-3.92;
limits[1][1]=0.82;
        limits[2][0]=-1.01;
limits[2][1]=4.27;
        limits[3][0]=-2.02;
limits[3][1]=3.36;
        limits[4][0]=-1.87;
limits[4][1]=1.86;
        limits[5][0]=-5.36;
limits[5][1]=5.35;


// theta 1 calculation
        theta[0][0]=atan2(x[1],x[0])-
atan2(l[1],sqrt(pow(x[0],2.0)+pow(x[1],2.0)-
pow(l[1],2.0)));
        theta[1][0]=atan2(x[1],x[0])-atan2(l[1],-
sqrt(pow(x[0],2.0)+pow(x[1],2.0)-pow(l[1],2.0)));
        theta[2][0]=theta[0][0];
        theta[3][0]=theta[1][0];

// theta 3 calculation
```

```
        k=(pow(x[0],2.0)+pow(x[1],2.0)+pow(x[2
],2.0)-pow(l[0],2.0)-pow(l[1],2.0)-pow(l[2],2.0)-
pow(l[3],2.0))/(2.0*l[0]);
        theta[0][2]=atan2(k,sqrt(pow(l[2],2.0)+po
w(l[3],2.0)-pow(k,2.0)))-atan2(l[2],l[3]);
        theta[1][2]=theta[0][2];
        theta[2][2]=atan2(k,-
sqrt(pow(l[2],2.0)+pow(l[3],2.0)-pow(k,2.0)))-
atan2(l[2],l[3]);
        theta[3][2]=theta[2][2];
        for (i=0;i<4;i++)
        {
                if (theta[i][2] < -1.01)
theta[i][2]=theta[i][2]+2.0*pi;
        }

// theta 2 calculation
        for (i=0;i<4;i++)
        {
                v1=l[2]+l[0]*cos(theta[i][2]);

        v2=x[0]*cos(theta[i][0])+x[1]*sin(theta[i]
[0]);
                v3=l[3]+l[0]*sin(theta[i][2]);
                theta[i][1]=atan2(v3*v2-
x[2]*v1,v1*v2+x[2]*v3)-theta[i][2];
                if (theta[i][1] > 0.82)
theta[i][1]=theta[i][1]-2.0*pi;
        }

// check joint limits
        for (i=0;i<4;i++)
        {
                for (j=0;j<3;j++)
                {
                        if ((limits[j][0] <
theta[i][j]) && (theta[i][j] < limits[j][1]))
                        {
                                valid[i]=1;
                        }
                        else
                        {
                                valid[i]=0;
                                break;
                        }
                }
        }

// find the closest valid solution to the old position
        for (i=0;i<4;i++)
        {
                if (valid[i] == 1)
                {
                        select=i;
                        sdist=0.0;
                        for (j=0;j<3;j++)
                        {

sdist=sdist+fabs(positiond[j]-theta[i][j]);
                        }
                        break;
                }
        }

        for (i=select+1;i<4;i++)
        {
                if (valid[i] == 1)
                {
                        dist=0.0;
                        for (j=0;j<3;j++)
                        {

dist=dist+fabs(positiond[j]-theta[i][j]);
                        }
                        if (dist < sdist)
                        {
                                sdist=dist;
                                select=i;
                        }
                }
        }

// select the solution
        for (i=0;i<3;i++)
positiond[i]=theta[select][i];
}
```

```
// jacobian.c

#include "puma.h"
#include "puma.ext"

void jacobian(void)
{
        int i;
        double c1,s1,c2,s2,c3,s3,c23,s23;
        double l1,l2,l3,l4;

        l1=0.4318;
        l2=0.15005;
        l3=-0.0191;
        l4=0.4331;
        c1=cos(position[0]);
        s1=sin(position[0]);

        c2=cos(position[1]);
        s2=sin(position[1]);

        c3=cos(position[2]);
        s3=sin(position[2]);

        c23=cos(position[1]+position[2]);
        s23=sin(position[1]+position[2]);

        eJr[0][0]=-c23*l2;
        eJr[0][1]=s3*l1+l4;
        eJr[0][2]=l4;

        eJr[1][0]=c2*l1+c23*l3+s23*l4;
        eJr[1][1]=0.0;
        eJr[1][2]=0.0;

        eJr[2][0]=-s23*l2;
        eJr[2][1]=-c3*l1-l3;
        eJr[2][2]=-l3;

        oJr[0][0]=-s1*(c23*l3+s23*l4+c2*l1)-
c1*l2;
        oJr[0][1]=c1*(-s23*l3+c23*l4-s2*l1);
        oJr[0][2]=c1*(-s23*l3+c23*l4);

        oJr[1][0]=c1*(c23*l3+s23*l4+c2*l1)-
s1*l2;
        oJr[1][1]=s1*(-s23*l3+c23*l4-s2*l1);
        oJr[1][2]=s1*(-s23*l3+c23*l4);

        oJr[2][0]=0.0;
        oJr[2][1]=-c23*l3-s23*l4-c2*l1;
        oJr[2][2]=-c23*l3-s23*l4;

// gravity compensation
        vg[0]=0.0;
        vg[2]=-1.1201*s23+0.0977*c23;
        vg[1]=0.2400*s2+2.1144*c2-
0.5304*vg[2];
//      vg[0]=0.0;
//      vg[1]=0.0;
//      vg[2]=0.0;
}
```

```
// forkin.c

#include "puma.h"
#include "puma.ext"

void forkin(void)
{
        double
c1,s1,c2,s2,c23,s23,c4,s4,c5,s5,c6,s6;
        double
v1,v2,v3,v4,v5,v6,v7,v8,v9,v10,v11;
        double A,B,C;
        double l[5],r[4][4];

        l[1]=0.4318;
        l[2]=0.15005;
        l[3]=-0.0191;
        l[4]=0.4331;

        c1=cos(position[0]);
        s1=sin(position[0]);

        c2=cos(position[1]);
        s2=sin(position[1]);

        c23=cos(position[1]+position[2]);
        s23=sin(position[1]+position[2]);

        c4=cos(position[3]);
        s4=sin(position[3]);

        c5=cos(position[4]);
        s5=sin(position[4]);

        c6=cos(position[5]);
        s6=sin(position[5]);

// end effector position
        x[0]=c1*(c23*l[3]+s23*l[4]+c2*l[1])-
s1*l[2];
        x[1]=s1*(c23*l[3]+s23*l[4]+c2*l[1])+c1
*l[2];
        x[2]=-s23*l[3]+c23*l[4]-s2*l[1];

        v1=c4*c5*c6-s4*s6;
        v2=s5*c6;
        v3=c23*v1-s23*v2;
        v4=s4*c5*c6+c4*s6;

        r[1][1]=c1*v3-s1*v4;
        r[2][1]=s1*v3+c1*v4;
        r[3][1]=-s23*v1-c23*v2;
        v5=c4*c5*s6+s4*c6;
        v6=s5*s6;
        v7=-c23*v5+s23*v6;
        v8=s4*c5*s6-c4*c6;

        r[1][2]=c1*v7+s1*v8;
        r[2][2]=s1*v7-c1*v8;
        r[3][2]=s23*v5+c23*v6;

        v9=c4*s5;
        v10=c23*v9+s23*c5;
        v11=s4*s5;

        r[1][3]=c1*v10-s1*v11;
        r[2][3]=s1*v10+c1*v11;
        r[3][3]=-s23*v9+c23*c5;

// end effector orientation
        B=atan2(sqrt(pow(r[3][1],2.0)+pow(r[3][
2],2.0)),r[3][3]);
        if (fabs(B) < 0.0001)
        {
                A=0.0;
                C=atan2(-r[1][2],r[1][1]);
        }
        else if (fabs(B-180.0) < 0.0001)
        {
                A=0.0;
                C=atan2(r[1][2],-r[1][1]);
        }
        else
        {
                A=atan2(r[2][3],r[1][3]);
                C=atan2(r[3][2],-r[3][1]);
        }

        x[3]=C;
        x[4]=B;
        x[5]=A;
}
```

```
// loadencoder.c

#include "puma.h"

void loadencoder(int channel,int value)
{
        outport(ENC_LOAD + 2*channel,
value);
}
```

```
// main.c

#include "puma.h"
#include "com.h"
#include "puma.gbl"

int dcount=0;

// Prototypes for interrupt service routines
static void interrupt far my_isr();
static void interrupt (*old_isr)();

void main(void)
{
        int i,safe_count;
        int cdiv,lb,hb;
        long BaudRate,safty;
        double fs;
        double base_freq=1192500.0;

        FILE *out;

// Initialize some variables
        init();

// Calibrate encoders to scratch mark values
        calibration2();

// Open serial port
        BaudRate=38400;
        com_init(BaudRate);

// Determine time base for control loop
        fs=300.0;
        cdiv=(int) ceil(base_freq/fs);
        outportb(0x43,0x36);
        hb=cdiv/255;
        lb=(int) fmod(cdiv,255);
        outport(0x40,lb);
        outport(0x40,hb);

// Save old interrupt serive routine
        disable();
        old_isr=getvect(0x1c);

// Activate new interrupt service routine
        setvect(0x1c,my_isr);
        enable();
        counter=0;
        safe_count=0;

// Enable arm power
        DISCRETE=DISCRETE | POWER_BIT;
        outport(DISCRETE_REG,DISCRETE);
        printf("Turn arm power on!!!\n");

// Loop in time - run controller
        while(!kbhit() && safty > 0)
        {
// Turn interrupt flag off
                disable();
                INTERRUPT_FLAG=0;

// Run control function
                control();
                enable();

                data_pts2[safe_count]=(double)
counter;
                safe_count++;
                if (safe_count > 998)
safe_count=0;
                safty=0;

// Wait for interrupt
                while(!INTERRUPT_FLAG)
safty++;
        }

// Loop in time - return to home position
        for(i=0;i<500;i++)
        {
// Turn interrupt flag off
                INTERRUPT_FLAG=0;

// Run control function
                home();

// Wait for interrupt
                while(!INTERRUPT_FLAG);
        }

// Disable arm power
        DISCRETE=DISCRETE |
(~POWER_BIT);
        outport(DISCRETE_REG,DISCRETE);

// Reactivate old interrupt service routine
        disable();
        setvect(0x1c,old_isr);
        enable();

// Change time base back to normal
        fs=18.3;
        cdiv=(int) ceil(base_freq/fs);
        outportb(0x43,0x36);
        hb=cdiv/255;
        lb=(int) fmod(cdiv,255);
        outport(0x40,lb);
        outport(0x40,hb);

// close serial port
        com_close();

// Output counter value
        printf("\n\ncounter = %ld\n",counter);

// Output some data
        if((out=fopen("out.dat","wt"))==NULL)
        {
                printf("Cannot open output file
OUT.DAT.\n");
                exit(1);
        }

        for (i=0;i<=999;i++)
        {
                fprintf(out,"%f    %f        %f
        %f\n",data_pts[0][i],data_pts[1][i],data_p
ts[2][i],data_pts2[i]);
        }
}

static void interrupt far my_isr()
{
        INTERRUPT_FLAG=1;
}
```

```
// path.c

#include "puma.h"
#include "puma.ext"

void path()
{
        double u,pi,radius,center[3];
        int counter_max;

        pi=3.14159;
        radius=0.3;
        center[0]=0.2;
        center[1]=0.0;
        center[2]=0.5;
        counter_max=2001;

        counter++;
        if (counter == counter_max)
        {
                counter=0;
                if (direction == 1)  direction=-1;
                else  direction=1;
        }

        u=((double) counter)/((double)
(counter_max-1));
        if (direction == -1)  u=1.0-u;

        x[0]=radius*cos(pi*u/2.0)+center[0];
        x[1]=radius*sin(pi*u/2.0)+center[1];
        x[2]=center[2];
}
```

```
// puma.ext

// External variables
extern long DISCRETE;
extern long STATUS;
extern long counter;
extern int data;
extern int ii;
extern int val[6];
extern double encoder_scale[6];
extern double encoder_offset[6];
extern double position[6];
extern double position_old[6];
extern double velocity[6];
extern double positiond[6];
extern double error[6];
extern double error_old[6];
extern double error_dot[6];
extern double voltage_out[6];
extern double voltage_imped;
extern int voltage_int[6];
extern double kp[6];
extern double kd[6];
extern double data_pts[3][2000];
extern double data_pts2[1003];
extern int data_max;
extern double w;
extern double time;
extern double x[6],xstart,xfinish;
extern double theta_old1,theta_old2;
extern int direction;
extern double eJr[6][6];
extern double oJr[6][6];
extern double theta_v;
extern double xv_old[6];
extern double xv_dot_old[6];
extern double xv_dot_way_old[6];
extern double vg[6];
extern int sync;
extern int serial[6][2];
```

```
// puma.gbl

// Global variables
int INTERRUPT_FLAG=1;
long DISCRETE;
long STATUS;
long counter;
int data=0;
int ii;
int val[6];
double encoder_scale[6];
double encoder_offset[6];
double position[6];
double position_old[6];
double velocity[6];
double positiond[6];
double error[6];
double error_old[6];
double error_dot[6];
double voltage_out[6];
double voltage_imped;
int voltage_int[6];
double kp[6];
double kd[6];
double data_pts[3][2000];
double data_pts2[1003];
int data_max=1000;
double w;
double time=0.0;
double x[6],xstart=.5,xfinish=-.5;
double theta_old1=1.57,theta_old2=0.0;
int direction=1;
double eJr[6][6];
double oJr[6][6];
double theta_v=0.0;
double xv_old[6];
double xv_dot_old[6];
double xv_dot_way_old[6];
double vg[6];
int sync=0;
int serial[6][2];
```

```
// puma.h

#include <stdio.h>
#include <dos.h>
#include <float.h>
#include <stdlib.h>
#include <bios.h>
#include <conio.h>
#include <io.h>
#include <math.h>
#include <string.h>

// prototypes
int readencoder(int);
void calibration2(void);
void loadencoder(int,int);
void control(void);
void path(void);
void invkin(void);
void jacobian(void);
void forkin(void);
void error_v(void);

double **matrix(int,int,int,int);
double ***array3(int,int,int,int,int,int);
double *vector(int,int);
int *ivector(int,int);
void free_vector(double *, int);
void free_ivector(int *, int);
void free_matrix(double **, int, int,int);


// define some addresses that will be needed
#define BASE
0x0300
#define AD_MUX_SELECT
BASE + 0x002C
#define AD_START_PULSE
BASE + 0x001E
#define DISCRETE_REG
BASE + 0x002E
#define STATUS_REG
BASE + 0x000C
#define AD_VALUE
BASE + 0x001C
#define ENC_COUNTER

                BASE + 0x0010
#define ENC_LOAD

                BASE + 0x0020

// define some bit masks that will be needed
#define AD_MASK
0x4000
#define AD_STATUS_MASK
0x4000
#define POWER_BIT

                                0x0001
```

```
// readencoder.c

#include "puma.h"

int readencoder(int channel)
{
        int val;

        val=(int)(inport(ENC_COUNTER +
2*channel));
        return val;
}
```

NURBS Curve

```
// calibraton2.c

#include "puma.h"

void calibration2(void)
{
        loadencoder(0,0);
        loadencoder(1,0);
        loadencoder(2,0);
        loadencoder(3,8000);
        loadencoder(4,0);
        loadencoder(5,0);
}
```

```
// control.c
```

```
#include "puma.h"

extern double encoder_scale[6];
extern double encoder_offset[6];
extern double position[6];
extern double position_old[6];
extern double vg[6];
extern double error_old[6];
extern double forced[6];
extern double forced_old[6];
extern double f_fil_old[6];
extern double f_fil_way_old[6];
extern double data_pts[3][1500];
extern double time;
extern int data;
extern double evJI[3][3];
extern double eJw[3][3];
extern double x[6];
extern double x_old[6];
extern double joint_limit;
extern double fv[6];
extern double r[3][3];
extern double rv[3][3];
extern double eJI[3][3];
extern double f_flag;

void control(void)
{
        int i,j;
        int val[6];
        double kp[6];
        double kd[6];
        int voltage_int[6];
        double voltage_out[6];
        double error[6];
        double error_dot[6];
        double f_fil[6];
        double positiond[6];
        double voltage_imped;
        double wn = 2.0*5.0*3.14159;
        double zeta = 1.0;
        double T = 1./300.;
        double KplKpw;
        double fb[6];
        double fev[6];

// Read encoders
        for (i=0;i<6;i++)
        {
                val[i]=readencoder(i);
                position[i]=encoder_scale[i]*((double)
(val[i]) - encoder_offset[i]);
                }

// Get gravity compensation
                gravity();

// Forward kinematics
                forkin();

// Error calculation
                error_v();

// Get end effector force
                get_force();

// Evaluate the PUMA jacobian
                jacobian();

// Position gains
                kp[0]=27.6;
                kd[0]=3.5;
                kp[1]=-71.9;
                kd[1]=-9.0;
                kp[2]=51.5;
                kd[2]=3.7;
                kp[3]=-15.0;
                kd[3]=-1.0;
                kp[4]=-25.2;
                kd[4]=-1.2;
                kp[5]=-10.0;
                kd[5]=-1.0;

                positiond[0]=-0.50;
                positiond[1]=-0.17;
                positiond[2]=0.35;
                positiond[3]=0.0;
                positiond[4]=0.0;
                positiond[5]=0.0;

// Calculate control command
                        for (i=0;i<6;i++)
                        {
                                error[i]=positiond[i]-
position[i];
                                error_dot[i]=(error[i]-
error_old[i])*300.0;

                f_fil[i]=(forced[i]*T*T*wn*wn+f_fil_old
[i]*(2.0*zeta*wn*T+2.0)-
f_fil_way_old[i])/(1.0+2.0*zeta*wn*T+wn*wn*T
*T);

        voltage_out[i]=kp[i]*error[i]+kd[i]*error
_dot[i];
                }

                for (i=0;i<3;i++)
                {
                        fb[i]=0.0;
                        for (j=0;j<3;j++)
                        {
                                fb[i] +=
r[i][j]*f_fil[j];
                        }
                }
                for (i=0;i<3;i++)
                {
                        fev[i]=0.0;
                        for (j=0;j<3;j++)
                        {
                                fev[i] +=
rv[i][j]*fb[j];
                        }
                }

// I can't believe I am trying this unattended
                if (time > 5.0)
                {
                        joint_limit = 1.4;

                        fev[0]=0.0;
                        fev[1]=0.0;
                        if (f_flag == 1)
fev[2]=0.0;

                        for (j=0;j<3;j++)
                        {

        voltage_out[j]=0.0;
                                for
(i=0;i<3;i++)
                                {

        voltage_out[j] += evJl[i][j]*(fv[i]-
0.5*fev[i]);
                                }
                        }


        voltage_out[0]=voltage_out[0]*-1.0;

                voltage_out[2]=voltage_out[2]*-1.0;
                        }

// Implement impedence control to protect the
joints
        voltage_imped=0.02*pow((1.0/(position[
0]-
2.7)),3.0)+0.02*pow((1.0/(position[0]+2.7)),3.0);
        voltage_out[0] += voltage_imped+vg[0];
        if (position[0] > position_old[0])
voltage_out[0] += 1.0;
        if (position[0] < position_old[0])
voltage_out[0] -= 0.9;

        voltage_imped=-
0.02*pow((1.0/(position[1]-0.7)),3.0)-
0.02*pow((1.0/(position[1]+3.7)),3.0);
        voltage_out[1] += voltage_imped+vg[1];
        if (position[1] > position_old[1])
        {
                if (position[1] > -1.57)
voltage_out[1] -= 0.3;
                else  voltage_out[1] -= 0.9;
        }
        if (position[1] < position_old[1])
        {
                if (position[1] > -1.57)
voltage_out[1] += 0.9;
                else voltage_out[1] += 0.6;
        }

        voltage_imped=0.02*pow((1.0/(position[
2]-
joint_limit)),3.0)+0.02*pow((1.0/(position[2]+0.9)
),3.0);
        voltage_out[2] += voltage_imped+vg[2];
        if (position[2] > position_old[2])
voltage_out[2] += 0.47;
        if (position[2] < position_old[2])
voltage_out[2] -= 0.47;

        voltage_imped=-
0.02*pow((1.0/(position[3]-3.2)),3.0)-
0.02*pow((1.0/(position[3]+1.8)),3.0);
        voltage_out[3] += voltage_imped;

        voltage_imped=-
0.02*pow((1.0/(position[4]-1.7)),3.0)-
0.02*pow((1.0/(position[4]+1.7)),3.0);
        voltage_out[4] += voltage_imped;
        voltage_imped=-
0.02*pow((1.0/(position[5]-5.2)),3.0)-
0.02*pow((1.0/(position[5]+5.2)),3.0);
        voltage_out[5] += voltage_imped;

// Convert voltages into integers to output to
trident board
        for (i=0;i<6;i++)
        {
                if (fabs(voltage_out[i]) > 9.9)
voltage_out[i]=9.9*voltage_out[i]/fabs(voltage_out
[i]);
                voltage_int[i]=(int)
(4095.0*(voltage_out[i]+10.0)/20.0);
        }

// Output voltages
        for (i=0;i<6;i++)
        {
                outport(BASE + 0x0030 + 2*i,
voltage_int[i]);
        }

// Save old position values
        position_old[0]=position[0];
        position_old[1]=position[1];
        position_old[2]=position[2];
        position_old[3]=position[3];
        position_old[4]=position[4];
        position_old[5]=position[5];

// Save old error values
        error_old[0]=error[0];
        error_old[1]=error[1];
        error_old[2]=error[2];
        error_old[3]=error[3];
        error_old[4]=error[4];
        error_old[5]=error[5];

// Save old force values
        forced_old[0]=forced[0];
        forced_old[1]=forced[1];
        forced_old[2]=forced[2];
        forced_old[3]=forced[3];
        forced_old[4]=forced[4];
        forced_old[5]=forced[5];

        f_fil_way_old[0]=f_fil_old[0];
        f_fil_way_old[1]=f_fil_old[1];
        f_fil_way_old[2]=f_fil_old[2];
        f_fil_way_old[3]=f_fil_old[3];
        f_fil_way_old[4]=f_fil_old[4];
        f_fil_way_old[5]=f_fil_old[5];

        f_fil_old[0]=f_fil[0];
        f_fil_old[1]=f_fil[1];
        f_fil_old[2]=f_fil[2];
        f_fil_old[3]=f_fil[3];
        f_fil_old[4]=f_fil[4];
        f_fil_old[5]=f_fil[5];

        x_old[0]=x[0];
        x_old[1]=x[1];
        x_old[2]=x[2];
        x_old[3]=x[3];
        x_old[4]=x[4];
        x_old[5]=x[5];

        time = time + 1./300.;

// Take some data
        if (data < 1000)
        {
                data_pts[0][data]=x[0];
                data_pts[1][data]=x[1];
                data_pts[2][data]=x[2];
                data++;
                if (data == 999) data=0;
        }
}
```

```
// error.c

#include "puma.h"

extern double u_vm;
extern double x[6];
extern double xv_old[3];
extern double xv_dot_old[3];
extern double xv_dot_way_old[3];
extern double fv[6];
extern double rv[3][3];
extern int f_flag;

void error_v(void)
{
        double xv[3],xv_dot[3];
        double xcp[3],ycp[3],zcp[3];
        double b[3],db[3],vup[3];
        double xc[3],e[3],xdot[3],d_u_vm,mag;
        double xh[3],yh[3],zh[3];
        int i,flag;
        double kp[3],kv[3];
        double wn,z,T;

        i=0;

        xcp[0]=0.5;
        xcp[1]=0.5;
        xcp[2]=0.5;

        ycp[0]=-0.1;
        ycp[1]=0.0;
        ycp[2]=0.1;

        zcp[0]=0.5;
        zcp[1]=0.3;
        zcp[2]=0.5;

// inverse kinematics of virtual manipuator
        while(i != 30 && flag != 1)
        {
                b[0]=(1.0-u_vm)*(1.0-u_vm);
                b[1]=2.0*(1.0-u_vm)*u_vm;
                b[2]=u_vm*u_vm;


        xc[0]=xcp[0]*b[0]+xcp[1]*b[1]+xcp[2]*b
[2];

        xc[1]=ycp[0]*b[0]+ycp[1]*b[1]+ycp[2]*b
[2];

        xc[2]=zcp[0]*b[0]+zcp[1]*b[1]+zcp[2]*b
[2];

                e[0]=x[0]-xc[0];
                e[1]=x[1]-xc[1];
                e[2]=x[2]-xc[2];

                db[0]=-2.0*(1.0-u_vm);
                db[1]=2.0*(1.0-u_vm)-
2.0*u_vm;
                db[2]=2.0*u_vm;


        xdot[0]=xcp[0]*db[0]+xcp[1]*db[1]+xcp[
2]*db[2];

        xdot[1]=ycp[0]*db[0]+ycp[1]*db[1]+ycp[
2]*db[2];

        xdot[2]=zcp[0]*db[0]+zcp[1]*db[1]+zcp[
2]*db[2];

                d_u_vm=e[0]*xdot[0]+e[1]*xdot[1]+e[2]
*xdot[2];
                        u_vm=u_vm+d_u_vm;

// parameter in bounds
                        if (u_vm < 0.0)
                        {
                                u_vm=0.0;
                                flag=1;
                        }
                        else if (u_vm > 1.0)
                        {
                                u_vm=1.0;
                                flag=1;
                        }
// parameter not changing
                        else if (fabs(d_u_vm) < 0.01)
                        {
                                flag=1;
                        }

                        i++;
                }

// forward kinematics
                b[0]=(1.0-u_vm)*(1.0-u_vm);
                b[1]=2.0*(1.0-u_vm)*u_vm;
                b[2]=u_vm*u_vm;

                xc[0]=xcp[0]*b[0]+xcp[1]*b[1]+xcp[2]*b
[2];
                xc[1]=ycp[0]*b[0]+ycp[1]*b[1]+ycp[2]*b
[2];
                xc[2]=zcp[0]*b[0]+zcp[1]*b[1]+zcp[2]*b
[2];


                db[0]=-2.0*(1.0-u_vm);
                db[1]=2.0*(1.0-u_vm)-2.0*u_vm;
                db[2]=2.0*u_vm;

                xdot[0]=xcp[0]*db[0]+xcp[1]*db[1]+xcp[
2]*db[2];
                xdot[1]=ycp[0]*db[0]+ycp[1]*db[1]+ycp[
2]*db[2];
                xdot[2]=zcp[0]*db[0]+zcp[1]*db[1]+zcp[
2]*db[2];

                mag=sqrt(xdot[0]*xdot[0]+xdot[1]*xdot[
1]+xdot[2]*xdot[2]);
```

```
        zh[0]=xdot[0]/mag;
        zh[1]=xdot[1]/mag;
        zh[2]=xdot[2]/mag;

        vup[0]=0.0;
        vup[1]=0.0;
        vup[2]=1.0;

        yh[0]=zh[1]*vup[2]-vup[1]*zh[2];
        yh[1]=vup[0]*zh[2]-zh[0]*vup[2];
        yh[2]=zh[0]*vup[1]-vup[0]*zh[1];

        mag=sqrt(yh[0]*yh[0]+yh[1]*yh[1]+yh[2
]*yh[2]);

        yh[0]=yh[0]/mag;
        yh[1]=yh[1]/mag;
        yh[2]=yh[2]/mag;

        xh[0]=yh[1]*zh[2]-zh[1]*yh[2];
        xh[1]=zh[0]*yh[2]-yh[0]*zh[2];
        xh[2]=yh[0]*zh[1]-zh[0]*yh[1];

        mag=sqrt(xh[0]*xh[0]+xh[1]*xh[1]+xh[2
]*xh[2]);

        xh[0]=xh[0]/mag;
        xh[1]=xh[1]/mag;
        xh[2]=xh[2]/mag;

        rv[0][0]=xh[0];
        rv[0][1]=xh[1];
        rv[0][2]=xh[2];
        rv[1][0]=yh[0];
        rv[1][1]=yh[1];
        rv[1][2]=yh[2];
        rv[2][0]=zh[0];
        rv[2][1]=zh[1];
        rv[2][2]=zh[2];

// Determine the position of the robot in the
virtual manipulator's
// end effect space
        xv[0]=xh[0]*x[0]+xh[1]*x[1]+xh[2]*x[2]
-xh[0]*xc[0]-xh[1]*xc[1]-xh[2]*xc[2];
        xv[1]=yh[0]*x[0]+yh[1]*x[1]+yh[2]*x[2]
-yh[0]*xc[0]-yh[1]*xc[1]-yh[2]*xc[2];
        xv[2]=zh[0]*x[0]+zh[1]*x[1]+zh[2]*x[2]
-zh[0]*xc[0]-zh[1]*xc[1]-zh[2]*xc[2];

        wn=60.0;
        T=1.0/300.0;

        z=0.7071;

        for (i=0;i<3;i++)
        {
                xv_dot[i]=(wn*wn*T*(xv[i]-
xv_old[i])+xv_dot_old[i]*(2.0+2.0*z*wn*T)-
xv_dot_way_old[i])/(1.0+2.0*z*wn*T+wn*wn*T*
T);
        }

        for (i=0;i<3;i++)
        {

xv_dot_way_old[i]=xv_dot_old[i];
                xv_dot_old[i]=xv_dot[i];
                xv_old[i]=xv[i];
        }

        if (u_vm <= 0.0)
        {
                kp[2]=470.0;
                kv[2]=30.0;
                f_flag=1;
        }
        else if (u_vm >= 1.0)
        {
                kp[2]=470.0;
                kv[2]=30.0;
                f_flag=1;
        }
        else
        {
                kp[2]=0.0;
                kv[2]=0.0;
                f_flag=0;
        }

        kp[0]=470.0;            kv[0]=30.0;
        kp[1]=470.0;            kv[1]=30.0;

        for (i=0;i<3;i++)
        {

        fv[i]=kp[i]*xv[i]+kv[i]*xv_dot[i];
        }
}
```

```
// ft.c

#include "puma.h"
```

```
extern int stop;
extern double forced[6];
extern double r[3][3];
extern double fbasis[6];

void init_force(void)
{
      int status,data;

          printf("Initialize ATI force
transducer\n");
          printf("Written at LARCC\n");
          printf("copyright 1996, Jim Edwards\n");

          status=inportb(STATUS_FT);
          printf("\n\nCheck STATUS_FT register:
%d\n",status);
          status=inportb(CONFIG);
          printf("Check configuration register:
%d\n",status);

      if (inportb(STATUS_FT) & 0x10)
          {
                 data=inportb(PORT_B) << 8 |
inportb(PORT_A);
                 printf("preload 1 %d\n",data);
          }
      if (inportb(STATUS_FT) & 0x10)
          {
                 data=inportb(PORT_B) << 8 |
inportb(PORT_A);
                 printf("preload 2 %d\n",data);
          }

// send CPP to switch to parallel board
          printf("Switch to parallel board\n");
      while((inportb(STATUS_FT) & 0x80) == 0)
sleep(1);
          send(67); // C
          printf("C");
      while((inportb(STATUS_FT) & 0x80) == 0)
sleep(1);
          send(80); // P
          printf("P");
      while((inportb(STATUS_FT) & 0x80) == 0)
sleep(1);
          send(80); // P
          printf("P");
          while((inportb(STATUS_FT) & 0x80) ==
0) sleep(1);
          send(13); // <cr>
          printf("<cr>\n");

// wait for acknowledgment
      while((inportb(STATUS_FT) & 0x10) == 0);
//printf("xx\n");
          data=inportb(PORT_B) << 8 |
inportb(PORT_A);
          printf("%d\n",data);
      while((inportb(STATUS_FT) & 0x10) == 0);
//printf("xx\n");
          data=inportb(PORT_B) << 8 |
inportb(PORT_A);
          printf("%d\n",data);
          while((inportb(STATUS_FT) & 0x10) ==
0); //printf("xx\n");
          data=inportb(PORT_B) << 8 |
inportb(PORT_A);
          printf("%d\n",data);
      while((inportb(STATUS_FT) & 0x10) == 0);
//printf("xx\n");
          data=inportb(PORT_B) << 8 |
inportb(PORT_A);
          printf("%d\n",data);

      if (inportb(STATUS_FT) & 0x10)
          {
                 data=inportb(PORT_B) << 8 |
inportb(PORT_A);
                 printf(".%d\n",data);
          }
          if (inportb(STATUS_FT) & 0x10)
          {
                 data=inportb(PORT_B) << 8 |
inportb(PORT_A);
                 printf("..%d\n",data);
          }
      if (inportb(STATUS_FT) & 0x10)
          {
                 data=inportb(PORT_B) << 8 |
inportb(PORT_A);
                 printf("...%d\n",data);
          }
      if (inportb(STATUS_FT) & 0x10)
          {
                 data=inportb(PORT_B) << 8 |
inportb(PORT_A);
                 printf("....%d\n",data);
          }
      if (inportb(STATUS_FT) & 0x10)
          {
                data=inportb(PORT_B) << 8 |
inportb(PORT_A);
                printf(".....%d\n",data);
        }
        if (inportb(STATUS_FT) & 0x10)
        {
                data=inportb(PORT_B) << 8 |
inportb(PORT_A);
                printf("......%d\n",data);
        }

// send CDB
        printf("\n Set to communicate binary
mode\n");
     while((inportb(STATUS_FT) & 0x80) == 0)
sleep(1);
        send(67);
     while((inportb(STATUS_FT) & 0x80) == 0)
sleep(1);
     while((inportb(STATUS_FT) & 0x10) == 0);
        data=inportb(PORT_B) << 8 |
inportb(PORT_A);
        printf("%d\n",data);

        while((inportb(STATUS_FT) & 0x80) ==
0)  sleep(1);
        send(68);
     while((inportb(STATUS_FT) & 0x80) == 0)
sleep(1);
     while((inportb(STATUS_FT) & 0x10) == 0);
        data=inportb(PORT_B) << 8 |
inportb(PORT_A);
        printf("%d\n",data);
     while((inportb(STATUS_FT) & 0x80) == 0)
sleep(1);
        send(66);
     while((inportb(STATUS_FT) & 0x80) == 0)
sleep(1);
        while((inportb(STATUS_FT) & 0x10) ==
0); //printf("xx\n");
        data=inportb(PORT_B) << 8 |
inportb(PORT_A);
        printf("%d\n",data);
        while((inportb(STATUS_FT) & 0x80) ==
0)  sleep(1);
        send(13);
     while((inportb(STATUS_FT) & 0x80) == 0)
sleep(1);
     while((inportb(STATUS_FT) & 0x10) == 0);
//printf("xx\n");
        data=inportb(PORT_B) << 8 |
inportb(PORT_A);
        printf("%d\n",data);

     while((inportb(STATUS_FT) & 0x10) == 0);
//printf("xx\n");
        data=inportb(PORT_B) << 8 |
inportb(PORT_A);
        printf("%d\n",data);
     while((inportb(STATUS_FT) & 0x10) == 0);
//printf("xx\n");
        data=inportb(PORT_B) << 8 |
inportb(PORT_A);
        printf("%d\n",data);
        while((inportb(STATUS_FT) & 0x10) ==
0); //printf("xx\n");
        data=inportb(PORT_B) << 8 |
inportb(PORT_A);
        printf("%d\n",data);
     while((inportb(STATUS_FT) & 0x10) == 0);
//printf("xx\n");
        data=inportb(PORT_B) << 8 |
inportb(PORT_A);
        printf("%d\n",data);
     while((inportb(STATUS_FT) & 0x10) == 0);
//printf("xx\n");
        data=inportb(PORT_B) << 8 |
inportb(PORT_A);
        printf("%d\n",data);
     while((inportb(STATUS_FT) & 0x10) == 0);
//printf("xx\n");
        data=inportb(PORT_B) << 8 |
inportb(PORT_A);
        printf("%d\n",data);
}

void get_force(void)
{
        int force[7],i,j;

        send(14);

        while((inportb(STATUS_FT) & 0x10) ==
0); //printf("xx\n");
        force[6]=inportb(PORT_B) << 8 |
inportb(PORT_A);

        while((inportb(STATUS_FT) & 0x10) ==
0); //printf("xx\n");
        force[0]=inportb(PORT_B) << 8 |
inportb(PORT_A);

        while((inportb(STATUS_FT) & 0x10) ==
0); //printf("xx\n");
```

```
        force[1]=inportb(PORT_B) << 8 |
inportb(PORT_A);

        while((inportb(STATUS_FT) & 0x10) ==
0); //printf("xx\n");
        force[2]=inportb(PORT_B) << 8 |
inportb(PORT_A);

        while((inportb(STATUS_FT) & 0x10) ==
0); //printf("xx\n");
        force[3]=inportb(PORT_B) << 8 |
inportb(PORT_A);

        while((inportb(STATUS_FT) & 0x10) ==
0); //printf("xx\n");
        force[4]=inportb(PORT_B) << 8 |
inportb(PORT_A);

        while((inportb(STATUS_FT) & 0x10) ==
0); //printf("xx\n");
        force[5]=inportb(PORT_B) << 8 |
inportb(PORT_A);

        if (force[6] != 0)  stop=0;

        forced[0]=((double) (force[0]))*0.1-
fbasis[0]+3.6*r[2][0];
        forced[1]=((double) (force[1]))*0.1-
fbasis[1]+3.6*r[2][1];
        forced[2]=((double) (force[2]))*0.1-
fbasis[2]+3.6*r[2][2];
        forced[3]=((double) (force[3]))*0.005-
fbasis[3];
        forced[4]=((double) (force[4]))*0.005-
fbasis[4];
        forced[5]=((double) (force[5]))*0.005-
fbasis[5];


}


void send(int data)
{
        int msb, lsb;
        long i;

        msb=(data & 0x00) >> 8;
        lsb=data & 0xFF;

        outportb(PORT_C,lsb);
        outportb(PORT_D,msb);
```

```
}
```

```
// gravity.c

#include "puma.h"

extern double vg[6];
extern double position[6];

void gravity(void)
{
        double c2,s2,c23,s23;

        c2=cos(position[1]);
        s2=sin(position[1]);

        c23=cos(position[1]+position[2]);
        s23=sin(position[1]+position[2]);

// gravity compensation
        vg[0]=0.0;
        vg[2]=-1.1201*s23+0.0977*c23;
        vg[1]=0.2400*s2+2.1144*c2-
0.5304*vg[2];
}
```

```
// home.c

#include "puma.h"

extern double encoder_scale[6];
extern double encoder_offset[6];
extern double position[6];
extern double error_old[6];

void home(void)
{
        int i;
        int val[6];
        double positiond[6];
        double kp[6];
        double kd[6];
        double error[6];
        double error_dot[6];
        double voltage_out[6];
        int voltage_int[6];

// Read encoders
```

```
        for (i=0;i<6;i++)
        {
                val[i]=readencoder(i);

        position[i]=encoder_scale[i]*((double)
(val[i]) - encoder_offset[i]);
        }

// Desired trajectory
        positiond[0]=0.0;
        positiond[1]=-1.57;
        positiond[2]=1.57;
        positiond[3]=0.0;
        positiond[4]=0.0;
        positiond[5]=0.0;

// Control law
        kp[0]=27.6;
        kd[0]=3.5;
        kp[1]=-71.9;
        kd[1]=-9.0;
        kp[2]=51.5;
        kd[2]=3.7;
        kp[3]=-5.0;
        kd[3]=0.0;
        kp[4]=-15.2;
        kd[4]=-1.2;
        kp[5]=-5.0;
        kd[5]=0.0;

        for (i=0;i<6;i++)
        {
                error[i]=positiond[i]-position[i];
                error_dot[i]=(error[i]-
error_old[i])*300.0;


        voltage_out[i]=kp[i]*error[i]+kd[i]*error
_dot[i];

                if (fabs(voltage_out[i]) > 9.9)
voltage_out[i]=9.9*voltage_out[i]/fabs(voltage_out
[i]);

                voltage_int[i]=(int)
(4095.0*(voltage_out[i]+10.0)/20.0);
        }

// Output voltages
        for (i=0;i<6;i++)
        {
                        outport(BASE + 0x0030 + 2*i,
voltage_int[i]);
        }

// Save old error values
        error_old[0]=error[0];
        error_old[1]=error[1];
        error_old[2]=error[2];
        error_old[3]=error[3];
        error_old[4]=error[4];
        error_old[5]=error[5];
}
```

```
// init.c

#include "puma.h"

extern long DISCRETE;
extern double encoder_scale[6];
extern double encoder_offset[6];
extern double error_old[6];
extern double position_old[6];
extern double forced_old[6];
extern double f_fil_old[6];
extern double f_fil_way_old[6];
extern double x_old[6];
extern double xv_old[3];
extern double xv_dot_old[3];
extern double xv_dot_way_old[3];
extern double fbasis[6];

void init(void)
{
        int i;

        DISCRETE=0x0000;

        encoder_scale[0]=0.00010035;
        encoder_scale[1]=-0.000073156;
        encoder_scale[2]=0.000117;
        encoder_scale[3]=-0.000082663;
        encoder_scale[4]=-0.000087376;
        encoder_scale[5]=-0.00016377;

        encoder_offset[0]=0.0;
        encoder_offset[1]=-21472.0;
        encoder_offset[2]=-13426.0;
        encoder_offset[3]=8000.0;
        encoder_offset[4]=0.0;
        encoder_offset[5]=0.0;
```

```
    error_old[0]=0.0;
    error_old[1]=0.0;
    error_old[2]=0.0;
    error_old[3]=0.0;
    error_old[4]=0.0;
    error_old[5]=0.0;

    position_old[0]=0.0;
    position_old[1]=0.0;
    position_old[2]=0.0;
    position_old[3]=0.0;
    position_old[4]=0.0;
    position_old[5]=0.0;

    forced_old[0] = 0.0;
    forced_old[1] = 0.0;
    forced_old[2] = 0.0;
    forced_old[3] = 0.0;
    forced_old[4] = 0.0;
    forced_old[5] = 0.0;

    f_fil_old[0] = 0.0;
    f_fil_old[1] = 0.0;
    f_fil_old[2] = 0.0;
    f_fil_old[3] = 0.0;
    f_fil_old[4] = 0.0;
    f_fil_old[5] = 0.0;

    f_fil_way_old[0] = 0.0;
    f_fil_way_old[1] = 0.0;
    f_fil_way_old[2] = 0.0;
    f_fil_way_old[3] = 0.0;
    f_fil_way_old[4] = 0.0;
    f_fil_way_old[5] = 0.0;

    x_old[0] = 0.0;
    x_old[1] = 0.0;
    x_old[2] = 0.0;
    x_old[3] = 0.0;
    x_old[4] = 0.0;
    x_old[5] = 0.0;

    for (i=0;i<3;i++)
    {
            xv_old[i]=0.0;
            xv_dot_old[i]=0.0;
            xv_dot_way_old[i]=0.0;
    }

    for (i=0;i<6;i++)
    {
                    fbasis[i]=0.0;
            }
}
```

```
// invkin.c

#include "puma.h"
#include "puma.ext"

void invkin(void)
{
        double l[4],theta[4][6],pi,k,v1,v2,v3;
        double valid[4],limits[6][2],dist,sdist;
        double ca,sa,cb,sb,cc,sc,r[4][4];
        double c1,s1,c23,s23,c4,s4,c5,s5,c6,s6;
        double r11,r12,r21,r22,r23,r13,r33;
        int i,j,select;

        pi=3.14159;

        l[0]=0.4318;
        l[1]=0.15005;
        l[2]=-0.0191;
        l[3]=0.4331;

        valid[0]=1;
        valid[1]=1;
        valid[2]=1;
        valid[3]=1;

        select=0;

        limits[0][0]=-2.92;
limits[0][1]=2.89;
        limits[1][0]=-3.92;
limits[1][1]=0.82;
        limits[2][0]=-1.01;
limits[2][1]=4.27;
        limits[3][0]=-2.02;
limits[3][1]=3.36;
        limits[4][0]=-1.87;
limits[4][1]=1.86;
        limits[5][0]=-5.36;
limits[5][1]=5.35;


// theta 1 calculation
        theta[0][0]=atan2(x[1],x[0])-
atan2(l[1],sqrt(pow(x[0],2.0)+pow(x[1],2.0)-
pow(l[1],2.0)));
```

```
        theta[1][0]=atan2(x[1],x[0])-atan2(l[1],-
sqrt(pow(x[0],2.0)+pow(x[1],2.0)-pow(l[1],2.0)));
        theta[2][0]=theta[0][0];
        theta[3][0]=theta[1][0];

// theta 3 calculation
        k=(pow(x[0],2.0)+pow(x[1],2.0)+pow(x[2
],2.0)-pow(l[0],2.0)-pow(l[1],2.0)-pow(l[2],2.0)-
pow(l[3],2.0))/(2.0*l[0]);
        theta[0][2]=atan2(k,sqrt(pow(l[2],2.0)+po
w(l[3],2.0)-pow(k,2.0)))-atan2(l[2],l[3]);
        theta[1][2]=theta[0][2];
        theta[2][2]=atan2(k,-
sqrt(pow(l[2],2.0)+pow(l[3],2.0)-pow(k,2.0)))-
atan2(l[2],l[3]);
        theta[3][2]=theta[2][2];
        for (i=0;i<4;i++)
        {
                if (theta[i][2] < -1.01)
theta[i][2]=theta[i][2]+2.0*pi;
        }

// theta 2 calculation
        for (i=0;i<4;i++)
        {
                v1=l[2]+l[0]*cos(theta[i][2]);

        v2=x[0]*cos(theta[i][0])+x[1]*sin(theta[i]
[0]);
                v3=l[3]+l[0]*sin(theta[i][2]);
                theta[i][1]=atan2(v3*v2-
x[2]*v1,v1*v2+x[2]*v3)-theta[i][2];
                if (theta[i][1] > 0.82)
theta[i][1]=theta[i][1]-2.0*pi;
        }

// check joint limits
        for (i=0;i<4;i++)
        {
                for (j=0;j<3;j++)
                {
                        if ((limits[j][0] <
theta[i][j]) && (theta[i][j] < limits[j][1]))
                        {
                                valid[i]=1;
                        }
                        else
                        {
                                valid[i]=0;
                                break;
                        }
                }
        }

// find the closest valid solution to the old position
        for (i=0;i<4;i++)
        {
                if (valid[i] == 1)
                {
                        select=i;
                        sdist=0.0;
                        for (j=0;j<3;j++)
                        {

        sdist=sdist+fabs(positiond[j]-theta[i][j]);
                        }
                        break;
                }
        }

        for (i=select+1;i<4;i++)
        {
                if (valid[i] == 1)
                {
                        dist=0.0;
                        for (j=0;j<3;j++)
                        {

        dist=dist+fabs(positiond[j]-theta[i][j]);
                        }
                        if (dist < sdist)
                        {
                                sdist=dist;
                                select=i;
                        }
                }
        }

// select the solution
        for (i=0;i<3;i++)
positiond[i]=theta[select][i];
}
```

```
// jacobian.c

#include "puma.h"

extern double position[6];
extern double evJl[3][3];
extern double eJw[3][3];
extern double r[3][3];
extern double rv[3][3];
extern double eJl[3][3];

void jacobian(void)
{
        double c2,c3,s3,c23,s23;
        double l1,l2,l3,l4;
        double bJl[3][3];
        int i,j,k;

        l1=0.4318;
        l2=0.15005;
        l3=-0.0191;
        l4=0.4331;

        c2=cos(position[1]);

        c3=cos(position[2]);
        s3=sin(position[2]);

        c23=cos(position[1]+position[2]);
        s23=sin(position[1]+position[2]);

        eJl[0][0]=-c23*l2;
        eJl[0][1]=s3*l1+l4;
        eJl[0][2]=l4;

        eJl[1][0]=c2*l1+c23*l3+s23*l4;
        eJl[1][1]=0.0;
        eJl[1][2]=0.0;

        eJl[2][0]=-s23*l2;
        eJl[2][1]=-c3*l1-l3;
        eJl[2][2]=-l3;

        for (i=0;i<3;i++)
        {
                for (j=0;j<3;j++)
                {
                        bJl[i][j]=0.0;
                        for (k=0;k<3;k++)
                        {
                                bJl[i][j] +=
r[i][k]*eJl[k][j];
                        }
                }
        }
        for (i=0;i<3;i++)
        {
                for (j=0;j<3;j++)
                {
                        evJl[i][j]=0.0;
                        for (k=0;k<3;k++)
                        {
                                evJl[i][j] +=
rv[i][k]*bJl[k][j];
                        }
                }
        }

        eJw[0][0]=-s23;
        eJw[0][1]=0.0;
        eJw[0][2]=0.0;

        eJw[1][0]=0.0;
        eJw[1][1]=1.0;
        eJw[1][2]=1.0;

        eJw[2][0]=c23;
        eJw[2][1]=0.0;
        eJw[2][2]=0.0;
}
```

```
// forkin.c

#include "puma.h"

extern double position[6];
extern double x[6];
extern double r[3][3];

void forkin(void)
{
        double
c1,s1,c2,s2,c23,s23,c4,s4,c5,s5,c6,s6;
        double l[5];
        double
v1,v2,v3,v4,v5,v6,v7,v8,v9,v10,v11;

        l[1]=0.4318;
        l[2]=0.15005;
        l[3]=-0.0191;
        l[4]=0.4331;

        c1=cos(position[0]);
        s1=sin(position[0]);

        c2=cos(position[1]);
        s2=sin(position[1]);

        c23=cos(position[1]+position[2]);
        s23=sin(position[1]+position[2]);
        c4=cos(position[3]);
        s4=sin(position[3]);

        c5=cos(position[4]);
        s5=sin(position[4]);

        c6=cos(position[5]);
        s6=sin(position[5]);

        x[0]=c1*(c23*l[3]+s23*l[4]+c2*l[1])-
s1*l[2];
        x[1]=s1*(c23*l[3]+s23*l[4]+c2*l[1])+c1
*l[2];
        x[2]=-s23*l[3]+c23*l[4]-s2*l[1];
        x[3]=0.0;
        x[4]=0.0;
        x[5]=0.0;

        v1=c4*c5*c6-s4*s6;
        v2=s5*c6;
        v3=c23*v1-s23*v2;
        v4=s4*c5*c6+c4*s6;

        r[0][0]=c1*v3-s1*v4;
        r[1][0]=s1*v3+c1*v4;
        r[2][0]=-s23*v1-c23*v2;

        v5=c4*c5*s6+s4*c6;
        v6=s5*s6;
        v7=-c23*v5+s23*v6;
        v8=s4*c5*s6-c4*c6;

        r[0][1]=c1*v7+s1*v8;
        r[1][1]=s1*v7-c1*v8;
        r[2][1]=s23*v5+c23*v6;

        v9=c4*s5;
        v10=c23*v9+s23*c5;
        v11=s4*s5;

        r[0][2]=c1*v10-s1*v11;
        r[1][2]=s1*v10+c1*v11;
        r[2][2]=-s23*v9+c23*c5;

}
```

```
// loadencoder.c

#include "puma.h"

void loadencoder(int channel,int value)
{
        outport(ENC_LOAD + 2*channel,
value);
}
```

```
// main.c

#include "puma.h"

// global variables
int board; // daq card board number
int err_num; // daq card error number
int stop=1; // flag used to stop program in event
of ft error
long DISCRETE; // PUMA discrete input control
word
double encoder_scale[6]; // scale factor to convert
encoder counts to radians
double encoder_offset[6]; // encoder counts in
home position
double position[6];
double position_old[6];
double error_old[6];
double forced[6];
double forced_old[6];
double f_fil_old[6];
double f_fil_way_old[6];
double vg[6];
double data_pts[3][1500];
double time = 0.0;
int data = 0;
double evJl[3][3];
double eJw[3][3];
double x[6];
double x_old[6];
double joint_limit = 4.;
double r[3][3];
double u_vm=0.0;
double xv_old[3];
double xv_dot_old[3];
double xv_dot_way_old[3];
double fv[6];
double rv[3][3];
double eJl[3][3];
double fbasis[6];
int f_flag;

void main(void)
{
        int ctr1=1; // counter 1
        int ctr2=2; // counter 2
        int overflow; // counter overflow error
flag
        int i; // counting variable
        int safety=1; // flag used to stop program
if control loop takes to long
        int count=0; // counter variable used in
homing robot
        unsigned int count1; // counter 1 value
        unsigned int count2; // counter 2 value
        double time1=0.0; // time associated
with counter 1
        double time2=0.0; // time associated
with counter 2
        double dt=1.0/300.0; // desired control
loop refresh time
        double error; // actual control loop
refresh time
        int kbh=1; // flag used to determine if
keyboard event has occured
        FILE *out; // output data file

// PUMA controller
        printf("PUMA control designed at
LARCC\n");
        printf("Laboratory for Advanced Robotics
and Computer Control\n");
        printf("Iowa State University\n");
        printf("\n\nwritten by Jim Edwards and
Brian Miller\n");
        printf("All rights reserved\n");

// Get the board number of the daq card
        board=getDeviceToUse();

// Initialize some gobal variables
        init();

// Initialize force transducer
        init_force();
        get_force();
        printf("%f %f %f %f %f
%f\n",forced[0],forced[1],forced[2],forced[3],force
d[4],forced[5]);
        fbasis[0]=forced[0];
        fbasis[1]=forced[1];
        fbasis[2]=forced[2]+3.6;
        fbasis[3]=forced[3];
        fbasis[4]=forced[4];
        fbasis[5]=forced[5];

// Calibrate encoders to scratch mark values
        calibration2();

// Set up counter 1 and 2
        err_num=CTR_EvCount(board,ctr1,1,1);
        ErrPrint("CTR_EvCount",err_num);

        err_num=CTR_EvCount(board,ctr2,0,0);
        ErrPrint("CTR_EvCount",err_num);

// Enable arm power
        DISCRETE=DISCRETE | POWER_BIT;
        outport(DISCRETE_REG,DISCRETE);
        printf("Turn arm power on!!!\n");

        while(count<500 && safety && stop)
        {
// if a keyboard event occurs switch the flag
                if (kbhit()) kbh=0;

// perform control loop if no keyboard event
                if (kbh)
                {
                        control();
                }
// if keyboard event send the robot home
                else
                {
                        home();
// it only has so long to get there
                        count++;
                }


                do
                {
// read counter 1 and 2

        err_num=CTR_EvRead(board,ctr2,&over
flow,&count2);

        err_num=CTR_EvRead(board,ctr1,&over
flow,&count1);

// based on counters determine what time it is
                        time1=(double)
(count1)*0.000001;
                        time1 += (double)
(count2)*0.065535;

// if it's time run the loop again
                } while((time1-time2) < dt-
.00002);
// determine how long the loop took
                error=time1-time2;

// if it took to long better stop
                if (fabs((error-dt) > 0.00003))
safety=0;

// save current time for next loop
                time2=time1;

// do it all again
        }

// Disable arm power
        DISCRETE=DISCRETE |
(~POWER_BIT);
        outport(DISCRETE_REG,DISCRETE);

        printf("\n\ncounter: %d\n",count);
// Output some data
        if((out=fopen("out.dat","wt"))==NULL)
        {
                printf("Cannot open output file
OUT.DAT.\n");
                exit(1);
        }

        for (i=0;i<999;i++)
        {
                fprintf(out,"%f    %f
        %f\n",data_pts[0][i],data_pts[1][i],data_p
ts[2][i]);
        }
        fclose(out);
}
```

```
// path.c

#include "puma.h"
#include "puma.ext"

void path()
{
        double u,pi,radius,center[3];
        int counter_max;

        pi=3.14159;
        radius=0.3;
        center[0]=0.2;
        center[1]=0.0;
        center[2]=0.5;
        counter_max=2001;

        counter++;
        if (counter == counter_max)
        {
                counter=0;
                if (direction == 1)  direction=-1;
                else  direction=1;
        }

        u=((double) counter)/((double)
(counter_max-1));
        if (direction == -1)  u=1.0-u;

        x[0]=radius*cos(pi*u/2.0)+center[0];
        x[1]=radius*sin(pi*u/2.0)+center[1];
        x[2]=center[2];
}
```

```
// puma.h

#include <stdio.h>
#include <dos.h>
#include <float.h>
#include <stdlib.h>
#include <bios.h>
#include <conio.h>
#include <io.h>
#include <math.h>
#include <string.h>
#include "nidaq.h"
#include "nidaqcns.h"
#include "nidaqerr.h"

// prototypes
void init(void);
int getDeviceToUse(void);
void ErrPrint(char[],int);
int CTR_EvCount(int,int,int,int);
int CRT_EvRead(int,int,int *,int *);
int readencoder(int);
void calibration2(void);
void loadencoder(int,int);
void control(void);
void home(void);
//void path(void);
//void invkin(void);
void jacobian(void);
```

```
//void forkin(void);
//void error_v(void);
void send(int);
void init_force(void);
void get_force(void);
void gravity(void);

// define some addresses that will be needed
#define BASE
0x0300
#define AD_MUX_SELECT
BASE + 0x002C
#define AD_START_PULSE
BASE + 0x001E
#define DISCRETE_REG
BASE + 0x002E
#define STATUS_REG
BASE + 0x000C
#define AD_VALUE
BASE + 0x001C
#define ENC_COUNTER
                    BASE + 0x0010
#define ENC_LOAD
                    BASE + 0x0020

// define some bit masks that will be needed
#define AD_MASK
0x4000
#define AD_STATUS_MASK
0x4000
#define POWER_BIT
                              0x0001

// addresses for the ATI force transducer
#define FT_BASE                    0x280
#define PORT_A          FT_BASE
#define PORT_B          FT_BASE + 0x01
#define PORT_C          FT_BASE + 0x02
#define PORT_D          FT_BASE + 0x03
#define STATUS_FT       FT_BASE + 0x04
#define CONFIG          FT_BASE + 0x05
```

```
// readencoder.c

#include "puma.h"

int readencoder(int channel)
{
        int val;
        val=(int)(inport(ENC_COUNTER +
2*channel));
        return val;
}
```

NURBS Surface

```
// error.c

#include "puma.h"

void error(pumaFile* pumaData)
{
        double xv[3],xv_dot[3];
        double xcp[3][3],ycp[3][3],zcp[3][3];
        double bu[3],dbu[3],bv[3],dbv[3];
        double xc[3],tu[3],tv[3],mag;
        double xh[3],yh[3],zh[3];
        double e[3], d_u_vm, d_v_vm;
        int i, j, k;
        int flag=0;
        double kp[3],kv[3];
        double wn,z;
        double spring,damper;
        int normal=0;
        double fend[6], fbase[6], rv[3][3];
        double rd[3][3], xv_ori[3][3], xyz[3],
xyzd[3];
        double xyz_dot[3];

        i=0;

        if (normal == 1)
        {
                xcp[0][0]=0.5;
                xcp[0][1]=0.5;
                xcp[0][2]=0.3;

                xcp[1][0]=0.5;
                xcp[1][1]=0.5;
                xcp[1][2]=0.3;

                xcp[2][0]=0.5;
                xcp[2][1]=0.5;
                xcp[2][2]=0.3;

                ycp[0][0]=-0.1;
                ycp[0][1]=-0.1;
                ycp[0][2]=-0.1;
                ycp[1][0]=0.0;
                ycp[1][1]=0.0;
                ycp[1][2]=0.0;

                ycp[2][0]=0.1;
                ycp[2][1]=0.1;
                ycp[2][2]=0.1;

                zcp[0][0]=0.3;
                zcp[0][1]=0.4;
                zcp[0][2]=0.5;

                zcp[1][0]=0.3;
                zcp[1][1]=0.4;
                zcp[1][2]=0.5;

                zcp[2][0]=0.3;
                zcp[2][1]=0.4;
                zcp[2][2]=0.5;
        }
        else
        {
                xcp[0][0]=0.5;
                xcp[0][1]=0.5;
                xcp[0][2]=0.5;

                xcp[1][0]=0.5;
                xcp[1][1]=0.1;
                xcp[1][2]=0.5;

                xcp[2][0]=0.5;
                xcp[2][1]=0.5;
                xcp[2][2]=0.5;

                ycp[0][0]=-0.1;
                ycp[0][1]=-0.1;
                ycp[0][2]=-0.1;

                ycp[1][0]=0.0;
                ycp[1][1]=0.0;
                ycp[1][2]=0.0;

                ycp[2][0]=0.1;
                ycp[2][1]=0.1;
                ycp[2][2]=0.1;

                zcp[0][0]=0.3;
                zcp[0][1]=0.4;
                zcp[0][2]=0.5;

                zcp[1][0]=0.3;
                zcp[1][1]=0.4;
                zcp[1][2]=0.5;

                zcp[2][0]=0.3;
                zcp[2][1]=0.4;
                zcp[2][2]=0.5;
        }

// inverse kinematics of virtual manipuator
        while(i != 30 && flag != 1)
        {
// evaluate basis functions
                bu[0]=(1.0-pumaData-
>u_vm)*(1.0-pumaData->u_vm);
                bu[1]=2.0*(1.0-pumaData-
>u_vm)*pumaData->u_vm;
                bu[2]=pumaData-
>u_vm*pumaData->u_vm;


                bv[0]=(1.0-pumaData-
>v_vm)*(1.0-pumaData->v_vm);
                bv[1]=2.0*(1.0-pumaData-
>v_vm)*pumaData->v_vm;
                bv[2]=pumaData-
>v_vm*pumaData->v_vm;

// determine what the cartesian coordinates are for
u_vm and v_vm

        xc[0]=bu[0]*(xcp[0][0]*bv[0]+xcp[0][1]*
bv[1]+xcp[0][2]*bv[2])

+bu[1]*(xcp[1][0]*bv[0]+xcp[1][1]*bv[1]+xcp[1][
2]*bv[2])

+bu[2]*(xcp[2][0]*bv[0]+xcp[2][1]*bv[1]+xcp[2][
2]*bv[2]);


        xc[1]=bu[0]*(ycp[0][0]*bv[0]+ycp[0][1]*
bv[1]+ycp[0][2]*bv[2])

+bu[1]*(ycp[1][0]*bv[0]+ycp[1][1]*bv[1]+ycp[1][
2]*bv[2])

+bu[2]*(ycp[2][0]*bv[0]+ycp[2][1]*bv[1]+ycp[2][
2]*bv[2]);


        xc[2]=bu[0]*(zcp[0][0]*bv[0]+zcp[0][1]*
bv[1]+zcp[0][2]*bv[2])
```

```
+bu[1]*(zcp[1][0]*bv[0]+zcp[1][1]*bv[1]+zcp[1][
2]*bv[2])

+bu[2]*(zcp[2][0]*bv[0]+zcp[2][1]*bv[1]+zcp[2][
2]*bv[2]);

// determine the error between the robot is and the
point on the surface
                    e[0]=pumaData->x[0]-xc[0];
                    e[1]=pumaData->x[1]-xc[1];
                    e[2]=pumaData->x[2]-xc[2];

// evaluate derivatives of the basis functions
                    dbu[0]=-2.0*(1.0-pumaData-
>u_vm);
                    dbu[1]=2.0*(1.0-pumaData-
>u_vm)-2.0*pumaData->u_vm;
                    dbu[2]=2.0*pumaData->u_vm;

                    dbv[0]=-2.0*(1.0-pumaData-
>v_vm);
                    dbv[1]=2.0*(1.0-pumaData-
>v_vm)-2.0*pumaData->v_vm;
                    dbv[2]=2.0*pumaData->v_vm;

// determine the u and v direction tangents

        tu[0]=dbu[0]*(xcp[0][0]*bv[0]+xcp[0][1]
*bv[1]+xcp[0][2]*bv[2])

+dbu[1]*(xcp[1][0]*bv[0]+xcp[1][1]*bv[1]+xcp[1]
[2]*bv[2])

+dbu[2]*(xcp[2][0]*bv[0]+xcp[2][1]*bv[1]+xcp[2]
[2]*bv[2]);


        tu[1]=dbu[0]*(ycp[0][0]*bv[0]+ycp[0][1]
*bv[1]+ycp[0][2]*bv[2])

+dbu[1]*(ycp[1][0]*bv[0]+ycp[1][1]*bv[1]+ycp[1]
[2]*bv[2])

+dbu[2]*(ycp[2][0]*bv[0]+ycp[2][1]*bv[1]+ycp[2]
[2]*bv[2]);


        tu[2]=dbu[0]*(zcp[0][0]*bv[0]+zcp[0][1]
*bv[1]+zcp[0][2]*bv[2])

+dbu[1]*(zcp[1][0]*bv[0]+zcp[1][1]*bv[1]+zcp[1]
[2]*bv[2])

+dbu[2]*(zcp[2][0]*bv[0]+zcp[2][1]*bv[1]+zcp[2]
[2]*bv[2]);


        tv[0]=bu[0]*(xcp[0][0]*dbv[0]+xcp[0][1]
*dbv[1]+xcp[0][2]*dbv[2])

+bu[1]*(xcp[1][0]*dbv[0]+xcp[1][1]*dbv[1]+xcp[
1][2]*dbv[2])

+bu[2]*(xcp[2][0]*dbv[0]+xcp[2][1]*dbv[1]+xcp[
2][2]*dbv[2]);


        tv[1]=bu[0]*(ycp[0][0]*dbv[0]+ycp[0][1]
*dbv[1]+ycp[0][2]*dbv[2])

+bu[1]*(ycp[1][0]*dbv[0]+ycp[1][1]*dbv[1]+ycp[
1][2]*dbv[2])

+bu[2]*(ycp[2][0]*dbv[0]+ycp[2][1]*dbv[1]+ycp[
2][2]*dbv[2]);


        tv[2]=bu[0]*(zcp[0][0]*dbv[0]+zcp[0][1]
*dbv[1]+zcp[0][2]*dbv[2])

+bu[1]*(zcp[1][0]*dbv[0]+zcp[1][1]*dbv[1]+zcp[
1][2]*dbv[2])

+bu[2]*(zcp[2][0]*dbv[0]+zcp[2][1]*dbv[1]+zcp[
2][2]*dbv[2]);

// determine how much to change the parameter
estimates u_vm and v_vm

        d_u_vm=e[0]*tu[0]+e[1]*tu[1]+e[2]*tu[2
];

        d_v_vm=e[0]*tv[0]+e[1]*tv[1]+e[2]*tv[2
];

// update the parameter estimates
                    pumaData->u_vm=pumaData-
>u_vm+d_u_vm;
                    pumaData->v_vm=pumaData-
>v_vm+d_v_vm;
```

```
// parameter in bounds
                if (pumaData->u_vm < 0.0)
                {
                        pumaData->u_vm=0.0;
                }
                else if (pumaData->u_vm > 1.0)
                {
                        pumaData->u_vm=1.0;
                }

                if (pumaData->v_vm < 0.0)
                {
                        pumaData->v_vm=0.0;
                }
                else if (pumaData->v_vm > 1.0)
                {
                        pumaData->v_vm=1.0;
                }

// parameter not changing
                else if
(sqrt(d_u_vm*d_u_vm+d_v_vm*d_v_vm) < 0.02)
                {
                        flag=1;
                }

                i++;
        }

// forward kinematics
        mag=sqrt(tu[0]*tu[0]+tu[1]*tu[1]+tu[2]*t
u[2]);

        xh[0]=tu[0]/mag;
        xh[1]=tu[1]/mag;
        xh[2]=tu[2]/mag;

        mag=sqrt(tv[0]*tv[0]+tv[1]*tv[1]+tv[2]*t
v[2]);

        tv[0]=tv[0]/mag;
        tv[1]=tv[1]/mag;
        tv[2]=tv[2]/mag;

        zh[0]=xh[1]*tv[2]-tv[1]*xh[2];
        zh[1]=tv[0]*xh[2]-xh[0]*tv[2];
        zh[2]=xh[0]*tv[1]-tv[0]*xh[1];

        mag=sqrt(zh[0]*zh[0]+zh[1]*zh[1]+zh[2
]*zh[2]);

        zh[0]=zh[0]/mag;
        zh[1]=zh[1]/mag;
        zh[2]=zh[2]/mag;

        yh[0]=zh[1]*xh[2]-xh[1]*zh[2];
        yh[1]=xh[0]*zh[2]-zh[0]*xh[2];
        yh[2]=zh[0]*xh[1]-xh[0]*zh[1];

        mag=sqrt(yh[0]*yh[0]+yh[1]*yh[1]+yh[2
]*yh[2]);

        yh[0]=yh[0]/mag;
        yh[1]=yh[1]/mag;
        yh[2]=yh[2]/mag;

        rv[0][0]=xh[0];
        rv[0][1]=xh[1];
        rv[0][2]=xh[2];
        rv[1][0]=yh[0];
        rv[1][1]=yh[1];
        rv[1][2]=yh[2];
        rv[2][0]=zh[0];
        rv[2][1]=zh[1];
        rv[2][2]=zh[2];

// Determine the position of the robot in the
virtual manipulator's
// end effect space
        xv[0]=xh[0]*pumaData-
>x[0]+xh[1]*pumaData->x[1]+xh[2]*pumaData-
>x[2]-xh[0]*xc[0]-xh[1]*xc[1]-xh[2]*xc[2];
        xv[1]=yh[0]*pumaData-
>x[0]+yh[1]*pumaData->x[1]+yh[2]*pumaData-
>x[2]-yh[0]*xc[0]-yh[1]*xc[1]-yh[2]*xc[2];
        xv[2]=zh[0]*pumaData-
>x[0]+zh[1]*pumaData->x[1]+zh[2]*pumaData-
>x[2]-zh[0]*xc[0]-zh[1]*xc[1]-zh[2]*xc[2];

        if (normal == 1)
        {
                rd[0][0]=xh[0];
                rd[0][1]=yh[0];
                rd[0][2]=zh[0];
                rd[1][0]=xh[1];
                rd[1][1]=yh[1];
                rd[1][2]=zh[1];
                rd[2][0]=xh[2];
                rd[2][1]=yh[2];
                rd[2][2]=zh[2];
        }
        else
        {
```

```
		rd[0][0]=0.0;
		rd[0][1]=0.0;
		rd[0][2]=1.0;
		rd[1][0]=1.0;
		rd[1][1]=0.0;
		rd[1][2]=0.0;
		rd[2][0]=0.0;
		rd[2][1]=1.0;
		rd[2][2]=0.0;
	}

	for (i=0;i<3;i++)
	{
		for (j=0;j<3;j++)
		{
			xv_ori[i][j]=0.0;
			for (k=0;k<3;k++)
			{
				xv_ori[i][j] +=
rv[i][k]*pumaData->r[k][j];
			}
		}
	}

	xyz[1]=atan2(-
xv_ori[2][0],sqrt(xv_ori[0][0]*xv_ori[0][0]+xv_or
i[1][0]*xv_ori[1][0]));
	if (fabs(xyz[1]-1.5708) < 0.01)
	{
		xyz[2]=0.0;

	xyz[0]=atan2(xv_ori[0][1],xv_ori[1][1]);
	}
	else if (fabs(xyz[1]+1.5708) < 0.01)
	{
		xyz[2]=0.0;
		xyz[0]=-
atan2(xv_ori[0][1],xv_ori[1][1]);
	}
	else
	{

	xyz[2]=atan2(xv_ori[1][0],xv_ori[0][0]);

	xyz[0]=atan2(xv_ori[2][1],xv_ori[2][2]);
	}

	for (i=0;i<3;i++)
	{
		for (j=0;j<3;j++)
		{
			xv_ori[i][j]=0.0;
			for (k=0;k<3;k++)
			{
				xv_ori[i][j] +=
rv[i][k]*rd[k][j];
			}
		}
	}

	xyzd[1]=atan2(-
xv_ori[2][0],sqrt(xv_ori[0][0]*xv_ori[0][0]+xv_or
i[1][0]*xv_ori[1][0]));
	if (fabs(xyzd[1]-1.5708) < 0.01)
	{
		xyzd[2]=0.0;

	xyzd[0]=atan2(xv_ori[0][1],xv_ori[1][1]);
	}
	else if (fabs(xyzd[1]+1.5708) < 0.01)
	{
		xyzd[2]=0.0;
		xyzd[0]=-
atan2(xv_ori[0][1],xv_ori[1][1]);
	}
	else
	{

	xyzd[2]=atan2(xv_ori[1][0],xv_ori[0][0]);

	xyzd[0]=atan2(xv_ori[2][1],xv_ori[2][2]);
	}

	xyz[0]=-xyzd[0]+xyz[0];
	xyz[1]=-xyzd[1]+xyz[1];
	xyz[2]=-xyzd[2]+xyz[2];

	wn=60.0;
	z=0.7071;

	for (i=0;i<3;i++)
	{
		xv_dot[i]=(wn*wn*pumaData-
>dt*(xv[i]-pumaData->xv_old[i])+pumaData-
>xv_dot_old[i]*(2.0+2.0*z*wn*pumaData->dt)-
pumaData-
>xv_dot_way_old[i])/(1.0+2.0*z*wn*pumaData-
>dt+wn*wn*pumaData->dt*pumaData->dt);
		xyz_dot[i]=(wn*wn*pumaData-
>dt*(xyz[i]-pumaData->xyz_old[i])+pumaData-
>xyz_dot_old[i]*(2.0+2.0*z*wn*pumaData->dt)-
pumaData-
>xyz_dot_way_old[i])/(1.0+2.0*z*wn*pumaData-
>dt+wn*wn*pumaData->dt*pumaData->dt);
```

```
	}

	for (i=0;i<3;i++)
	{
		pumaData-
>xv_dot_way_old[i]=pumaData->xv_dot_old[i];
		pumaData-
>xv_dot_old[i]=xv_dot[i];
		pumaData->xv_old[i]=xv[i];
		pumaData-
>xyz_dot_way_old[i]=pumaData->xyz_dot_old[i];
		pumaData-
>xyz_dot_old[i]=xyz_dot[i];
		pumaData->xyz_old[i]=xyz[i];
	}

	spring=400.0;
	damper=30.0;

	if (pumaData->u_vm <= 0.0)
	{
		kp[0]=spring;
		kv[0]=damper;
	}
	else if (pumaData->u_vm >= 1.0)
	{
		kp[0]=spring;
		kv[0]=damper;
	}
	else
	{
		kp[0]=0.0;
		kv[0]=0.0;
	}

	if (pumaData->v_vm <= 0.0)
	{
		kp[1]=spring;
		kv[1]=damper;
	}
	else if (pumaData->v_vm >= 1.0)
	{
		kp[1]=spring;
		kv[1]=damper;
	}
	else
	{
		kp[1]=0.0;
		kv[1]=0.0;
	}

//	kp[0]=spring;			kv[0]=damper;
//	kp[1]=spring;			kv[1]=damper;
	kp[2]=spring;			kv[2]=damper;

	for (i=0;i<3;i++)
	{

	fend[i]=kp[i]*xv[i]+kv[i]*xv_dot[i];

	fend[i+3]=30.0*xyz[i]+2.0*xyz_dot[i];
	}

// force end effector vm to base puma
	for (i=0; i<3; i++)
	{
		fbase[i]=0.0;
		for (j=0; j<3; j++)
		{

	fbase[i]=fbase[i]+rv[j][i]*fend[j];
		}
	}

	for (i=0; i<3; i++)
	{
		fbase[i+3]=0.0;
		for (j=0; j<3; j++)
		{

	fbase[i+3]=fbase[i+3]+rv[j][i]*fend[j+3];
		}
	}

// force base puma to end effector puma
	for (i=0; i<3; i++)
	{
		pumaData->fv[i]=0.0;
		for (j=0; j<3; j++)
		{
			pumaData-
>fv[i]=pumaData->fv[i]+pumaData-
>r[j][i]*fbase[j];
		}
	}

	for (i=0; i<3; i++)
	{
		pumaData->fv[i+3]=0.0;
		for (j=0; j<3; j++)
		{
			pumaData-
>fv[i+3]=pumaData->fv[i+3]+pumaData-
>r[j][i]*fbase[j+3];
```

```
                }
        }

// zero some stuff
/*      fend[3]=0.0;
        fend[4]=0.0;
        fend[5]=0.0;
        fbase[3]=0.0;
        fbase[4]=0.0;
        fbase[5]=0.0;
        pumaData->fv[3]=0.0;
        pumaData->fv[4]=0.0;
        pumaData->fv[5]=0.0;*/
}
```

```
// friction.c

#include "puma.h"

void friction(pumaFile* pumaData)
{
        int i;
        double tau=0.05305;

        if (pumaData->theta[0] > pumaData-
>theta_old[0]) pumaData->v_fric[0]=1.0;
        if (pumaData->theta[0] <= pumaData-
>theta_old[0]) pumaData->v_fric[0]=-0.9;
        pumaData->v_fric[0]=(pumaData-
>v_fric[0]*pumaData->dt+pumaData-
>v_fric_old[0]*tau)/(pumaData->dt+tau);

        if (pumaData->theta[1] > pumaData-
>theta_old[1])
        {
                if (pumaData->theta[1] > -1.57)
pumaData->v_fric[1]=-0.3;
                else pumaData->v_fric[1]=-0.9;
        }
        if (pumaData->theta[1] <= pumaData-
>theta_old[1])
        {
                if (pumaData->theta[1] > -1.57)
pumaData->v_fric[1]=0.9;
                else pumaData->v_fric[1]=0.6;
        }
        pumaData->v_fric[1]=(pumaData-
>v_fric[1]*pumaData->dt+pumaData-
>v_fric_old[1]*tau)/(pumaData->dt+tau);

        if (pumaData->theta[2] > pumaData-
>theta_old[2]) pumaData->v_fric[2]=0.47;
        if (pumaData->theta[2] <= pumaData-
>theta_old[2]) pumaData->v_fric[2]=-0.47;
        pumaData->v_fric[2]=(pumaData-
>v_fric[2]*pumaData->dt+pumaData-
>v_fric_old[2]*tau)/(pumaData->dt+tau);

        if (pumaData->theta[3] > pumaData-
>theta_old[3]) pumaData->v_fric[3]=-0.35;
        else if (pumaData->theta[3] <=
pumaData->theta_old[3]) pumaData-
>v_fric[3]=0.35;
        else pumaData->v_fric[3]=0.0;
        pumaData->v_fric[3]=(pumaData-
>v_fric[3]*pumaData->dt+pumaData-
>v_fric_old[3]*tau)/(pumaData->dt+tau);

        if (pumaData->theta[4] > pumaData-
>theta_old[4]) pumaData->v_fric[4]=-0.4;
        else if (pumaData->theta[4] < pumaData-
>theta_old[4]) pumaData->v_fric[4]=0.4;
        else pumaData->v_fric[4]=0.0;
        pumaData->v_fric[4]=(pumaData-
>v_fric[4]*pumaData->dt+pumaData-
>v_fric_old[4]*tau)/(pumaData->dt+tau);

        if (pumaData->theta[5] > pumaData-
>theta_old[5]) pumaData->v_fric[5]=-0.5;
        else if (pumaData->theta[5] < pumaData-
>theta_old[5]) pumaData->v_fric[5]=0.5;
        else pumaData->v_fric[5]=0.0;
        pumaData->v_fric[5]=(pumaData-
>v_fric[5]*pumaData->dt+pumaData-
>v_fric_old[5]*tau)/(pumaData->dt+tau);

        for (i=0;i<6;i++)
        {
                pumaData-
>v_fric_old[i]=pumaData->v_fric[i];
        }
}
```

```
// gravity.c

#include "puma.h"

void gravity(pumaFile* pumaData)
{
        double c2,s2,c23,s23;
        c2=cos(pumaData->theta[1]);
        s2=sin(pumaData->theta[1]);

        c23=cos(pumaData->theta[1]+pumaData-
>theta[2]);
        s23=sin(pumaData->theta[1]+pumaData-
>theta[2]);

// gravity compensation
        pumaData->vg[0]=0.0;
        pumaData->vg[2]=-
1.1201*s23+0.0977*c23;
        pumaData-
>vg[1]=0.2400*s2+2.1144*c2-0.5304*pumaData-
>vg[2];
        pumaData->vg[3]=0.0;
        pumaData->vg[4]=0.0;
        pumaData->vg[5]=0.0;
}
```

```
// impedence.c

#include "puma.h"

void impedence(pumaFile* pumaData)
{
        pumaData-
>vim[0]=0.02*pow((1.0/(pumaData->theta[0]-
2.7)),3.0)+0.02*pow((1.0/(pumaData-
>theta[0]+2.7)),3.0);

        pumaData->vim[1]=-
0.02*pow((1.0/(pumaData->theta[1]-0.7)),3.0)-
0.02*pow((1.0/(pumaData->theta[1]+3.7)),3.0);

        pumaData-
>vim[2]=0.02*pow((1.0/(pumaData->theta[2]-
pumaData-
>jlimit3)),3.0)+0.02*pow((1.0/(pumaData-
>theta[2]+0.9)),3.0);

        pumaData->vim[3]=-
0.02*pow((1.0/(pumaData->theta[3]-3.2)),3.0)-
0.02*pow((1.0/(pumaData->theta[3]+1.8)),3.0);

        pumaData->vim[4]=-
0.02*pow((1.0/(pumaData->theta[4]-1.7)),3.0)-
0.02*pow((1.0/(pumaData->theta[4]+pumaData-
>jlimit5)),3.0);

        pumaData->vim[5]=-
0.02*pow((1.0/(pumaData->theta[5]-5.2)),3.0)-
0.02*pow((1.0/(pumaData->theta[5]+5.2)),3.0);
}
```

```
// jacobian.c

#include "puma.h"

void jacobian(pumaFile* pumaData)
{
        double
c1,s1,c2,s2,c3,s3,c23,s23,c4,s4,c5,s5,c6,s6;
        double l[5];
        l[1]=0.4318;
        l[2]=0.15005;
        l[3]=-0.0191;
        l[4]=0.4331;

        c1=cos(pumaData->theta[0]);
        s1=sin(pumaData->theta[0]);

        c2=cos(pumaData->theta[1]);
        s2=sin(pumaData->theta[1]);

        c3=cos(pumaData->theta[2]);
        s3=sin(pumaData->theta[2]);

        c23=cos(pumaData->theta[1]+pumaData-
>theta[2]);
        s23=sin(pumaData->theta[1]+pumaData-
>theta[2]);

        c4=cos(pumaData->theta[3]);
        s4=sin(pumaData->theta[3]);

        c5=cos(pumaData->theta[4]);
        s5=sin(pumaData->theta[4]);

        c6=cos(pumaData->theta[5]);
        s6=sin(pumaData->theta[5]);

// jacobian
        pumaData->eJr[0][0]=c5*c6*(-
c23*c4*l[2]+s4*(c2*l[1]+c23*l[3]+s23*l[4]))+s6*
(c23*s4*l[2]+c4*(c2*l[1]+c23*l[3]+s23*l[4]))+s5
*c6*s23*l[2];
```

```
        pumaData-
>eJr[0][1]=c5*c6*(c4*(s3*l[1]+l[4]))+s6*(-
s4*(s3*l[1]+l[4]))-s5*c6*(-c3*l[1]-l[3]);
        pumaData->eJr[0][2]=c5*c6*c4*l[4]-
s6*s4*l[4]+s5*c6*l[3];
        pumaData->eJr[0][3]=0.0;
        pumaData->eJr[0][4]=0.0;
        pumaData->eJr[0][5]=0.0;

        pumaData->eJr[1][0]=-c5*s6*(-
c23*c4*l[2]+s4*(c2*l[1]+c23*l[3]+s23*l[4]))+c6*
(c23*s4*l[2]+c4*(c2*l[1]+c23*l[3]+s23*l[4]))-
s5*s6*s23*l[2];
        pumaData->eJr[1][1]=-
c5*s6*(c4*(s3*l[1]+l[4]))+c6*(-
s4*(s3*l[1]+l[4]))+s5*s6*(-c3*l[1]-l[3]);
        pumaData->eJr[1][2]=-c5*s6*c4*l[4]-
c6*s4*l[4]-s5*s6*l[3];
        pumaData->eJr[1][3]=0.0;
        pumaData->eJr[1][4]=0.0;
        pumaData->eJr[1][5]=0.0;

        pumaData->eJr[2][0]=s5*(-
c23*c4*l[2]+s4*(c2*l[1]+c23*l[3]+s23*l[4]))-
c5*s23*l[2];
        pumaData-
>eJr[2][1]=s5*(c4*(s3*l[1]+l[4]))+c5*(-c3*l[1]-
l[3]);
        pumaData->eJr[2][2]=s5*c4*l[4]-c5*l[3];
        pumaData->eJr[2][3]=0.0;
        pumaData->eJr[2][4]=0.0;
        pumaData->eJr[2][5]=0.0;

        pumaData->eJr[3][0]=s23*(s4*s6-
c4*c5*c6)-c23*s5*c6;
        pumaData->eJr[3][1]=s4*c5*c6+c4*s6;
        pumaData->eJr[3][2]=s4*c5*c6+c4*s6;
        pumaData->eJr[3][3]=-s5*c6;
        pumaData->eJr[3][4]=s6;
        pumaData->eJr[3][5]=0.0;

        pumaData-
>eJr[4][0]=s23*(c4*c5*s6+s4*c6)+c23*s5*s6;
        pumaData->eJr[4][1]=-s4*c5*s6+c4*c6;
        pumaData->eJr[4][2]=-s4*c5*s6+c4*c6;
        pumaData->eJr[4][3]=s5*s6;
        pumaData->eJr[4][4]=c6;
        pumaData->eJr[4][5]=0.0;

        pumaData->eJr[5][0]=-
s23*c4*s5+c23*c5;
        pumaData->eJr[5][1]=s4*s5;
        pumaData->eJr[5][2]=s4*s5;
        pumaData->eJr[5][3]=c5;
        pumaData->eJr[5][4]=0.0;
        pumaData->eJr[5][5]=1.0;
}
```

```
// kinematics.c

#include "puma.h"

void kinematics(pumaFile* pumaData)
{
        double
c1,s1,c2,s2,c3,s3,c23,s23,c4,s4,c5,s5,c6,s6;
        double l[5];
        double
v1,v2,v3,v4,v5,v6,v7,v8,v9,v10,v11;

        l[1]=0.4318;
        l[2]=0.15005;
        l[3]=-0.0191;
        l[4]=0.4331;

        c1=cos(pumaData->theta[0]);
        s1=sin(pumaData->theta[0]);

        c2=cos(pumaData->theta[1]);
        s2=sin(pumaData->theta[1]);

        c3=cos(pumaData->theta[2]);
        s3=sin(pumaData->theta[2]);

        c23=cos(pumaData->theta[1]+pumaData-
>theta[2]);
        s23=sin(pumaData->theta[1]+pumaData-
>theta[2]);

        c4=cos(pumaData->theta[3]);
        s4=sin(pumaData->theta[3]);

        c5=cos(pumaData->theta[4]);
        s5=sin(pumaData->theta[4]);

        c6=cos(pumaData->theta[5]);
        s6=sin(pumaData->theta[5]);

        pumaData-
>x[0]=c1*(c23*l[3]+s23*l[4]+c2*l[1])-s1*l[2];
        pumaData-
>x[1]=s1*(c23*l[3]+s23*l[4]+c2*l[1])+c1*l[2];
```

```
        pumaData->x[2]=-s23*l[3]+c23*l[4]-
s2*l[1];
        pumaData->x[3]=0.0;
        pumaData->x[4]=0.0;
        pumaData->x[5]=0.0;

        v1=c4*c5*c6-s4*s6;
        v2=s5*c6;
        v3=c23*v1-s23*v2;
        v4=s4*c5*c6+c4*s6;

        pumaData->r[0][0]=c1*v3-s1*v4;
        pumaData->r[1][0]=s1*v3+c1*v4;
        pumaData->r[2][0]=-s23*v1-c23*v2;

        v5=c4*c5*s6+s4*c6;
        v6=s5*s6;
        v7=-c23*v5+s23*v6;
        v8=s4*c5*s6-c4*c6;

        pumaData->r[0][1]=c1*v7+s1*v8;
        pumaData->r[1][1]=s1*v7-c1*v8;
        pumaData->r[2][1]=s23*v5+c23*v6;

        v9=c4*s5;
        v10=c23*v9+s23*c5;
        v11=s4*s5;

        pumaData->r[0][2]=c1*v10-s1*v11;
        pumaData->r[1][2]=s1*v10+c1*v11;
        pumaData->r[2][2]=-s23*v9+c23*c5;
}
```

```
// main.c

#include "puma.h"

void main(void)
{
// robot stuff
        pumaFile *pumaData;
        int stop;
        int homecount;

// window's stuff
        HANDLE hprocess;
        HANDLE hthread;
        int processerror;

// timer stuff
        BOOL result;
        LARGE_INTEGER lifrequency;
        LARGE_INTEGER licount;
        LONGLONG frequency;
        double dfrequency;
        LONGLONG startcount;
        LONGLONG count;
        double currenttime;
        double dtactual;
        double dterror;
        double dtmax;

// error flags
        int timererror;
        int timeroverrun;
        int DeviceStop;
        int errorSocket;

// socket stuff
        int err;
        char szDataSend[100];
        int gcount;

// data file stuff
        double data[3][2000];
        int datalength=2000;
        int datacount;
        int datacycle;
        int datamax;
        int fileerror;
        FILE *out;

// general stuff (counter and the like)
        int i;

////////////////////////////////////////////////////////////////////
// Taking Care of Business
////////////////////////////////////////////////////////////////////

        printf("PUMA control program\n");
        printf("written by Jim Edwards for
LARCC\n");
        printf("All rights reserved\n\n\n\n");

////////////////////////////////////////////////////////////////////
// Code Initialization Section
////////////////////////////////////////////////////////////////////
// set counter error flag to pass
        timererror=1;

// set counter overrun flag to pass
        timeroverrun=1;
```

```
// start taking data at zero
        datacount=0;

// set data pass to zero
        datacycle=0;

// set process error flag to pass
        processerror=0;

// set maximum delta-t to zero
        dtmax=0.0;

// set stop to pass
        stop=1;

// set homecount to zero
        homecount=0;

// set socket error to none
        errorSocket=0;

// set graphics dump counter to zero
        gcount=0;

/////////////////////////////////////////////////////////////
////// Hardware Initialization
/////////////////////////////////////////////////////////////
// get process handle
        hprocess=GetCurrentProcess();

// set process priority
        result=SetPriorityClass(hprocess,
REALTIME_PRIORITY_CLASS);
        if (result == 0)  processerror=1;

// get thread handle
        hthread=GetCurrentThread();

// set thread priority
        result=SetThreadPriority(hthread,
THREAD_PRIORITY_TIME_CRITICAL);
        if (result == 0)  processerror=2;

// allocate memory for puma structure
        pumaData=(pumaFile
*)malloc(sizeof(pumaFile));

// connect to the puma kernel device
        DeviceStop=1;
        pumaData-
>PumaDevice=HwNewDevice(NULL);
        HwSetErrorHandler(pumaData-
>PumaDevice, MyErrorHandler);
        if (!HwConnectDevice(pumaData-
>PumaDevice, "puma"))
        {
                printf("Failed to connect to puma
device!\n");
                HwDeleteDevice(pumaData-
>PumaDevice);
                DeviceStop=0;
        }

// setup puma
        pumaInitialization(pumaData);

// open socket - useSocket = 1 use socket, = 0
don't use socket
        pumaData->useSocket=1;
        pumaData->activeSocket=0;
        openSocket(pumaData);

// test socket
        testSocket(pumaData);

// get frequency of high performance counter
        result=QueryPerformanceFrequency(&lifr
equency);
        if (result == TRUE)
        {
                frequency=lifrequency.QuadPart;
                dfrequency=((double)
frequency);
                printf("clock frequency:  %f
MHz\n\n\n\n",dfrequency);
        }
        else
        {

        printf("QueryPerformanceFrequency:
failure\n");
                timererror=0;
        }

// get starting count
        printf("\n\n\nTurn Arm Power On!!!!\n");
        result=QueryPerformanceCounter(&licou
nt);
        if (result == TRUE)
        {
                startcount=licount.QuadPart;
        }
        else
        {
        printf("QueryPerformanceCoumter:
failure\n");
                timererror=0;
        }
// disengage the brakes
        HwOutpw(pumaData->PumaDevice,
0x02e, 0x0001);

////////////////////////////////////////////////////////////////
// Main Control Loop
////////////////////////////////////////////////////////////////

        while((homecount < 2000) &&
(DeviceStop == 1) && (timererror == 1) &&
(timeroverrun == 1) && (processerror == 0))
        {
// control code
                if (kbhit()) stop=0;
                if (stop == 1)
                {
                pumaControl(pumaData);
                }
                else
                {
                homecount++;
                        pumaHome(pumaData);
                }

// increment graphics dump counter
                gcount++ ;

// send data to graphics engine
                if (gcount == 5)
                {
                gcount=0;
// but only if there is an active socket for
communication
                        if (pumaData-
>activeSocket == 1)
                {

        sprintf(szDataSend,"%4.3f %4.3f %4.3f
%4.3f %4.3f %4.3f %4.3f ",
                        pumaData->time,
                                pumaData-
>theta[0],
                                pumaData-
>theta[1],
                                pumaData-
>theta[2],
                                pumaData-
>theta[3],
                                pumaData-
>theta[4],
                                pumaData-
>theta[5]);

                        err=send(pumaData->hSock,
(LPSTR) szDataSend, 51, 0) ;
                                if
(err==SOCKET_ERROR) errorSocket=1;
                        }
                        }

// timing code
                        do
                        {
// get the current count of performance counter

        result=QueryPerformanceCounter(&licou
nt);
                                if (result == TRUE)
                                {

        count=licount.QuadPart;
// convert into time since program started

        currenttime=((double) (count-
startcount))/dfrequency;

                                }
                                else
                                {

        printf("QueryPerformanceCounter:
failure\n");
                                timererror=0;
                                }

                                dtactual=currenttime-
pumaData->time;
                        } while(dtactual < pumaData-
>dt);

// get maximum delta-t
                        if (dtactual > dtmax)
dtmax=dtactual;

// get error in delta-t
                        dterror=dtactual-pumaData->dt;
```

```
			if (fabs(dterror) > pumaData-
>dt)  timeroverrun=0;

// take some data
			if (stop == 1)
			{
				// time
//
		data[0][datacount]=pumaData->time;
				// fresh frequency
//
		data[1][datacount]=1.0/dtactual;
				// voltage to axis 5
//
		data[2][datacount]=pumaData-
>voltage_out[4];


		data[0][datacount]=pumaData->x[0];

		data[1][datacount]=pumaData->x[1];

		data[2][datacount]=pumaData->x[2];


				if (datacount == 1999)
				{
					datacount=0;
					datacycle=1;
				}
				else  datacount++;
			}

// update absolute time base
			pumaData->time=pumaData-
>time+pumaData->dt;
		} // end main control loop

// engage the brakes
		HwOutpw(pumaData->PumaDevice,
0x02e, 0x0000);

/////////////////////////////////////////////////////////////////
// Hardware Clean-Up
/////////////////////////////////////////////////////////////////
// kernal device
		HwDeleteDevice(pumaData-
>PumaDevice);

// close socket
		closeSocket(pumaData);

/////////////////////////////////////////////////////////////////
// Take some data
/////////////////////////////////////////////////////////////////

// open the data file
		if ((out=fopen("out.dat","wt"))==NULL)
fileerror=0;
		else
		{
// write data
				fileerror=1;

				fprintf(out,"max dt is
%f\n\n\n",dtmax);

				if (datacycle == 1)
datamax=datalength;
				else  datamax=datacount;

				for (i=0; i<datamax; i++)
				{
					fprintf(out,"%f, %f,
%f\n",data[0][i],data[1][i],data[2][i]);
				}

// close file
				fclose(out);
		}

/////////////////////////////////////////////////////////////////
// Final Error Messages
/////////////////////////////////////////////////////////////////
		printf("\n\n\nError Messages:\n");
		if (timererror == 0)  printf("timer
malfunction\n");
		else if (timeroverrun == 0)  printf("timer
over run\n");
		else if (DeviceStop == 0)  printf("DriverX
error\n");
		else if (fileerror == 0)  printf("could not
open data file\n");
		else if (processerror == 1)  printf("could
not set process priority\n");
		else if (processerror == 2)  printf("could
not set thread priority\n");
		else if (errorSocket == 1)  printf("error
sending data over socket\n");
		else  printf("all went well\n");


		Sleep(3000);
}
```

```
//////////////////////////////////////////////////////////
////// DriverX Error Handler
//////////////////////////////////////////////////////////
void MyErrorHandler(HWDEVICE* pDevice,
DWORD nError)
{
        printf("Critical DriverX error: %d\n",
nError);
        exit(nError);
}
```

```
// puma.h

// include files
#include <windows.h>
#include <winsock.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include "DriverX.h"

// structures
typedef struct
{
// needed for all
        HWDEVICE* PumaDevice;
        double dt;
        double time;
        double encoder_scale[6];
        double encoder_offset[6];
        double theta[6];
        double voltage_out[6];

// socket stuff
        SOCKET hSock;
        int useSocket;
        int activeSocket;

// kinematics
        double x[6];
        double r[3][3];
        double eJr[6][6];

// virtual manipulator
        double fv[6];
        double u_vm;
        double v_vm;
        double xv_old[3];
        double xv_dot_old[3];
        double xv_dot_way_old[3];
        double xyz_old[3];
        double xyz_dot_old[3];
        double xyz_dot_way_old[3];

// needed for me
        int first_flag;
        int last_flag;
        double kp[6];
        double kd[6];
        double error[6];
        double errorold[6];
        double errordot[6];
        double thetad[6];
        double theta_old[6];
        double thetao[6];
        double timeh;
        double vg[6];
        double v_fric[6];
        double v_fric_old[6];
        double vim[6];
        double jlimit3;
        double jlimit5;
} pumaFile;

// prototypes
void main(void);
void MyErrorHandler(HWDEVICE * , DWORD);
void pumaInitialization(pumaFile *);
void pumaControl(pumaFile *);
void pumaHome(pumaFile *);
void openSocket(pumaFile *);
void closeSocket(pumaFile *);
void testSocket(pumaFile *);
void gravity(pumaFile *);
void friction(pumaFile *);
void impedence(pumaFile *);
void kinematics(pumaFile *);
void jacobian(pumaFile *);
void error(pumaFile *);
```

```
// pumaControl.c

#include "puma.h"

void pumaControl(pumaFile* pumaData)
{
        short val[6];
        int voltage_int[6];
        int i, j;
        double thetaf[6];
        double tf=5.0;

//  read encoders
        val[0]=HwInpw(pumaData-
>PumaDevice, 0x010);
        val[1]=HwInpw(pumaData-
>PumaDevice, 0x012);
        val[2]=HwInpw(pumaData-
>PumaDevice, 0x014);
        val[3]=HwInpw(pumaData-
>PumaDevice, 0x016);
        val[4]=HwInpw(pumaData-
>PumaDevice, 0x018);
        val[5]=HwInpw(pumaData-
>PumaDevice, 0x01a);

//  convert encoders to radians
        for (i=0; i<6; i++)
        {
                pumaData->theta[i]=pumaData-
>encoder_scale[i]*(((double) val[i]) - pumaData-
>encoder_offset[i]);
        }

//  gravity compensation
        gravity(pumaData);

//  forward kinematics and Jacobian
        kinematics(pumaData);

//  virtual manipulator control
        error(pumaData);

//  evaluate jacobian
        jacobian(pumaData);

//  friction compensation
        friction(pumaData);

//  impedence protection
        impedence(pumaData);

//  first time through get current position
        if (pumaData->first_flag==1)
        {
                pumaData-
>thetao[0]=pumaData->theta[0];
                pumaData-
>thetao[1]=pumaData->theta[1];
                pumaData-
>thetao[2]=pumaData->theta[2];
                pumaData-
>thetao[3]=pumaData->theta[3];
                pumaData-
>thetao[4]=pumaData->theta[4];
                pumaData-
>thetao[5]=pumaData->theta[5];
                pumaData->first_flag=2;
        }

//  final position
        thetaf[0]=-0.4965;
        thetaf[1]=0.3013;
        thetaf[2]=-0.0805;
        thetaf[3]=0.4976;
        thetaf[4]=1.3767;
        thetaf[5]=1.4526;

//  do cubic spline interpolation
        if (pumaData->time <= tf)
        {
                pumaData-
>thetad[0]=pumaData->thetao[0]-3.0*(pumaData-
>thetao[0]-thetaf[0])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[0]-thetaf[0])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                pumaData-
>thetad[1]=pumaData->thetao[1]-3.0*(pumaData-
>thetao[1]-thetaf[1])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[1]-thetaf[1])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                pumaData-
>thetad[2]=pumaData->thetao[2]-3.0*(pumaData-
>thetao[2]-thetaf[2])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[2]-thetaf[2])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                pumaData-
>thetad[3]=pumaData->thetao[3]-3.0*(pumaData-
>thetao[3]-thetaf[3])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[3]-thetaf[3])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                pumaData-
>thetad[4]=pumaData->thetao[4]-3.0*(pumaData-
>thetao[4]-thetaf[4])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[4]-thetaf[4])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                    pumaData-
>thetad[5]=pumaData->thetao[5]-3.0*(pumaData-
>thetao[5]-thetaf[5])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[5]-thetaf[5])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
          }
// after tf stay put at final position
          else if (pumaData->time > tf)
          {
                    pumaData->thetad[0]=thetaf[0];
                    pumaData->thetad[1]=thetaf[1];
                    pumaData->thetad[2]=thetaf[2];
                    pumaData->thetad[3]=thetaf[3];
                    pumaData->thetad[4]=thetaf[4];
                    pumaData->thetad[5]=thetaf[5];
          }

//
// control section
//
          for (i=0;i<6;i++)
          {
// calculate error
                    pumaData->error[i]=pumaData-
>thetad[i]-pumaData->theta[i];

// calculate rate of change of the error
                    pumaData-
>errordot[i]=(pumaData->error[i]-pumaData-
>errorold[i])/pumaData->dt;

// evaluate local PD control law
                    pumaData-
>voltage_out[i]=pumaData->kp[i]*pumaData-
>error[i]+pumaData->kd[i]*pumaData-
>errordot[i];
          }

// impedence based control law
          if (pumaData->time > 6.0)
          {
                    for (j=0;j<3;j++)
                    {
                    pumaData->voltage_out[j]=0.0;
                              for (i=0;i<3;i++)
                              {
                    pumaData->voltage_out[j] +=
pumaData->eJr[i][j]*pumaData->fv[i];
                              }
                    }
                    for (j=3;j<6;j++)
                    {
                    pumaData->voltage_out[j]=0.0;
                              for (i=3;i<6;i++)
                              {
                    pumaData->voltage_out[j] +=
pumaData->eJr[i][j]*pumaData->fv[i];
                              }
                    }

                    pumaData-
>voltage_out[0]=pumaData->voltage_out[0]*-1.0;
                    pumaData-
>voltage_out[2]=pumaData->voltage_out[2]*-1.0;
          }

// Convert voltages into integers to output to
trident board
          for (i=0;i<6;i++)
          {
                    pumaData-
>voltage_out[i]=pumaData-
>voltage_out[i]+pumaData->vg[i]+pumaData-
>v_fric[i]+pumaData->vim[i];
                    if (fabs(pumaData-
>voltage_out[i]) > 9.9)
          pumaData-
>voltage_out[i]=9.9*pumaData-
>voltage_out[i]/fabs(pumaData->voltage_out[i]);
                    voltage_int[i]=(int)
(4095.0*(pumaData->voltage_out[i]+10.0)/20.0);
          }

// Output voltages to trident hardware
          HwOutpw(pumaData->PumaDevice,
0x030, voltage_int[0]);
          HwOutpw(pumaData->PumaDevice,
0x032, voltage_int[1]);
          HwOutpw(pumaData->PumaDevice,
0x034, voltage_int[2]);
          HwOutpw(pumaData->PumaDevice,
0x036, voltage_int[3]);
          HwOutpw(pumaData->PumaDevice,
0x038, voltage_int[4]);
          HwOutpw(pumaData->PumaDevice,
0x03a, voltage_int[5]);

// save some old information
```

```
        for (i=0;i<6;i++)
        {
                pumaData-
>errorold[i]=pumaData->error[i];
                pumaData-
>theta_old[i]=pumaData->theta[i];
        }
}
```

```
// pumaHome.c

#include "puma.h"

void pumaHome(pumaFile* pumaData)
{
        short val[6];
        int voltage_int[6];
        int i;
        double thetaf[6];
        double localtime;
        double tf=5.0;

// read encoders
        val[0]=HwInpw(pumaData-
>PumaDevice, 0x010);
        val[1]=HwInpw(pumaData-
>PumaDevice, 0x012);
        val[2]=HwInpw(pumaData-
>PumaDevice, 0x014);
        val[3]=HwInpw(pumaData-
>PumaDevice, 0x016);
        val[4]=HwInpw(pumaData-
>PumaDevice, 0x018);
        val[5]=HwInpw(pumaData-
>PumaDevice, 0x01a);

// convert encoders to radians
        for (i=0; i<6; i++)
        {
                pumaData->theta[i]=pumaData-
>encoder_scale[i]*(((double) val[i]) - pumaData-
>encoder_offset[i]);
        }

// first time through get current position
        if (pumaData->last_flag==1)
        {
                pumaData-
>thetao[0]=pumaData->theta[0];
                pumaData-
>thetao[1]=pumaData->theta[1];
                pumaData-
>thetao[2]=pumaData->theta[2];
                pumaData-
>thetao[3]=pumaData->theta[3];
                pumaData-
>thetao[4]=pumaData->theta[4];
                pumaData-
>thetao[5]=pumaData->theta[5];
                pumaData->last_flag=0;
                pumaData->timeh=pumaData-
>time;
        }

// final position
        thetaf[0]=0.0;
        thetaf[1]=-1.57;
        thetaf[2]=1.57;
        thetaf[3]=0.0;
        thetaf[4]=0.0;
        thetaf[5]=0.0;

// time that home has been running
        localtime=pumaData->time-pumaData-
>timeh;

// do cubic spline interpolation
        if (localtime <= tf)
        {
                pumaData-
>thetad[0]=pumaData->thetao[0]-3.0*(pumaData-
>thetao[0]-
thetaf[0])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[0]-
thetaf[0])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[1]=pumaData->thetao[1]-3.0*(pumaData-
>thetao[1]-
thetaf[1])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[1]-
thetaf[1])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[2]=pumaData->thetao[2]-3.0*(pumaData-
>thetao[2]-
thetaf[2])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[2]-
thetaf[2])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[3]=pumaData->thetao[3]-3.0*(pumaData-
>thetao[3]-
thetaf[3])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[3]-
thetaf[3])*localtime*localtime*localtime/(tf*tf*tf);
                    pumaData-
>thetad[4]=pumaData->thetao[4]-3.0*(pumaData-
>thetao[4]-
thetaf[4])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[4]-
thetaf[4])*localtime*localtime*localtime/(tf*tf*tf);
                    pumaData-
>thetad[5]=pumaData->thetao[5]-3.0*(pumaData-
>thetao[5]-
thetaf[5])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[5]-
thetaf[5])*localtime*localtime*localtime/(tf*tf*tf);
          }
// after tf stay put in the final position
          else if (localtime > tf)
          {
                    pumaData->thetad[0]=thetaf[0];
                    pumaData->thetad[1]=thetaf[1];
                    pumaData->thetad[2]=thetaf[2];
                    pumaData->thetad[3]=thetaf[3];
                    pumaData->thetad[4]=thetaf[4];
                    pumaData->thetad[5]=thetaf[5];
          }


//
// control section
//
          for (i=0;i<6;i++)
          {
// calculate error
                    pumaData->error[i]=pumaData-
>thetad[i]-pumaData->theta[i];

// calculate rate of change of the error
                    pumaData-
>errordot[i]=(pumaData->error[i]-pumaData-
>errorold[i])/pumaData->dt;

// evaluate local PD control law
                    pumaData-
>voltage_out[i]=pumaData->kp[i]*pumaData-
>error[i]+pumaData->kd[i]*pumaData-
>errordot[i];
          }

// Convert voltages into integers to output to
trident board
          for (i=0;i<6;i++)
          {
                    pumaData-
>voltage_out[i]=pumaData-
>voltage_out[i];//+pumaData->vg[i]+pumaData-
>v_fric[i]+pumaData->vim[i];
                    if (fabs(pumaData-
>voltage_out[i]) > 9.9)
          pumaData-
>voltage_out[i]=9.9*pumaData-
>voltage_out[i]/fabs(pumaData->voltage_out[i]);
                    voltage_int[i]=(int)
(4095.0*(pumaData->voltage_out[i]+10.0)/20.0);
          }

// Output voltages to trident hardware
          HwOutpw(pumaData->PumaDevice,
0x030, voltage_int[0]) ;
          HwOutpw(pumaData->PumaDevice,
0x032, voltage_int[1]) ;
          HwOutpw(pumaData->PumaDevice,
0x034, voltage_int[2]) ;
          HwOutpw(pumaData->PumaDevice,
0x036, voltage_int[3]) ;
          HwOutpw(pumaData->PumaDevice,
0x038, voltage_int[4]) ;
          HwOutpw(pumaData->PumaDevice,
0x03a, voltage_int[5]) ;

// save some old information
          for (i=0;i<6;i++)
          {
                    pumaData-
>errorold[i]=pumaData->error[i] ;
                    pumaData-
>theta_old[i]=pumaData->theta[i] ;
  }
}
```

```
// pumaInitialization.c

#include "puma.h"

void pumaInitialization(pumaFile* pumaData)
{
          double frequency;
          int i;

// desired refresh rate (Hz)
          frequency=300.0;

// desired delta-t
```

```
        pumaData->dt=1.0/frequency;

// initialize absolute time base to zero
        pumaData->time=0.0;

// set some joint limits for impedence fields
        pumaData->jlimit3=4.0;
        pumaData->jlimit5=1.7;

// set flags for slow up and down
        pumaData->first_flag=1;
        pumaData->last_flag=1;

// encoder stuff
        pumaData-
>encoder_scale[0]=0.00010035;
        pumaData->encoder_scale[1]=-
0.000073156;
        pumaData->encoder_scale[2]=0.000117;
        pumaData->encoder_scale[3]=-
0.000082663;
        pumaData->encoder_scale[4]=-
0.000087376;
        pumaData->encoder_scale[5]=-
0.00016377;

        pumaData->encoder_offset[0]=0.0;
        pumaData->encoder_offset[1]=-21472.0;
        pumaData->encoder_offset[2]=-13426.0;
        pumaData->encoder_offset[3]=8000.0;
        pumaData->encoder_offset[4]=0.0;
        pumaData->encoder_offset[5]=0.0;

// initialize feedback gains
        pumaData->kp[0]=118.0;
        pumaData->kd[0]=15.4;
        pumaData->kp[1]=-288.0;
        pumaData->kd[1]=-24.0;
        pumaData->kp[2]=200.0;
        pumaData->kd[2]=20.0;
        pumaData->kp[3]=-15.0;
        pumaData->kd[3]=-2.0;
        pumaData->kp[4]=-25.2;
        pumaData->kd[4]=-2.2;
        pumaData->kp[5]=-10.0;
        pumaData->kd[5]=-2.0;

        pumaData->u_vm=0.0;
        pumaData->v_vm=0.0;

// initialize some variables
        for (i=0; i<6; i++)
        {
                pumaData->errorold[i]=0.0;
                        // error values
                pumaData->theta_old[i]=0.0;
                        // angular positions
                pumaData->v_fric_old[i]=0.0;
                // friction voltages
        }

        for (i=0; i<3; i++)
        {
                pumaData->xv_old[i]=0.0;
                pumaData->xv_dot_old[i]=0.0;
                pumaData-
>xv_dot_way_old[i]=0.0;
                pumaData->xyz_old[i]=0.0;
                pumaData->xyz_dot_old[i]=0.0;
                pumaData-
>xyz_dot_way_old[i]=0.0;
        }

// calibrate encoders
        HwOutpw(pumaData->PumaDevice,
0x020, 0x0000);
        HwOutpw(pumaData->PumaDevice,
0x022, 0x0000);
        HwOutpw(pumaData->PumaDevice,
0x024, 0x0000);
        HwOutpw(pumaData->PumaDevice,
0x026, 0x1f40);
        HwOutpw(pumaData->PumaDevice,
0x028, 0x0000);
        HwOutpw(pumaData->PumaDevice,
0x02a, 0x0000);
}
```

```
// socket.c

#include "puma.h"

SOCKADDR_IN stLclName;
SOCKADDR_IN stRmtName;

void openSocket(pumaFile* pumaData)
{
        int server=0;
        int nRet;

        // ip for snow
//      char szHost[] = "129.186.232.46";
```

```
        // ip for hood
        char szHost[] = "129.186.232.34";
        // ip for mammoth
//      char szHost[] = "129.186.232.54";

        char szDataReceive[] = {0};
        unsigned long addr;
        WORD WSA_VERSION;
        WSADATA stWSAData;

        WSA_VERSION = MAKEWORD(1, 1);
  nRet=WSAStartup(WSA_VERSION,
&stWSAData);
        if (nRet==0)  printf("attached to winsock
dll\n");
        else  printf("could not attach winsock
dll\n");

        if (pumaData->useSocket == 1)
        {
                pumaData-
>hSock=socket(AF_INET, SOCK_DGRAM, 0);

                if (pumaData-
>hSock==INVALID_SOCKET)  printf("could not
get a valid socket handle\n");
                else
                {
                        if (server==1)
                        {

        stLclName.sin_family = PF_INET;

        stLclName.sin_port=htons(1026);

        stLclName.sin_addr.s_addr=INADDR_A
NY;


        nRet=bind(pumaData->hSock,
(LPSOCKADDR) &stLclName, sizeof(struct
sockaddr));
                                        if
(nRet==SOCKET_ERROR)  printf("could not
bind server socket\n");
                                        else
printf("server socket:  Open\n");


        nRet=recv(pumaData->hSock, (LPSTR)
szDataReceive, 5, 0);
                                        if
(nRet==SOCKET_ERROR)  printf("server socket
could not receive data\n");
                                        else
printf("sever socket received data\n");
                        }
                        else
                        {

        addr=inet_addr((LPSTR) szHost);
                                        if
(addr==INADDR_NONE)  printf("could not find
address of server\n");


        stRmtName.sin_family = PF_INET;

        stRmtName.sin_port=htons(1026);

        stRmtName.sin_addr.s_addr=addr;


        nRet=connect(pumaData->hSock,
(LPSOCKADDR) &stRmtName, sizeof(struct
sockaddr));
                                        if
(nRet==SOCKET_ERROR)  printf("could not
connect to server socket\n");
                                        else
                                        {

        printf("Socket Open\n");

        pumaData->activeSocket=1;
                                        }
                        }
                }
        }
}

void closeSocket(pumaFile* pumaData)
{
        int nRet;

        if (pumaData->activeSocket == 1)
        {
                nRet=closesocket(pumaData-
>hSock);
                if (nRet==SOCKET_ERROR)
printf("error closing socket\n");
                else  printf("Socket Closed\n");
        }
```

```
	nRet=WSACleanup();

}

void testSocket(pumaFile* pumaData)
{
	int nRet;
	char szDataSend[100];
	double t0=0.0;
	double t1=1.571;
	double t2=-1.571;

	sprintf(szDataSend,"%4.3f %4.3f %4.3f
%4.3f %4.3f %4.3f %4.3f ",t0,t0,t2,t1,t0,t0,t0);

	if (pumaData->activeSocket == 1)
	{
		nRet=send(pumaData->hSock,
(LPSTR) szDataSend, 51, 0);
		if (nRet==SOCKET_ERROR)
printf("Socket test failed\n");
		else  printf("Socket test
passed\n");
	}
}
```

Darth Vader

```
// error.c

#include "puma.h"

void error(pumaFile* pumaData)
{
	double xv[3],xv_dot[3];
	double e[3];
	double xv_ori[3][3],xyz[3],xyz_dot[3];
	double rd[3][3],R[3],C[3];
	double ctheta,theta,mag;
	double wn,z;
	int i,j,k;
	double rv[3][3];

// linear error - world space
	e[0]=0.0; //0.4175-x[0];
	e[1]=0.0; //0.1505-x[1];
	e[2]=0.0; //0.4310-x[2];

// linear error - end effector space
	for (j=0;j<3;j++)
	{
		xv[j]=0.0;
		for (i=0;i<3;i++)
		{
			xv[j] += pumaData-
>r[i][j]*e[i];
		}
	}

// rotational error - world space
	R[0]=-pumaData->r[0][2];
	R[1]=-pumaData->r[1][2];
	R[2]=0.0;
	mag=sqrt(R[0]*R[0]+R[1]*R[1]+R[2]*R[
2]);
	R[0]=R[0]/mag;
	R[1]=R[1]/mag;
	R[2]=R[2]/mag;

	C[0]=pumaData->x[0]-1.0;
	C[1]=pumaData->x[1]-0.0;
	C[2]=0.0;
	mag=sqrt(C[0]*C[0]+C[1]*C[1]+C[2]*C[
2]);
	C[0]=C[0]/mag;
	C[1]=C[1]/mag;
	C[2]=C[2]/mag;

	ctheta=R[0]*C[0]+R[1]*C[1]+R[2]*C[2];
	theta=acos(ctheta);

// check to see which solution of arccos is needed
	theta=theta*fabs(R[0]*C[1]-
R[1]*C[0])/(R[0]*C[1]-R[1]*C[0]);

// desired orientation - world space
	rd[0][0]=cos(theta);
	rd[0][1]=-sin(theta);
	rd[0][2]=0.0;

	rd[1][0]=sin(theta);
	rd[1][1]=cos(theta);
	rd[1][2]=0.0;

	rd[2][0]=0.0;
	rd[2][1]=0.0;
	rd[2][2]=1.0;

	for (i=0;i<3;i++)
	{
		for (j=0;j<3;j++)
```

```
                {
                        rv[i][j]=0.0;
                        for (k=0;k<3;k++)
                        {
                                rv[i][j] +=
rd[i][k]*pumaData->r[k][j];
                        }
                }
        }

// desired orientation - end effector space
        for (i=0;i<3;i++)
        {
                for (j=0;j<3;j++)
                {
                        xv_ori[i][j]=0.0;
                        for (k=0;k<3;k++)
                        {
                                xv_ori[i][j] +=
pumaData->r[k][i]*rv[k][j];
                        }
                }
        }

// rotational error - end effector space
        xyz[1]=atan2(-
xv_ori[2][0],sqrt(xv_ori[0][0]*xv_ori[0][0]+xv_or
i[1][0]*xv_ori[1][0]));
        if (fabs(xyz[1]-1.5708) < 0.01)
        {
                xyz[2]=0.0;

        xyz[0]=atan2(xv_ori[0][1],xv_ori[1][1]);
        }
        else if (fabs(xyz[1]+1.5708) < 0.01)
        {
                xyz[2]=0.0;
                xyz[0]=-
atan2(xv_ori[0][1],xv_ori[1][1]);
        }
        else
        {

        xyz[2]=atan2(xv_ori[1][0],xv_ori[0][0]);

        xyz[0]=atan2(xv_ori[2][1],xv_ori[2][2]);
        }

// derivatives of linear and rotational error
        wn=60.0;
        z=0.7071;
        for (i=0;i<3;i++)
        {
                xv_dot[i]=(wn*wn*pumaData-
>dt*(xv[i]-pumaData->xv_old[i])+pumaData-
>xv_dot_old[i]*(2.0+2.0*z*wn*pumaData->dt)-
pumaData-
>xv_dot_way_old[i])/(1.0+2.0*z*wn*pumaData-
>dt+wn*wn*pumaData->dt*pumaData->dt);
                xyz_dot[i]=(wn*wn*pumaData-
>dt*(xyz[i]-pumaData->xyz_old[i])+pumaData-
>xyz_dot_old[i]*(2.0+2.0*z*wn*pumaData->dt)-
pumaData-
>xyz_dot_way_old[i])/(1.0+2.0*z*wn*pumaData-
>dt+wn*wn*pumaData->dt*pumaData->dt);
        }

        for (i=0;i<3;i++)
        {
                pumaData-
>xv_dot_way_old[i]=pumaData->xv_dot_old[i];
                pumaData-
>xv_dot_old[i]=xv_dot[i];
                pumaData->xv_old[i]=xv[i];
                pumaData-
>xyz_dot_way_old[i]=pumaData->xyz_dot_old[i];
                pumaData-
>xyz_dot_old[i]=xyz_dot[i];
                pumaData->xyz_old[i]=xyz[i];
        }

// Evaluate virtual spring force
        for (i=0;i<3;i++)
        {
                pumaData->fv[i]=-470.0*xv[i]-
30.0*xv_dot[i];
        }

//      if (fabs(theta) <=0.05)
        if (theta > 0.0)
        {
                pumaData->contact=1.0;
                for (i=0;i<3;i++)
                {
                        pumaData->fv[i+3]=-
60.0*xyz[i]-3.*xyz_dot[i];
                }
        }
        else
        {
                pumaData->contact=0.0;
                for (i=0;i<3;i++)
                {
                        pumaData-
>fv[i+3]=0.0;
                }
        }
}
```

```
// friction.c

#include "puma.h"

void friction(pumaFile* pumaData)
{
        int i;
        double tau=0.05305;

        if (pumaData->theta[0] > pumaData-
>theta_old[0]) pumaData->v_fric[0]=1.0;
        if (pumaData->theta[0] <= pumaData-
>theta_old[0]) pumaData->v_fric[0]=-0.9;
        pumaData->v_fric[0]=(pumaData-
>v_fric[0]*pumaData->dt+pumaData-
>v_fric_old[0]*tau)/(pumaData->dt+tau);

        if (pumaData->theta[1] > pumaData-
>theta_old[1])
        {
                if (pumaData->theta[1] > -1.57)
pumaData->v_fric[1]=-0.3;
                else pumaData->v_fric[1]=-0.9;
        }
        if (pumaData->theta[1] <= pumaData-
>theta_old[1])
        {
                if (pumaData->theta[1] > -1.57)
pumaData->v_fric[1]=0.9;
                else pumaData->v_fric[1]=0.6;
        }
        pumaData->v_fric[1]=(pumaData-
>v_fric[1]*pumaData->dt+pumaData-
>v_fric_old[1]*tau)/(pumaData->dt+tau);

        if (pumaData->theta[2] > pumaData-
>theta_old[2]) pumaData->v_fric[2]=0.47;
        if (pumaData->theta[2] <= pumaData-
>theta_old[2]) pumaData->v_fric[2]=-0.47;
        pumaData->v_fric[2]=(pumaData-
>v_fric[2]*pumaData->dt+pumaData-
>v_fric_old[2]*tau)/(pumaData->dt+tau);

        if (pumaData->theta[3] > pumaData-
>theta_old[3]) pumaData->v_fric[3]=-0.35;
        else if (pumaData->theta[3] <=
pumaData->theta_old[3]) pumaData-
>v_fric[3]=0.35;
        else pumaData->v_fric[3]=0.0;
        pumaData->v_fric[3]=(pumaData-
>v_fric[3]*pumaData->dt+pumaData-
>v_fric_old[3]*tau)/(pumaData->dt+tau);

        if (pumaData->theta[4] > pumaData-
>theta_old[4]) pumaData->v_fric[4]=-0.4;
        else if (pumaData->theta[4] < pumaData-
>theta_old[4]) pumaData->v_fric[4]=0.4;
        else pumaData->v_fric[4]=0.0;
        pumaData->v_fric[4]=(pumaData-
>v_fric[4]*pumaData->dt+pumaData-
>v_fric_old[4]*tau)/(pumaData->dt+tau);

        if (pumaData->theta[5] > pumaData-
>theta_old[5]) pumaData->v_fric[5]=-0.5;
        else if (pumaData->theta[5] < pumaData-
>theta_old[5]) pumaData->v_fric[5]=0.5;
        else pumaData->v_fric[5]=0.0;
        pumaData->v_fric[5]=(pumaData-
>v_fric[5]*pumaData->dt+pumaData-
>v_fric_old[5]*tau)/(pumaData->dt+tau);

        for (i=0;i<6;i++)
        {
                pumaData-
>v_fric_old[i]=pumaData->v_fric[i];
        }
}
```

```
// ft.c

#include "puma.h"

void ftask(pumaFile* pumaData)
{
        short data;
        BYTE bhigh, blow;

        data=14;
        bhigh=(BYTE) ((data & 0xFF00) >> 8);
        blow=(BYTE) (data & 0xFF);
        HwOutp(pumaData-
>FTDevice,0x02,blow);
```

```
        HwOutp(pumaData-
>FTDevice,0x03,bhigh);
}

int ftget(pumaFile* pumaData)
{
        short force[7], data;
        BYTE bhigh, blow;
        double wn=2.0*5.0*3.14159;
        double zeta=1.0;
        double f_fil[6];
        int i;

// get a force measurement

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        force[6]=data;

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        force[0]=data;

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        force[1]=data;

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        force[2]=data;

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        force[3]=data;

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        force[4]=data;

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        force[5]=data;

        if (force[6] != 0)  return(0);

// convert counts to force adjust for fbasis and
weight of handle
        pumaData->forced[0]=((double)
(force[0]))*0.1-pumaData-
>fbasis[0];+3.6*pumaData->r[2][0];
        pumaData->forced[1]=((double)
(force[1]))*0.1-pumaData-
>fbasis[1];+3.6*pumaData->r[2][1];
        pumaData->forced[2]=((double)
(force[2]))*0.1-pumaData-
>fbasis[2];+3.6*pumaData->r[2][2];
        pumaData->forced[3]=((double)
(force[3]))*0.005-pumaData->fbasis[3]-
0.09*pumaData->forced[1];-0.06*2.6*pumaData-
>r[2][1];
        pumaData->forced[4]=((double)
(force[4]))*0.005-pumaData-
```

```
>fbasis[4]+0.09*pumaData-
>forced[0];+0.06*2.6*pumaData->r[2][0];
        pumaData->forced[5]=((double)
(force[5]))*0.005-pumaData->fbasis[5];

// filter force data
        for (i=0;i<6;i++)
        {
                f_fil[i]=(pumaData-
>forced[i]*pumaData->dt*pumaData-
>dt*wn*wn+
                pumaData-
>f_fil_old[i]*(2.0*zeta*wn*pumaData->dt+2.0)-
                pumaData-
>f_fil_way_old[i])/(1.0+2.0*zeta*wn*pumaData-
>dt+wn*wn*pumaData->dt*pumaData->dt);
        }

        for (i=0;i<3;i++)
        {
                pumaData->ft[i]=0.3*f_fil[i];
                pumaData-
>ft[i+3]=0.5*f_fil[i+3];
        }

        for (i=0;i<6;i++)
        {
                pumaData-
>f_fil_way_old[i]=pumaData->f_fil_old[i];
                pumaData->f_fil_old[i]=f_fil[i];
        }

        return(1);
}
```

```
// ftinitialize.c

#include "puma.h"

int ftInitialize(pumaFile* pumaData)
{
        short data, force[7];
        BYTE bhigh, blow;

// clear any data in the buffer
        if (HwInp(pumaData->FTDevice,0x04) &
0x10)
        {
                blow=HwInp(pumaData-
>FTDevice,0x00);
                bhigh=HwInp(pumaData-
>FTDevice,0x01);
                data=(short) (bhigh << 8);
                data=data | (short) (blow &
0xFF);
                printf("preload 1 %d\n",data);
                Sleep(500);
        }
        if (HwInp(pumaData->FTDevice,0x04) &
0x10)
        {
                blow=HwInp(pumaData-
>FTDevice,0x00);
                bhigh=HwInp(pumaData-
>FTDevice,0x01);
                data=(short) (bhigh << 8);
                data=data | (short) (blow &
0xFF);
                printf("preload 2 %d\n",data);
                Sleep(500);
        }

// send CPP to switch to parallel board
        printf("Switch to parallel board\n");
        Sleep(1000);

        while((HwInp(pumaData-
>FTDevice,0x04) & 0x80) == 0) Sleep(1000);
        data=67;
        bhigh=(BYTE) ((data & 0xFF00) >> 8);
        blow=(BYTE) (data & 0xFF);
        HwOutp(pumaData-
>FTDevice,0x02,blow);
        HwOutp(pumaData-
>FTDevice,0x03,bhigh);
        printf("C\n");
        Sleep(500);

        while((HwInp(pumaData-
>FTDevice,0x04) & 0x80) == 0) Sleep(1000);
        data=80;
        bhigh=(BYTE) ((data & 0xFF00) >> 8);
        blow=(BYTE) (data & 0xFF);
        HwOutp(pumaData-
>FTDevice,0x02,blow);
        HwOutp(pumaData-
>FTDevice,0x03,bhigh);
        printf("P\n");
        Sleep(500);

        while((HwInp(pumaData-
>FTDevice,0x04) & 0x80) == 0) Sleep(1000);
```

```
        data=80;
        bhigh=(BYTE) ((data & 0xFF00) >> 8);
        blow=(BYTE) (data & 0xFF);
        HwOutp(pumaData-
>FTDevice,0x02,blow);
        HwOutp(pumaData-
>FTDevice,0x03,bhigh);
        printf("P\n");
        Sleep(500);

        while((HwInp(pumaData-
>FTDevice,0x04) & 0x80) == 0)  Sleep(1000);
        data=13;
        bhigh=(BYTE) ((data & 0xFF00) >> 8);
        blow=(BYTE) (data & 0xFF);
        HwOutp(pumaData-
>FTDevice,0x02,blow);
        HwOutp(pumaData-
>FTDevice,0x03,bhigh);
        printf("<cr>\n");
        Sleep(500);

//  wait for acknowledgment
        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        printf("%d\n",data);
        Sleep(500);

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        printf("%d\n",data);
        Sleep(500);

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        printf("%d\n",data);
        Sleep(500);

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        printf("%d\n",data);
        Sleep(500);

        if (HwInp(pumaData->FTDevice,0x04) &
0x10)
        {
                blow=HwInp(pumaData-
>FTDevice,0x00);
                bhigh=HwInp(pumaData-
>FTDevice,0x01);
                data=(short) (bhigh << 8);
                data=data | (short) (blow &
0xFF);
                printf(".%d\n",data);
                Sleep(500);
        }

        if (HwInp(pumaData->FTDevice,0x04) &
0x10)
        {
                blow=HwInp(pumaData-
>FTDevice,0x00);
                bhigh=HwInp(pumaData-
>FTDevice,0x01);
                data=(short) (bhigh << 8);
                data=data | (short) (blow &
0xFF);
                printf("..%d\n",data);
                Sleep(500);
        }

        if (HwInp(pumaData->FTDevice,0x04) &
0x10)
        {
                blow=HwInp(pumaData-
>FTDevice,0x00);
                bhigh=HwInp(pumaData-
>FTDevice,0x01);
                data=(short) (bhigh << 8);
                data=data | (short) (blow &
0xFF);
                printf("...%d\n",data);
                Sleep(500);
        }

        if (HwInp(pumaData->FTDevice,0x04) &
0x10)
        {
                blow=HwInp(pumaData-
>FTDevice,0x00);
                bhigh=HwInp(pumaData-
>FTDevice,0x01);
                data=(short) (bhigh << 8);
                data=data | (short) (blow &
0xFF);
                printf("....%d\n",data);
                Sleep(500);
        }

        if (HwInp(pumaData->FTDevice,0x04) &
0x10)
        {
                blow=HwInp(pumaData-
>FTDevice,0x00);
                bhigh=HwInp(pumaData-
>FTDevice,0x01);
                data=(short) (bhigh << 8);
                data=data | (short) (blow &
0xFF);
                printf(".....%d\n",data);
                Sleep(500);
        }

        if (HwInp(pumaData->FTDevice,0x04) &
0x10)
        {
                blow=HwInp(pumaData-
>FTDevice,0x00);
                bhigh=HwInp(pumaData-
>FTDevice,0x01);
                data=(short) (bhigh << 8);
                data=data | (short) (blow &
0xFF);
                printf("......%d\n",data);
                Sleep(500);
        }

// send CDB
        printf("Set to communicate binary
mode\n");
        Sleep(1000);
```

```
        while((HwInp(pumaData-
>FTDevice,0x04) & 0x80) == 0)  Sleep(1000);
        data=67;
        bhigh=(BYTE) ((data & 0xFF00) >> 8);
        blow=(BYTE) (data & 0xFF);
        HwOutp(pumaData-
>FTDevice,0x02,blow);
        HwOutp(pumaData-
>FTDevice,0x03,bhigh);
        while((HwInp(pumaData-
>FTDevice,0x04) & 0x80) == 0)  Sleep(1000);
        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        printf("%d\n",data);
        Sleep(500);
        if (data != 67)  return(0);

        while((HwInp(pumaData-
>FTDevice,0x04) & 0x80) == 0)  Sleep(1000);
        data=68;
        bhigh=(BYTE) ((data & 0xFF00) >> 8);
        blow=(BYTE) (data & 0xFF);
        HwOutp(pumaData-
>FTDevice,0x02,blow);
        HwOutp(pumaData-
>FTDevice,0x03,bhigh);
        while((HwInp(pumaData-
>FTDevice,0x04) & 0x80) == 0)  Sleep(1000);
        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        printf("%d\n",data);
        Sleep(500);
        if (data != 68)  return(0);

        while((HwInp(pumaData-
>FTDevice,0x04) & 0x80) == 0)  Sleep(1000);
        data=66;
        bhigh=(BYTE) ((data & 0xFF00) >> 8);
        blow=(BYTE) (data & 0xFF);
        HwOutp(pumaData-
>FTDevice,0x02,blow);
        HwOutp(pumaData-
>FTDevice,0x03,bhigh);
        while((HwInp(pumaData-
>FTDevice,0x04) & 0x80) == 0)  Sleep(1000);
        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        printf("%d\n",data);
        Sleep(500);
        if (data != 66)  return(0);

        while((HwInp(pumaData-
>FTDevice,0x04) & 0x80) == 0)  Sleep(1000);
        data=13;
        bhigh=(BYTE) ((data & 0xFF00) >> 8);
        blow=(BYTE) (data & 0xFF);
        HwOutp(pumaData-
>FTDevice,0x02,blow);
        HwOutp(pumaData-
>FTDevice,0x03,bhigh);
        while((HwInp(pumaData-
>FTDevice,0x04) & 0x80) == 0)  Sleep(1000);
        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        printf("%d\n",data);
        Sleep(500);
        if (data != 13)  return(0);

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        printf("%d\n",data);
        Sleep(500);
        if (data != 10)  return(0);

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        printf("%d\n",data);
        Sleep(500);
        if (data != 6)  return(0);

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        printf("%d\n",data);
        Sleep(500);
        if (data != 6)  return(0);

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data ! (short) (blow & 0xFF);
        printf("%d\n",data);
        Sleep(500);
        if (data != 13)  return(0);

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        printf("%d\n",data);
        Sleep(500);
        if (data != 10)  return(0);

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
```

```
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        printf("%d\n",data);
        Sleep(500);
        if (data != 62)  return(0);

// get a force measurement for the basis
        while((HwInp(pumaData-
>FTDevice,0x04) & 0x80) == 0)  Sleep(1000);
        data=14;
        bhigh=(BYTE) ((data & 0xFF00) >> 8);
        blow=(BYTE) (data & 0xFF);
        HwOutp(pumaData-
>FTDevice,0x02,blow);
        HwOutp(pumaData-
>FTDevice,0x03,bhigh);

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        force[6]=data;

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        force[0]=data;

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        force[1]=data;

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        force[2]=data;

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        force[3]=data;

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        force[4]=data;

        while ((HwInp(pumaData-
>FTDevice,0x04) & 0x10) == 0);
        blow=HwInp(pumaData-
>FTDevice,0x00);
        bhigh=HwInp(pumaData-
>FTDevice,0x01);
        data=(short) (bhigh << 8);
        data=data | (short) (blow & 0xFF);
        force[5]=data;

        if (force[6] != 0)
        {
                printf("Error getting force
basis\n");
                return(0);
        }

        pumaData->fbasis[0]=((double)
(force[0]))*0.1;
        pumaData->fbasis[1]=((double)
(force[1]))*0.1;
```

```
        pumaData->fbasis[2]=((double)
(force[2]))*0.1+3.6;
        pumaData->fbasis[3]=((double)
(force[3]))*0.005-0.09*pumaData->fbasis[1];
        pumaData->fbasis[4]=((double)
(force[4]))*0.005+0.09*pumaData->fbasis[0];
        pumaData->fbasis[5]=((double)
(force[5]))*0.005;

        printf("FT initialized OK\n");
        return(1);
}
```

```
// gravity.c

#include "puma.h"

void gravity(pumaFile* pumaData)
{
        double c2,s2,c23,s23;

        c2=cos(pumaData->theta[1]);
        s2=sin(pumaData->theta[1]);

        c23=cos(pumaData->theta[1]+pumaData-
>theta[2]);
        s23=sin(pumaData->theta[1]+pumaData-
>theta[2]);

// gravity compensation
        pumaData->vg[0]=0.0;
        pumaData->vg[2]=-
1.1201*s23+0.0977*c23;
        pumaData-
>vg[1]=0.2400*s2+2.1144*c2-0.5304*pumaData-
>vg[2];
        pumaData->vg[3]=0.0;
        pumaData->vg[4]=0.0;
        pumaData->vg[5]=0.0;
}
```

```
// impedence.c

#include "puma.h"

void impedence(pumaFile* pumaData)
{
        pumaData-
>vim[0]=0.02*pow((1.0/(pumaData->theta[0]-
2.7)),3.0)+0.02*pow((1.0/(pumaData-
>theta[0]+2.7)),3.0);

        pumaData->vim[1]=-
0.02*pow((1.0/(pumaData->theta[1]-0.7)),3.0)-
0.02*pow((1.0/(pumaData->theta[1]+3.7)),3.0);

        pumaData-
>vim[2]=0.02*pow((1.0/(pumaData->theta[2]-
pumaData-
>jlimit3)),3.0)+0.02*pow((1.0/(pumaData-
>theta[2]+0.9)),3.0);

        pumaData->vim[3]=-
0.02*pow((1.0/(pumaData->theta[3]-3.2)),3.0)-
0.02*pow((1.0/(pumaData->theta[3]+1.8)),3.0);

        pumaData->vim[4]=-
0.02*pow((1.0/(pumaData->theta[4]-1.7)),3.0)-
0.02*pow((1.0/(pumaData->theta[4]+pumaData-
>jlimit5)),3.0);

        pumaData->vim[5]=-
0.02*pow((1.0/(pumaData->theta[5]-5.2)),3.0)-
0.02*pow((1.0/(pumaData->theta[5]+5.2)),3.0);
}
```

```
// kinematics.c

#include "puma.h"

void kinematics(pumaFile* pumaData)
{
        double
c1,s1,c2,s2,c3,s3,c23,s23,c4,s4,c5,s5,c6,s6;
        double l[5];
        double
v1,v2,v3,v4,v5,v6,v7,v8,v9,v10,v11;

        l[1]=0.4318;
        l[2]=0.15005;
        l[3]=-0.0191;
        l[4]=0.4331;

        c1=cos(pumaData->theta[0]);
        s1=sin(pumaData->theta[0]);

        c2=cos(pumaData->theta[1]);
        s2=sin(pumaData->theta[1]);

        c3=cos(pumaData->theta[2]);
        s3=sin(pumaData->theta[2]);

        c23=cos(pumaData->theta[1]+pumaData-
>theta[2]);
        s23=sin(pumaData->theta[1]+pumaData-
>theta[2]);

        c4=cos(pumaData->theta[3]);
        s4=sin(pumaData->theta[3]);

        c5=cos(pumaData->theta[4]);
        s5=sin(pumaData->theta[4]);

        c6=cos(pumaData->theta[5]);
        s6=sin(pumaData->theta[5]);

        pumaData-
>x[0]=c1*(c23*l[3]+s23*l[4]+c2*l[1])-s1*l[2];
        pumaData-
>x[1]=s1*(c23*l[3]+s23*l[4]+c2*l[1])+c1*l[2];
        pumaData->x[2]=-s23*l[3]+c23*l[4]-
s2*l[1];
        pumaData->x[3]=0.0;
        pumaData->x[4]=0.0;
        pumaData->x[5]=0.0;

        v1=c4*c5*c6-s4*s6;
        v2=s5*c6;
        v3=c23*v1-s23*v2;
        v4=s4*c5*c6+c4*s6;

        pumaData->r[0][0]=c1*v3-s1*v4;
        pumaData->r[1][0]=s1*v3+c1*v4;
        pumaData->r[2][0]=-s23*v1-c23*v2;

        v5=c4*c5*s6+s4*c6;
        v6=s5*s6;
        v7=-c23*v5+s23*v6;
        v8=s4*c5*s6-c4*c6;

        pumaData->r[0][1]=c1*v7+s1*v8;
        pumaData->r[1][1]=s1*v7-c1*v8;
        pumaData->r[2][1]=s23*v5+c23*v6;

        v9=c4*s5;
        v10=c23*v9+s23*c5;
        v11=s4*s5;

        pumaData->r[0][2]=c1*v10-s1*v11;
        pumaData->r[1][2]=s1*v10+c1*v11;
        pumaData->r[2][2]=-s23*v9+c23*c5;\

// jacobian
        pumaData->eJr[0][0]=c5*c6*(-
c23*c4*l[2]+s4*(c2*l[1]+c23*l[3]+s23*l[4]))+s6*
(c23*s4*l[2]+c4*(c2*l[1]+c23*l[3]+s23*l[4]))+s5
*c6*s23*l[2];
        pumaData-
>eJr[0][1]=c5*c6*(c4*(s3*l[1]+l[4]))+s6*(-
s4*(s3*l[1]+l[4]))-s5*c6*(-c3*l[1]-l[3]);
        pumaData->eJr[0][2]=c5*c6*c4*l[4]-
s6*s4*l[4]+s5*c6*l[3];
        pumaData->eJr[0][3]=0.0;
        pumaData->eJr[0][4]=0.0;
        pumaData->eJr[0][5]=0.0;

        pumaData->eJr[1][0]=-c5*s6*(-
c23*c4*l[2]+s4*(c2*l[1]+c23*l[3]+s23*l[4]))+c6*
(c23*s4*l[2]+c4*(c2*l[1]+c23*l[3]+s23*l[4]))-
s5*s6*s23*l[2];
        pumaData->eJr[1][1]=-
c5*s6*(c4*(s3*l[1]+l[4]))+c6*(-
s4*(s3*l[1]+l[4]))+s5*s6*(-c3*l[1]-l[3]);
        pumaData->eJr[1][2]=-c5*s6*c4*l[4]-
c6*s4*l[4]-s5*s6*l[3];
        pumaData->eJr[1][3]=0.0;
        pumaData->eJr[1][4]=0.0;
        pumaData->eJr[1][5]=0.0;

        pumaData->eJr[2][0]=s5*(-
c23*c4*l[2]+s4*(c2*l[1]+c23*l[3]+s23*l[4]))-
c5*s23*l[2];
        pumaData-
>eJr[2][1]=s5*(c4*(s3*l[1]+l[4]))+c5*(-c3*l[1]-
l[3]);
        pumaData->eJr[2][2]=s5*c4*l[4]-c5*l[3];
        pumaData->eJr[2][3]=0.0;
        pumaData->eJr[2][4]=0.0;
        pumaData->eJr[2][5]=0.0;

        pumaData->eJr[3][0]=s23*(s4*s6-
c4*c5*c6)-c23*s5*c6;
        pumaData->eJr[3][1]=s4*c5*c6+c4*s6;
        pumaData->eJr[3][2]=s4*c5*c6+c4*s6;
        pumaData->eJr[3][3]=-s5*c6;
        pumaData->eJr[3][4]=s6;
        pumaData->eJr[3][5]=0.0;

        pumaData-
>eJr[4][0]=s23*(c4*c5*s6+s4*c6)+c23*s5*s6;
        pumaData->eJr[4][1]=-s4*c5*s6+c4*c6;
        pumaData->eJr[4][2]=-s4*c5*s6+c4*c6;
        pumaData->eJr[4][3]=s5*s6;
        pumaData->eJr[4][4]=c6;
        pumaData->eJr[4][5]=0.0;

        pumaData->eJr[5][0]=-
s23*c4*s5+c23*c5;
        pumaData->eJr[5][1]=s4*s5;
        pumaData->eJr[5][2]=s4*s5;
        pumaData->eJr[5][3]=c5;
        pumaData->eJr[5][4]=0.0;
        pumaData->eJr[5][5]=1.0;
}
```

```
// main.c

#include "puma.h"

void main(void)
{
// robot stuff
        pumaFile *pumaData;
        int stop;
        int homecount;

// window's stuff
        HANDLE hprocess;
        HANDLE hthread;
        int processerror;

// timer stuff
        BOOL result;
        LARGE_INTEGER lifrequency;
        LARGE_INTEGER licount;
        LONGLONG frequency;
        double dfrequency;
        LONGLONG startcount;
        LONGLONG count;
        double currenttime;
        double dtactual;
        double dterror;
        double dtmax;

// error flags
        int timererror;
        int timeroverrun;
        int DeviceStop;
        int errorSocket;

// socket stuff
        int err;
        char szDataSend[100];
        int gcount;

// ft stuff
        int ftinitok;

// data file stuff
        double data[3][2000];
        int datalength=2000;
        int datacount;
        int datacycle;
        int datamax;
        int fileerror;
        FILE *out;

// general stuff (counter and the like)
        int i;

//////////////////////////////////////////////////////////////////
// Taking Care of Business
//////////////////////////////////////////////////////////////////

        printf("PUMA control program\n");
        printf("written by Jim Edwards for
LARCC\n");
        printf("All rights reserved\n\n\n\n");

//////////////////////////////////////////////////////////////////
// Code Initialization Section
//////////////////////////////////////////////////////////////////
// set counter error flag to pass
        timererror=1;

// set counter overrun flag to pass
        timeroverrun=1;

// start taking data at zero
        datacount=0;

// set data pass to zero
        datacycle=0;

// set process error flag to pass
        processerror=0;

// set maximum delta-t to zero
        dtmax=0.0;

// set stop to pass
        stop=1;
```

```
// set homecount to zero
        homecount=0;

// set socket error to none
        errorSocket=0;

// set graphics dump counter to zero
        gcount=0;

////////////////////////////////////////////////////////////
////// Hardware Initialization
////////////////////////////////////////////////////////////
// get process handle
        hprocess=GetCurrentProcess();

// set process priority
        result=SetPriorityClass(hprocess,
REALTIME_PRIORITY_CLASS);
        if (result == 0) processerror=1;

// get thread handle
        hthread=GetCurrentThread();

// set thread priority
        result=SetThreadPriority(hthread,
THREAD_PRIORITY_TIME_CRITICAL);
        if (result == 0) processerror=2;

// allocate memory for puma structure
        pumaData=(pumaFile
*)malloc(sizeof(pumaFile));

// connect to the puma kernel device
        DeviceStop=1;
        pumaData-
>PumaDevice=HwNewDevice(NULL);
        HwSetErrorHandler(pumaData-
>PumaDevice, MyErrorHandler);
        if (!HwConnectDevice(pumaData-
>PumaDevice, "puma"))
        {
                printf("Failed to connect to puma
device!\n");
                HwDeleteDevice(pumaData-
>PumaDevice);
                DeviceStop=0;
        }

// connect to the ft kernel device
        pumaData-
>FTDevice=HwNewDevice(NULL);
        HwSetErrorHandler(pumaData-
>FTDevice, MyErrorHandler);
        if (!HwConnectDevice(pumaData-
>FTDevice, "ft"))
        {
                printf("Failed to connect to ft
device!\n");
                HwDeleteDevice(pumaData-
>FTDevice);
                DeviceStop=0;
        }

// setup ft
        ftinitok=ftInitialize(pumaData);
        pumaData->ftgetok=1;

// setup puma
        pumaInitialization(pumaData);

// open socket - useSocket = 1 use socket, = 0
don't use socket
        pumaData->useSocket=1;
        pumaData->activeSocket=0;
        openSocket(pumaData);

// test socket
        testSocket(pumaData);

// get frequency of high performance counter
        result=QueryPerformanceFrequency(&lifr
equency);
        if (result == TRUE)
        {
                frequency=lifrequency.QuadPart;
                dfrequency=((double)
frequency);
                printf("clock frequency: %f
MHz\n\n\n\n",dfrequency);
        }
        else
        {

        printf("QueryPerformanceFrequency:
failure\n");
                timererror=0;
        }

// get starting count
        printf("\n\n\nTurn Arm Power On!!!!\n");
        result=QueryPerformanceCounter(&licou
nt);
        if (result == TRUE)
        {
                startcount=licount.QuadPart;
        }
        else
        {

        printf("QueryPerformanceCounter:
failure\n");
                timererror=0;
        }

// disengage the brakes
        HwOutpw(pumaData->PumaDevice,
0x02e, 0x0001);

/////////////////////////////////////////////////////////////
// Main Control Loop
/////////////////////////////////////////////////////////////

        while((homecount < 2000) &&
(pumaData->ftgetok == 1) && (ftinitok == 1) &&
(DeviceStop == 1) && (timererror == 1) &&
(timeroverrun == 1) && (processerror == 0))
        {
// control code
                if (kbhit())  stop=0;
                if (stop == 1)
                {
                pumaControl(pumaData);
                }
                else
                {
                homecount++;
                        pumaHome(pumaData);
                }

// increment graphics dump counter
                gcount++ ;

// send data to graphics engine
                if (gcount == 5)
                {
                gcount=0;
// but only if there is an active socket for
communication
                        if (pumaData-
>activeSocket == 1)
                {

        sprintf(szDataSend,"%4.3f %4.3f %4.3f
%4.3f %4.3f %4.3f %4.3f ",
                        pumaData->time,
                                                pumaData-
>theta[0],
                                                pumaData-
>theta[1],
                                                pumaData-
>theta[2],
                                                pumaData-
>theta[3],
                                                pumaData-
>theta[4],
                                                pumaData-
>theta[5]);

                . err=send(pumaData->hSock,
(LPSTR) szDataSend, 51, 0) ;
                                        if
(err==SOCKET_ERROR) errorSocket=1;
                }
                }

// timing code
                do
                {
// get the current count of performance counter

        result=QueryPerformanceCounter(&licou
nt);
                        if (result == TRUE)
                        {

        count=licount.QuadPart;
// convert into time since program started

        currenttime=((double) (count-
startcount))/dfrequency;

                        }
                        else
                        {

        printf("QueryPerformanceCounter:
failure\n");
                                timererror=0;
                        }

                        dtactual=currenttime-
pumaData->time;
                } while(dtactual < pumaData-
>dt);

// get maximum delta-t
```

```
                if (dtactual > dtmax)
dtmax=dtactual;

// get error in delta-t
                dterror=dtactual-pumaData->dt;
                if (fabs(dterror) > pumaData-
>dt) timeroverrun=0;

// take some data
                if (stop == 1)
                {

        data[0][datacount]=pumaData-
>forced[1];

        data[1][datacount]=pumaData->contact;

        data[2][datacount]=pumaData->x[1];

                        if (datacount == 1999)
                        {
                                datacount=0;
                                datacycle=1;
                        }
                        else datacount++;
                }

// update absolute time base
                pumaData->time=pumaData-
>time+pumaData->dt;
        } // end main control loop

// engage the brakes
        HwOutpw(pumaData->PumaDevice,
0x02e, 0x0000);

/////////////////////////////////////////////////////////////////
// Hardware Clean-Up
/////////////////////////////////////////////////////////////////
// kernal device
        HwDeleteDevice(pumaData-
>PumaDevice);
        HwDeleteDevice(pumaData->FTDevice);

// close socket
        closeSocket(pumaData);

/////////////////////////////////////////////////////////////////
// Take some data
/////////////////////////////////////////////////////////////////

// open the data file
        if ((out=fopen("out.dat","wt"))==NULL)
fileerror=0;
        else
        {
// write data
                fileerror=1;

                fprintf(out,"max dt is
%f\n\n\n",dtmax);

                if (datacycle == 1)
datamax=datalength;
                else datamax=datacount;

                for (i=0; i<datamax; i++)
                {
                        fprintf(out,"%f, %f,
%f\n",data[0][i],data[1][i],data[2][i]);
                }

// close file
                fclose(out);
        }

/////////////////////////////////////////////////////////////////
// Final Error Messages
/////////////////////////////////////////////////////////////////
        printf("\n\n\nError Messages:\n");
        if (timererror == 0) printf("timer
malfunction\n");
        else if (timeroverrun == 0) printf("timer
over run\n");
        else if (DeviceStop == 0) printf("DriverX
error\n");
        else if (fileerror == 0) printf("could not
open data file\n");
        else if (processerror == 1) printf("could
not set process priority\n");
        else if (processerror == 2) printf("could
not set thread priority\n");
        else if (errorSocket == 1) printf("error
sending data over socket\n");
        else printf("all went well\n");


        Sleep(3000);
}

///////////////////////////////////////////////////////////
////// DriverX Error Handler
///////////////////////////////////////////////////////////
```

```
void MyErrorHandler(HWDEVICE* pDevice,
DWORD nError)
{
	printf("Critical DriverX error: %d\n",
nError);
	exit(nError);
}
```

```
// puma.h

// include files
#include <windows.h>
#include <winsock.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include "DriverX.h"

// structures
typedef struct
{
// needed for all
	HWDEVICE* PumaDevice;
	HWDEVICE* FTDevice;
	double dt;
	double time;
	double encoder_scale[6];
	double encoder_offset[6];
	double theta[6];
	double voltage_out[6];

// standard robot stuff
	double r[3][3];
	double eJr[6][6];
	double x[6];

// virtual manipulator stuff
	double fv[6];
	double xv_old[3];
	double xv_dot_old[3];
	double xv_dot_way_old[3];
	double xyz_old[3];
	double xyz_dot_old[3];
	double xyz_dot_way_old[3];

// socket stuff
	SOCKET hSock;
	int useSocket;
	int activeSocket;

// ft stuff
	double fbasis[6];
	double forced[6];
	int ftgetok;
	double ft[6];
	double f_fil_old[6];
	double f_fil_way_old[6];

// needed for me
	int first_flag;
	int last_flag;
	double kp[6];
	double kd[6];
	double error[6];
	double erroroid[6];
	double errordot[6];
	double thetad[6];
	double theta_old[6];
	double thetao[6];
	double timeh;
	double vg[6];
	double v_fric[6];
	double v_fric_old[6];
	double vim[6];
	double jlimit3;
	double jlimit5;
	double contact;
} pumaFile;

// prototypes
void main(void);
void MyErrorHandler(HWDEVICE *, DWORD);
void pumaInitialization(pumaFile *);
int ftInitialize(pumaFile *);
void pumaControl(pumaFile *);
void pumaHome(pumaFile *);
void openSocket(pumaFile *);
void closeSocket(pumaFile *);
void testSocket(pumaFile *);
void gravity(pumaFile *);
void friction(pumaFile *);
void impedence(pumaFile *);
void ftask(pumaFile *);
int ftget(pumaFile *);
void kinematics(pumaFile *);
void error(pumaFile *);
```

```
// pumaControl.c

#include "puma.h"
```

```
void pumaControl(pumaFile* pumaData)
{
        short val[6];
        int voltage_int[6];
        int i, j;
        double thetaf[6];
        double tf=5.0;

// read encoders
        val[0]=HwInpw(pumaData-
>PumaDevice, 0x010);
        val[1]=HwInpw(pumaData-
>PumaDevice, 0x012);
        val[2]=HwInpw(pumaData-
>PumaDevice, 0x014);
        val[3]=HwInpw(pumaData-
>PumaDevice, 0x016);
        val[4]=HwInpw(pumaData-
>PumaDevice, 0x018);
        val[5]=HwInpw(pumaData-
>PumaDevice, 0x01a);

// convert encoders to radians
        for (i=0; i<6; i++)
        {
                pumaData->theta[i]=pumaData-
>encoder_scale[i]*(((double) val[i]) - pumaData-
>encoder_offset[i]);
        }

// ask for some forces
        ftask(pumaData);

// forward kinematics and Jacobian
        kinematics(pumaData);

// virtual manipulator control
        error(pumaData);

// gravity compensation
        gravity(pumaData);

// friction compensation
        friction(pumaData);

// get forces
        pumaData->ftgetok=ftget(pumaData);

// impedence protection
        impedence(pumaData);

// first time through get current position
        if (pumaData->first_flag==1)
        {
                pumaData-
>thetao[0]=pumaData->theta[0];
                pumaData-
>thetao[1]=pumaData->theta[1];
                pumaData-
>thetao[2]=pumaData->theta[2];
                pumaData-
>thetao[3]=pumaData->theta[3];
                pumaData-
>thetao[4]=pumaData->theta[4];
                pumaData-
>thetao[5]=pumaData->theta[5];
                pumaData->first_flag=2;
        }

// final position
        thetaf[0]=0.0 ;
        thetaf[1]=0.0 ;
        thetaf[2]=0.0 ;
        thetaf[3]=-0.2593 ;
        thetaf[4]=1.3638 ;
        thetaf[5]=0.2776 ;

// do cubic spline interpolation
        if (pumaData->time <= tf)
        {
                pumaData-
>thetad[0]=pumaData->thetao[0]-3.0*(pumaData-
>thetao[0]-thetaf[0])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[0]-thetaf[0])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                pumaData-
>thetad[1]=pumaData->thetao[1]-3.0*(pumaData-
>thetao[1]-thetaf[1])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[1]-thetaf[1])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                pumaData-
>thetad[2]=pumaData->thetao[2]-3.0*(pumaData-
>thetao[2]-thetaf[2])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[2]-thetaf[2])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                pumaData-
>thetad[3]=pumaData->thetao[3]-3.0*(pumaData-
>thetao[3]-thetaf[3])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[3]-thetaf[3])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                    pumaData-
>thetad[4]=pumaData->thetao[4]-3.0*(pumaData-
>thetao[4]-thetaf[4])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[4]-thetaf[4])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
                    pumaData-
>thetad[5]=pumaData->thetao[5]-3.0*(pumaData-
>thetao[5]-thetaf[5])*pumaData-
>time*pumaData->time/(tf*tf)+2.0*(pumaData-
>thetao[5]-thetaf[5])*pumaData-
>time*pumaData->time*pumaData-
>time/(tf*tf*tf);
          }
// after tf stay put at final position
          else if (pumaData->time > tf)
          {
                    pumaData->thetad[0]=thetaf[0];
                    pumaData->thetad[1]=thetaf[1];
                    pumaData->thetad[2]=thetaf[2];
                    pumaData->thetad[3]=thetaf[3];
                    pumaData->thetad[4]=thetaf[4];
                    pumaData->thetad[5]=thetaf[5];
          }

//
// control section
//
          for (i=0;i<6;i++)
          {
// calculate error
                    pumaData->error[i]=pumaData-
>thetad[i]-pumaData->theta[i];

// calculate rate of change of the error
                    pumaData-
>errordot[i]=(pumaData->error[i]-pumaData-
>errorold[i])/pumaData->dt;

// evaluate local PD control law
                    pumaData-
>voltage_out[i]=pumaData->kp[i]*pumaData-
>error[i]+pumaData->kd[i]*pumaData-
>errordot[i];
          }

// impedence based control law
          if (pumaData->time > 6.0)
          {
                    pumaData->jlimit3=1.4;
                    pumaData->jlimit5=-0.2;
                    for (j=0;j<6;j++)
                    {
                    pumaData->voltage_out[j]=0.0;
                              for (i=0;i<6;i++)
                              {
                    pumaData->voltage_out[j] +=
pumaData->eJr[i][j]*(pumaData->fv[i]-pumaData-
>ft[i]);
                              }
                    }
                    pumaData-
>voltage_out[0]=pumaData->voltage_out[0]*-1.0;
                    pumaData-
>voltage_out[2]=pumaData->voltage_out[2]*-1.0;

          }

// Convert voltages into integers to output to
trident board
          for (i=0;i<6;i++)
          {
                    pumaData-
>voltage_out[i]=pumaData-
>voltage_out[i]+pumaData->vg[i]+pumaData-
>v_fric[i]+pumaData->vim[i];
                    if (fabs(pumaData-
>voltage_out[i]) > 9.9)
          pumaData-
>voltage_out[i]=9.9*pumaData-
>voltage_out[i]/fabs(pumaData->voltage_out[i]);
                    voltage_int[i]=(int)
(4095.0*(pumaData->voltage_out[i]+10.0)/20.0);
          }

// Output voltages to trident hardware
          HwOutpw(pumaData->PumaDevice,
0x030, voltage_int[0]);
          HwOutpw(pumaData->PumaDevice,
0x032, voltage_int[1]);
          HwOutpw(pumaData->PumaDevice,
0x034, voltage_int[2]);
          HwOutpw(pumaData->PumaDevice,
0x036, voltage_int[3]);
          HwOutpw(pumaData->PumaDevice,
0x038, voltage_int[4]);
          HwOutpw(pumaData->PumaDevice,
0x03a, voltage_int[5]);
```

```
// save some old information
        for (i=0;i<6;i++)
        {
                pumaData-
>errorold[i]=pumaData->error[i];
                pumaData-
>theta_old[i]=pumaData->theta[i];
        }
}
```

```
// pumaHome.c

#include "puma.h"

void pumaHome(pumaFile* pumaData)
{
        short val[6];
        int voltage_int[6];
        int i;
        double thetaf[6];
        double localtime;
        double tf=5.0;

// read encoders
        val[0]=HwInpw(pumaData-
>PumaDevice, 0x010);
        val[1]=HwInpw(pumaData-
>PumaDevice, 0x012);
        val[2]=HwInpw(pumaData-
>PumaDevice, 0x014);
        val[3]=HwInpw(pumaData-
>PumaDevice, 0x016);
        val[4]=HwInpw(pumaData-
>PumaDevice, 0x018);
        val[5]=HwInpw(pumaData-
>PumaDevice, 0x01a);

// convert encoders to radians
        for (i=0; i<6; i++)
        {
                pumaData->theta[i]=pumaData-
>encoder_scale[i]*(((double) val[i]) - pumaData-
>encoder_offset[i]);
        }

// first time through get current position
        if (pumaData->last_flag==1)
        {
                pumaData-
>thetao[0]=pumaData->theta[0];
                pumaData-
>thetao[1]=pumaData->theta[1];
                pumaData-
>thetao[2]=pumaData->theta[2];
                pumaData-
>thetao[3]=pumaData->theta[3];
                pumaData-
>thetao[4]=pumaData->theta[4];
                pumaData-
>thetao[5]=pumaData->theta[5];
                pumaData->last_flag=0;
                pumaData->timeh=pumaData-
>time;
        }

// final position
        thetaf[0]=0.0;
        thetaf[1]=-1.57;
        thetaf[2]=1.57;
        thetaf[3]=0.0;
        thetaf[4]=0.0;
        thetaf[5]=0.0;

// time that home has been running
        localtime=pumaData->time-pumaData-
>timeh;

// do cubic spline interpolation
        if (localtime <= tf)
        {
                pumaData-
>thetad[0]=pumaData->thetao[0]-3.0*(pumaData-
>thetao[0]-
thetaf[0])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[0]-
thetaf[0])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[1]=pumaData->thetao[1]-3.0*(pumaData-
>thetao[1]-
thetaf[1])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[1]-
thetaf[1])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[2]=pumaData->thetao[2]-3.0*(pumaData-
>thetao[2]-
thetaf[2])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[2]-
thetaf[2])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[3]=pumaData->thetao[3]-3.0*(pumaData-
>thetao[3]-
thetaf[3])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[3]-
thetaf[3])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[4]=pumaData->thetao[4]-3.0*(pumaData-
>thetao[4]-
thetaf[4])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[4]-
thetaf[4])*localtime*localtime*localtime/(tf*tf*tf);
                pumaData-
>thetad[5]=pumaData->thetao[5]-3.0*(pumaData-
>thetao[5]-
thetaf[5])*localtime*localtime/(tf*tf)+2.0*(pumaD
ata->thetao[5]-
thetaf[5])*localtime*localtime*localtime/(tf*tf*tf);
        }
// after tf stay put in the final position
        else if (localtime > tf)
        {
                pumaData->thetad[0]=thetaf[0];
                pumaData->thetad[1]=thetaf[1];
                pumaData->thetad[2]=thetaf[2];
                pumaData->thetad[3]=thetaf[3];
                pumaData->thetad[4]=thetaf[4];
                pumaData->thetad[5]=thetaf[5];
        }


//
// control section
//
        for (i=0;i<6;i++)
        {
// calculate error
                pumaData->error[i]=pumaData-
>thetad[i]-pumaData->theta[i];

// calculate rate of change of the error
                pumaData-
>errordot[i]=(pumaData->error[i]-pumaData-
>errorold[i])/pumaData->dt;

// evaluate local PD control law
                pumaData-
>voltage_out[i]=pumaData->kp[i]*pumaData-
>error[i]+pumaData->kd[i]*pumaData-
>errordot[i];
        }

// Convert voltages into integers to output to
trident board
        for (i=0;i<6;i++)
        {
                pumaData-
>voltage_out[i]=pumaData-
>voltage_out[i];//+pumaData->vg[i]+pumaData-
>v_fric[i]+pumaData->vim[i];
                if (fabs(pumaData-
>voltage_out[i]) > 9.9)
        pumaData-
>voltage_out[i]=9.9*pumaData-
>voltage_out[i]/fabs(pumaData->voltage_out[i]);
                voltage_int[i]=(int)
(4095.0*(pumaData->voltage_out[i]+10.0)/20.0);
        }

// Output voltages to trident hardware
        HwOutpw(pumaData->PumaDevice,
0x030, voltage_int[0]) ;
        HwOutpw(pumaData->PumaDevice,
0x032, voltage_int[1]) ;
        HwOutpw(pumaData->PumaDevice,
0x034, voltage_int[2]) ;
        HwOutpw(pumaData->PumaDevice,
0x036, voltage_int[3]) ;
        HwOutpw(pumaData->PumaDevice,
0x038, voltage_int[4]) ;
        HwOutpw(pumaData->PumaDevice,
0x03a, voltage_int[5]) ;

// save some old information
        for (i=0;i<6;i++)
        {
                pumaData-
>errorold[i]=pumaData->error[i] ;
                pumaData-
>theta_old[i]=pumaData->theta[i] ;
  }
}
```

```
// pumaInitialize.c

#include "puma.h"

void pumaInitialization(pumaFile* pumaData)
{
        double frequency;
        int i;

// desired refresh rate (Hz)
        frequency=300.0;
// desired delta-t
        pumaData->dt=1.0/frequency;

// initialize absolute time base to zero
        pumaData->time=0.0;

// set some joint limits for impedence fields
        pumaData->jlimit3=4.0;
        pumaData->jlimit5=1.7;

// set flags for slow up and down
        pumaData->first_flag=1;
        pumaData->last_flag=1;

// encoder stuff
        pumaData-
>encoder_scale[0]=0.00010035;
        pumaData->encoder_scale[1]=-
0.000073156;
        pumaData->encoder_scale[2]=0.000117;
        pumaData->encoder_scale[3]=-
0.000082663;
        pumaData->encoder_scale[4]=-
0.000087376;
        pumaData->encoder_scale[5]=-
0.00016377;

        pumaData->encoder_offset[0]=0.0;
        pumaData->encoder_offset[1]=-21472.0;
        pumaData->encoder_offset[2]=-13426.0;
        pumaData->encoder_offset[3]=8000.0;
        pumaData->encoder_offset[4]=0.0;
        pumaData->encoder_offset[5]=0.0;

// initialize feedback gains
        pumaData->kp[0]=118.0;
        pumaData->kd[0]=15.4;
        pumaData->kp[1]=-288.0;
        pumaData->kd[1]=-24.0;
        pumaData->kp[2]=200.0;
        pumaData->kd[2]=20.0;
        pumaData->kp[3]=-15.0;
        pumaData->kd[3]=-2.0;
        pumaData->kp[4]=-25.2;
        pumaData->kd[4]=-2.2;
        pumaData->kp[5]=-10.0;
        pumaData->kd[5]=-2.0;

// initialize some variables
        for (i=0; i<6; i++)
        {
                pumaData->errorold[i]=0.0;
                        // error values
                pumaData->theta_old[i]=0.0;
                        // angular positions
                pumaData->v_fric_old[i]=0.0;
                // friction voltages
                pumaData-
>f_fil_way_old[i]=0.0;
                pumaData->f_fil_old[i]=0.0;
        }

        for (i=0; i<3; i++)
        {
                pumaData->xv_old[i]=0.0;
                pumaData->xv_dot_old[i]=0.0;
                pumaData-
>xv_dot_way_old[i]=0.0;
                pumaData->xyz_old[i]=0.0;
                pumaData->xyz_dot_old[i]=0.0;
                pumaData-
>xyz_dot_way_old[i]=0.0;
        }

// calibrate encoders
        HwOutpw(pumaData->PumaDevice,
0x020, 0x0000);
        HwOutpw(pumaData->PumaDevice,
0x022, 0x0000);
        HwOutpw(pumaData->PumaDevice,
0x024, 0x0000);
        HwOutpw(pumaData->PumaDevice,
0x026, 0x1f40);
        HwOutpw(pumaData->PumaDevice,
0x028, 0x0000);
        HwOutpw(pumaData->PumaDevice,
0x02a, 0x0000);
}
```

```
// socket.c

#include "puma.h"

SOCKADDR_IN stLclName;
SOCKADDR_IN stRmtName;

void openSocket(pumaFile* pumaData)
{
        int server=0;
        int nRet;
        // ip for snow
//      char szHost[] = "129.186.232.46";
        // ip for hood
//      char szHost[] = "129.186.232.34";
        // ip for mammoth
//      char szHost[] = "129.186.232.54";
        // mammoth direct
//      char szHost[]= "192.168.1.3";
        // racer
//      char szHost[]= "129.186.232.66";
        // tiny
        char szHost[]= "129.186.232.49";

        char szDataReceive[] = {0};
        unsigned long addr;
        WORD WSA_VERSION;
        WSADATA stWSAData;

        WSA_VERSION = MAKEWORD(1, 1);
  nRet=WSAStartup(WSA_VERSION,
&stWSAData);
        if (nRet==0)  printf("attached to winsock
dll\n");
        else  printf("could not attach winsock
dll\n");

        if (pumaData->useSocket == 1)
        {
                pumaData-
>hSock=socket(AF_INET, SOCK_DGRAM, 0);

                if (pumaData-
>hSock==INVALID_SOCKET)  printf("could not
get a valid socket handle\n");
                else
                {
                        if (server==1)
                        {

        stLclName.sin_family = PF_INET;

        stLclName.sin_port=htons(1036);

        stLclName.sin_addr.s_addr=INADDR_A
NY;


        nRet=bind(pumaData->hSock,
(LPSOCKADDR) &stLclName, sizeof(struct
sockaddr));
                                                if
(nRet==SOCKET_ERROR)  printf("could not
bind server socket\n");
                                                else
printf("server socket:  Open\n");


        nRet=recv(pumaData->hSock, (LPSTR)
szDataReceive, 5, 0);
                                                if
(nRet==SOCKET_ERROR)  printf("server socket
could not receive data\n");
                                                else
printf("sever socket received data\n");
                                }
                                else
                                {

        addr=inet_addr((LPSTR) szHost);
                                                if
(addr==INADDR_NONE)  printf("could not find
address of server\n");


        stRmtName.sin_family = PF_INET;

        stRmtName.sin_port=htons(1036);

        stRmtName.sin_addr.s_addr=addr;


        nRet=connect(pumaData->hSock,
(LPSOCKADDR) &stRmtName, sizeof(struct
sockaddr));
                                                if
(nRet==SOCKET_ERROR)  printf("could not
connect to server socket\n");
                                                else
                                                {

        printf("Socket Open\n");

        pumaData->activeSocket=1;
                                                }
                                }
                        }
                }
}

void closeSocket(pumaFile* pumaData)
{
        int nRet;
        if (pumaData->activeSocket == 1)
        {
                nRet=closesocket(pumaData-
>hSock);
                if (nRet==SOCKET_ERROR)
printf("error closing socket\n");
                else  printf("Socket Closed\n");
        }

        nRet=WSACleanup();

}

void testSocket(pumaFile* pumaData)
{
        int nRet;
        char szDataSend[100];
        double t0=0.0;
        double t1=1.571;
        double t2=-1.571;

        sprintf(szDataSend,"%4.3f %4.3f %4.3f
%4.3f %4.3f %4.3f %4.3f ",t0,t0,t2,t1,t0,t0,t0);

        if (pumaData->activeSocket == 1)
        {
                nRet=send(pumaData->hSock,
(LPSTR) szDataSend, 51, 0);
                if (nRet==SOCKET_ERROR)
printf("Socket test failed\n");
                else  printf("Socket test
passed\n");
        }
}
```

Dynamic Surface

```
// basisfunc.c

#include "JimsCave.h"

void dboy_basisfuns(int i,double u,int p,int
n,double *U,double **ders)
{
/*      Compute nonzero basis functions and
their
        derivatives. */

        double **ndu,*left,*right,**a;
        double saved,temp,d;
        int j,r,s1,s2,k,rk,pk,j1,j2;

        ndu=dboy_DoubleMatrix(0,p,0,p);
        left=dboy_DoubleVector(0,p);
        right=dboy_DoubleVector(0,p);
        a=dboy_DoubleMatrix(0,p,0,p);

        ndu[0][0]=1.0;
        for (j=1;j<=p;j++)
        {
                left[j]=u-U[i+1-j];
                right[j]=U[i+j]-u;
                saved=0.0;
                for (r=0;r<j;r++)
                {

        ndu[j][r]=right[r+1]+left[j-r];
                        temp=ndu[r][j-
1]/ndu[j][r];

        ndu[r][j]=saved+right[r+1]*temp;
                        saved=left[j-r]*temp;
                }
                ndu[j][j]=saved;
        }
        for (j=0;j<=p;j++)  ders[0][j]=ndu[j][p];
        for (r=0;r<=p;r++)
        {
                s1=0;  s2=1;
                a[0][0]=1.0;
                for (k=1;k<=n;k++)
                {
                        d=0.0;
                        rk=r-k;  pk=p-k;
                        if (r>=k)
                        {

        a[s2][0]=a[s1][0]/ndu[pk+1][rk];

        d=a[s2][0]*ndu[rk][pk];
                        }
                        if (rk>=-1)  j1=1;
                        else                j1=-
rk;
                        if (r-1<=pk)  j2=k-1;
                        else                j2=p-
r;
                        for (j=j1;j<=j2;j++)
                        {
```

```
                a[s2][j]=(a[s1][j]-a[s1][j-
1])/ndu[pk+1][rk+j];
                                        d +=
a[s2][j]*ndu[rk+j][pk];
                                }
                                if (r<=pk)
                                {
                                        a[s2][k]= -
a[s1][k-1]/ndu[pk+1][r];
                                        d +=
a[s2][k]*ndu[r][pk];
                                }
                                ders[k][r]=d;
                                j=s1; s1=s2; s2=j;
                        }
                }
                r=p;
                for (k=1;k<=n;k++)
                {
                        for (j=0;j<=p;j++) ders[k][j] *=
r;
                        r *= (p-k);
                }

                dboy_freeDoubleMatrix(ndu,0,p,0,p);
                dboy_freeDoubleVector(left,0,p);
                dboy_freeDoubleVector(right,0,p);
                dboy_freeDoubleMatrix(a,0,p,0,p);
}
```

```
// compute.c

#include "JimsCave.h"

/* declare external shared memory pointers */
extern int *dboy_NCPTS;
extern double **dboy_M, **dboy_K;
extern double *dboy_PNEW, *dboy_PCURRENT,
*dboy_POLD, *dboy_PREF;
extern double **dboy_PCONST;
extern double *dboy_WAND;

extern dboy_File *dboy_parameter;

/* compute performs all necessary computations
for drawScene() */
void dboy_Compute(void)
{
        int i, j;
        double dt=0.05;
        double **sconst;

        sconst=dboy_DoubleMatrix(1,dboy_NCP
TS[0]*dboy_NCPTS[1],1,2);

/* zero secondary position constraints */
        for
(i=1;i<=dboy_NCPTS[0]*dboy_NCPTS[1];i++)
        {
                sconst[i][1]=0.0;
                sconst[i][2]=0.0;
        }

/* invert wand tip coordinates into parametric
coordinates */
        dboy_inversion(dboy_WAND,sconst);

/* Step forward in time */
        dboy_step(dboy_PNEW,dboy_POLD,dboy
_PCURRENT,dboy_K,dboy_M,dt,dboy_PCONST,
sconst);

/* Up-data control points */
        for (i=0;i<dboy_NCPTS[1];i++)
        {
                for (j=0;j<dboy_NCPTS[0];j++)
                {

        dboy_POLD[dboy_NCPTS[0]*i+j+1]=db
oy_PCURRENT[dboy_NCPTS[0]*i+j+1];

        dboy_PCURRENT[dboy_NCPTS[0]*i+j+
1]=dboy_PNEW[dboy_NCPTS[0]*i+j+1];
                        dboy_parameter-
>cpts[j][i][2]=dboy_PNEW[dboy_NCPTS[0]*i+j+
1];
                }
        }

        dboy_freeDoubleMatrix(sconst,1,dboy_N
CPTS[0]*dboy_NCPTS[1],1,2);

/* check to see if the surface should be reset and
killed */
        dboy_reset();
}
```

```
// dpythag.c
```

```
#include "JimsCave.h"

double dboy_dpythag(double a, double b)
{
        double absa,absb;
        absa=fabs(a);
        absb=fabs(b);
        if (absa > absb) return
absa*sqrt(1.0+dboy_DSQR(absb/absa));
        else return (absb == 0.0 ? 0.0 :
absb*sqrt(1.0+dboy_DSQR(absa/absb)));
}
```

```
// draw.c

#include "JimsCave.h"

/* declare external shared memory pointers */
extern int *dboy_NCPTS;
extern int *dboy_P;
extern int *dboy_npatch;
extern double *dboy_U, *dboy_V;
extern double **dboy_X, **dboy_Y, **dboy_Z;
extern double **dboy_X_DU, **dboy_Y_DU;
extern double **dboy_X_DV, **dboy_Y_DV;
extern double ***dboy_U_BAS,
***dboy_U_BAS_DER, ***dboy_V_BAS,
***dboy_V_BAS_DER;
extern double *dboy_WAND;
extern double *dboy_TRANS;

extern int dboy_backTexIndex;
extern int dboy_texture;

extern dboy_File *dboy_parameter;

extern int dboy_sound;
extern int *dboy_Playme;

extern awSound *dboy_sowood;
extern awSound *dboy_soboing;
extern awSound *dboy_socreaky;

/* drawScene() is called every frame */
void dboy_drawScene(void)
{
        int i,j;
        double u, v;
        double *tpts, *tpts_du, *tpts_dv;
        double **temp, **temp_du;
        double *norm;
        double **tpts_store, **norm_store;
        int nu, nv;
        int uspan, vspan;
        int s, r;
        double mag;
        float ori[3];
        float pos[3];
        double point[3];
        int bstate;
        double time;

        glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);

        if (dboy_sound)
        {

          if (CAVEDistribMaster() &&
CAVEMasterDisplay() && CAVEEye ==
CAVE_LEFT_EYE)
            {
                time = awGetClockSecs();
                awFrame( time );

                if (dboy_Playme[0]==1)
                {
                  awProp ( dboy_soboing,
AWSND_STATE, AWSND_ON );
                  dboy_Playme[0]=0;
                }
                if (dboy_Playme[1]==1)
                {
                  awProp ( dboy_sowood,
AWSND_STATE, AWSND_ON );
                  dboy_Playme[1]=0;
                }
                if (dboy_Playme[2]==1)
                {
                  awProp ( dboy_socreaky,
AWSND_STATE, AWSND_ON );
                  dboy_Playme[2]=0;
                }
                if (dboy_Playme[3]==1)
                {
                  awProp ( dboy_socreaky,
AWSND_STATE, AWSND_OFF );
                  dboy_Playme[3]=0;
                }
            }
        }
```

```
/*  get wand position  */
        CAVEGetPosition(CAVE_WAND,pos);
        CAVEGetWandFront(ori[0],ori[1],ori[2])
;
        dboy_WAND[0]=(double)
(pos[0]+1.0*ori[0]);
        dboy_WAND[1]=(double)
(pos[1]+1.0*ori[1]);
        dboy_WAND[2]=(double)
(pos[2]+1.0*ori[2]);
        CAVEGetWandOrientation(ori[0],ori[1],
ori[2]);

/*  draw wand  */
        glColor3f(0.0,1.0,0.0);
        if (dboy_parameter->intersect)
glColor3f(0.0, 1.0, 1.0);

        glPushMatrix();
        glTranslated(dboy_WAND[0],
dboy_WAND[1], dboy_WAND[2]);
        glRotated((double) (ori[0]), 0.0, 1.0, 0.0);
        glRotated((double) (ori[1]), 1.0, 0.0, 0.0);
        glRotated((double) (ori[2]), 0.0, 0.0, 1.0);
        glBegin(GL_TRIANGLE_STRIP);

        for (i=0;i<=10;i++)
        {
                u=((double) (i))/10.0;

        point[0]=0.05*cos(2.0*3.14159*u);

        point[1]=0.05*sin(2.0*3.14159*u);
                point[2]=0.0;
                glVertex3dv(point);

                point[2] += 1.0;
                glVertex3dv(point);
        }

        glEnd();
        glPopMatrix();

/*  draw control point net in red  */
        glPushMatrix();

        bstate=CAVEButtonChange(2);
        if (dboy_parameter->intersect)
        {
                if (CAVEBUTTON2)
                {
                        if (bstate==1)
                        {

        dboy_TRANS[0]=dboy_parameter-
>surf[0]-2.5;

        dboy_TRANS[1]=dboy_parameter-
>surf[1]+2.5;

        dboy_TRANS[2]=dboy_parameter-
>surf[2];
                        }
                        dboy_parameter-
>wand_old[0]=dboy_WAND[0]-dboy_TRANS[0];
                        dboy_parameter-
>wand_old[1]=dboy_WAND[1]-dboy_TRANS[1];
                        dboy_parameter-
>wand_old[2]=dboy_WAND[2]-dboy_TRANS[2];
                        dboy_parameter-
>wand_old[3]=dboy_parameter-
>wand_old[3]+((double) ori[0])-dboy_TRANS[3];
                        dboy_parameter-
>wand_old[4]=dboy_parameter-
>wand_old[4]+((double) ori[1])-dboy_TRANS[4];
                        dboy_parameter-
>wand_old[5]=dboy_parameter-
>wand_old[5]+((double) ori[2])-dboy_TRANS[5];
                }
        }

        dboy_TRANS[3]=((double) ori[0]);
        dboy_TRANS[4]=((double) ori[1]);
        dboy_TRANS[5]=((double) ori[2]);

        glTranslated(dboy_parameter-
>wand_old[0], dboy_parameter->wand_old[1],
dboy_parameter->wand_old[2]);
        glTranslated(dboy_TRANS[0],
dboy_TRANS[1], dboy_TRANS[2]);
        glRotated(dboy_parameter->wand_old[3],
0.0, 1.0, 0.0);
        glRotated(dboy_parameter->wand_old[4],
1.0, 0.0, 0.0);
        glRotated(dboy_parameter->wand_old[5],
0.0, 0.0, 1.0);
        glTranslated(-dboy_TRANS[0], -
dboy_TRANS[1], -dboy_TRANS[2]);
        glTranslated(-2.5,2.5,0.0);

        glColor3f(1,0,0);
        for (i=0;i<dboy_NCPTS[0];i++)
        {
                glBegin(GL_LINE_STRIP);
```

```
                        for
(j=0;j<dboy_NCPTS[1];j++)
                        {

        glVertex3dv(dboy_parameter->cpts[i][j]);
                        }
                glEnd();
        }
        for (i=0;i<dboy_NCPTS[1];i++)
        {
                glBegin(GL_LINE_STRIP);

                        for
(j=0;j<dboy_NCPTS[0];j++)
                        {

        glVertex3dv(dboy_parameter->cpts[j][i]);
                        }
                glEnd();
        }

/* Evaluate B-spline suface */
        glColor3f(0.0. 0.369. 0.165);

   if (dboy_texture)
   {
        glEnable(GL_TEXTURE_2D);
        glCallList(dboy_backTexIndex);
   }

        nu=dboy_NCPTS[0]-1;
        nv=dboy_NCPTS[1]-1;

        tpts=dboy_DoubleVector(0.2);
        tpts_du=dboy_DoubleVector(0.2);
        tpts_dv=dboy_DoubleVector(0,2);
        temp=dboy_DoubleMatrix(0.2,0.dboy_P[
1]);
        temp_du=dboy_DoubleMatrix(0.2,0.dboy
_P[1]);
        norm=dboy_DoubleVector(0,2);
        tpts_store=dboy_DoubleMatrix(0.dboy_n
patch[0],0.2);
        norm_store=dboy_DoubleMatrix(0.dboy_
npatch[0],0.2);

        for (i=0;i<=dboy_npatch[0];i++)
        {
                u=((double) i)/((double)
dboy_npatch[0]);
```

```
        uspan=dboy_findspan(u.dboy_U,nu.dboy_
P[0]);

        glBegin(GL_TRIANGLE_STRIP);
                for (j=0;j<=dboy_npatch[0];j++)
                {
                        v=((double) j)/((double)
dboy_npatch[0]);

        vspan=dboy_findspan(v.dboy_V.nv.dboy_
P[1]);
                        for
(s=0;s<=dboy_P[1];s++)
                        {

        temp[2][s]=0.0;

        temp_du[2][s]=0.0;
                                for
(r=0;r<=dboy_P[0];r++)
                                {

        temp[2][s]=temp[2][s]+dboy_U_BAS[i][j]
[r]*dboy_parameter->cpts[uspan-
dboy_P[0]+r][vspan-dboy_P[1]+s][2];

        temp_du[2][s]=temp_du[2][s]+dboy_U_B
AS_DER[i][j][r]*dboy_parameter->cpts[uspan-
dboy_P[0]+r][vspan-dboy_P[1]+s][2];
                                }
                        }

                        tpts[2]=0.0;
                        tpts_du[2]=0.0;
                        tpts_dv[2]=0.0;
                        for
(s=0;s<=dboy_P[1];s++)
                        {

        tpts[2]=tpts[2]+dboy_V_BAS[i][j][s]*tem
p[2][s];

        tpts_du[2]=tpts_du[2]+dboy_V_BAS[i][j]
[s]*temp_du[2][s];

        tpts_dv[2]=tpts_dv[2]+dboy_V_BAS_DE
R[i][j][s]*temp[2][s];
                        }

                        tpts[0]=dboy_X[i][j];
                        tpts[1]=dboy_Y[i][j];
```

```
                    dboy_Z[i][j]=tpts[2];


        tpts_du[0]=dboy_X_DU[i][j];

        tpts_du[1]=dboy_Y_DU[i][j];


        tpts_dv[0]=dboy_X_DV[i][j];

        tpts_dv[1]=dboy_Y_DV[i][j];


        norm[0]=tpts_du[1]*tpts_dv[2]-
tpts_du[2]*tpts_dv[1];

        norm[1]=tpts_du[0]*tpts_dv[2]-
tpts_du[2]*tpts_dv[0];

        norm[2]=tpts_du[0]*tpts_dv[1]-
tpts_du[1]*tpts_dv[0];

        mag=sqrt(pow(norm[0],2.0)+pow(norm[1
],2.0)+pow(norm[2],2.0));

        norm[0]=norm[0]/(mag);

        norm[1]=norm[1]/(mag);

        norm[2]=norm[2]/(mag);

                    if (i != 0)
                    {

        glNormal3dv(norm_store[j]);

        glVertex3dv(tpts_store[j]);

        glNormal3dv(norm);

        glVertex3dv(tpts);
                    }
                    tpts_store[j][0]=tpts[0];
                    tpts_store[j][1]=tpts[1];
                    tpts_store[j][2]=tpts[2];

        norm_store[j][0]=norm[0];

        norm_store[j][1]=norm[1];

        norm_store[j][2]=norm[2];
                }
```

```
                    glEnd();
                }

        if (dboy_texture)
        {
            glDisable(GL_TEXTURE_2D);
            glEnd();
        }


            glPopMatrix();

            dboy_freeDoubleVector(tpts,0,2);
            dboy_freeDoubleVector(tpts_du,0,2);
            dboy_freeDoubleVector(tpts_dv,0,2);
            dboy_freeDoubleMatrix(temp,0,2,0,dboy_
P[1]);
            dboy_freeDoubleMatrix(temp_du,0,2,0,db
oy_P[1]);
            dboy_freeDoubleVector(norm,0,2);
            dboy_freeDoubleMatrix(tpts_store,0,dboy
_npatch[0],0,2);
            dboy_freeDoubleMatrix(norm_store,0,dbo
y_npatch[0],0,2);
}
```

```
// dsvbksb.c

#include "JimsCave.h"

void dboy_dsvbksb(double **u,double *w,double
**v,int m,int n,double *b,double *x)
{
        int jj,j,i;
        double s,*tmp;

        tmp=dboy_DoubleVector(1,n);
        for (j=1;j<=n;j++) {
                s=0.0;
                if (w[j]) {
                        for (i=1;i<=m;i++) s +=
u[i][j]*b[i];
                        s /= w[j];
                }
                tmp[j]=s;
        }
        for (j=1;j<=n;j++) {
                s=0.0;
                for (jj=1;jj<=n;jj++) s +=
v[j][jj]*tmp[jj];
                x[j]=s;
        }
        dboy_freeDoubleVector(tmp,1,n);
}
```

```
// dsvdcmp.c

#include "JimsCave.h"

void dboy_dsvdcmp(double **a,int m,int n,double
*w,double **v)
{
        int flag,i,its,j,jj,k,l,nm;
        double anorm,c,f,g,h,s,scale,x,y,z,*rv1;

        rv1=dboy_DoubleVector(1,n);
        g=scale=anorm=0.0;
        for (i=1;i<=n;i++) {
                l=i+1;
                rv1[i]=scale*g;
                g=s=scale=0.0;
                if (i <= m) {
                        for (k=i;k<=m;k++)
scale += fabs(a[k][i]);
                        if (scale) {
                                for
(k=i;k<=m;k++) {
                                        a[k][i]
/= scale;
                                        s +=
a[k][i]*a[k][i];
                                }
                                f=a[i][i];
                                g = -
dboy_SIGN(sqrt(s),f);
                                h=f*g-s;
                                a[i][i]=f-g;
                                for
(j=l;j<=n;j++) {
                                        for
(s=0.0,k=i;k<=m;k++) s += a[k][i]*a[k][j];
                                        f=s/h;
                                        for
(k=i;k<=m;k++) a[k][j] += f*a[k][i];
                                }
                                for
(k=i;k<=m;k++) a[k][i] *= scale;
                        }
                }
                w[i]=scale *g;
                g=s=scale=0.0;
                if (i <= m && i != n) {
                        for (k=l;k<=n;k++)
scale += fabs(a[i][k]);
                        if (scale) {
                                for
(k=l;k<=n;k++) {
                                        a[i][k]
/= scale;
                                        s +=
a[i][k]*a[i][k];
                                }
                                f=a[i][l];
                                g = -
dboy_SIGN(sqrt(s),f);
                                h=f*g-s;
                                a[i][l]=f-g;
                                for
(k=l;k<=n;k++) rv1[k]=a[i][k]/h;
                                for
(j=l;j<=m;j++) {
                                        for
(s=0.0,k=l;k<=n;k++) s += a[j][k]*a[i][k];
                                        for
(k=l;k<=n;k++) a[j][k] += s*rv1[k];
                                }
                                for
(k=l;k<=n;k++) a[i][k] *= scale;
                        }
                }

        anorm=dboy_DMAX(anorm,(fabs(w[i])+f
abs(rv1[i])));
        }
        for (i=n;i>=1;i--) {
                if (i < n) {
                        if (g) {
                                for
(j=l;j<=n;j++) v[j][i]=(a[i][j]/a[i][l])/g;
                                for
(j=l;j<=n;j++) {
                                        for
(s=0.0,k=l;k<=n;k++) s += a[i][k]*v[k][j];
                                        for
(k=l;k<=n;k++) v[k][j] += s*v[k][i];
                                }
                        }
                        for (j=l;j<=n;j++)
v[i][j]=v[j][i]=0.0;
                }
                v[i][i]=1.0;
                g=rv1[i];
                l=i;
        }
        for (i=dboy_IMIN(m,n);i>=1;i--) {
                l=i+1;
                g=w[i];
                for (j=l;j<=n;j++) a[i][j]=0.0;
                if (g) {
                        g=1.0/g;
                        for (j=l;j<=n;j++) {
                                for
(s=0.0,k=l;k<=m;k++) s += a[k][i]*a[k][j];
                                f=(s/a[i][i])*g;
                                for
(k=i;k<=m;k++) a[k][j] += f*a[k][i];
                        }
                        for (j=i;j<=m;j++)
a[j][i] *= g;
                } else for (j=i;j<=m;j++)
a[j][i]=0.0;
                ++a[i][i];
        }
        for (k=n;k>=1;k--) {
                for (its=1;its<=30;its++) {
                        flag=1;
                        for (l=k;l>=1;l--) {
                                nm=l-1;
                                if
((double)(fabs(rv1[l])+anorm) == anorm) {

        flag=0;
                                        break;
                                }
                                if
((double)(fabs(w[nm])+anorm) == anorm) break;
                        }
                        if (flag) {
                                c=0.0;
                                s=1.0;
                                for
(i=l;i<=k;i++) {

        f=s*rv1[i];

        rv1[i]=c*rv1[i];
                                        if
((double)(fabs(f)+anorm) == anorm) break;

        g=w[i];

        h=dboy_dpythag(f,g);

        w[i]=h;
```

```
        h=1.0/h;

        c=g*h;
                                        s = -
f*h;
                                        for
(j=1;j<=m;j++) {

        y=a[j][nm];

        z=a[j][i];

        a[j][nm]=y*c+z*s;

        a[j][i]=z*c-y*s;
                                        }
                                }
                        }
                        z=w[k];
                        if (l == k) {
                                if (z < 0.0) {
                                        w[k] =
-z;
                                        for
(j=1;j<=n;j++) v[j][k] = -v[j][k];
                                }
                                break;
                        }
                        if (its == 30) printf("no
convergence in 30 dsvdcmp iterations\n");
                        x=w[l];
                        nm=k-1;
                        y=w[nm];
                        g=rv1[nm];
                        h=rv1[k];
                        f=((y-z)*(y+z)+(g-
h)*(g+h))/(2.0*h*y);
                        g=dboy_dpythag(f,1.0);
                        f=((x-
z)*(x+z)+h*((y/(f+dboy_SIGN(g,f)))-h))/x;
                        c=s=1.0;
                        for (j=l;j<=nm;j++) {
                                i=j+1;
                                g=rv1[i];
                                y=w[i];
                                h=s*g;
                                g=c*g;

        z=dboy_dpythag(f,h);
                                rv1[j]=z;
                                c=f/z;
```

```
                              s=h/z;
                              f=x*c+g*s;
                              g = g*c-x*s;
                              h=y*s;
                              y *= c;
                              for
(jj=1;jj<=n;jj++) {

      x=v[jj][j];

      z=v[jj][i];

      v[jj][j]=x*c+z*s;

      v[jj][i]=z*c-x*s;
                              }
      z=dboy_dpythag(f,h);
                              w[j]=z;
                              if (z) {

      z=1.0/z;
                                   c=f*z;

      s=h*z;
                              }
                              f=c*g+s*y;
                              x=c*y-s*g;
                              for
(jj=1;jj<=m;jj++) {

      y=a[jj][j];

      z=a[jj][i];

      a[jj][j]=y*c+z*s;

      a[jj][i]=z*c-y*s;
                              }
                         }
                         rv1[l]=0.0;
                         rv1[k]=f;
                         w[k]=x;
                    }
               }
      dboy_freeDoubleVector(rv1,1,n);
}
```

```
// element.c
```

```
#include "JimsCave.h"

extern int *dboy_NCPTS;
extern int *dboy_P;
extern double *dboy_U, *dboy_V;
extern double **dboy_M, **dboy_K;

extern dboy_File *dboy_parameter;

void dboy_element(void)
{
        int i,j,k,l,ii,jj,iii,jjj;
        int ng;
        int s,r;
        double
**temp,**temp_du,dx_du,dy_du,dx_dv,dy_dv,ma
g_j,**jm,**jinv;
        double a,b,c,d,dGauss;
        double u,v;
        int uspan,vspan,nu,nv,du,dv;
        double *g,*w,*gu,*gv;
        double
**uders,**vders,*Ni,*Ni_du,*Ni_dv;

/*  Gauss quadrature points and weights  */
        ng=4;
        g=dboy_DoubleVector(0,ng-1);
        gu=dboy_DoubleVector(0,ng-1);
        gv=dboy_DoubleVector(0,ng-1);
        g[0]=0.861136312;
        g[1]=0.339981044;
        g[2]=-0.339981044;
        g[3]=-0.861136312;

        w=dboy_DoubleVector(0,ng-1);
        w[0]=0.34785485;
        w[1]=0.65214515;
        w[2]=0.65214515;
        w[3]=0.34785485;

/*  allocate some memory  */
        uders=dboy_DoubleMatrix(0,dboy_P[0],0
,dboy_P[0]);
        vders=dboy_DoubleMatrix(0,dboy_P[1],0
,dboy_P[1]);
        Ni=dboy_DoubleVector(0,(dboy_P[0]+1)
*(dboy_P[1]+1)-1);
        Ni_du=dboy_DoubleVector(0,(dboy_P[0]
+1)*(dboy_P[1]+1)-1);
        Ni_dv=dboy_DoubleVector(0,(dboy_P[0]
+1)*(dboy_P[1]+1)-1);
```

```
        temp=dboy_DoubleMatrix(0,1,0,dboy_P[
1]);
        temp_du=dboy_DoubleMatrix(0,1,0,dboy
_P[1]);
        jm=dboy_DoubleMatrix(0,1,0,1);
        jinv=dboy_DoubleMatrix(0,1,0,1);

/* evaluate mass and stiffness matrices */

/* zero matrices */
        for
(i=0;i<=dboy_NCPTS[0]*dboy_NCPTS[1]-1;i++)
        {
                for
(j=0;j<=dboy_NCPTS[0]*dboy_NCPTS[1]-1;j++)
                {
                        dboy_M[i][j]=0.0;
                        dboy_K[i][j]=0.0;
                }
        }

/* integrate each element separately */
        for (i=dboy_P[0];i<=dboy_NCPTS[0]-
1;i++)
        {
                for
(j=dboy_P[1];j<=dboy_NCPTS[1]-1;j++)
                {

/* scale quadrature points to correct range */
                        a=dboy_U[i];
                        b=dboy_U[i+1];

                        c=dboy_V[j];
                        d=dboy_V[j+1];

                        for (k=0;k<ng;k++)
                        {

        gu[k]=((b+a)/2.0)+((b-a)/2.0)*g[k];

        gv[k]=((d+c)/2.0)+((d-c)/2.0)*g[k];
                        }

/* determine magnitude change due to rescaling
*/
                        dGauss=((b-a)/2.0)*((d-
c)/2.0);

/* initialize stuff for basis function evaluation */
                        nu=dboy_NCPTS[0]-1;
                        du=1;

                        nv=dboy_NCPTS[1]-1;
                        dv=1;

/* for each Gauss point */
                        for (k=0;k<ng;k++)
                        {
/* evaluate interpolation (basis) functions */
                                u=gu[k];

        uspan=dboy_findspan(u,dboy_U,nu,dboy_
P[0]);

        dboy_basisfuns(uspan,u,dboy_P[0],du,dbo
y_U,uders);
                                for
(l=0;l<ng;l++)
                                {

        v=gv[l];

        vspan=dboy_findspan(v,dboy_V,nv,dboy_
P[1]);

        dboy_basisfuns(vspan,v,dboy_P[1],dv,dbo
y_V,vders);
                                        for
(ii=0;ii<=dboy_P[1];ii++)
                                        {

        for (jj=0;jj<=dboy_P[0];jj++)

        {

        Ni[jj+(dboy_P[0]+1)*ii]=uders[0][jj]*vde
rs[0][ii];

        Ni_du[jj+(dboy_P[0]+1)*ii]=uders[1][jj]*
vders[0][ii];

        Ni_dv[jj+(dboy_P[0]+1)*ii]=uders[0][jj]*
vders[1][ii];

        }
                                        }
/* evaluate Jacobian matrix */
                                        for
(s=0;s<=dboy_P[1];s++)
```

```
                                        {

        temp[0][s]=0.0;

        temp[1][s]=0.0;


        temp_du[0][s]=0.0;

        temp_du[1][s]=0.0;


        for (r=0;r<=dboy_P[0];r++)

        {


        temp[0][s]=temp[0][s]+uders[0][r]*dboy_
parameter->cpts[uspan-dboy_P[0]+r][vspan-
dboy_P[1]+s][0];


        temp[1][s]=temp[1][s]+uders[0][r]*dboy_
parameter->cpts[uspan-dboy_P[0]+r][vspan-
dboy_P[1]+s][1];



        temp_du[0][s]=temp_du[0][s]+uders[1][r]
*dboy_parameter->cpts[uspan-
dboy_P[0]+r][vspan-dboy_P[1]+s][0];


        temp_du[1][s]=temp_du[1][s]+uders[1][r]
*dboy_parameter->cpts[uspan-
dboy_P[0]+r][vspan-dboy_P[1]+s][1];

        }
                                        }


        dx_du=0.0;

        dy_du=0.0;


        dx_dv=0.0;

        dy_dv=0.0;

                                        for
(s=0;s<=dboy_P[1];s++)
```

```
                                        {

        dx_du=dx_du+vders[0][s]*temp_du[0][s]
;

        dy_du=dy_du+vders[0][s]*temp_du[1][s];


        dx_dv=dx_dv+vders[1][s]*temp[0][s];

        dy_dv=dy_dv+vders[1][s]*temp[1][s];
                                        }


        jm[0][0]=dx_du;

        jm[0][1]=dy_du;

        jm[1][0]=dx_dv;

        jm[1][1]=dy_dv;

        mag_j=jm[0][0]*jm[1][1]-
jm[0][1]*jm[1][0];

/*  evaluate the inverse of the Jacobian matrix  */

        jinv[0][0]=jm[1][1]/mag_j;

        jinv[0][1]=-jm[0][1]/mag_j;

        jinv[1][0]=-jm[1][0]/mag_j;

        jinv[1][1]=jm[0][0]/mag_j;

/*  evaluate integral and assemble into global
matrix  */
                                        for
(ii=0;ii<=(dboy_P[0]+1)*(dboy_P[1]+1)-1;ii++)
                                        {

        for
(jj=ii;jj<=(dboy_P[0]+1)*(dboy_P[1]+1)-1;jj++)

        {

                iii=ii-
(dboy_P[0]+1)*(ii/(dboy_P[0]+1))+uspan-
dboy_P[0]+dboy_NCPTS[0]*((ii/(dboy_P[0]+1))+
vspan-dboy_P[1]);

                jjj=jj-
(dboy_P[0]+1)*(jj/(dboy_P[0]+1))+uspan-
dboy_P[0]+dboy_NCPTS[0]*((jj/(dboy_P[0]+1))+
vspan-dboy_P[1]);

                    dboy_M[iii][jjj] +=
Ni[ii]*Ni[jj]*w[k]*w[l]*mag_j*dGauss;

                    dboy_K[iii][jjj] +=
(Ni_du[ii]*Ni_du[jj]*(jinv[0][0]*jinv[0][0]+jinv[1
][0]*jinv[1][0])


+Ni_dv[ii]*Ni_du[jj]*(jinv[0][0]*jinv[0][1]+jinv[1
][0]*jinv[1][1])


+Ni_du[ii]*Ni_dv[jj]*(jinv[0][0]*jinv[0][1]+jinv[1
][0]*jinv[1][1])


+Ni_dv[ii]*Ni_dv[jj]*(jinv[0][1]*jinv[0][1]+jinv[1
][1]*jinv[1][1]))*w[k]*w[l]*mag_j*dGauss;

          }
                                        }
                              }
                    }
          }
     }

/* fill in remaining elements */
     for
(k=0;k<=dboy_NCPTS[0]*dboy_NCPTS[1]-
1;k++)
     {
          for
(l=k;l<=dboy_NCPTS[0]*dboy_NCPTS[1]-1;l++)
          {
               if (l != k)
               {

     dboy_M[l][k]=dboy_M[k][l];

     dboy_K[l][k]=dboy_K[k][l];
               }
          }
     }

/* free some memory */
     dboy_freeDoubleVector(g,0,ng-1);
     dboy_freeDoubleVector(w,0,ng-1);
     dboy_freeDoubleVector(gu,0,ng-1);
     dboy_freeDoubleVector(gv,0,ng-1);
     dboy_freeDoubleMatrix(uders,0,dboy_P[0
],0,dboy_P[0]);
     dboy_freeDoubleMatrix(vders,0,dboy_P[1
],0,dboy_P[1]);
     dboy_freeDoubleVector(Ni,0,(dboy_P[0]+
1)*(dboy_P[1]+1)-1);
     dboy_freeDoubleVector(Ni_du,0,(dboy_P[
0]+1)*(dboy_P[1]+1)-1);
     dboy_freeDoubleVector(Ni_dv,0,(dboy_P[
0]+1)*(dboy_P[1]+1)-1);
     dboy_freeDoubleMatrix(temp,0,1,0,dboy_
P[1]);
     dboy_freeDoubleMatrix(temp_du,0,1,0,db
oy_P[1]);
     dboy_freeDoubleMatrix(jm,0,1,0,1);
     dboy_freeDoubleMatrix(jinv,0,1,0,1);
}
```

```
// findspan.c

#include "JimsCave.h"

int dboy_findspan(double u,double *U,int n,int p)
{
     int low,mid,high;
     if (u==U[n+1])  return (n);
     low = p;
     high = n+1;
     while (low <= high)
     {
          mid = (low+high)/2;
          if (u==U[mid])
          {
               while (u==U[mid+1])
mid++;
               return (mid);
          }
          if (u<U[mid])  high=mid-1;
               else      low=mid+1;
     }
     if (u==U[high])
     {
          while (u==U[high])  high++;
          return (high);
     }
     return (low-1);
}
```

```
// init_cp.c

#include "JimsCave.h"

/* declare external pointer to shared memory
arena */
extern void *dboy_sharedData;

/* declare external shared memory pointers */
extern int *dboy_NCPTS;
extern int *dboy_P;
extern int *dboy_npatch;
extern double *dboy_U, *dboy_V;
extern double **dboy_M, **dboy_K;
extern double *dboy_PNEW, *dboy_PCURRENT,
*dboy_POLD, *dboy_PREF;
extern double **dboy_PCONST;
extern double **dboy_X, **dboy_Y, **dboy_Z;
extern double **dboy_X_DU, **dboy_Y_DU;
extern double **dboy_X_DV, **dboy_Y_DV;
extern double ***dboy_U_BAS,
***dboy_U_BAS_DER, ***dboy_V_BAS,
***dboy_V_BAS_DER;
extern double *dboy_WAND;
extern double *dboy_TRANS;
extern int *dboy_Playme;

extern dboy_File *dboy_parameter;

void dboy_initCompute(void)
{
        int i, j;
        int m, n;
        int umin, umax, vmin, vmax;

        int nu, du, nv, dv;
        double **uders, **vders;
        double *tpts, *tpts_du, *tpts_dv;
        double **temp, **temp_du;
        int uspan, vspan;
        double u, v;
        int s, r;

/* create a shared memory arena */
        dboy_sharedData=CAVEUserSharedMe
mory(51200000);

/* allocate shared memory for data structure */
   dboy_parameter=(dboy_File *)
amalloc(sizeof(dboy_File),dboy_sharedData);

        dboy_NCPTS=dboy_sharedIntVector(0,1,
dboy_sharedData);

/* number of control points in u direction */
        dboy_NCPTS[0]=8;

/* number of control points in v direction */
        dboy_NCPTS[1]=8;

        dboy_P=dboy_sharedIntVector(0,1,dboy_
sharedData);

/* degree in u direction */
        dboy_P[0]=2;

/* d3egree in v direction */
        dboy_P[1]=2;

/* length of the knot vector in u direction */
        m=dboy_NCPTS[0]+dboy_P[0];

/* length of the knot vector in u direction */
        n=dboy_NCPTS[1]+dboy_P[1];

        dboy_U=dboy_sharedDoubleVector(0,m,d
boy_sharedData);

/* evaluate uniform knot vector in u direction */
        dboy_knot(dboy_U,m,dboy_P[0]);

        dboy_V=dboy_sharedDoubleVector(0,n,d
boy_sharedData);

/* evaluate uniform knot vector in v direction */
        dboy_knot(dboy_V,n,dboy_P[1]);

/* initial control point positions */
        for (i=0;i<dboy_NCPTS[0];i++)
        {
                for (j=0;j<dboy_NCPTS[1];j++)
                {
                        dboy_parameter-
>cpts[i][j][0]=5.0*(((double) i)/((double)
(dboy_NCPTS[0] - 1)));
                        dboy_parameter-
>cpts[i][j][1]=5.0*(((double) j)/((double)
(dboy_NCPTS[1] - 1)));
                        dboy_parameter-
>cpts[i][j][2]=0.0;
                }
        }
        dboy_M=dboy_sharedDoubleMatrix(0,db
oy_NCPTS[0]*dboy_NCPTS[1]-
1,0,dboy_NCPTS[0]*dboy_NCPTS[1]-
1,dboy_sharedData);
        dboy_K=dboy_sharedDoubleMatrix(0,dbo
y_NCPTS[0]*dboy_NCPTS[1]-
1,0,dboy_NCPTS[0]*dboy_NCPTS[1]-
1,dboy_sharedData);

/* integrate for FEA model */
        dboy_element();

        dboy_PNEW=dboy_sharedDoubleVector(
1,dboy_NCPTS[0]*dboy_NCPTS[1],dboy_shared
Data);
        dboy_PCURRENT=dboy_sharedDoubleV
ector(1,dboy_NCPTS[0]*dboy_NCPTS[1],dboy_sh
aredData);
        dboy_POLD=dboy_sharedDoubleVector(
1,dboy_NCPTS[0]*dboy_NCPTS[1],dboy_shared
Data);
        dboy_PREF=dboy_sharedDoubleVector(1
,dboy_NCPTS[0]*dboy_NCPTS[1],
dboy_sharedData);

/* establish initial conditions */
        for (i=0;i<dboy_NCPTS[1];i++)
        {
                for (j=0;j<dboy_NCPTS[0];j++)
                {

        dboy_PCURRENT[dboy_NCPTS[0]*i+j+
1]=dboy_parameter->cpts[j][i][2];
                }
        }

        for
(i=1;i<=dboy_NCPTS[0]*dboy_NCPTS[1];i++)
        {

        dboy_POLD[i]=dboy_PCURRENT[i];
                dboy_PNEW[i]=0.0;
                dboy_PREF[i]=0.0;
        }

        dboy_PCONST=dboy_sharedDoubleMatri
x(1,dboy_NCPTS[0]*dboy_NCPTS[1],1,2,dboy_sh
aredData);

/* zero the primary contraint matrix */
        for
(i=1;i<=dboy_NCPTS[0]*dboy_NCPTS[1];i++)
        {
                dboy_PCONST[i][1]=0.0;
                dboy_PCONST[i][2]=0.0;
        }

/* set boundary conditions */
        umin=0;
        umax=dboy_NCPTS[0]-1;
        vmin=0;
        vmax=dboy_NCPTS[1]-1;
        for (i=umin;i<=umax;i++)
        {
                dboy_PCONST[i+1][1]=10.0;
                dboy_PCONST[i+1][2]=0.0;

        dboy_PCONST[i+dboy_NCPTS[0]*vmax
+1][1]=10.0;

        dboy_PCONST[i+dboy_NCPTS[0]*vmax
+1][2]=0.0;
        }
        for (i=vmin+1;i<vmax;i++)
        {

        dboy_PCONST[dboy_NCPTS[0]*i+1][1]
=10.0;

        dboy_PCONST[dboy_NCPTS[0]*i+1][2]
=0.0;

        dboy_PCONST[dboy_NCPTS[0]*i+umax
+1][1]=10.0;

        dboy_PCONST[dboy_NCPTS[0]*i+umax
+1][2]=0.0;
        }

/* evaluate x and y components for B-spline
surface */
        dboy_npatch=dboy_sharedIntVector(0,0,d
boy_sharedData);
        dboy_npatch[0]=20;
        dboy_X=dboy_sharedDoubleMatrix(0,dbo
y_npatch[0]+1,0,dboy_npatch[0]+1,dboy_sharedD
ata);
        dboy_Y=dboy_sharedDoubleMatrix(0,dbo
y_npatch[0]+1,0,dboy_npatch[0]+1,dboy_sharedD
ata);
        dboy_Z=dboy_sharedDoubleMatrix(0,dbo
y_npatch[0]+1,0,dboy_npatch[0]+1,dboy_sharedD
ata);
```

```
        dboy_X_DU=dboy_sharedDoubleMatrix(
0,dboy_npatch[0]+1,0,dboy_npatch[0]+1,dboy_sha
redData);
        dboy_Y_DU=dboy_sharedDoubleMatrix(
0,dboy_npatch[0]+1,0,dboy_npatch[0]+1,dboy_sha
redData);
        dboy_X_DV=dboy_sharedDoubleMatrix(
0,dboy_npatch[0]+1,0,dboy_npatch[0]+1,dboy_sha
redData);
        dboy_Y_DV=dboy_sharedDoubleMatrix(
0,dboy_npatch[0]+1,0,dboy_npatch[0]+1,dboy_sha
redData);
        dboy_U_BAS=dboy_sharedDouble3Tenso
r(0,dboy_npatch[0]+1,0,dboy_npatch[0]+1,0,dboy
_P[0],dboy_sharedData);
        dboy_U_BAS_DER=dboy_sharedDouble3
Tensor(0,dboy_npatch[0]+1,0,dboy_npatch[0]+1,0
,dboy_P[0],dboy_sharedData);
        dboy_V_BAS=dboy_sharedDouble3Tenso
r(0,dboy_npatch[0]+1,0,dboy_npatch[0]+1,0,dboy
_P[1],dboy_sharedData);
        dboy_V_BAS_DER=dboy_sharedDouble
3Tensor(0,dboy_npatch[0]+1,0,dboy_npatch[0]+1,
0,dboy_P[1],dboy_sharedData);

        nu=dboy_NCPTS[0]-1;
        du=1;
        uders=dboy_DoubleMatrix(0,dboy_P[0],0
,dboy_P[0]);

        nv=dboy_NCPTS[1]-1;
        dv=1;
        vders=dboy_DoubleMatrix(0,dboy_P[1],0
,dboy_P[1]);

        tpts=dboy_DoubleVector(0,1);
        tpts_du=dboy_DoubleVector(0,1);
        tpts_dv=dboy_DoubleVector(0,1);
        temp=dboy_DoubleMatrix(0,1,0,dboy_P[
1]);
        temp_du=dboy_DoubleMatrix(0,1,0,dboy
_P[1]);

        for (i=0;i<=dboy_npatch[0];i++)
        {
                u=((double) i)/((double)
dboy_npatch[0]);

        uspan=dboy_findspan(u,dboy_U,nu,dboy_
P[0]);

        dboy_basisfuns(uspan,u,dboy_P[0],du,dbo
y_U,uders);
                for (j=0;j<=dboy_npatch[0];j++)
                {
                        v=((double) j)/((double)
dboy_npatch[0]);

        vspan=dboy_findspan(v,dboy_V,nv,dboy_
P[1]);

        dboy_basisfuns(vspan,v,dboy_P[1],dv,dbo
y_V,vders);
                        for
(s=0;s<=dboy_P[1];s++)
                        {

        temp[0][s]=0.0;

        temp[1][s]=0.0;

        temp_du[0][s]=0.0;

        temp_du[1][s]=0.0;
                                for
(r=0;r<=dboy_P[0];r++)
                                {

        temp[0][s]=temp[0][s]+uders[0][r]*dboy_
parameter->cpts[uspan-dboy_P[0]+r][vspan-
dboy_P[1]+s][0];

        temp[1][s]=temp[1][s]+uders[0][r]*dboy_
parameter->cpts[uspan-dboy_P[0]+r][vspan-
dboy_P[1]+s][1];

        temp_du[0][s]=temp_du[0][s]+uders[1][r]
*dboy_parameter->cpts[uspan-
dboy_P[0]+r][vspan-dboy_P[1]+s][0];

        temp_du[1][s]=temp_du[1][s]+uders[1][r]
*dboy_parameter->cpts[uspan-
dboy_P[0]+r][vspan-dboy_P[1]+s][1];
                                }
                        }

                        tpts[0]=0.0;
                        tpts[1]=0.0;
                        tpts_du[0]=0.0;
                        tpts_du[1]=0.0;
                        tpts_dv[0]=0.0;
                        tpts_dv[1]=0.0;
                        for
(s=0;s<=dboy_P[1];s++)
                        {

        tpts[0]=tpts[0]+vders[0][s]*temp[0][s];

        tpts[1]=tpts[1]+vders[0][s]*temp[1][s];

        tpts_du[0]=tpts_du[0]+vders[0][s]*temp_
du[0][s];

        tpts_du[1]=tpts_du[1]+vders[0][s]*temp_
du[1][s];

        tpts_dv[0]=tpts_dv[0]+vders[1][s]*temp[
0][s];

        tpts_dv[1]=tpts_dv[1]+vders[1][s]*temp[
1][s];
                        }

                        dboy_X[i][j]=tpts[0];
                        dboy_Y[i][j]=tpts[1];

        dboy_X_DU[i][j]=tpts_du[0];

        dboy_Y_DU[i][j]=tpts_du[1];

        dboy_X_DV[i][j]=tpts_dv[0];

        dboy_Y_DV[i][j]=tpts_dv[1];
                        for
(r=0;r<=dboy_P[0];r++)
                        {

        dboy_U_BAS[i][j][r]=uders[0][r];

        dboy_U_BAS_DER[i][j][r]=uders[1][r];
                        }
                        for
(r=0;r<=dboy_P[1];r++)
                        {

        dboy_V_BAS[i][j][r]=vders[0][r];

        dboy_V_BAS_DER[i][j][r]=vders[1][r];
                        }
                }
        }

        dboy_freeDoubleMatrix(uders,0,dboy_P[0
],0,dboy_P[0]);
        dboy_freeDoubleMatrix(vders,0,dboy_P[1
],0,dboy_P[1]);
        dboy_freeDoubleVector(tpts,0,1);
        dboy_freeDoubleVector(tpts_du,0,1);
        dboy_freeDoubleVector(tpts_dv,0,1);
        dboy_freeDoubleMatrix(temp,0,1,0,dboy_
P[1]);
        dboy_freeDoubleMatrix(temp_du,0,1,0,db
oy_P[1]);

/* allocate memory for wand position */
        dboy_Playme=dboy_sharedIntVector(0,4,
dboy_sharedData);
        dboy_Playme[0]=0;
        dboy_Playme[1]=0;
        dboy_Playme[2]=0;
        dboy_Playme[3]=0;
        dboy_WAND=dboy_sharedDoubleVector
(0,3,dboy_sharedData);
        dboy_TRANS=dboy_sharedDoubleVector
(0,6,dboy_sharedData);
}
```

```
// init_gr.c

#include "JimsCave.h"
#include <loadImage.h>

int dboy_backTexIndex;
extern int dboy_texture;

/* InitScene() is called only once, at the start of
the program */
void dboy_initScene(void)
{
        static unsigned long *texImage;
        static long sizex, sizey;
        GLfloat mat_specular[] = { 0.5, 0.5, 0.5,
1.0 };
        GLfloat mat_shininess[] = { 100.0 };
        GLfloat light0_ambient[] = { .1, .1, .1,
1.0 };
        GLfloat light0_diffuse[] = { .8, 0.8, 0.8,
1.0 };
        GLfloat light0_specular[] = { 0.9, 0.9,
0.9, 1.0 };
        GLfloat light0_position[] = {10.0, 10.0,
5.0, 1.0 };
```

```
        static float texGenParam[1] =
{GL_OBJECT_LINEAR};
        static float texGenTParam[4] = {0.0, 0.2,
0.0, 0.0};
        static float texGenSParam[4] = {0.2, 0.0,
0.0, 0.0};

        glMaterialfv(GL_FRONT,
GL_SPECULAR, mat_specular);
        glMaterialfv(GL_FRONT,
GL_SHININESS, mat_shininess);
        glLightfv(GL_LIGHT0, GL_AMBIENT,
light0_ambient);
        glLightfv(GL_LIGHT0, GL_DIFFUSE,
light0_diffuse);
        glLightfv(GL_LIGHT0, GL_SPECULAR,
light0_specular);
        glLightfv(GL_LIGHT0, GL_POSITION,
light0_position);


        glEnable(GL_DEPTH_TEST);
        glEnable(GL_NORMALIZE);

        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);

        glEnable(GL_COLOR_MATERIAL);
        glShadeModel(GL_SMOOTH);

  if (dboy_texture)
  {
        dboy_backTexIndex = glGenLists(1);
        texImage =
readLongImageData("gdb.rgb", &sizex, &sizey);

        glNewList(dboy_backTexIndex,
GL_COMPILE_AND_EXECUTE);
        glEnable(GL_TEXTURE_2D);

        glTexImage2D(GL_TEXTURE_2D, 0,
4, (int)sizex, (int)sizey, 0, GL_RGBA,
GL_UNSIGNED_BYTE,
            texImage);

        glTexEnvf(GL_TEXTURE_ENV,
GL_TEXTURE_ENV_MODE,
GL_MODULATE);
        glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);

        glTexGenfv(GL_S,GL_TEXTURE_GEN
_MODE, texGenParam);
        glTexGenfv(GL_T,GL_TEXTURE_GEN
_MODE, texGenParam);
        glTexGenfv(GL_T,GL_OBJECT_PLANE
, texGenTParam);
        glTexGenfv(GL_S,GL_OBJECT_PLANE
, texGenSParam);

        glEnable(GL_TEXTURE_GEN_S);
        glEnable(GL_TEXTURE_GEN_T);

        glColor4f(1.0, 1.0, 1.0, 0.3);
        glEndList();
   }
}
```

```
// init_so.c

#include "JimsCave.h"

extern awSound *dboy_sowood;
extern awSoumd *dboy_soboing;
extern awSound *dboy_socreaky;

void dboy_initSound(void)
{
/* only needs to be local, then one can delete
adfFile[100] */
   static char adfFile[100];

   /* print the adf file name to a string */

   sprintf(adfFile, "%s",
"/home/usr1/trisha/research/CAVE/Sound/jims.adf
");


   /* initialize the AudioWorks system */

   printf( "Load adf file [%s]\n", adfFile);

   awInitSys();
   awDefineSys(adfFile);
   awConfigSys();
   /* find the sound we want */

   printf ( "Find the mysound wave\n");
   dboy_sowood = awFindSnd("wood4");
   dboy_soboing = awFindSnd("boing2");
   dboy_socreaky = awFindSnd("creaky");
   printf ( "Located mysound\n");

   if ( dboy_sowood == 0 || dboy_soboing==0 ||
dboy_socreaky==0 )
   {
      awNotify ( AW_FATAL, AW_APP,"Unable
to locate mysound\n");
   }

   awProp(dboy_sowood, AWSND_RETRIGGER,
AW_ON);
   awProp(dboy_soboing, AWSND_RETRIGGER,
AW_ON);
}
```

```
// inversion.c

#include "JimsCave.h"

extern int *dboy_NCPTS;
extern int *dboy_P;
extern double *dboy_U, *dboy_V;
extern double **dboy_X, **dboy_Y, **dboy_Z;
extern double ***dboy_U_BAS, ***dboy_V_BAS;
extern int *dboy_npatch;
extern double *dboy_TRANS;

extern int dboy_old_int;
extern int dboy_play;

extern dboy_File *dboy_parameter;

extern int *dboy_Playme;

void dboy_inversion(double *WAND, double
**sconst)
{
        int i,j,ii,jj;
        double shortdistance, distance;
        double ustart, vstart;
        int usint, vsint, neqn;
        int nu,nv;
        int uspan,vspan;
        double dz;
        double wmin, wmax;
        double *bb, *b, **a, *w, **v, *x;
        double up, vp;
        double wand[3];
        double cx,sx,cy,sy,cz,sz;

        cy=cos(dboy_parameter-
>wand_old[3]*3.14159/180.0);
        sy=sin(dboy_parameter-
>wand_old[3]*3.14159/180.0);
        cx=cos(dboy_parameter-
>wand_old[4]*3.14159/180.0);
        sx=sin(dboy_parameter-
>wand_old[4]*3.14159/180.0);
        cz=cos(dboy_parameter-
>wand_old[5]*3.14159/180.0);
        sz=sin(dboy_parameter-
>wand_old[5]*3.14159/180.0);

        wand[0]=(cz*cy+sz*sx*sy)*WAND[0]+s
z*cx*WAND[1]+(-cz*sy+sz*sx*cy)*WAND[2];
        wand[0]=wand[0]+(-dboy_parameter-
>wand_old[0]-
dboy_TRANS[0])*(cz*cy+sz*sx*sy)+(-
dboy_parameter->wand_old[1]-
dboy_TRANS[1])*sz*cx+(-dboy_parameter-
>wand_old[2]-dboy_TRANS[2])*(sz*sx*cy-
cz*sy)+dboy_TRANS[0]+2.5;
        wand[1]=(-
sz*cy+cz*sx*sy)*WAND[0]+cz*cx*WAND[1]+(s
z*sy+cz*sx*cy)*WAND[2];
        wand[1]=wand[1]+(-dboy_parameter-
>wand_old[0]-dboy_TRANS[0])*(cz*sx*sy-
sz*cy)+(-dboy_parameter->wand_old[1]-
dboy_TRANS[1])*cz*cx+(-dboy_parameter-
>wand_old[2]-
dboy_TRANS[2])*(cz*sx*cy+sz*sy)+dboy_TRAN
S[1]-2.5;
        wand[2]=cx*sy*WAND[0]-
sx*WAND[1]+cx*cy*WAND[2]+(-
dboy_parameter->wand_old[0]-
dboy_TRANS[0])*cx*sy-(-dboy_parameter-
>wand_old[1]-dboy_TRANS[1])*sx+(-
dboy_parameter->wand_old[2]-
dboy_TRANS[2])*cx*cy+dboy_TRANS[2];


/*      wand[0]=WAND[0]-dboy_parameter-
>wand_old[0]+2.5;
```

```
        wand[1]=WAND[1]-dboy_parameter-
>wand_old[1]-2.5;
        wand[2]=WAND[2]-dboy_parameter-
>wand_old[2];*/

/* Do point inverstion */
        shortdistance=1000.0;
        for (i=0;i<=dboy_npatch[0];i++)
        {
                for (j=0;j<=dboy_npatch[0];j++)
                {

        distance=pow(dboy_X[i][j]-
wand[0],2.0)+pow(dboy_Y[i][j]-
wand[1],2.0)+pow(-wand[2],2.0);
                        if (distance <
shortdistance)
                        {

        ustart=((double) i)/((double)
dboy_npatch[0]);

        vstart=((double) i)/((double)
dboy_npatch[0]);
                                usint=i;
                                vsint=j;

        shortdistance=distance;
                        }
                }
        }
        dboy_parameter-
>surf[0]=dboy_X[usint][vsint];
        dboy_parameter-
>surf[1]=dboy_Y[usint][vsint];
        dboy_parameter-
>surf[2]=dboy_Z[usint][vsint]-0.1;

/* Do collision detection */
        dboy_parameter->intersect=0;
/*      if ((dboy_Z[usint][vsint] >= wand[2]) &&
(shortdistance < 0.1)) dboy_parameter-
>intersect=1;*/
        if (dboy_Z[usint][vsint] >= wand[2])
dboy_parameter->intersect=1;



/* If collision, performed free-form deformation
*/
        if (dboy_parameter->intersect == 1 &&
dboy_old_int == 0)
        {
          dboy_Playme[1]=1;
        }
        dboy_old_int=dboy_parameter->intersect;
        if (dboy_parameter->intersect &&
CAVEBUTTON1)
        {
                if (dboy_play == 0)
                {
                  dboy_Playme[2]=1;
                  dboy_play=1;
                }


        neqn=(dboy_P[0]+1)*(dboy_P[1]+1);


        a=dboy_DoubleMatrix(1,neqn,1,neqn);
                w=dboy_DoubleVector(1,neqn);

        v=dboy_DoubleMatrix(1,neqn,1,neqn);
                x=dboy_DoubleVector(1,neqn);
                b=dboy_DoubleVector(1,neqn);
                bb=dboy_DoubleVector(0,neqn-
1);

                dz=wand[2]-
dboy_Z[usint][vsint];

                for (i=0;i<=dboy_P[0];i++)
                {
                        for
(j=0;j<=dboy_P[1];j++)
                        {

        bb[i+(dboy_P[0]+1)*j]=dboy_U_BAS[usi
nt][vsint][i]*dboy_V_BAS[usint][vsint][j];

        b[i+(dboy_P[0]+1)*j+1]=bb[i+(dboy_P[0]
+1)*j]*dz;
                        }
                }

                for (i=0;i<neqn;i++)
                {
                        for (j=0;j<neqn;j++)
                        {

        a[i+1][j+1]=bb[i]*bb[j];
                        }
                }
        dboy_dsvdcmp(a,neqn,neqn,w,v);
                wmax=0.0;
                for (j=1;j<=neqn;j++) if (w[j] >
wmax) wmax=w[j];
                wmin=wmax*0.000001;
                for (j=1;j<=neqn;j++) if (w[j] <
wmin) w[j]=0.0;

        dboy_dsvbksb(a,w,v,neqn,neqn,b,x);

                nu=dboy_NCPTS[0]-1;
                nv=dboy_NCPTS[1]-1;
                up=((double) usint)/((double)
dboy_npatch[0]);
                vp=((double) vsint)/((double)
dboy_npatch[0]);

        uspan=dboy_findspan(up,dboy_U,nu,dboy
_P[0]);

        vspan=dboy_findspan(vp,dboy_V,nv,dboy
_P[1]);

                for (i=0;i<=dboy_P[0];i++)
                {
                        for
(j=0;j<=dboy_P[1];j++)
                        {
                                jj=uspan-
dboy_P[0]+i;
                                ii=vspan-
dboy_P[1]+j;

        sconst[dboy_NCPTS[0]*ii+jj+1][1]=10.0;

        sconst[dboy_NCPTS[0]*ii+jj+1][2]=dboy
_parameter->cpts[jj][ii][2]+x[i+(dboy_P[0]+1)*j];
                        }
                }


        dboy_freeDoubleMatrix(a,1,neqn,1,neqn);

        dboy_freeDoubleVector(w,1,neqn);

        dboy_freeDoubleMatrix(v,1,neqn,1,neqn)
;

        dboy_freeDoubleVector(x,1,neqn);

        dboy_freeDoubleVector(b,1,neqn);
        dboy_freeDoubleVector(bb,0,neqn-1);
        }
        else
        {
          if (dboy_play==1)
          {
                dboy_Playme[3]=1;
                dboy_play=0;
          }
        }
}
```

```
// jimscave.h

#include <cave_ogl.h>
#include <unistd.h>
#include <math.h>
#include <stdio.h>
#include <aw.h>
#include <stdlib.h>
#include "util.h"

/* function prototypes */
void dboy_initScene(void);
void dboy_drawScene(void);
void dboy_Compute(void);
void dboy_initCompute(void);
void dboy_knot(double *, int, int);
void dboy_element(void);
int dboy_findspan(double, double *, int, int);
void dboy_basisfuns(int, double, int, int, double *,
double **);
void dboy_step(double *, double *, double *,
double **, double **, double, double **, double
**);
void dboy_dsvdcmp(double **, int, int, double *,
double **);
double dboy_dpythag(double, double);
void dboy_dsvbksb(double **, double *, double **,
int, int, double *, double *);
void dboy_inversion(double *, double **);
void dboy_initSound(void);
void dboy_initialize(void);
void dboy_process(void);
void dboy_reset(void);

typedef struct
{
  int intersect;
```

```
   double surf[3];
   double wand_old[6];
   double cpts[8][8][3];
} dboy_File;
```

```
// knot.c

#include "JimsCave.h"

/*      This function returns a uniform knot
vector,
        U. given the order of the B-spline. p. and
        the length of the knot vector. m.  */

void dboy_knot(double *U,int m,int p)
{
        int i;
        for (i=0;i<=p;i++)
        {
                U[i]=0.0;
        }
        for (i=p+1;i<=m-p-1;i++)
        {
                U[i]=((double) (i-p))/((double)
(m-2*p));
        }
        for (i=m-p;i<=m;i++)
        {
                U[i]=1.0;
        }
}
```

```
// load.image.c

#include <stdio.h>
#include <malloc.h>
#include <unistd.h>
#include <stdlib.h>

#include "load.image.h"



void bwToCpack(unsigned short *, unsigned long
*, int);
void rgbToCpack( unsigned short *,  unsigned
short *, unsigned short *,
          unsigned long *,  int);
void rgbaToCpack( unsigned short *, unsigned
short *, unsigned short *,
          unsigned short *, unsigned long *,  int);
void rgbaToCpackCond( unsigned short *,
unsigned short *, unsigned short *,
          unsigned short *, unsigned long *,  int,
          unsigned short,  unsigned short,
unsigned short,  unsigned short);

unsigned long *readLongImageData(char *name,
long *width, long *height)
{
   unsigned long *base, *lptr;
   unsigned short *rbuf, *gbuf, *bbuf, *abuf;
   IMAGE *image;
   int y,  jj;

   image = iopen(name,"r");
   if(!image) {
        return NULL;
   }
   *width = image->xsize;
   *height = image->ysize;
   base = (unsigned long *)malloc(image-
>xsize*image->ysize*sizeof(unsigned long));
   rbuf = (unsigned short *)malloc(image-
>xsize*sizeof(short));
   gbuf = (unsigned short *)malloc(image-
>xsize*sizeof(short));
   bbuf = (unsigned short *)malloc(image-
>xsize*sizeof(short));
   abuf = (unsigned short *)malloc(image-
>xsize*sizeof(short));
   if(!base || !rbuf || !gbuf || !bbuf) {
        fprintf(stderr,"readLongImageData: can't
malloc enough memory\n");
        exit(1);
   }
   lptr = base;
   for(y=0; y<image->ysize; y++) {
        if(image->zsize>=4) {
          if(y==0)
                printf("Doing 4 component
texture : %s \n",  name);
          /* getrow(image,rbuf,y,0);
            getrow(image,gbuf,y,1);
            getrow(image,bbuf,y,2);
            getrow(image,abuf,y,3); */

            getrow(image,abuf,y,0);
            getrow(image,bbuf,y,1);
            getrow(image,gbuf,y,2);
            getrow(image,rbuf,y,3);
rgbaToCpack(rbuf,gbuf,bbuf,abuf,lptr,image-
>xsize);
            lptr += image->xsize;
         } else if(image->zsize==3) {
            if(y==0)
                  printf("Doing 3 component
texture : %s \n", name);
            getrow(image,bbuf,y,0);
            getrow(image,gbuf,y,1);
            getrow(image,rbuf,y,2);
            rgbToCpack(rbuf,gbuf,bbuf,lptr,image-
>xsize);
            lptr += image->xsize;
         } else {
            if(y==0)
                  printf("Doing 1 component
texture : %s \n", name);
            getrow(image,rbuf,y,0);
            bwToCpack(rbuf,lptr,image->xsize);
            lptr += image->xsize;
         }
   }
   iclose(image);
   free(rbuf);
   free(gbuf);
   free(bbuf);
   free(abuf);
   return base;
}

void bwToCpack( unsigned short *b, unsigned
long *l, int n)
{
   while(n>=8) {
         l[0] = 0x00010101*b[0];
         l[1] = 0x00010101*b[1];
         l[2] = 0x00010101*b[2];
         l[3] = 0x00010101*b[3];
         l[4] = 0x00010101*b[4];
         l[5] = 0x00010101*b[5];
         l[6] = 0x00010101*b[6];
         l[7] = 0x00010101*b[7];
         l += 8;
         b += 8;
         n -= 8;
   }
   while(n--)
         *l++ = 0x00010101*(*b++);
}
```

```
void rgbToCpack( unsigned short *r, unsigned
short *g, unsigned short *b,
         unsigned long *l, int n)
{
unsigned short a, aval;

  aval = 255;

  a = aval; /* Object non transparent */

  while(n>=8) {
         l[0] = a | (r[0]<<8) | (g[0]<<16) |
(b[0]<<24);
         l[1] = a | (r[1]<<8) | (g[1]<<16) |
(b[1]<<24);
         l[2] = a | (r[2]<<8) | (g[2]<<16) |
(b[2]<<24);
         l[3] = a | (r[3]<<8) | (g[3]<<16) |
(b[3]<<24);
         l[4] = a | (r[4]<<8) | (g[4]<<16) |
(b[4]<<24);
         l[5] = a | (r[5]<<8) | (g[5]<<16) |
(b[5]<<24);
         l[6] = a | (r[6]<<8) | (g[6]<<16) |
(b[6]<<24);
         l[7] = a | (r[7]<<8) | (g[7]<<16) |
(b[7]<<24);

         /*
         l[0] = r[0] | (g[0]<<8) | (b[0]<<16) | a
<<24;
         a = aval;
         l[1] = r[1] | (g[1]<<8) | (b[1]<<16) | a
<<24;
         a = aval;
         l[2] = r[2] | (g[2]<<8) | (b[2]<<16) | a
<<24;
         a = aval;
         l[3] = r[3] | (g[3]<<8) | (b[3]<<16) | a
<<24;
         a = aval;
         l[4] = r[4] | (g[4]<<8) | (b[4]<<16) | a
<<24;
         a = aval;
         l[5] = r[5] | (g[5]<<8) | (b[5]<<16) | a
<<24;
         a = aval;
         l[6] = r[6] | (g[6]<<8) | (b[6]<<16) | a
<<24;
         a = aval;
         l[7] = r[7] | (g[7]<<8) | (b[7]<<16) | a
<<24;
```

```
        */
        l += 8;
        r += 8;
        g += 8;
        b += 8;
        n -= 8;
  }
  while(n--)
  {
        a = aval;
     *l++ = *r++ | ((*g++)<<8) | ((*b++)<<16) |
a<<24;
        }
}

void rgbaToCpack( unsigned short *r, unsigned
short *g, unsigned short *b,
        unsigned short *a, unsigned long *l, int
n)
{
  while(n>=8) {

#if 0
  r[0] = 0;
  r[1] = 0;
  r[2] = 0;
  r[3] = 0;
  r[4] = 0;
  r[5] = 0;
  r[6] = 0;
  r[7] = 0;

  if(*a == 0&& *g == 0 && *b == 0)
        *r = 0;
#endif


        l[0] = r[0] | (g[0]<<8) | (b[0]<<16) |
(a[0]<<24);
        l[1] = r[1] | (g[1]<<8) | (b[1]<<16) |
(a[1]<<24);
        l[2] = r[2] | (g[2]<<8) | (b[2]<<16) |
(a[2]<<24);
        l[3] = r[3] | (g[3]<<8) | (b[3]<<16) |
(a[3]<<24);
        l[4] = r[4] | (g[4]<<8) | (b[4]<<16) |
(a[4]<<24);
        l[5] = r[5] | (g[5]<<8) | (b[5]<<16) |
(a[5]<<24);
        l[6] = r[6] | (g[6]<<8) | (b[6]<<16) |
(a[6]<<24);
        l[7] = r[7] | (g[7]<<8) | (b[7]<<16) |
(a[7]<<24);
        l += 8;
        r += 8;
        g += 8;
        b += 8;
        a += 8;
        n -= 8;
  }

  while(n--)
  {

     *l++ = *r++ | ((*g++)<<8) | ((*b++)<<16) |
((*a++)<<24);
        }
}




unsigned long *readLongImageDataCond(char
*name, long *width, long *height,
                                unsigned short
rt, unsigned short gt, unsigned short bt,
                                unsigned short
comp)
{
  unsigned long *base, *lptr;
  unsigned short *rbuf, *gbuf, *bbuf, *abuf;
  IMAGE *image;
  int y, jj;

  image = iopen(name,"r");
  if(!image) {
        return NULL;
  }
  *width = image->xsize;
  *height = image->ysize;
  base = (unsigned long *)malloc(image-
>xsize*image->ysize*sizeof(unsigned long));
  rbuf = (unsigned short *)malloc(image-
>xsize*sizeof(short));
  gbuf = (unsigned short *)malloc(image-
>xsize*sizeof(short));
  bbuf = (unsigned short *)malloc(image-
>xsize*sizeof(short));
  abuf = (unsigned short *)malloc(image-
>xsize*sizeof(short));
  if(!base || !rbuf || !gbuf || !bbuf) {
        fprintf(stderr,"readLongImageData: can't
malloc enough memory\n");
        exit(1);
    }
    lptr = base;
    for(y=0; y<image->ysize; y++) {
        if(image->zsize>=4) {
          if(y==0)
                printf("Doing 4 component
texture : %s \n", name);
          /* getrow(image,rbuf,y,0);
          getrow(image,gbuf,y,1);
          getrow(image,bbuf,y,2);
          getrow(image,abuf,y,3); */

          getrow(image,abuf,y,0);
          getrow(image,bbuf,y,1);
          getrow(image,gbuf,y,2);
          getrow(image,rbuf,y,3);

rgbaToCpackCond(rbuf,gbuf,bbuf,abuf,lptr,image-
>xsize, rt, gt, bt, comp);
          lptr += image->xsize;
        } else if(image->zsize==3) {
          if(y==0)
                printf("Doing 3 component
texture : %s \n", name);
          getrow(image,rbuf,y,0);
          getrow(image,gbuf,y,1);
          getrow(image,bbuf,y,2);
          rgbToCpack(rbuf,gbuf,bbuf,lptr,image-
>xsize);
          lptr += image->xsize;
        } else {
          if(y==0)
                printf("Doing 1 component
texture : %s \n", name);
          getrow(image,rbuf,y,0);
          bwToCpack(rbuf,lptr,image->xsize);
          lptr += image->xsize;
        }
    }
    iclose(image);
    free(rbuf);
    free(gbuf);
    free(bbuf);
    free(abuf);
    return base;
}

void rgbaToCpackCond( unsigned short *r,
unsigned short *g, unsigned short *b,
        unsigned short *a, unsigned long *l, int
n,
        unsigned short rt, unsigned short gt,
unsigned short bt, unsigned short comp)
{
  while(n>=8) {

        switch(comp)
        {
        case 0:
          if(a[0]  == rt && g[0] == gt && b[0]
== bt)    r[0] = 0;
          if(a[1]  == rt && g[1] == gt && b[1]
== bt)    r[1] = 0;
          if(a[2]  == rt && g[2] == gt && b[2]
== bt)    r[2] = 0;
          if(a[3]  == rt && g[3] == gt && b[3]
== bt)    r[3] = 0;
          if(a[4]  == rt && g[4] == gt && b[4]
== bt)    r[4] = 0;
          if(a[5]  == rt && g[5] == gt && b[5]
== bt)    r[5] = 0;
          if(a[6]  == rt && g[6] == gt && b[6]
== bt)    r[6] = 0;
          if(a[7]  == rt && g[7] == gt && b[7]
== bt)    r[7] = 0;
          break;
        case 1:
          if(a[0]  <= rt && g[0] <= gt && b[0]
<= bt)    r[0] = 0;
          if(a[1]  <= rt && g[1] <= gt && b[1]
<= bt)    r[1] = 0;
          if(a[2]  <= rt && g[2] <= gt && b[2]
<= bt)    r[2] = 0;
          if(a[3]  <= rt && g[3] <= gt && b[3]
<= bt)    r[3] = 0;
          if(a[4]  <= rt && g[4] <= gt && b[4]
<= bt)    r[4] = 0;
          if(a[5]  <= rt && g[5] <= gt && b[5]
<= bt)    r[5] = 0;
          if(a[6]  <= rt && g[6] <= gt && b[6]
<= bt)    r[6] = 0;
          if(a[7]  <= rt && g[7] <= gt && b[7]
<= bt)    r[7] = 0;
          break;
        case 2:
          if(a[0]  >= rt && g[0] >= gt && b[0]
>= bt)    r[0] = 0;
          if(a[1]  >= rt && g[1] >= gt && b[1]
>= bt)    r[1] = 0;
          if(a[2]  >= rt && g[2] >= gt && b[2]
>= bt)    r[2] = 0;
```

```
            if(a[3]  >= rt && g[3] >= gt && b[3]
>= bt)      r[3] = 0;
            if(a[4]  >= rt && g[4] >= gt && b[4]
>= bt)      r[4] = 0;
            if(a[5]  >= rt && g[5] >= gt && b[5]
>= bt)      r[5] = 0;
            if(a[6]  >= rt && g[6] >= gt && b[6]
>= bt)      r[6] = 0;
            if(a[7]  >= rt && g[7] >= gt && b[7]
>= bt)      r[7] = 0;
            break;
        }

        l[0] = r[0] | (g[0]<<8) | (b[0]<<16) |
(a[0]<<24);
        l[1] = r[1] | (g[1]<<8) | (b[1]<<16) |
(a[1]<<24);
        l[2] = r[2] | (g[2]<<8) | (b[2]<<16) |
(a[2]<<24);
        l[3] = r[3] | (g[3]<<8) | (b[3]<<16) |
(a[3]<<24);
        l[4] = r[4] | (g[4]<<8) | (b[4]<<16) |
(a[4]<<24);
        l[5] = r[5] | (g[5]<<8) | (b[5]<<16) |
(a[5]<<24);
        l[6] = r[6] | (g[6]<<8) | (b[6]<<16) |
(a[6]<<24);
        l[7] = r[7] | (g[7]<<8) | (b[7]<<16) |
(a[7]<<24);
        l += 8;
        r += 8;
        g += 8;
        b += 8;
        a += 8;
        n -= 8;
  }

  while(n--)
  {
        if(*a == 0&& *g == 0 && *b == 0)
          *r = 0;
     *l++ = *r++ | ((*g++)<<8) | ((*b++)<<16) |
((*a++)<<24);
        }
}
```

```
// load.image.h

#ifndef  __GL_IMAGE_H__
#define  __GL_IMAGE_H__

#ifdef __cplusplus
extern "C" {
#endif


/*
 *      Defines for image files . . . .
 *
 *                          Paul Haeberli - 1984
 *      Modified Carolina Cruz-Neira - 1995
 *   Look in
/usr/people/4Dgifts/iristools/imgtools for example
code!
 *
 */

#include <stdio.h>

#define IMAGIC          0732

/* colormap of images */
#define CM_NORMAL               0       /* file
contains rows of values which
                                         * are
either RGB values (zsize == 3)
                                         * or
greyramp values (zsize == 1) */
#define CM_DITHERED             1
#define CM_SCREEN               2       /* file
contains data which is a screen
                                         *
image; getrow returns buffer which
                                         * can
be displayed directly with
                                         *
writepixels */
#define CM_COLORMAP                     3
        /* a colormap file */

#define TYPEMASK                0xff00
#define BPPMASK                         0x00ff
#define ITYPE_VERBATIM
        0x0000
#define ITYPE_RLE               0x0100
#define ISRLE(type)             (((type) &
0xff00) == ITYPE_RLE)
#define ISVERBATIM(type)        (((type) &
0xff00) == ITYPE_VERBATIM)
#define BPP(type)               ((type) &
BPPMASK)
#define RLE(bpp)                (ITYPE_RLE |
(bpp))
```

```
#define VERBATIM(bpp)
        (ITYPE_VERBATIM | (bpp))
#define IBUFSIZE(pixels)
        ((pixels+(pixels>>6))<<2)
#define RLE_NOP                         0x00

#define ierror(p)       (((p)-
>flags&_IOERR)!=0)
#define ifileno(p)              ((p)->file)
#define getpix(p)               (--(p)->cnt>=0
? *(p)->ptr++ : ifilbuf(p))
#define putpix(p,x)             (--(p)->cnt>=0
\
                                ? ((int)(*(p)-
>ptr++=(unsigned)(x))) \
                                :
iflsbuf(p,(unsigned)(x)))

typedef struct {
   unsigned short imagic;       /* stuff saved
on disk . . */
   unsigned short       type;
   unsigned short       dim;
   unsigned short       xsize;
   unsigned short       ysize;
   unsigned short       zsize;
   unsigned long min;
   unsigned long max;
   unsigned long wastebytes;
   char         name[80];
   unsigned long colormap;

   long         file;           /* stuff used in
core only */
   unsigned short       flags;
   short        dorev;
   short        x;
   short        y;
   short        z;
   short        cnt;
   unsigned short *ptr;
   unsigned short *base;
   unsigned short *tmpbuf;
   unsigned long offset;
   unsigned long rleend;        /* for rle
images */
   unsigned long *rowstart;     /* for rle
images */
   long         *rowsize;       /* for rle
images */
} IMAGE;
```

```
IMAGE *icreate();
/*
 * IMAGE *iopen(char *file, char *mode,
unsigned int type, unsigned int dim,
 *                      unsigned int xsize, unsigned int
ysize, unsigned int zsize);
 * IMAGE *fiopen(int f, char *mode, unsigned int
type, unsigned int dim,
 *                      unsigned int xsize, unsigned int
ysize, unsigned int zsize);
 *
 * ...while iopen and fiopen can take an extended
set of parameters, the
 * last five are optional, so a more correct
prototype would be:
 *
 * IMAGE *iopen(char *file, char *mode, ...);
 * IMAGE *fiopen(int f, char *mode, ...);
 *
 * unsigned short *ibufalloc(IMAGE *image);
 * int ifilbuf(IMAGE *image);
 * int iflush(IMAGE *image);
 * unsigned int iflsbuf(IMAGE *image, unsigned
int c);
 * void isetname(IMAGE *image, char *name);
 * void isetcolormap(IMAGE *image, int
colormap);
 * int iclose(IMAGE *image);
 *
 * int putrow(IMAGE *image, unsigned short
*buffer, unsigned int y, unsigned int z);
 * int getrow(IMAGE *image, unsigned short
*buffer, unsigned int y, unsigned int z);
 *
 */

/*IMAGE *iopen(); */
IMAGE *iopen(char *file, char *mode);
IMAGE *icreate();
int iclose(IMAGE *);
unsigned short *ibufalloc();

unsigned long *readLongImageData(char *, long
*, long *);

unsigned long *readLongImageDataCond(char *,
long *, long *,
                                        unsigned short
, unsigned short , unsigned short ,
                                        unsigned short
);
```

```
#define IMAGEDEF                /* for
backwards compatibility */
#ifdef __cplusplus
}
#endif
#endif  /* !__GL_IMAGE_H__ */
```

```
// main.c

#include "JimsCave.h"

/* declare pointer to shared memory arena */
void *dboy_sharedData;

/* declare shared memory pointers */
int *dboy_NCPTS;
int *dboy_P;
int *dboy_npatch;
double *dboy_U, *dboy_V;
double **dboy_M, **dboy_K;
double *dboy_PNEW, *dboy_PCURRENT,
*dboy_POLD, *dboy_PREF;
double **dboy_PCONST;
double **dboy_X, **dboy_Y, **dboy_Z;
double **dboy_X_DU, **dboy_Y_DU;
double **dboy_X_DV, **dboy_Y_DV;
double ***dboy_U_BAS, ***dboy_U_BAS_DER,
***dboy_V_BAS, ***dboy_V_BAS_DER;
double *dboy_WAND;
double *dboy_TRANS;
int *dboy_Playme;

awSound *dboy_sowood;
awSound *dboy_soboing;
awSound *dboy_socreaky;

int dboy_old_int=0;
int dboy_play=0;
int dboy_sound=1;
int dboy_texture=1;

dboy_File *dboy_parameter;

/* main() starts the CAVE tracking, drawing, and
computing */
void main(int argc, char **argv)
{
        CAVEConfigure(&argc, argv, NULL);

        dboy_initialize();

        CAVEInit();

        CAVEInitApplication(dboy_initScene,
0);

        CAVEDisplay( dboy_drawScene, 0);

        CAVEDistribConnect();
        sleep(1);

        while
(!CAVEgetbutton(CAVE_ESCKEY))
        {
          dboy_process();
        }

        if(dboy_sound) awExit();
        CAVEExit();
}
```

```
# makefile

COMPILER =    CC

CAVE_DIR =    -L/home/vr/CAVE/lib/test
CAVE_INC_DIR =  -
I/home/vr/CAVE/include/test

CAVELIBS      = -lcave_ogl -ll -ly -lGLU -llimg

AW2_LIB =     -law -lawhwi -lpsi -laudiofile -lC
AW2_LDIR =    -L/usr/lib/PSI
AW2_INC_DIR =         -I/usr/include/PSI

# These are assorted other libraries that must be
linked with
GLLIBS =   -lGL
XLIBS =              -lX11 -lXt -lXi
OTHERLIBS = -lm -lgutil -limage

LIBS =  $(AW2_LDIR) $(AW2_LIB)
$(CAVE_DIR) $(CAVELIBS) $(XLIBS)
$(GLLIBS) $(OTHERLIBS)

CFLAGS = -32 -mips2 -O $(CAVE_INC_DIR)
$(AW2_INC_DIR) -I. -DSGI

OBJS = main.o util.o compute.o draw.o init_gr.o
init_cp.o knot.o element.o findspan.o bfuns.o
```

```
step.o dsvdcmp.o dpythag.o dsvbksb.o inversion.o
init_so.o minit.o mproc.o reset.o

all: run

# Does linking of the files and libraries
run:  $(OBJS)
        @echo "Currently compiling"
        $(COMPILER) $(CFLAGS) $(OBJS) -o
run $(LIBS)
        @echo "Compiled and linked"

# Compiles the files if necessary
.c.o:
        $(COMPILER) $(CFLAGS) -c $@ $*.c

#clean up code
clean:
        strip run
        rm -f *.o
        @echo "All Neat Now"
```

```
// minit.c

#include "JimsCave.h"

extern int dboy_sound;

void dboy_initialize(void)
{
  dboy_initCompute();
  if(dboy_sound) dboy_initSound();
}
```

```
// mproc.c

#include "JimsCave.h"

extern dboy_File *dboy_parameter;

void dboy_process(void)
{
  if(CAVEDistribMaster())
  {
        dboy_Compute();

        CAVEDistribBarrier();
        CAVEDistribSend(dboy_parameter,
sizeof(dboy_File));
  }
        else
  {
        CAVEDistribBarrier();
        CAVEDistribReceive(dboy_parameter,
sizeof(dboy_File));
  }
  sginap(0);
}
```

```
// reset.c

#include "JimsCave.h"

extern dboy_File *dboy_parameter;
extern double *dboy_PCURRENT, *dboy_POLD,
*dboy_PREF;
extern int *dboy_NCPTS;

extern int *dboy_Playme;

void dboy_reset(void)
{
  int i, j;

  if(CAVEBUTTON3)
  {
/* reset the surface */
        dboy_Playme[0]=1;

        for (i=0;i<dboy_NCPTS[1];i++)
        {
          for (j=0;j<dboy_NCPTS[0];j++)
          {
dboy_POLD[dboy_NCPTS[0]*i+j+1]=0.0;

dboy_PCURRENT[dboy_NCPTS[0]*i+j+1]=0.0;

dboy_PREF[dboy_NCPTS[0]*i+j+1]=0.0;
                        dboy_parameter-
>cpts[j][i][2]=0.0;
          }
        }

/* check to see if the process should be terminated
*/
        if (CAVEBUTTON2)
```

```
          {
            printf("allen\n");
          }
    }
}
```

```
// step.c

#include "JimsCave.h"

extern double *dboy_PREF;
extern int *dboy_NCPTS;

void dboy_step(double *Unew,double
*Uold,double *Ucurrent,double **K,double
**M,double delta,double **pconst,double
**sconst)
{
        double **a,*w,**v,*b,*x;
        double alpha=20.0;
        double beta=10.0;
        double tau=0.1;
        double wmin,wmax;
        int i,j,m;

        m=dboy_NCPTS[0]*dboy_NCPTS[1];

        a=dboy_DoubleMatrix(1,m,1,m);
        w=dboy_DoubleVector(1,m);
        v=dboy_DoubleMatrix(1,m,1,m);
        x=dboy_DoubleVector(1,m);
        b=dboy_DoubleVector(1,m);

        for (i=1;i<=m;i++)
        {
                for (j=1;j<=m;j++)
                {

        a[i][j]=(1.0+alpha*delta)*M[i-1][j-1];

                }
        }

        for (i=1;i<=m;i++)
        {
                b[i]=0.0;
                for (j=1;j<=m;j++)
                {
                /*        b[i] +=
((2.0+alpha*delta)*M[i-1][j-1]-
pow(delta,2.0)*K[i-1][j-1])*Ucurrent[j]-M[i-1][j-
1]*Uold[j];*/
                                b[i] +=
(2.0+alpha*delta)*M[i-1][j-1]*Ucurrent[j]-
pow(delta,2.0)*beta*K[i-1][j-1]*(Ucurrent[j]-
dboy_PREF[j])-M[i-1][j-1]*Uold[j];
                }
        }

/*  Apply primary constraints to the matrix  */
        for (i=1;i<=m;i++)
        {
                if (pconst[i][1] > 1.0)
                {
                        for (j=1;j<=m;j++)
                        {
                                a[i][j]=0.0;
                                if (i==j)
a[i][j]=1.0;

        b[i]=pconst[i][2];
                        }
                }
        }

/*  Apply secondary constraints to the matrix  */
        for (i=1;i<=m;i++)
        {
                if (sconst[i][1] > 1.0)
                {
                        for (j=1;j<=m;j++)
                        {
                                a[i][j]=0.0;
                                if (i==j)
a[i][j]=1.0;

        b[i]=sconst[i][2];
                        }
                }
        }

        for (i=1;i<=m;i++)
        {

        dboy_PREF[i]=(Ucurrent[i]*delta+dboy_
PREF[i]*tau)/(delta+tau);
        }

        dboy_dsvdcmp(a,m,m,w,v);
        wmax=0.0;
        for (j=1;j<=m;j++) if (w[j] > wmax)
wmax=w[j];
```

```
        wmin=wmax*0.000001;
        for (j=1;j<=m;j++) if (w[j] < wmin)
w[j]=0.0;
        dboy_dsvbksb(a,w,v,m,m,b,x);
        for (i=1;i<=m;i++)  Unew[i]=x[i];

        dboy_freeDoubleMatrix(a,1,m,1,m);
        dboy_freeDoubleVector(w,1,m);
        dboy_freeDoubleMatrix(v,1,m,1,m);
        dboy_freeDoubleVector(x,1,m);
        dboy_freeDoubleVector(b,1,m);
}
```

```
// util.c

#include "util.h"

/*  allocate a double number in shared memory  */
double *dboy_sharedDouble(void *sharedMemory)
{
        double *d;

        d=(double *)
amalloc(sizeof(double),sharedMemory);
        return d;
}

/*  allocate a double vector in shared memory  */
double *dboy_sharedDoubleVector(long nl, long
nh, void *sharedMemory)
{
        double *v;

        v=(double *) amalloc((nh-
nl+1+NR_END)*sizeof(double),sharedMemory);
        if (!v)  printf("allocation failure in
sharedDoubleVector\n");
        return v-nl+NR_END;
}

/*  allocate a double matrix in shared memory  */
double **dboy_sharedDoubleMatrix(long nrl, long
nrh, long ncl, long nch, void *sharedMemory)
{
        long i, nrow=nrh-nrl+1, ncol=nch-ncl+1;
        double **m;

        m=(double **)
amalloc((nrow+NR_END)*sizeof(double*),shared
Memory);
        if (!m)  printf("allocation failure 1 in
sharedDoubleMatrix\n");
        m += NR_END;
        m -= nrl;

        m[nrl]=(double *)
amalloc((nrow*ncol+NR_END)*sizeof(double),sh
aredMemory);
        if (!m[nrl])  printf("allocation failure 2 in
sharedDoubleMatrix\n");
        m[nrl] += NR_END;
        m[nrl] -= ncl;

        for(i=nrl+1;i<=nrh;i++) m[i]=m[i-
1]+ncol;

        return m;
}

/*  allocate a double 3D tensor in shared memory
*/
double ***dboy_sharedDouble3Tensor(long nrl,
long nrh, long ncl, long nch, long ndl, long ndh,
void *sharedMemory)
{
        long i, j, nrow=nrh-nrl+1, ncol=nch-
ncl+1, ndep=ndh-ndl+1;
        double ***t;

        t=(double ***)
amalloc((nrow+NR_END)*sizeof(double**),share
dMemory);
        if (!t)  printf("allocation failure 1 in
sharedDouble3Tensor\n");
        t += NR_END;
        t -= nrl;

        t[nrl]=(double **)
amalloc((nrow*ncol+NR_END)*sizeof(double*),s
haredMemory);
        if (!t[nrl])  printf("allocation failure 2 in
sharedDouble3Tensor\n");
        t[nrl] += NR_END;
        t[nrl] -= ncl;

        t[nrl][ncl]=(double *)
amalloc((nrow*ncol*ndep+NR_END)*sizeof(doub
le),sharedMemory);
        if (!t[nrl][ncl])  printf("allocation failure
3 in sharedDouble3Tensor\n");
        t[nrl][ncl] += NR_END;
        t[nrl][ncl] -= ndl;
        for(i=ncl+1;i<=nch;i++)
t[nrl][i]=t[nrl][i-1]+ndep;
        for(i=nrl+1;i<=nrh;i++)
        {
                t[i]=t[i-1]+ncol;
                t[i][ncl]=t[i-1][ncl]+ncol*ndep;
                for(j=ncl+1;j<=nch;j++)
t[i][j]=t[i][j-1]+ndep;
        }

        return t;
}

/*  allocate a float vector in shared memory  */
float *dboy_sharedFloatVector(long nl, long nh,
void *sharedMemory)
{
        float *v;

        v=(float *) amalloc((nh-
nl+1+NR_END)*sizeof(float),sharedMemory);
        if (!v)  printf("allocation failure in
sharedFloatVector\n");
        return v-nl+NR_END;
}

/*  allocate an integer vector in shared memory  */
int *dboy_sharedIntVector(long nl, long nh, void
*sharedMemory)
{
        int *v;

        v=(int *) amalloc((nh-
nl+1+NR_END)*sizeof(int),sharedMemory);
        if (!v)  printf("allocation failure in
sharedIntVector\n");
        return v-nl+NR_END;
}



/*  allocate a double vector  */
double *dboy_DoubleVector(long nl, long nh)
{
        double *v;

        v=(double *) malloc((nh-
nl+1+NR_END)*sizeof(double));
        if (!v)  printf("allocation failure in
DoubleVector\n");
        return v-nl+NR_END;
}

/*  allocate a double matrix  */
double **dboy_DoubleMatrix(long nrl, long nrh,
long ncl, long nch)
{
        long i, nrow=nrh-nrl+1, ncol=nch-ncl+1;
        double **m;

        m=(double **)
malloc((nrow+NR_END)*sizeof(double*));
        if (!m)  printf("allocation failure 1 in
DoubleMatrix\n");
        m += NR_END;
        m -= nrl;

        m[nrl]=(double *)
malloc((nrow*ncol+NR_END)*sizeof(double));
        if (!m[nrl])  printf("allocation failure 2 in
DoubleMatrix\n");
        m[nrl] += NR_END;
        m[nrl] -= ncl;

        for(i=nrl+1;i<=nrh;i++) m[i]=m[i-
1]+ncol;

        return m;
}



/*  free a double vector  */
void dboy_freeDoubleVector(double *v, long nl,
long nh)
{
        free(v+nl-NR_END);
        nh=nh;
}

/*  free a double vector  */
void dboy_freeDoubleMatrix(double **m, long
nrl, long nrh, long ncl, long nch)
{
        free(m[nrl]+ncl-NR_END);
        free(m+nrl-NR_END);
        nrh=nrh;
        nch=nch;
}
```

```
// util.h

#include <sys/types.h>
#include <malloc.h>

#define NR_END 1

static double dsqrarg;
#define dboy_DSQR(a) ((dsqrarg=(a)) == 0.0 ? 0.0
: dsqrarg*dsqrarg)

static double dmaxarg1,dmaxarg2;
#define dboy_DMAX(a,b)
(dmaxarg1=(a),dmaxarg2=(b),(dmaxarg1) >
(dmaxarg2) ? (dmaxarg1) : (dmaxarg2))

static int iminarg1,iminarg2;
#define dboy_IMIN(a,b)
(iminarg1=(a),iminarg2=(b),(iminarg1) <
(iminarg2) ? (iminarg1) : (iminarg2))

#define dboy_SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -
fabs(a))



/* memory allocation utilities - prototypes */
double *dboy_sharedDouble(void *);
double *dboy_sharedDoubleVector(long, long,
void *);
double **dboy_sharedDoubleMatrix(long, long,
long, long, void *);
double ***dboy_sharedDouble3Tensor(long, long,
long, long, long, long, void *);
float *dboy_sharedFloatVector(long, long, void *);
int *dboy_sharedIntVector(long, long, void *);



double *dboy_DoubleVector(long, long);
double **dboy_DoubleMatrix(long, long, long,
long);



void dboy_freeDoubleVector(double *, long, long);
void dboy_freeDoubleMatrix(double **, long,
long, long, long);
```

REFERENCES

1. Adachi, Y., T. Kumano, and K. Ogino, "Sensory evaluation of virtual haptic push-buttons," *ASME Dynamic Systems and Control*, DSC-Vol. 55-1, pp. 361-368, Chicago, IL, 1994.
2. Anand, V.B., *Computer Graphics and Geometric Modeling for Engineers*, John Wiley & Sons, New York, NY.
3. Anderson, R.J. and M.W. Spong, "Asymptotic stability for force reflecting teleoperators with time delay," *International Journal of Robotics Research*, **11**(2): pp. 135-149, 1992.
4. Asimov, I., *I, Robot*, Double Day, Garden City, NJ, 1950.
5. Astley, O. and V. Hayward, "An experimental procedure for autonomous joint sensor estimation using adaptive control," *Proceedings of IEEE International Conference on Robotics and Automation*, Albuquerque, NM, 1997.
6. Aulbach, B., "Continuous and discrete dynamics new manifolds of equilbria," *Lecture Notes in Mathematics*, **1058**: pp. 1-142, 1984.

7. Baraff, D., "Analytical methods for dynamic simulations of non-penetrating rigid bodies," *Proceedings of SIGGRAPH '89*, **23**(3): pp. 223-232, July, 1989.

8. Barr, A.H., "Global and local deformations of solid primitives," *Proceedings of SIGGRAPH '84*, **18**(3): pp. 21-30, July, 1984.

9. Bergamasco, M., D.M. De Micheli, G. Parrini, F. Salsedo, S. Marchese, "Design considerations for glove-like advanced interfaces," *Proceedings of the Fifth International Conference on Advanced Robotics*, Pisa, Italy, 1991.

10. Bonitz, R.G. and T.C. Hsia, "Force decomposition in cooperating manipulators using the theory of metric spaces and generalized inverses," *Proceedings of IEEE International Conference on Robotics and Automation*, San Diego, CA, 1994.

11. Brickman, W.B., *A First Course in the Finite Element Method*, Irwin, Homewood, IL, 1990.

12. Brooks, F.P., "Walkthrough: A dynamic graphics system for simulating virtual buildings," *SIGGRAPH Workshop on 3D Graphics*, 1986.

13. Burdea, G., J. Zhaung, E. Roskos, D. Silver, N. Langrana, "A portable dexterous master with force feedback," *Presence-Teleoperators and Virtual Environments*, **1**(1): pp. 18-29, 1992.

14. Burdea, G. and P. Coiffet, *Virtual reality technology*, John Wiley & Sons, New York, NY, 1994.

15. Buttolo, P., B. Bratthen, and B. Hannaford, "Sliding control of a force reflecting teleoperation: Preliminary studies," *Presence*, **3**(2): pp. 158-172, 1994.

16. Buttolo, P., D. Kung, and B. Hannaford, "Manipulation in real, virtual and remote environments," *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, Vancouver, BC, Can, 1995.

17. Celniker, G. and D. Gossard, "Deformable curve and surface finite-element for free-form shape design," *Proceedings of SIGGRAPH '91*, **25**(4): pp. 257-266, July, 1991.

18. Chadwick, J.E., D.R. Haumann, and R.E. Parent, "Layered construction for deformable animated characters," *Proceedings of SIGGRAPH '89*, **23**(3): pp. 243-252, July, 1989.

19. Clover, C.L., "Control system design for robots used in simulating dynamic force and moment interaction in virtual reality applications," Ph.D. Thesis, Department of Mechanical Engineering, Iowa State University, 1996.

20. Codella, C., L. Koved, J.B. Lewis, "Interactive simulation in a multi-person virtual world," *Proceedings of Human Factors in Computing Systems*, Monterey, CA, 1992.

21. Colgate, J.E., P.E. Grafing, M.C. Stanley, G. Schenkel, "Implementation of stiff virtual walls in force-reflecting interfaces," *Proceedings of IEEE Annual Virtual Reality International Symposium*, Seattle, WA, 1993.

22. Colgate, J.E. and J.M. Brown, "Factors affecting the Z-width of a haptic display," *Proceedings of IEEE International Conference on Robotics and Automation*, San Diego, CA, 1994.

23. Corke, P.I. and B. Armstrong-Helouvry, "Search for consensus among model parameters reported for the PUMA 560 robot," *Proceedings of IEEE International Conference on Robotics and Automation*, San Diego, CA, 1994.

24. Craig, J.J., *Introduction to Robotics, Mechanics and Control*, Addison-Wesley, Reading, MA, 1989.

25. Cruz-Neira, C., D.J. Sandin, and T.A. DeFanti, "Surround-screen projection-based virtual reality: the design and implementation of the CAVE," *Proceedings of SIGGRAPH '93*, Anaheim, CA, 1993.

26. Dede, C.J., M. Salzman, and R.B. Loftin, "Development of a virtual world for learning Newtonian mechanics," *Proceedings of International Conference on Multimedia, Hypermedia, and Virtual Reality*, Moscow, Russia, 1994.

27. Deering, M., "High resolution virtual reality," *Proceedings of SIGGRAPH '92*, Chicago, IL, 1992.

28. Doty, K.L., C. Melchiorri, and C. Bonivento, "A theory of generalized inverses applied to robotics" *International Journal of Robotics Research*, **12**(1): pp. 1-18, 1993.

29. Duffy, J., M. Griffis, and M. Swinson, "Fallacy of modern hybrid controll theory for the simultaneous control of force and motion," *Symposium on Advances in Robot Kinematics 2nd Workshop on Advances in Robot Kinematics*, 1990.

30. Edwards, J.C. and G.R. Luecke, "Physically based dynamic models for use in a force feedback virtual environment," *Proceedings of Twenty-Fourth Midwestern Mechanics Conference*, Ames, IA, 1995.

31. Edwards, J.C. and G.R. Luecke, "Physically based models for use in a force feedback virtual environment," *Proceedings of Japan-USA Symposium on Flexible Automation*, Boston, MA, 1996.

32. Fuchs, H.M., M. Levoy, and S.M. Pizer, "Interactive visualization of 3-D medical data," *IEEE Computer*, pp. 46-50, 1989.

33. Galyean, T. and J.F. Hughes, "Sculpting: An interactive volumetric modeling technique," *Proceedings of SIGGRAPH '91*, **25**(4): pp. 267-274, July, 1991.

34. Gardner, L.R.T., G.A. Gardner, S.I. Zaki, Z. El Sahrawi, "B-spline finite element studies of the non-linear Schroedinger equation," *Computer Methods in Applied Mechanics and Engineering*, **108**(3-4): pp. 303-318, 1993.

35. Gardner, L.R.T., G.A. Gardner, F.A. Ayoub, N.K. Amein, "Modeling an undular bore with B-splines," *Computer Methods in Applied Mechanics and Engineering*, **147**(1-2): pp. 147-152, 1997.

36. Gillespie, B. and M. Cutkosky, "Interactive dynamics with haptic display," *Proceedings of ASME Winter Annual Meeting*, New Orleans, LA, 1993.

37. Gourret, J.-P., N.M. Thalmann, and D. Thalmann, "Simulation of object and human skin deformation in a grasp task," *Proceedings of SIGGRAPH '89*, **23**(3): pp. 21-30, July, 1989.

38. Gudukbay, U. and B. Ozguc, "Free-form solid modeling using deformations," *Computers and Graphics*, **14**(3): pp. 491-500, 1990.

39. "Handykey Corporation," http://www.handykey.com, (December 10, 1997).

40. Hannaford, B., L. Wood, D.A. McAffee, H. Zak, "Performance evaluation of a six-axis generalized force-reflecting teleoperator," *IEEE Transactions on Systems, Man and Cybernetics*, **21**(3): pp. 620-633, 1991.

41. Hatada, T., "Psychological and physiological analysis of stereo-scopic vision," *Journal of Robotics and Mechatronics*, **4**(1): p. 13-19, 1992.

42. Hogan, N., "Impedance control: An approach to manipulation," *Journal of Dynamic Systems, Measurement and Control*, **107**(1): pp. 1-7, 1985.

43. Hogan, N., "Controlling impedance at the man/machine interface," *Proceedings of IEEE International Conference on Robotics and Automation*, Scottsdale, AZ, 1989.

44. Horn, R.A. and C.R. Johnson, *Matrix Analysis*, Cambridge University Press, New York, NY, 1985.

45. Hsu, W.M., J.F. Hughes, and H. Kaufman, "Direct manipulation of free-form deformation," *Proceedings of SIGGRAPH '92*, Chicago, IL, 1992.

46. "I - Force," iforce.html, http:www.force-feedback.com/iforce, (December 10, 1997).

47. "IMMERSION CORP: Impulse Engine 2000," research.html, http://www.force-feedback.com/research, (December 10, 1997).

48. Inman, D.J., *Engineering Vibration*, Prentice Hall, Englewood Cliffs, NJ, 1994.

49. Ishii, M. and M. Sato, "A 3D interface device with force feedback: A virtual work space for pick-and-place tasks," *Proceedings of IEEE Virtual Reality Annual International Symposium*, 1993.

50. Iwata, H., "Artificial reality with force-feedback. Development of desktop virtual space with compact master manipulator," *Proceedings of SIGGRAPH '90*, **24**(4): pp. 165-170, August, 1990.

51. Joly, L.D. and C. Andriot, "Imposing motion constraints to a force reflecting telerobot through real-time simulation of a virtual mechanism," *Proceedings of IEEE International Conference on Robotics and Automation*, Nagoya, Japan, 1995.

52. Joly, L.D., C. Andriot, and V. Hayward, "Mechanical analogies in hybrid position/force control," *Proceedings of IEEE International Conference on Robotics and Automation*, Albuquerque, NM, 1997.

53. Kaneko, K., H. Tokashiki, K. Tanie, K. Komoriya, "Impedance shaping based on force feedback bilaterial control in macro-mirco teleoperation system," *Proceedings of IEEE International Conference on Robotics and Automation*, Albuquerque, NM, 1997.

54. Kapusinski, C.L., "Motor selection and damper design for a six degree of freedom haptic display," M.S. Thesis, Department of Mechanical Engineering, North Western University, 1997.

55. Kazerooni, H. and M.-G. Her, "Dynamics and control of a haptic interface device," *IEEE Transactions on Robotics and Automation*, **10**(4): pp. 453-464, 1994.

56. Kelley, A.J. and S.E. Salcudean, "On the development of a force-feedback mouse and its integration into a graphical user interface," *Proceedings of International Mechanical Engineering Congress and Exposition*, Chicago, IL, 1994.

57. Khalil, H.K., *Nonlinear Systems*, Prentice Hall, Upper Saddle River, NJ.

58. Khatib, O., "Operational space framework," *JSME International Journal*, **36**(3): pp. 277-287, 1993.

59. Lawrence, D.A., L.Y. Pao, M.A. Salada, A.M. Dougherty, "Quantitative experimental analysis of transparency and stability in haptic interfaces," *Proceedings of ASME International Mechanical Engineering Congress and Exposition*, Atlanta, GA, 1996.

60. Luecke, G.R. and J. Winkler, "Magnetic interface for robot-supplied virtual forces," *Proceedings of International Mechanical Engineering Congress and Exposition*, Chicago, IL, 1994.

61. Luecke, G.R. and J.C. Edwards, "Virtual cooperating manipulators as a virtual reality haptic interface," *Proceedings of 3rd Annual Symposium on Human Interaction with Complex Systems*, Dayton, OH, 1996.

62. Luecke, G.R. and J.C. Edwards, "Force interactive virtual reality using local joint error control," *Proceedings of International Symposium on Intelligent Systems and Advanced Manufacturing*, Boston, MA, 1996.

63. Luecke, G.R., Y.H. Chai, and J.C. Edwards, "An exoskeleton manipulator for application of electromagnetic virtual forces," *Proceedings of ASME Dynamic Systems and Control Division*, Atlanta, GA, 1996.

64. Luecke, G.R., J.C. Edwards, and B.E. Miller, "Virtual cooperating manipulator control for haptic interaction with NURBS surfaces," *Proceedings of IEEE International Conference on Robotics and Automation*, Albuquerque, NM, 1997.

65. Luecke, G.R., Y.-H. Chai, and J.C. Edwards, "Force interactions in the synthetic environment using the ISU force reflecting exoskeleton," *Computer & Graphics*, **21**(4): pp. 431-442, 1997.

66. Mark, W.R., S.C. Randolph, M. Finch, J.M. Van Verth, R.M. Taylor, "Adding force feedback to graphics systems: Issues and solutions," *Proceedings of SIGGRAPH '96*, 1996: pp. 447-452, August.

67. Massie, T.H., "Design of a three degree of freedom force-reflecting haptic interface," B.S. Thesis, Department of Mechanical Engineering, Massachusetts Institute of Technology, 1993.

68. Massie, T.M. and J.K. Salisbury, "The PHANToM haptic interface: A device for probing virtual objects," *Proceedings of ASME Winter Annual Meeting*, 1994.

69. McDowall, I.E., M. Bolas, S. Pieper, S.S. Fisher, J. Humphries, "Implementation and integration of a counterbalanced CRT-based stereoscopic display for interactive viewpoint control in virtual environment applications," *Proceedings of SPIE*, 1990.

70. Metaxas, D. and D. Terzopoulos, "Dynamic deformation of solid primitives with constraints," *Proceedings of SIGGRAPH '92*, **26**(2) , July, 1992.

71. Micaelli, A., C. Bidard, and C. Andriot, "Decoupling control based on virtual mechanisms for telemanipulation," *Proceedings of IEEE International Conference on Robotics and Automation*, 1997.

72. "Microsoft Direct X DirectInput," default.asp, http://www.microsoft.com/directx/pavilion/dinput, (December 10, 1997).

73. Millman, P.A. and J.E. Colgate, "Design of a four degree-of-freedom force reflecting manipulandum with a special force/torque workspace," *Proceedings of IEEE International Conference on Robotics and Automation*, Sacramento, CA, 1991.

74. Millman, P.A., M. Stanley, and J.E. Colgate, "Design of a high performance interface to virtual environments," *Proceedings of IEEE Annual Virtual Reality International Symposium*, Seattle, WA, 1993.

75. Mitsuishi, M., T. Watanabe, H. Nakanishi, T. Hori, H. Watanabe, B. Kramer, "Tele-micro-surgery system across the internet with a fixed viewpoint/operation-point," *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, Pittsburgh, PA, USA, 1995.

76. Mizusawa, T. and T. Kato, "Application of the spline prism method to analyze vibration of thick circular cylindrical panels," *International Journal of Solids and Structures*, **33**(7): pp. 967-976, 1996.

77. Morizono, T., K. Kurahashi, and S. Kawamura, "Analysis and control of a force display system driven by parallel wire mechanism," *Japan-USA Symposium on Flexible Automation*, Boston, MA, 1996.

78. Orr, J.N., "Exotic CAD," *Computer Graphics World*, **12**(7): pp. 88-89, 1989.

79. Ouh-Young, M., M. Pique, J. Hughes, N. Srinivasan, F.P. Brooks, "Using a manipulator for force display in molecular docking," *Proceedings of IEEE International Conference on Robotics and Automation*, Philadelphia, PA, 1988.

80. "PC Gear - Force FX," CH Products, The new force FX, pcgear.html, http://www.chproducts.com, (December 10, 1997).

81. Pentland, A. and J. Williams, "Good Vibrations: Modal dynamics for graphics and animation," *Proceedings of SIGGRAPH '89*, **23**(3): pp. 215-222, July, 1989.

82. Piegl, L. and W. Tiller, *The NURBS Book*, Springer, New York, NY, 1995.

83. Press, W.H., *Numerical recipes in C*, Cambridge University Press, Cambridge, MA, 1992.

84. "Product Overview: VTI CyberGrasp," prod_cybergrasp.html, http://www.virtex.com, (December 10, 1997).

85. Qin, H. and D. Terzopoulos, "Dynamic NURBS swung surfaces for physics-based shape design," *Computer Aided Design*, **27**(2): pp. 111-127, 1995.

86. Qin, H. and D. Terzopoulos, "D-NURBS: A physics-based framework for geometric design," *IEEE Transactions on Visualization and Computer Graphics*, **2**(1): pp. 85-96, 1996.

87. Qin, H. and D. Terzopoulos, "Triangular NURBS and their dynamic generalizations," *Computer Aided Geometric Design*, **14**(4): pp. 325-347, 1997.

88. Raibert, M.H. and J.J. Craig, "Hybrid position/force control of manipulators," *Journal of Dynamic Systems, Measurement and Control*, **103**(2): pp. 126-133, 1981.

89. Sciavicco, L. and B. Siciliano, *Modeling and Control of Robot Manipulators*, McGraw-Hill, New York, NY, 1996.

90. Sederberg, T.W. and S.R. Parry, "Free-form deformation of solid geometric models," *Proceedings of SIGGRAPH '86*, **20**(4): pp. 151-160, August, 1986.

91. "SensAble Technologies: Community, in The SensAble Community," community.html, http://www.sensable.com, (December 10, 1997).

92. "SensAble Technologies: Product - Info," products.html#ghost, http://www.sensable.com, (December 10, 1997).

93. "SensAble Technologies: Products - Info," products.html#phantom, http://www.sensable.com, (December 10, 1997).

94. "Side Winder Force Feedback Pro - Overview," 488_ov.html, http://www.microsoft.com/products/prodnet, (December 10, 1997).

95. Song, G.J. and N.P. Reddy, "Towards surgical simulation of cutting in the VR environment," *Proceedings of the ASME Bioengineering Division ASME International Mechanical Engineering Congress and Exposition*, Chicago, IL, USA, 1994.

96. Stewart, P., P. Buttolo, and Y. Chen, "CAD data representations for haptic virtual prototyping," *Proceedings of ASME Design Engineering Technical Conferences*, Sacramento, CA, 1997.

97. Swokowski, E.W., *Calculus with analytic geometry*, PWS Kent Publishing Company, Boston, MA, 1988.

98. Tarn, T.J., S.H. Bejczy, and X. Yun, "Inertia parameters of Puma 560 robot arm, Washington University, St. Louis, MO, 1985.

99. Teitel, M.A., "Eyephone, a head mounted stereo display," *Proceedings of International Society for Optical Engineering Stereoscopic Displays and Applications*, Santa Clara, CA, 1990.

100. Terzopoulos, D. and K. Fleischer, "Modeling inelastic deformations: Viscoelasticity, plasticity, fracture," *Proceedings of SIGGRAPH '88*, **22**(4): pp. 269-278, August, 1988.

101. Terzopoulos, D., "Elastically deformable models," *Proceedings of SIGGRAPH '87*, **21**(4): pp. 205-214, July, 1987.

102. Thompson, T.V.I., D.E. Johnson, and E. Cohen, "Direct haptic rendering of sculptured models," *Symposium on Interactive 3D Graphics*, Providence, RI, 1997.

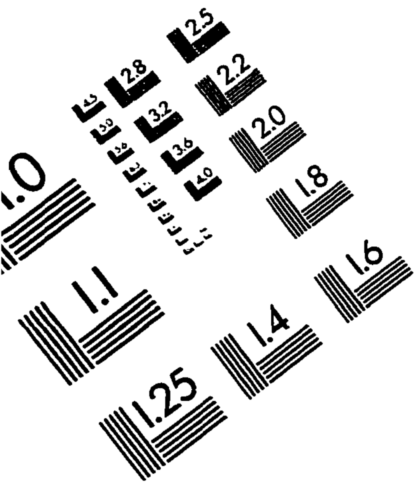
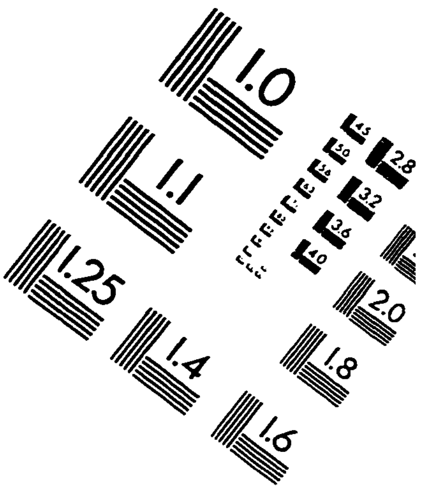
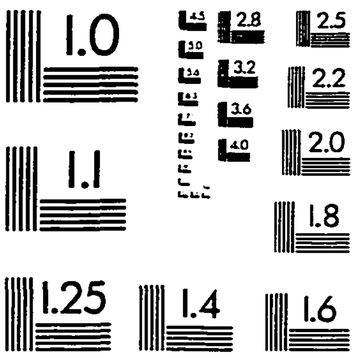
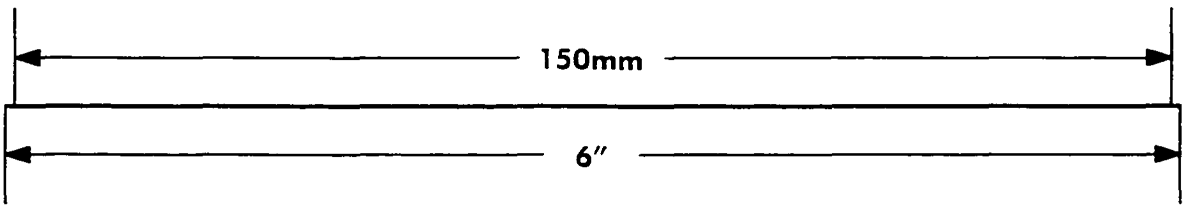
103. Van de Vegte, J.M.E., P. Milgram, and R.H. Kwong, "Teleoperator control models: Effects of time delay and imperfect system knowledge," *IEEE Transactions on Systems, Man and Cybernetics*, **20**(6): pp. 1258-1272, 1990.

104. Viswanadham, K.N.S.K. and S.R. Koneru, "Finite element method for one-dimensional and two-dimensional time dependent problems with B-splines," *Computer Methods in Applied Mechanics and Engineering*, **108**(3-4): pp. 201-222, 1993.

105. Yokoi, H., "Development of the virtual shape manipulating system," *Proceedings of Fourth International Conference on Artificial Reality and Tele-existence*, 1994.

106. Yokokohji, Y., R.L. Hollis, and T. Kanade, "Vision-based visual/haptic registration for WYSIWYF display," *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, Osaka, Japan, 1996.

107. Zimmerman, T.G., J. Lanier, C. Blanchard, S. Bryson, Y. Harvil, "Hand gesture interface device," *Human Factors in Computing Systems and Graphics Interfaces*, Toronto, Ont, Can, 1987.





150mm
6"