Simulation of Strategies to Minimize the Force of Impact Between a Redundant Manipulator and an Obstacle

Matthew Wayne Gertz

Submitted in partial fulfillment of the requirements
for the degree of Master of Science

Advanced Manipulators Laboratory
The Robotics Institute
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

February 22, 1991

This research was funded in part by NASA (grant number NAG-1-1075), the Department of Electrical and Computer Engineering at Carnegie Mellon University, and The Robotics Institute at Carnegie Mellon University.

The author gratefully acknowledges the following people for their assistance and advice during this study: Mr. John Lerchey at the Education Software Library, Carnegie Mellon University, for his helping set up the machine, software, and for tutoring the author in an alien operating system (VMS); Mr. William Doggett at NASA LaRC, for all of his assistance with ROBSIM, and who handled the author's numerous questions about the software with great patience; Mr. Jin-Oh Kim at the Robotics Institute, Carnegie Mellon University, who introduced the author to the problems of studying impact phenomena; and Dr. Pradeep K. Khosla, the author's advisor at The Robotics Institute who kept him on track.

Matthew Wayne Gertz
e-mail: gertz@bilge.ece.cmu.edu
Advanced Manipulators Laboratory
The Robotics Institute
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Table of Contents

List of Figures

List of Tables

Introduction

There has been much research in the past decade on the use of redundant manipulators. Redundant manipulators are robots which have more degrees-of-freedom than are required for their task. For instance, a robot with three joints (the rotation axes of which are parallel, so that the workspace of the robot is planar), is considered redundant because only two joints are required to reach any point in a planar workspace[6]. Some of the areas where redundant manipulators are particularily useful include collision avoidance, singularity avoidance, joint limit avoidance, and in general creating more options for task solutions than are possible with non-redundant manipulators [5]. Another area which has recently come to the fore is the use of redundancy to minimize the force of impact, which is the topic of this thesis. In broad terms, we wish to reduce the force of impact in order to reduce wear and tear on expensive and hard-to-replace equipment.

Much of the research done on reducing the force of impact has concentrated on the configuration of a manipulator as it affects the effective mass [1,3,4,10] or the joint velocity discontinuities [2] during an impact which is considered to be instantaneous. We present two theories which instead deal with impacts occuring in some finite duration of time. The first uses a damping torque to decrease the velocity of an end-effector into a surface (which in turn reduces the momentum and impact force). The second uses configuration control to reduce penetration, but takes into account the finite effects of impact. The damping torque model is simulated for a three degrees-of-freedom manipulator. The configuration-based model is simulated for both a three degrees-of-freedom planar manipulator and a seven degrees-of-freedom manipulator.

Because simulation is an efficient way to test control strategies, it is another growing area of study in the field of robotics. Simulators are useful in that they allow the user to test theories and tasks quickly and thoroughly before trying them out on expensive hardware. Furthermore, simulators usually allow system design, so that the user is not confined to only one or two manipulators to prove a theory. An accurate simulator can run many more useful tests than can a real manipulator, with no chance of accidents. In this thesis, the theories presented will be tested by use of the NASA Langley Robotics Simulation Program (ROBSIM) [14].

ROBSIM has changed much since its creation in 1982. Currently it is being upgraded by NASA Langley Research Center to conform with present conventions and to introduce new code, but especially to make it user-friendly. Upon our receipt of the source code in January 1990, ROBSIM was already in a state where a user could add his or her own code, but there was no literature on how this could be done, and programming could not be done easily in C, but only in FORTRAN. Since we began working with ROBSIM, necessity has caused us to create a C-interface for ROBSIM, in order that routines already existing in program libraries could be used on ROBSIM (with minor modifications), and also to create a preprocessor token programming language in C, to alleviate confusion due to cryptic names of variables. Furthermore, we have written literature on how user code may be added to ROBSIM, much of which will be included in future user manuals for ROBSIM. We have also helped modify ROBSIM's core code.

In Chapter 1 of this thesis, we discuss redundancy and introduce the two impact models. Additionally, we introduce the Premultiplier Diagram [7,8], which we used to develop the theory behind the models. We also provide results from the application of both model to manipulators simulated using ROBSIM. In Chapter 2, we discuss simulation, the creation of a C-interface and a token programming language (RPTL), how the models were simulated, and what the results mean. Finally, in Chapter 3, we summarize our results and discuss possible future research. The software generated for the tests are listed in Appendix A, and the C-interface and programming language are listed, along with an explanation of their use, in Appendix B.

1. Two Strategies for Minimizing the Force of Impact

1.1. Introduction

In most applications of robotics, the manipulator must invariably come into contact with the environment. In certain situations, how the end-effector contacts the environment is of great importance. For instance, if a robot which must apply a constant force to an object bounces upon contact with its surface, perhaps due to the stiffness of the object, then its effectiveness is greatly reduced. Similarly, if a robot on an automated space station damages its end-effector due to a high impact force when it intercepts some target object, it will be expensive to repair, not only due to the cost of the (perhaps specialized) end-effector, but also due to the resources needed to get to the robot and replace its end-effector. In both of these cases, a reduction of the impact force minimizes the danger of task failure.

There are many examples of how the minimization of the impact force increases the usefulness of a system. For instance, most screen doors have a dashpot damper attached to them, to keep the door from slamming. Screen doors without dashpots oscillate (rather noticeably) when the door is closed. By tuning the dashpot correctly, the component of angular velocity that moves the door into its frame is reduced. The door does not strike the frame with as much force as it would otherwise, and thus there is less rebound from the door frame. In another instance, consider a child bouncing on a trampoline which, for the purposes of argument, we will assume functions like an ideal spring. When she wishes to bounce higher, she pushes away from the trampoline. This force applied at the point of contact causes the surface of the trampoline to oscillate, and her own momentum, added to that of the surface of the trampoline, propels her into the air. When she wishes to get off the trampoline, she will start to bend her legs as they touch the canvas. The surface of the trampoline is not displaced as much, and she will not bounce as high, because the force incurred from impact has been reduced, thus reducing her momentum into the air.

Much of the work in the past on impact strategies have focused on the instantaneous effects of collision, where the duration of the impact occurs in an infinitesimally small period of time. Walker [1] takes advantage of kinematic redundancy to reduce the instantaneous force of impact by minimizing the effective mass of a manipulator along the direction normal to the surface with which the end-effector is colliding. Zheng and Hemami [2] explore the relationship between abrupt changes in generalized joint rate velocities and the intensity of collision during an instantaneous impact event, and also discuss how a collision affects the joint constraints of a manipulator. Cai [3] provides a method for determining the impulse of a perfectly elastic collision which also has duration much smaller than the controller's sampling period. Previously, however, very little research has been done where the duration of the collision is finite rather than instantaneous. Khatib and Burdick [4] offer a control method which models the impact event as a mass/spring oscillator, and adds pure damping to dissipate impact energy, for a non-redundant manipulator. This thesis introduces strategies for minimizing the the force of impact for redundant manipulators in which finite duration of impact is taken into consideration.

In this chapter, we shall introduce two methods which use some of the above observations to minimize the finite-duration impact force for a redundant manipulator. Section 1.2 discusses redundancy and its importance to minimizing the impact force. Section 1.3 briefly reviews the *Premultiplier Diagram*, which will be an important tool for creating the strategies presented in this paper. Section 1.4 offers a method for minimizing the impact force by introducing stiffness and damping torques to suppress motion into the surface of an object. Section 1.5 offers a method for reducing the impact force by choosing a configuration which maximizes the effective damping while minimizing the effective mass of the manipulator. In Section 1.6 we conclude this chapter.

1.2. The Importance of Redundancy in Reducing Impact Force

Before the strategies to reduce the impact force for collisions involving redundant manipulators are given, it is necessary to explain what redundant manipulators are, and how their properties allow us to reduce impact force in a way which is not possible by using nonredundant manipulators. Redundant manipulators are those robots which have more degrees of freedom than their task requires. These extra degrees of freedom allow the manipulator to reach a certain point in task space via an infinite set of configurations in joint space, instead of being limited to a finite number of configurations per task point. This characteristic of redundant manipulators makes them very useful in many applications in the field of robotics. Yoshikawa points out in [5] that redundant manipulators are useful in that they can achieve configurations which avoid singularities, motion limits on joints, and collisions with obstacles. A comprehensive review of the applications of redundancy is given in [6]. In this thesis, we will use redundancy to minimize the force resulting from an impact event of finite duration. The redundant manipulators we will use are a simulated three degrees-of-freedom planar manipulator and a simulated seven degrees-of-freedom manipulator. Their descriptions are given in Table 1.1 and 1.2, respectively. Although redundant manipulators have been used in the past to minimize the force of impact [1,4], it has always been with the assumption that the impact event is infinitesimally brief, and until now research has not been done which assumes a finite duration for the impact event.

In the example with the trampoline and the child, the system describing the contact of the child's feet with the canvas can be regarded, in a loose sense, as being kinematically redundant. The child can "configure" herself in an infinite number of body positions, and yet still remain in contact with some target (the trampoline), similar to the way that a redundant manipulator can reach a point through an infinite number of configurations. By changing her configuration, she can decrease her force of impact as described in the introduction, and thus her “redundancy” is necessary for the task of stopping her motion. While this fanciful illustration may oversimplify the

Link i	Mass	a_i	d_i	α_i	θ_{i-1}
1	12.0 kg	1.0 m	0	0	θ_1
2	12.0 kg	1.0 m	0	0	θ_2
3	11.04 kg	0.92 m	0	0	θ_3
tool	0.96 kg	0.08 m			0

(a)

Motor parameter	value
torque constant	0.01 n•m/rad
winding resistance	1.0 ohm
gear ratio	1:1
amplifier gain	0.1
coulomb coefficient	0.015
static coefficient	1.189
winding inductance	0.0 H
back emf	0.0 V
effective inertia	1.0e-5 kg
viscous damping coef.	1.0e-4

(b)

$$\mathbf{I}_1 = \begin{matrix} 0.00540 & 0.00000 & 0.00000 \\ 0.00000 & 1.00270 & 0.00000 \\ 0.00000 & 0.00000 & 1.00270 \end{matrix} \ \mathrm{kg{\bullet}m^2/rad^2} \qquad \mathbf{I}_2 = \begin{matrix} 0.00540 & 0.00000 & 0.00000 \\ 0.00000 & 1.00270 & 0.00000 \\ 0.00000 & 0.00000 & 1.00270 \end{matrix} \ \mathrm{kg{\bullet}m^2/rad^2}$$

$$\mathbf{I}_3 = \begin{matrix} 0.00497 & 0.00000 & 0.00000 \\ 0.00000 & 0.78117 & 0.00000 \\ 0.00000 & 0.00000 & 0.78117 \end{matrix} \ \mathrm{kg{\bullet}m^2/rad^2} \qquad \mathbf{I}_{tool} = \begin{matrix} 0.00019 & 0.00000 & 0.00000 \\ 0.00000 & 0.00061 & 0.00000 \\ 0.00000 & 0.00000 & 0.00061 \end{matrix} \ \mathrm{kg{\bullet}m^2/rad^2}$$

(c)

$$\mathbf{K}_D = \mathrm{Diag}(200,150,100)\ \mathrm{kg{\bullet}m^2/(sec{\bullet}rad^2)}$$

(d)

Tables 1.1(a)-(d) list the parameters for the simulated 3DOF planar manipulator. Table 1.1(a) lists the D-H parameters and the link masses, where a_i is the distance from frame i to frame $i + 1$ along the x-axis of frame i, d_i is the displacement between frames $i - 1$ and i along the joint axis i, α_i is the angle between the joint axes i and $i + 1$ measured around the x-axis of frame i, and θ_i is the angle between the links $i - 1$ and i measured around the joint axis i. Table 1.1(b) lists the parameters for the actuators at each joint. Table 1.1(c) gives the inertia matrices for each link in local coordinates, as calculated by ROBSIM (using a simple rod model for each link). Table 1.1(d) gives the terms of the constant joint damping matrix. Note that ROBSIM defines joints as rotating around the local y-axis and extending along the local x-axis.

The task frame was defined at the end of the tool, with axes parallel to the world axes.

Link i	Mass	a_i	d_i	α_i	θ_{i-1}
1	6.0 kg	0.5 m	0	+90	θ_1
2	6.0 kg	0.5 m	0	-90	θ_2
3	6.0 kg	0.5 m	0	+90	θ_3
4	6.0 kg	0.5 m	0	-90	θ_4
5	6.0 kg	0.5 m	0	+90	θ_5
6	6.0 kg	0.5 m	0	-90	θ_6
7	5.04 kg	0.42 m	0	+90	θ_7
tool	0.96 kg	0.08 m			0

(a)

Motor parameter	value
torque constant	0.01 n•m/rad
winding resistance	1.0 ohm
gear ratio	1:1
amplifier gain	0.1
coulomb coefficient	0.015
static coefficient	1.189
winding inductance	0.0 H
back emf	0.0 V
effective inertia	1.0e-5 kg
viscous damping coef.	1.0e-4

(b)

$$\mathbf{I}_1 = \mathbf{I}_2 = \mathbf{I}_3 = \mathbf{I}_4 = \mathbf{I}_5 = \mathbf{I}_6 = \begin{matrix} 0.00540 & 0.00000 & 0.00000 \\ 0.00000 & 1.00270 & 0.00000 \\ 0.00000 & 0.00000 & 1.00270 \end{matrix} \ \mathrm{kg \bullet m^2/rad^2}$$

$$\mathbf{I}_7 = \begin{matrix} 0.00227 & 0.00000 & 0.00000 \\ 0.00000 & 0.07522 & 0.00000 \\ 0.00000 & 0.00000 & 0.07522 \end{matrix} \ \mathrm{kg \bullet m^2/rad^2} \qquad \mathbf{I}_{tool} = \begin{matrix} 0.00019 & 0.00000 & 0.00000 \\ 0.00000 & 0.00061 & 0.00000 \\ 0.00000 & 0.00000 & 0.00061 \end{matrix} \ \mathrm{kg \bullet m^2/rad^2}$$

(c)

$$\mathbf{K}_D = \mathrm{Diag}(70,60,50,40,30,20,10)\ \mathrm{kg \bullet m^2/(sec \bullet rad^2)}$$

(d)

Tables 1.2(a)-(d) list the parameters for the simulated 7DOF manipulator. Table 1.2(a) lists the D-H parameters and the link masses (the definitions of the D-H parameters are given in the caption to Table 1.1). Table 1.2(b) lists the parameters for the actuators at each joint. Table 1.2(c) gives the inertia matrices for each link in local coordinates, as calculated by ROBSIM (using a simple rod model for each link). Table 1.1(d) gives the terms of the constant joint damping matrix. Note that ROBSIM defines joints as rotating around the local y-axis and extending along the local x-axis.

The task frame was defined at the end of the tool, with axes parallel to the world axes.

matter of impact force reduction, we can at least conclude that, generally speaking, redundancy is useful for reducing the force of impact, since the manipulator can be reconfigured and yet remain in contact with the object with which it collides.

It has been shown that, in the case of instantaneous impact, reduction of the effective task space inertia of the manipulator is beneficial to reducing the impact force[1]. The effective task space inertia is a measure of the inertia of the manipulator at the end-effector in Cartesian space along the direction of motion of the end-effector during the impact event. It has the properties of mass, and so we will refer to it as the *effective mass* for brevity's sake. The formulation of the effective mass with respect to the configuration of the manipulator is given in [15]. Components of the task space inertia not along the end-effector's direction of motion are orthogonal to it, and therefore do not effect the impact event [1]. By changing the configuration of the manipulator, we change the effective mass at the end-effector [1,4]. We will hypothesize (and later show), based on intuition and the results in [1], that certain configurations are better at reducing the penetration from collisions of finite duration. For now, we will justify this hypothesis by noting that reducing the effective mass should reduce the impact force, since:

$$\mathbf{F}_{e} = \frac{d\mathbf{p}_{e}}{dt} = m\frac{d\mathbf{v}_{e}}{dt} \tag{1}$$

where:

$\mathbf{F}_e$ is the force at the end-effector of the manipulator,

$\mathbf{p}_e$ is the momentum of the manipulator of the manipulator,

m is the effective mass of the manipulator,

and $\mathbf{v}_e$ is the Cartesian velocity of the manipulator.

and we will show that this is so in Section 1.5, although we will see that the reduction in effective mass must be coupled with increasing the effective damping as well.

In the case of nonredundant manipulators, there are only a finite number of configurations in which the manipulator tool will contact an object. When a nonredundant manipulator (such as that of Figure 1.2.1) impacts a surface, therefore, we cannot reduce the force of impact by reducing the effective mass, because we cannot change to another configuration. Even if we did desire to change the configuration of the manipulator (perhaps from "elbow up" to "elbow down"), we could not do it while remaining in contact with the surface without incurring larger forces at the end-effector. Nonredundant manipulators are capable of producing *net motion* only, meaning that any change in the configuration of the joints must also move the end-effector from its current location. Any joint motion will either move the end-effector away from the surface, thus losing its contact with the surface, or will drive the end-effector deeper into the surface (assuming the surface has enough pliability). Deeper penetration into the surface means that a larger impact force will be generated. To illustrate this, consider the case where a manipulator collides with a smooth vertical

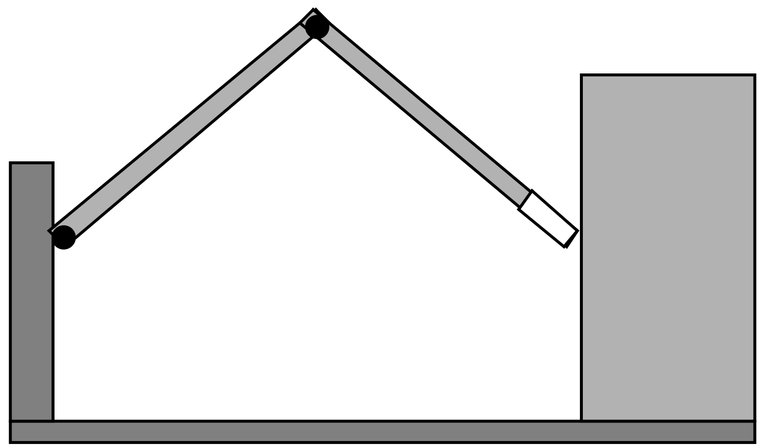


Figure 1.2.1: This two degrees-of-freedom manipulator only has two possible configurations which allow it to contact the wall at a particular point– elbow up and elbow down. Both configurations have the same effective task space inertia (or *effective mass*) along the vector normal to the wall.

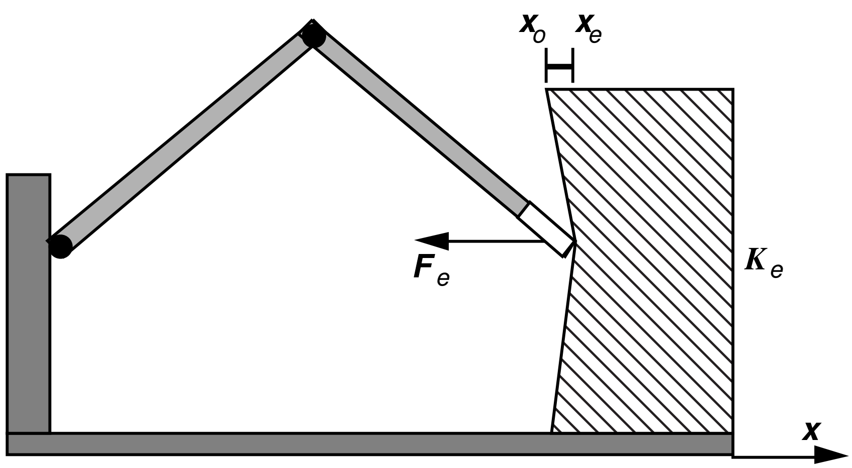


Figure 1.2.2 shows the penetration of the manipulator's end-effector into the surface of a normally smooth and flat wall. Here motion is assumed to be normal to the wall, and towards the wall, at the start of the impact event. The force at the end-effector is equal to the product of the stiffness of the wall in the direction of motion and the penetration into the object, as given in equation (2).

wall of some constant stiffness. We will assume that the momentum of the manipulator is in the direction of the wall, along the vector normal to the wall at the moment of impact. The impact force, which will also be along the vector normal to the wall [1], as in Figure 1.2.2, can be expressed as:

$$
\begin{aligned}
\mathbf{F}_e \quad &= \mathbf{K}_e \delta x_e, \ \delta x_e < 0 \\
&= 0, \qquad\qquad \delta x_e \quad 0 \qquad\qquad\qquad\qquad (2)
\end{aligned}
$$

where:

δx_e is the error is position with respect to the surface of the wall (referred to hereafter as *penetration*) along the vector normal to the wall, the magnitude of which is given as $\mathbf{x}_o$ - $\mathbf{x}_e$, where $\mathbf{x}_o$ and $\mathbf{x}_e$ are the locations of the wall and the end-effector, respectively, along the normal vector;

and $\mathbf{K}_e$ is the stiffness of the environment (in this case, the wall).

Clearly, given a surface with some fixed stiffness, the impact force is directly proportional to the penetration into the surface. (Our results in Sections 1.4 and 1.5 therefore use penetration as a gauge to judging the effectiveness of the impact strategies for a particular wall stiffness.) Therefore, a change in joint configuration for a nonredundant manipulator would result in the generation of higher forces at the end-effector, if contact with the surface was not in fact lost entirely.

On the other hand, a kinematically redundant manipulator, such as the three degrees-of-freedom planar manipulator of Figure 1.2.3, is quite capable of joint motion without moving the end-effector. This type of motion is called *null motion*. This null motion allows us to change the configuration of the manipulator without changing the location of the end-effector, and thereby change the effective mass of the manipulator. Figure 1.2.4 shows the effect that changing the configuration has on the component of effective mass in the direction of the impact force, using ROBSIM to calculate the effective mass of a three degrees-of-freedom planar manipulator (its parameters are given in Table 1.1). Since null motion does not move the position of the end-effector, we can even change the configuration of the manipulator during a collision without fear of pushing the end-effector deeper into the surface to detrimentally cause an increase in the impact force. Null motion cannot directly affect the impact force (although it may introduce a change in the effective mass sufficient to affect the outcome of the impact event).

To reduce the impact force, we can use the properties of kinematic redundancy to reduce the net motion into the surface of the impacted object. In order to do this, however, we will require a way to describe the current state of the redundant manipulator. This is not straightforward, since the normal relationships between torques, forces, joint velocities and Cartesian velocities used for nonredundant manipulator control strategies cannot be used for redundant manipulators. The reasons for this, and the tools used to relate the various aspects of the redundant manipulator, are discussed in the next section.

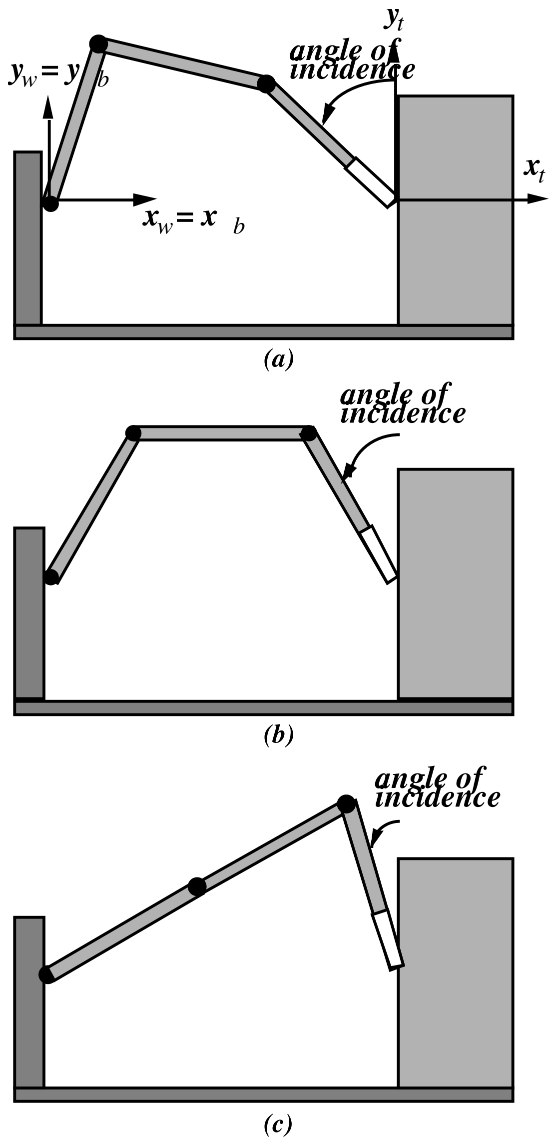


Figure 1.2.3: This three degrees-of-freedom manipulator has an infinite number of configurations that can make contact with a particular point on the wall. Configuration (a) has a relatively high effective mass with respect to the vector normal to the surface at the point of contact, since the link touching the wall is nearly linear with the adjacent link (think of jumping from some height and landing straight-legged, with all of your mass aligned behind your legs). Configuration (b) has less effective mass than (a), while (c) is the configuration with the lowest effective mass possible for this manipulator. The angle of incidence is used to describe the particular configuration. For each angle of incidence, there are two possible configurations – “elbow up” and “elbow down,” as shown in Figure 2.4.1. World, base, and task coordinate frames for the manipulator as defined in this research are given above in (a).

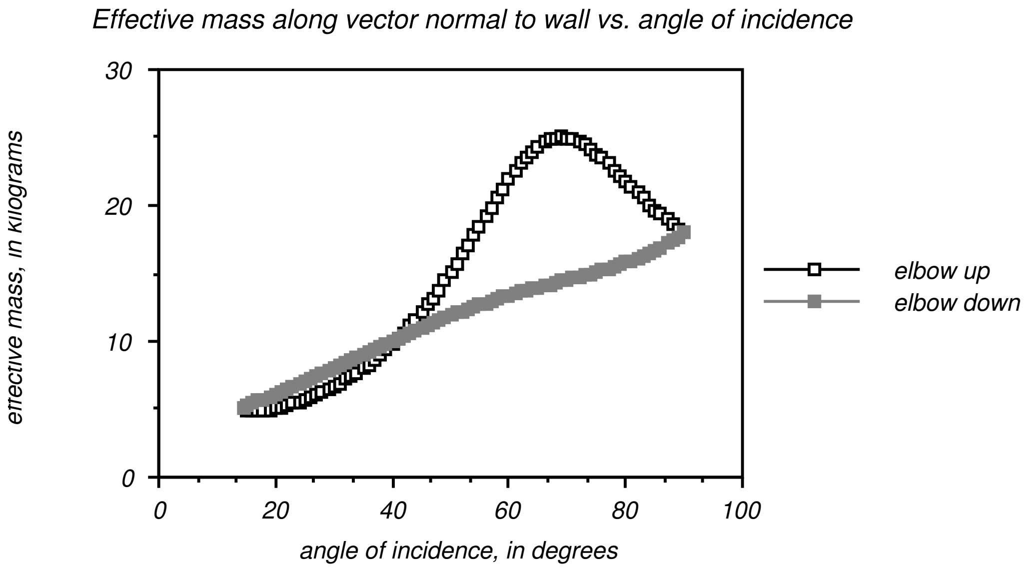


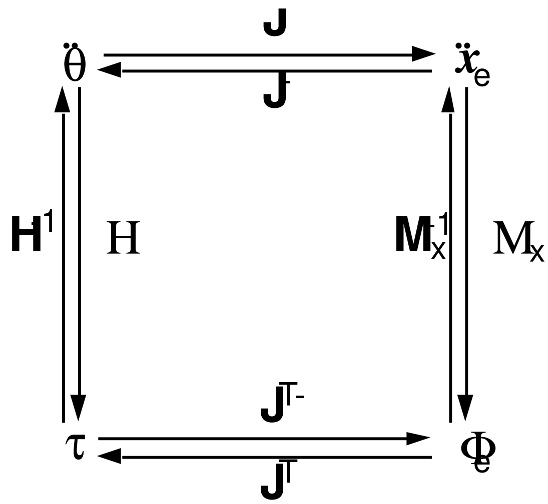
Figure 1.2.4: The effective mass of the three degrees-of-freedom planar manipulator of Figure 1.2.4 versus the angle of incidence (as defined in Figure 1.2.3). The manipulator parameters are given in Table 1.1. The angle at which the manipulator contacts its target ranges from approximately 14.5 to 90 , with dynamically equivalent solutions from 165.5 to 90 . There are two possible configurations for each angle of incidence ("elbow up" and "elbow down"), as illustrated in Figure 2.4.1.

1.3. The Premultiplier Diagram

Now that we have introduced redundant manipulators and explained why they are useful for reducing impact force, it is important to discuss the tools we have used to interrelate certain properties of a redundant manipulator (such as velocity, effective mass, etc.). To this end, we introduce the *Premultiplier Diagram* in this section.

The Premultiplier Diagram is a tool with which we can solve for certain pseudoinverses and equations when dealing with redundant manipulators. This is necessary since the Jacobian of a redundant manipulator is not square, and such things as "mass-weighted inverses" and the like take on different characteristics and meanings. In particular, the Premultiplier Diagram is used to convert joint-space matrices to Cartesian matrices. Its construction is given by Kim *et. al.* in [7,8]. The Premultiplier Diagram for impact dynamics is shown in Figure 1.3.1, and the diagram for stiffness is shown in Figure 1.3.2. The variables on the diagrams are as follows:

$\mathbf{F}_e$ is the force of the end-effector;
τ is the joint torque vector;



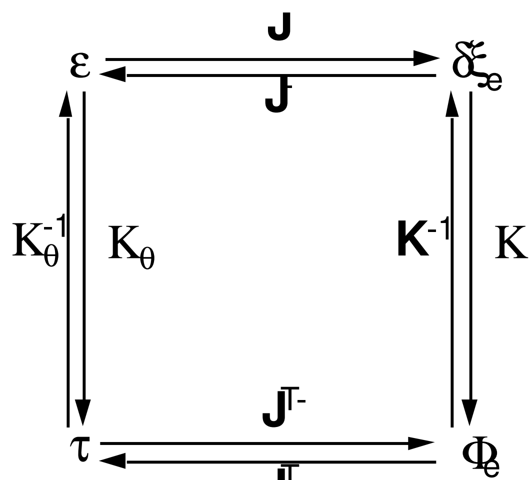


Figure 1.3.1: The Dynamic Premultiplier Diagram. Negligible changes in the Jacobian are assumed for the duration of the impact event. Futhermore, the effects of Coriolis and centripetal forces are also assumed to be negligible.

Figure 1.3.2: The Stiffness Premultiplier Diagram.

$\dot{\theta}$ is the joint velocity vector;

$\dot{\mathbf{x}}_e$ is the Cartesian velocity vector of the end-effector;

ε is the current joint position error vector, $(\theta_d - \theta)$;

δx_e is the Cartesian positional error vector of equation (1);

J is the Jacobian of the manipulator;

$\mathbf{J}^-$ is the pseudoinverse of the Jacobian (equals $\mathbf{J}^{-1}$ for a non-redundant manipulator in a non-singular configuration);

H is the manipulator inertia matrix;

$\mathbf{M}_x$ is the Cartesian matrix of inertia terms of the manipulator at the end-effector;

$\mathbf{K}_\theta$ is the joint stiffness matrix; and

$\mathbf{K}$ is the Cartesian effective stiffness matrix of the manipulator at the end-effector (not to be confused with $\mathbf{K}_e$, the stiffness of the environment).

The diagrams are used as follows:

1) Locate the corner vector that you wish to find on the Premultiplier Diagram. This will be the *target vector*.
2) Locate a corner vector that is *known* to you. This will be the *start vector*.
3) Beginning with the start vector, follow the arrows to the target vector, premultiplying the start vector with the matrix value of the arrow. Stop when you reach the target vector.

For example, we can solve for the change in Cartesian velocity from the force using Figure 1.3.1. Starting at the "force" corner, we premultiply successively by the transpose of the Jacobian, the inverse of the inertia matrix, and the Jacobian, so that:

$$\ddot{\mathbf{x}}_e = \mathbf{J}\,\mathbf{H}^{-1}\,\mathbf{J}^T\mathbf{F}_e \tag{3}$$

We can also find the value of the matrices along the arrows; for instance, the effective mass of the manipulator at the end-effector. First we note that another valid equation for the Cartesian acceleration is (assuming any changes in the Jacobian are negligible):

$$\ddot{\mathbf{x}}_e = \mathbf{M}_x^{-1}\,\mathbf{F}_e \tag{4}$$

Substituting (3) into (4) and solving, we find that:

$$\mathbf{M}_x = (\mathbf{J}\mathbf{H}^{-1}\mathbf{J}^T)^{-1} \tag{5}$$

We shall use both Premultiplier Diagrams to provide us with weighted pseudoinverses when we calculate torques in Sections 1.4 and 1.5.

1.4. Using Stiffness and Damping Torques to Minimize Impact Force

1.4.1. Introduction

With the explanation of why redundancy is important in Section 1.2, and the proper tools to define manipulator properties mathematically given in Section 1.3 in the form of the Premultiplier Diagram, we can now begin to create strategies to reduce the impact force at the end-effector. In this section, we present a method for reducing the impact force by introducing correcting torques at the joints to minimize penetration into an object during impact, thus minimizing the impact force.

1.4.2. Basic Assumptions

We will be modeling the impact event for a redundant manipulator that performs tasks in space, where it would be especially beneficial to reduce the impact force. In this model, we will make several initial assumptions. First of all, we assume that objects in the environment can be simulated as springs of various stiffness. Second, the links of our manipulator will be assumed to be rigid. Third, we assume that the collision event will have some duration (i.e. it is not instantaneous). Finally, we shall allow the end-effector to have only net motion at the time of impact. The last assumption is not necessary from a purely control point of view, but it will allow us to see more clearly the effects of our impact force reduction strategy on the joint motions.

The characteristics of the simulated three degrees-of-freedom manipulator used to test the strategy presented in this section are listed in Table 1.1.

1.4.3. Modeling the Impact and Correcting Torques

The motion of a manipulator is modeled as [9]:

(6)

$$\mathbf{H}(\theta)\ddot{\theta} + X(\theta,\dot{\theta}) + \Gamma(\theta) = \tau$$

where:

$\mathbf{H}(\theta)$ is the $n \times n$ mass matrix (also called the inertia matrix),

is the $n \times 1$ Coriolis/centripetal torque vector,

$$\mathbf{C}(\theta,\dot{\theta})$$

$\mathbf{G}(\theta)$ is the $n \times 1$ vector of gravity terms,

and τ is the $n \times 1$ vector of torques at each joint.

The manipulator operates in space, so the gravity terms become zero. We include the torque resulting from the impact force generated from the collision with the wall (Figure 1.4.1),

$\mathbf{J}^T\mathbf{F}_e$ (as mapped in both Premultiplier Diagrams). Equation (6) becomes:

$$\mathbf{H}(\theta)\,\ddot{\theta} + X(\theta,\dot{\theta}) = \tau + \vartheta^T\Phi_\varepsilon \tag{7}$$

where $\mathbf{F}_e$ is the current force at the end-effector (like the joint velocities and accelerations, it is a function of time). Thus, we can think of the motion of the manipulator as a function of the configuration combined with the relative joint velocities and accelerations of the joints, which creates some torque at the joints to move the manipulator. To draw on the analogy of the child on the trampoline, we wish to have the manipulator "bend its knees" when it comes into contact with a surface, in order to decrease penetration and thereby decrease impact force and its undesirable side effects. This implies the addition of some torque to rotate the joints. Because we do not want the end-effector to be forced deeper into the surface, we require that this torque introduces net motion in the direction opposite of that in which the end-effector is traveling. This torque will be referred to as the *correcting torque*.

We can introduce correcting torque as a function of the configuration (proportional feedback, or stiffness torque), a function of the joint velocity (derivative feedback, or damping torque) or a combination of the two. Additionally, we can introduce some extra torque at discrete

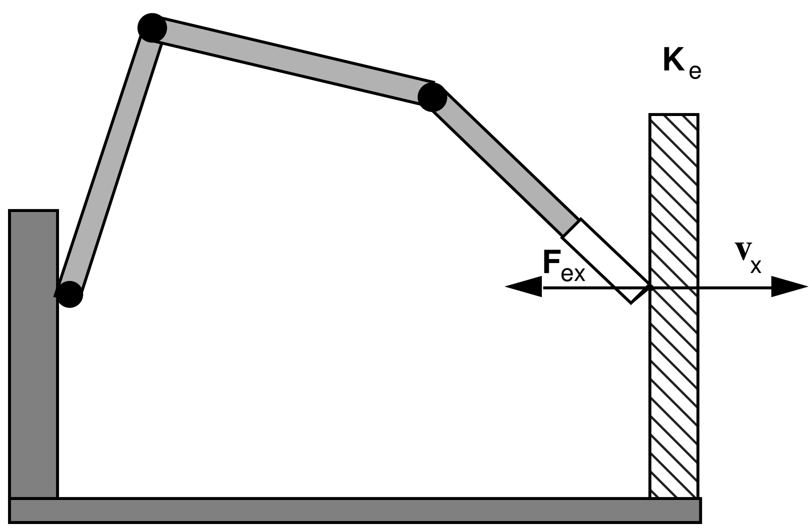


Figure 1.4.1: The end-effector of a three degrees-of-freedom colliding with a wall. The motion is normal to the surface of the wall, as is the force. The wall is assumed to have some stiffness K_e.

intervals to keep the end-effector in contact with the object. This latter torque would be necessarily small, and in opposition to the correcting torque. Substituting these for the torque vector in equation (7) gives us a model-based manipulator control law equation [4,10]:

$$\mathbf{H}(\theta)\,\ddot{\theta} + X(\theta,\dot{\theta})$$
$$= \tau_{\Sigma} + \tau_{\Delta э} - \tau_{\Phi}[\kappa] + \vartheta^{T}\Phi_{\varepsilon} \qquad (8)$$
$$= \mathbf{K}_{\theta}\,\varepsilon + K_{\Delta}\,\dot{\varepsilon} - \vartheta^{T}\Phi_{\delta}[\kappa] + \vartheta^{T}\Phi_{\varepsilon}$$

where:

τ_S is the added stiffness torque, $\mathbf{K}_\theta\,\varepsilon$,

$\tau_{\Delta э}$ ισ τηε αδδεδ δαµπινγ τορθυε, $K_{\Delta}\,\dot{\varepsilon}$,

τ_F is the added discrete contact torque, $\mathbf{J}^T\mathbf{F}_d[k]$,

$\mathbf{K}_\theta$ is an n x n matrix of joint stiffness terms,

$\mathbf{K}_D$ is an n x n matrix of joint damping terms,

ε is the joint position error (θ_d - θ),

$\dot{\varepsilon}$ ισ τηε φοιντ ϖελοχιτψ ερρορ $(\dot{\theta}_{\delta} - \dot{\theta})$,

$\mathbf{J}$ is the m x n Jacobian of the system,

and $\mathbf{F}_d[k]$ is the force at the end-effector taken at discrete time interval k, so that $\mathbf{F}_d[k]$(t) = $\mathbf{F}_e(mk)$ for mk t < $(n+1)k$., where m is an integer, reintroduced discretely (in stairstep form) to counter the impact force, and keep the end-effector in contact with the wall (not used in our research).

Both the stiffness and damping torque terms have a null component and a net component, which induce null and net motion, respectively. However, we only need to retard net motion, and therefore we need concern ourselves only with introducing net stiffness torque and net damping torque. We then have two ways to reduce the impact force. First, we can cancel the null component of the stiffness correcting torque, since we require that net damping should be high, and that null damping be low, and apply the resultant torque to the system. Second, we can reduce the size of the null component of the damping torque and increase the size of its net component and apply that torque as well. The null component of this damping torque should not be set to zero, as will be discussed later in Section 1.4.5.

1.4.4. The Stiffness Torque Term

Let us first examine the stiffness torque term, and find its net component. Kim *et al* show in [7,8] that a given joint position vector of a redundant manipulator can be decomposed into net (particular) and null (homogeneous) components. We can decompose the joint position error that we defined in equation (8) into its net and null components, ε_p and ε_h respectively, using stiffness-weighted minimum norm generalized inverses, so that:

$$\varepsilon = \varepsilon_p + \varepsilon_h$$

$$= \qquad (9)$$

$$\mathbf{J}^{+}_{K_\theta} \vartheta \varepsilon + (I - \vartheta^{+}_{K_\theta} \vartheta)\ \varepsilon$$

where:

is a Jacobian pseudoinverse weighted by $\mathbf{K}_\theta$ and $\mathbf{K}$ relating joint position error to

$$\mathbf{J}^{+}_{K_\theta} = K_\theta^{-1} \vartheta^T K$$

Cartesian position error, given by the Stiffness Premultiplier Diagram.

We can then extract the net portion of the stiffness torque, and solve for it in terms of the matrices that are specified for us, again using the Stiffness Premultiplier Diagram:

$$\tau_{Sp} = \mathbf{K}_\theta\, \varepsilon_p$$

$$=$$

$$\mathbf{K}_\theta\, \vartheta^{+}_{K_\theta} \vartheta \varepsilon$$

$$= \mathbf{K}_\theta\, (\mathbf{K}_\theta^{-1}\, \mathbf{J}^T\, \mathbf{K})\, \mathbf{J}\, \varepsilon$$

$$= \mathbf{J}^T\, \mathbf{K}\, \delta x$$

$$= \mathbf{J}^T\, (\mathbf{J}\, \mathbf{K}_\theta^{-1}\, \mathbf{J}^T)^{-1}\, \delta x \qquad (10)$$

where τ_{Sp} is the net stiffness torque.

Equation (10) gives us the pure net stiffness torque which counters the motion into the surface of the wall. We could increase this torque by scaling it by some factor and introduce the result into the system to decrease net motion without reducing the null component at all. This scale factor would have to be chosen based on a knowledge of the torque limits of the manipulator upon which this scheme is used, since too high a scale factor would overcome the torque limits, thereby ruining the usefulness of this strategy. The effectiveness of this scheme would also be dependent on the terms in the joint stiffness matrix, since they would directly affect the magnitude of the torque produced.

1.4.5. The Damping Torque Term

We will now turn to examining the damping torque. Rather than canceling its null component, which is unwise (no matter how hard we try to minimize null motion, there will almost always be some null joint velocity, which we will want damped if we wish the system to come to a halt at some point), we will derive the net and null components with the help of the Dynamics Premultiplier Diagram, and scale them in such a way that net damping is increased.

First, we should note that we wish the system to stop moving when it reaches its destination. That is, our desired angular velocity is zero. Therefore:

$$\mathbf{K}_D\dot{\varepsilon} = \mathbf{K}_D(\dot{\theta}_\delta - \dot{\theta}) = -\mathbf{K}_D\dot{\theta} = \tau_D \tag{11}$$

The decomposition of this torque into its net and null redundancy components is given as [7,8]:

$$\tau_\Delta = \mathbf{J}^T\mathbf{J}_H^{+T}\tau_\Delta + (I - \vartheta^T\vartheta_H^{+T})\,\tau_\Delta = \tau_{Dp} + \tau_{Dh} \tag{12}$$

where $\mathbf{J}_H^{+}$ is the mass-weighted generalized inverse of the Jacobian,

$$\mathbf{J}_H^{+} = \mathbf{H}^{-1}\mathbf{J}^T(\mathbf{J}\,\mathbf{H}^{-1}\mathbf{J}^T)^{-1} \tag{13}$$

as determined by the Dynamics Premultiplier Diagram.

Now, we wish to increase the net damping in order to slow down the forward progress of the manipulator. As mentioned previously, the null damping cannot be zero, because we would eventually like the system to come to a complete stop. (In a non-ideal system this would not be as big of a problem, since null damping would be caused by joint friction. However, there exist systems which have a very low friction coefficient, and the time that it would take the null motion to cease could be quite long in these systems.) Nevertheless, we do not want high null damping, as previously explained, so we must scale down the null component.

With the direction of the net and null vectors supplied by (12), we scale each component of the damping torque, to control the amount applied to the system:

$$\tau'_{Dp} = h_1 \, \tau_{Dp} \tag{14}$$

$$\tau'_{Dh} = h_2 \, \tau_{Dh} \tag{15}$$

$$\tau_D' = \tau'_{Dp} + \tau'_{Dh} \tag{16}$$

where:

τ'_{Dp} is the scaled net damping torque,
τ'_{Dh} is the scaled null damping torque,
and τ_D' is the total damping torque to be added.

Because we desire large net damping and low null damping, we conclude that h_1 should be much larger than h_2 . Increasing h_1 increases the net damping torque. An increase in the net damping torque causes the net velocity into the wall to decrease, and the resulting decrease in momentum causes the impact force to be reduced. If h_1 is too high, then the torque calculated by this method will be too large for the actuators, and saturation will occur. In our simulated ideal manipulator (no saturation limits), increasing h_1 caused the system to oscillate at a faster rate (without losing contact). At even higher values of h_1, the end-effector stayed sunk into the surface. Figure 1.4.2 shows the simulated effect of several different values of h_1 on the impact response over time.

Increasing h_2 had very little effect on the impact force, since null motion does not drive the arm deeper into the surface, and since our null motion was very small relative to the net motion at the start of the impact event. (In our experiments, increasing h_2 caused the amount of penetration to increase only slightly, even for extremely large values of h_2, as shown in Figure 1.4.3. Similar increases and decreases in h_1 cause much more dramatic changes in the penetration, as shown in

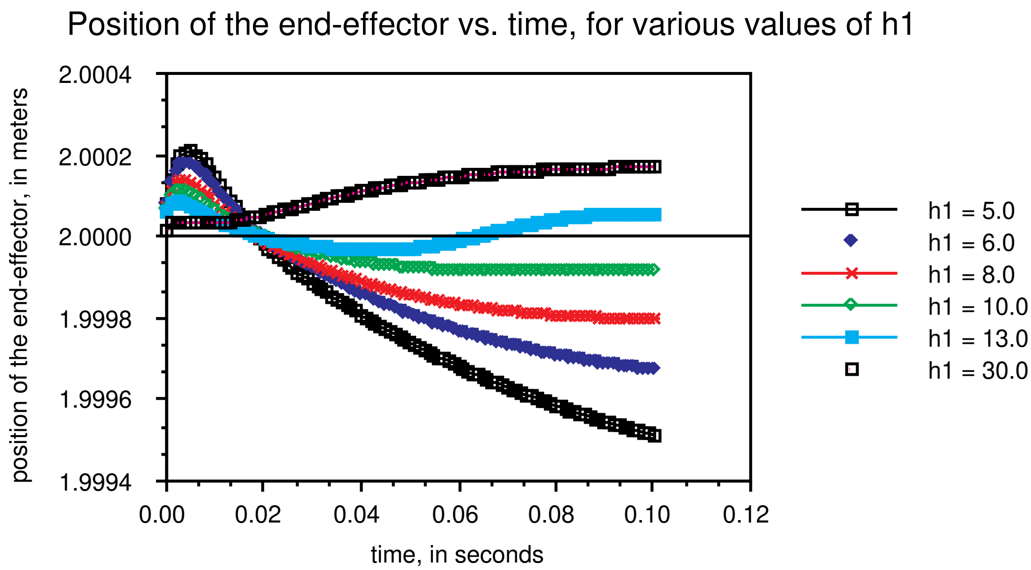


Figure 1.4.2: This plot shows the simulated impact event vs. time for several values of h_1. The configuration of the 3DOF planar manipulator is (60,-60,-60) degrees, h_2 is 0.01, and the stiffness of the environment is 1×10^5 n/m. The "wall" is 2.0 meters from the base of the manipulator. As can be seen, increasing h_1 decreases the penetration and the rebound from the "wall," until, for some high value of h_1, the excessive damping torque causes the end-effector to stay sunk into the wall.

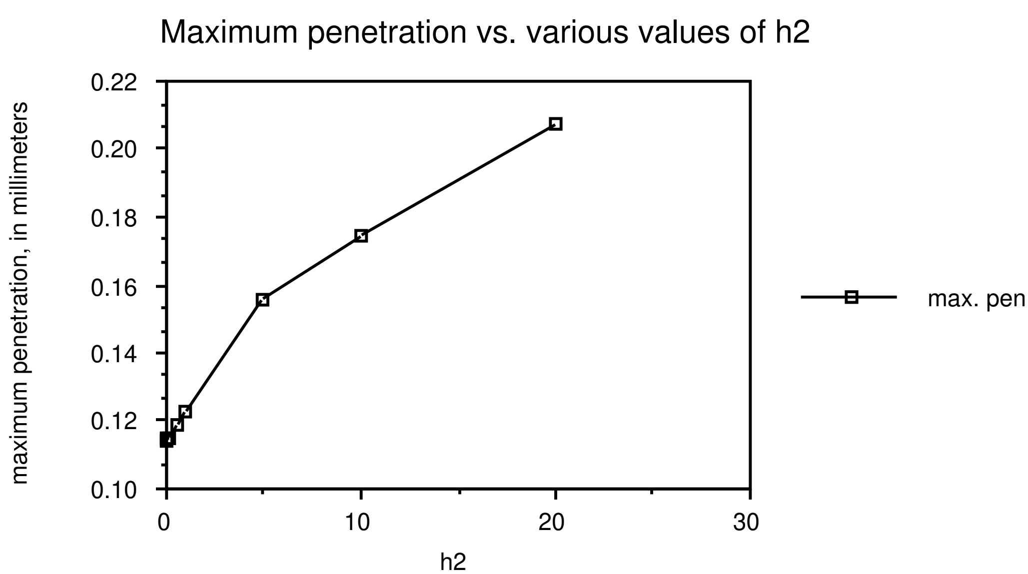


Figure 1.4.3: Penetration vs. several values of h_2, for configuration = (60,-60,-60) degrees, h_1=10.0, and wall stiffness = 1×10^5 n/m. Note that range of the penetration is less than 0.1 millimeters in this simulation, despite the unreasonably large domain of h_2.

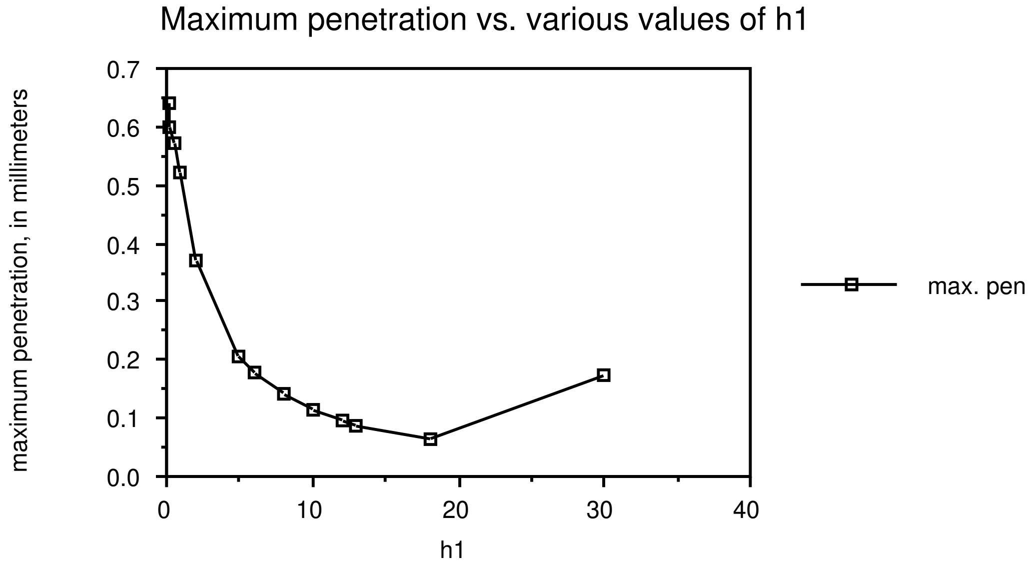


Figure 1.4.4: Penetration vs. several values of h_1, for configuration = (60,-60,-60) degrees, h_2=0.01, and wall stiffness = 1x10^5 n/m. Note that range of the penetration in this simulation is much greater than that experienced by changing h_2,as shown in Figure 1.4.3. For these tests, increasing net damping was necessary to decrease the impact force, while increasing null damping had little effect.

Figure 1.4.4.) A small h_2 is desirable if we do not wish to damp null motion.

The results of our simulations clearly show that, for our simulated ideal three degrees-of-freedom planar manipulator, the damping torque strategy presented in this section allows us to reduce the impact force between a manipulator and an object. These results are such that we feel we can conclude that this strategy would be effective for reducing the impact force for any actual redundant manipulator.

1.5. Using the Configuration to Minimize the Impact Force

1.5.1. Introduction

The strategy of the previous section introduced the idea that impact force can be reduced reduced by adding torques to minimize penetration into the surface of the object with which the end-effector collides. We now introduce a method which instead minimizes the impact force without the constant feedback of torques and velocities, using the configuration of the manipulator to increase effective damping and reduce effective mass.

Intuitively, there are certain configurations which are less useful than others when a collision occurs. Hitting a surface while in a configuration which causes a large effective mass at the end-effector will not be helpful in reducing the impact force, no matter what control scheme is used. Similarly, a configuration with low effective damping will not be of any use, because momentum into the surface must be damped. In this section we derive a means for deciding which configurations are useful during a collision, in order to find the best possible configuration to minimize impact force.

1.5.2. Basic Assumptions

As in Section 1.4, a few assumptions will be made. We will assume that we have prescribed a trajectory for the manipulator such that there is negligible null motion. In addition, we shall assume that a one-dimensional collision task can be generalized to a two- or three-dimensional case. We can justify this last assumption by choosing our world and object frames such that motion is along only one axis, and orthonormal to the other two axes.

The parameters of the simulated three degrees-of-freedom planar manipulator and seven degrees-of-freedom manipulator used to test the strategy of this section are listed in Tables 1.1 and 1.2, respectively.

1.5.3. The Configuration-Based Impact Strategy

We shall use the velocity damping control method of Section 1.4.5 to help create our configuration-based impact strategy. Recall that the torque of a manipulator can be calculated from (6). Slightly rearranged, this becomes:

$$\mathbf{H}(\theta)\,\ddot{\theta} + X(\theta,\dot{\theta}) - \vartheta^T\Phi_\varepsilon = \tau \tag{17}$$

We can separate the inertial component of the torque into net term and null terms:

$$\tau = \mathbf{H}\ddot{\theta}_\pi + \mathrm{H}\ddot{\theta}_\eta + X(\theta,\dot{\theta}) - \vartheta^T\Phi_\varepsilon \tag{18}$$

We must determine what the net angular acceleration is. To do this, we recall that velocity is given by [9]:

$$\dot{\mathbf{x}} = \mathbf{J}\dot{\theta} \tag{19}$$

Taking the derivative of (19), we get:

$$\ddot{\mathbf{x}} = \dot{\vartheta}\dot{\theta} + \vartheta\ddot{\theta} \tag{20}$$

Finally, we multiply both sides of the equation by the pseudoinverse of the Jacobian, which we get from the Premultiplier Diagram, and then solve for the angular acceleration. This gives us:

$$\ddot{\theta} = \vartheta_H^+\ddot{\xi} - \vartheta_H^+\dot{\vartheta}\dot{\theta} \tag{21}$$

We are concerned with the net terms only, having separated out the null terms in equation (18). We can substitute (21) into (18) for the net angular acceleration and get:

$$\tau = \mathbf{HJ}_H^+\ddot{\mathbf{x}} - \mathbf{HJ}_H^+\dot{\mathbf{J}}\dot{\theta} + H\ddot{\theta}_\eta + X(\theta,\dot{\theta}) + \vartheta^T\Phi_\varepsilon \tag{22}$$

Without loss of generality, we can define the position of the object as being at **x** = 0, and then, using equation (2), equation (22) becomes:

$$\tau = \mathbf{HJ}_H^+\ddot{\mathbf{x}} - \mathbf{HJ}_H^+\dot{\mathbf{J}}\dot{\theta} + H\ddot{\theta}_\eta + X(\theta,\dot{\theta}) + \vartheta^T K_\varepsilon \xi \tag{23}$$

Rearranging this, we get:

$$\mathbf{HJ}_H^+\ddot{\mathbf{x}} - \mathbf{HJ}_H^+\dot{\mathbf{J}}\dot{\theta}_\pi + H\ddot{\theta}_\eta + X(\theta,\dot{\theta}) - \tau + \vartheta^T K_\varepsilon \xi = 0 \tag{24}$$

Equation (24) can be simplified. We can drop the term relating to null acceleration because we have assumed that there is negligible null motion. The change in the Jacobian is negligible, because any one particular joint changes position very little during impact (for a sufficiently stiff surface of impact). (Note that this does not mean that we can cancel out the terms relating to Cartesian variables such as **x**, since they are a function of all joint positions, velocities, and accelerations, not just one). Finally, we note that the centripetal torque terms are dependent on the square of the joint velocities, and the Coriolis torque terms are dependent on the product of two joint velocities [9].

For our simulated manipulators we have arbitrarily chosen terms for the matrix of damping coefficients introduced in equation (8) that are on the order of 10^2 (kg•m^2)/(sec•rad^2), as shown in Tables 1.1 and 1.2. If we do not wish to saturate the actuators of the manipulator with torques which are too high for them too handle, then the joint velocities responsible for these torques must be made fairly small, on the order of 10^{-1} rad/sec or less. Therefore, the torque from the centripetal and Coriolis terms will be comparatively small with respect to the other torque terms, since the product of two joint velocities will be very small.

Canceling the aforementioned torque terms from (24), we are left with:

$$
\begin{aligned}
0 &= \mathbf{H}\mathbf{J}_H^{+}\ddot{\mathbf{x}} - \tau + \vartheta^T K_\varepsilon \xi \\
&= \mathbf{H}\mathbf{J}_H^{+}\ddot{\mathbf{x}} + \mathbf{K}_D\dot{\theta} + \vartheta^T K_\varepsilon \xi
\end{aligned}
\qquad (25)
$$

We can conclude from (25) that the impact event depends upon three parameters: the mass matrix and the mass-weighted generalized inverse of the Jacobian, which are configuration-dependent; the velocity damping matrix, which is fixed; the angular velocity, which is controllable. We can further separate (25) into net and null parts, introduced in the term having angular velocity. Since we have assumed the null motion to be negligible, (25) becomes:

$$\mathbf{H}\mathbf{J}_H^{+}\ddot{\mathbf{x}} + \mathbf{K}_D\dot{\theta}_\pi + \vartheta^T K_\varepsilon \xi = 0 \qquad (26)$$

The net angular velocity is given by multiplying both sides of equation (19) by the pseudoinverse of the Jacobian as determined from the Premultiplier Diagram, so that:

$$\mathbf{J}_H^{+}\dot{\mathbf{x}} = \dot{\theta}_\pi \qquad (27)$$

which we can substitute into (26) to get:

$$\mathbf{H}\mathbf{J}_H^{+}\ddot{\mathbf{x}} + \mathbf{K}_D\mathbf{J}_H^{+}\dot{\mathbf{x}} + \mathbf{J}^T\mathbf{K}_e\mathbf{x} = 0 \qquad (28)$$

Next, we normalize (28) so that the coefficient matrix of the acceleration vector is the identity matrix. To do this, we derive the pseudoinverse of the coefficient matrix of the acceleration from the Premultiplier Diagram:

$$(\mathbf{H}\mathbf{J}_H^+)^- = (\mathbf{J}_H^+)^- \mathbf{H}^- = \mathbf{J}\,\mathbf{H}^{-1} \tag{29}$$

We can then rewrite (30) by multiplying through by the inverse of the coefficient of the first term in the equation, so that we get:

$$\ddot{\mathbf{x}} + \mathbf{J}\mathbf{H}^{-1}\mathbf{K}_D\mathbf{J}_H^+\dot{\mathbf{x}} + \mathbf{J}\mathbf{H}^{-1}\mathbf{J}^T\mathbf{K}_e\mathbf{x} = 0 \tag{30}$$

As shown in Section 1.3, $\mathbf{J}\mathbf{H}^{-1}\mathbf{J}^T$ is the inverse of the effective mass of the manipulator. It can also be seen that $\mathbf{J}\mathbf{H}^{-1}\mathbf{K}_D\mathbf{J}_H^+$ simply maps terms involving the ratio of damping terms to mass terms from joint space to Cartesian space. For convenience, we simplify (30) as follows:

$$\ddot{\mathbf{x}} + \mathbf{A}_x\dot{\mathbf{x}} + \mathbf{M}_x^{-1}\mathbf{K}_e\mathbf{x} = 0 \tag{31}$$

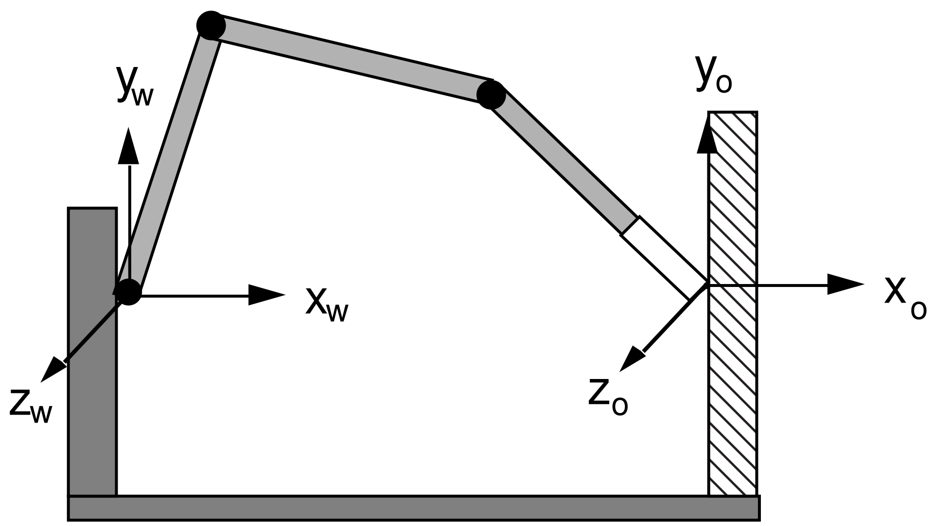


Figure 1.5.1: If the manipulator (world) frame and the object frame have the same orientation, then, if motion is only along the x-axis, only elements of matrices that deal exclusively with that axis need to be considered in the impact model. For instance, the effective mass of the manipulator along the y-axis will not affect motion along the x-axis, since the two quantities are orthogonal.

Thus, we have reduced the impact equations to a second-order differential equation in terms of $\mathbf{x}$, which, since we have defined the location of the object as the zero vector, is simply the penetration of the end-effector into the obstacle. Unfortunately, the coefficients of **x** and its derivatives are matrices, not scalars. However, the coefficients are task space matrices, of size m x m. We can certainly define our world and object frames so that the motion of the end-effector is along one of the axes (say, for instance, the x-axis), and orthogonal to the other axes (y and z), as in Figure 1.5.1. Since motion is orthogonal to the y- and z-axes, any coefficients involving terms along those axes do not contribute to the penetration into the wall in the direction of the x-axis. When we choose our frames this way, (31) becomes:

$$\ddot{\mathbf{x}} + a_x \dot{\mathbf{x}} + \frac{k_e}{m_x}\mathbf{x} = 0 \qquad (32)$$

where:

a_x is the element of $\mathbf{A}_x$ strictly along the vector of motion,

k_e is the element of $\mathbf{K}_e$ strictly along the vector of motion,

and m_x is the element of $\mathbf{M}_x$ strictly along the vector of motion.

Let's compare (32) with the standard second-order spring-mass-dashpot model. This is generally written as [11]:

$$m\ddot{\mathbf{x}} + d\dot{\mathbf{x}} + k\mathbf{x} = 0 \qquad (33)$$

or, dividing through by the mass:

$$\ddot{\mathbf{x}} + \frac{d}{m}\mathbf{x} + \frac{k}{m}\mathbf{x} = 0 \qquad (34)$$

Equation (34) compares very well with (32). If we let $m = m_x$ and $k = k_e$ in (34), then:

$$d = m_x a_x \qquad (35)$$

which is entirely appropriate, since a_x should be a ratio between the effective damping and the effective mass. Khatib and Burdick in [4] offer a similar second-order model having d=0 (pure oscillation).

We can now find a way to rate configurations based on their ability to damp the net motion, and thus reduce the impact force. In the spring-mass-dashpot model, increasing the damping ratio increases the damping of the system [11]. The damping ratio is given by:

$$\zeta = \frac{1}{2}\frac{\mu^{-1}\delta}{\sqrt{\mu^{-1}\kappa}} = \frac{1}{2}\frac{\delta}{\sqrt{\mu\kappa}} \tag{36}$$

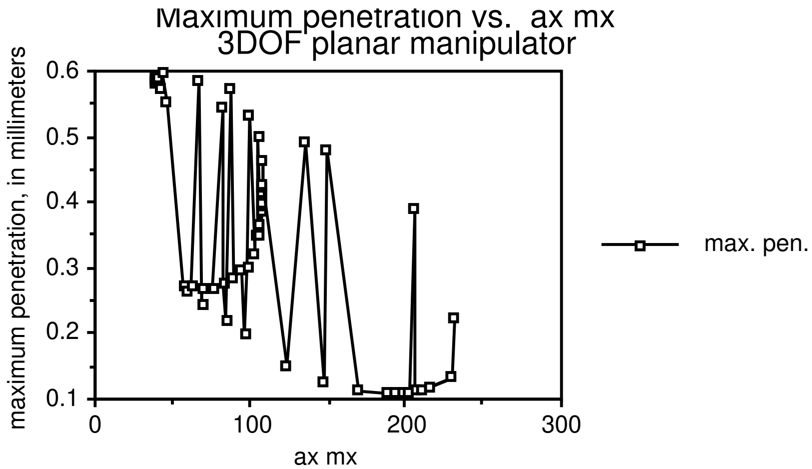
and substituting for *d*, *m*, and *k* in terms of the variables of our system gives us:

$$\zeta = \frac{1}{2}\frac{\delta}{\sqrt{\mu\kappa}} \tag{37}$$

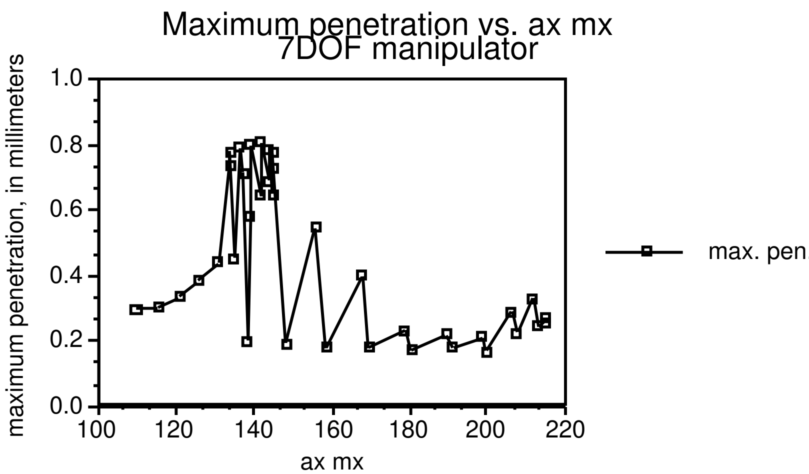
$$= \frac{1}{2}\frac{m_x a_x}{\sqrt{m_x k_e}}$$

$$\zeta = \frac{1}{2\sqrt{\kappa_\varepsilon}}\alpha_\xi\sqrt{\mu_\xi} \tag{38}$$

However, equation (38), which implies that increasing both a_x and m_x will be of benefit, failed to give good results as a criterion for minimizing penetration (and hence the impact force) when used in our ROBSIM simulations, as can be seen in Figures 1.5.2(a) and (b). The maximum values of penetration oscillate as a function of the damping ratio (perhaps due to the fact that the value we are taking for *d* in equation (34) is derived from a matrix which is a function of the mass matrix, intead of being an independent variable). In [1], Walker shows that minimizing the effective mass will reduce the impact force for the instantaneous impact event. Our ROBSIM experimental results, which simulate an impact event of finite duration, show that maximizing a_x was more conducive to minimizing penetration (and hence the impact force), but also show a tendency for lower effective mass to improve performance. While these results are from a limited number of simulations for one particular ideal manipulator (fifty different configurations were



(a)



(b)

Figure 1.5.2: A plot of the damping ratio vs. the penetration for a 3DOF planar manipulator (a) and a 7DOF manipulator (b). Obviously, this is not a good criterion for judging the effectiveness of a manipulator at reducing penetration and impact force.

simulated on the 3DOF planar manipulator, and thirty on the 7DOF manipulator), we feel that our simulations were designed with sufficient accuracy so that we can assume our results are valid for redundant manipulators in general.

Figure 1.5.3 shows a plot of penetration versus a_x for the simulated 3DOF manipulator, and Figure 1.5.4 shows a plot of penetration versus m_x for the same 3DOF manipulator. Clearly neither criterion alone is sufficient to rate the ability of the manipulator to minimize the impact force. Maximizing a_x is seen to be of paramount importance, while minimizing m_x is a secondary, though still important, concern. To take both observations into consideration, we weighted the

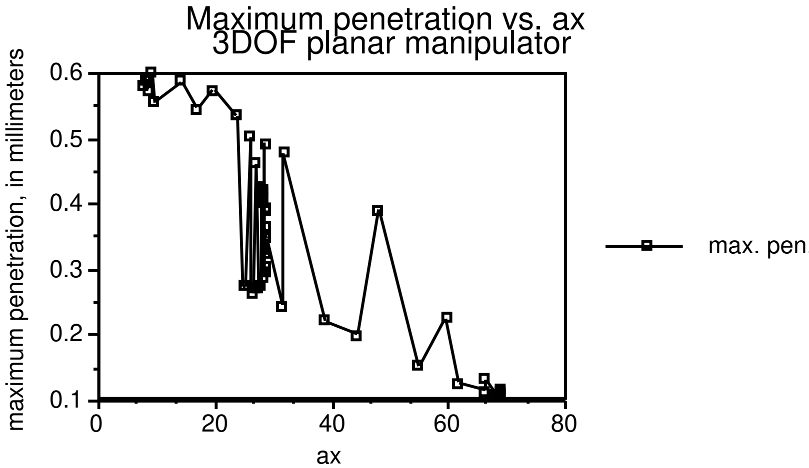


Figure 1.5.3: As can be seen, configurations with a lower effective damping value (a_x) tended to minimize penetration better on this simulated 3DOF planar manipulator. However, the plot is by no means clean.

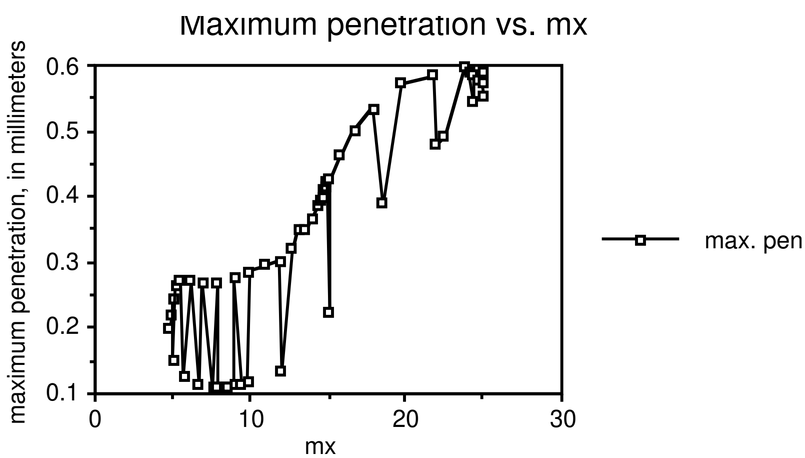


Figure 1.5.4: Configurations with a lower effective mass (m_x) tended to minimize the penetration into an object during impact for the simulated 3DOF planar manipulator, but the plot is even less clean than Figure 1.5.3.

damping ratio of (38) in such a way that the new damping ratio criterion would increase when the mass decreases, by dividing by the effective mass:

$$\hat{\zeta} = \frac{1}{2\sqrt{\kappa_\varepsilon}} \frac{\alpha_\xi}{\mu_\xi} \sqrt{\mu_\xi} = \chi \frac{\alpha_\xi}{\sqrt{\mu_\xi}} \tag{39}$$

where:

$$c = \frac{1}{2\sqrt{k_e}} \tag{40}$$

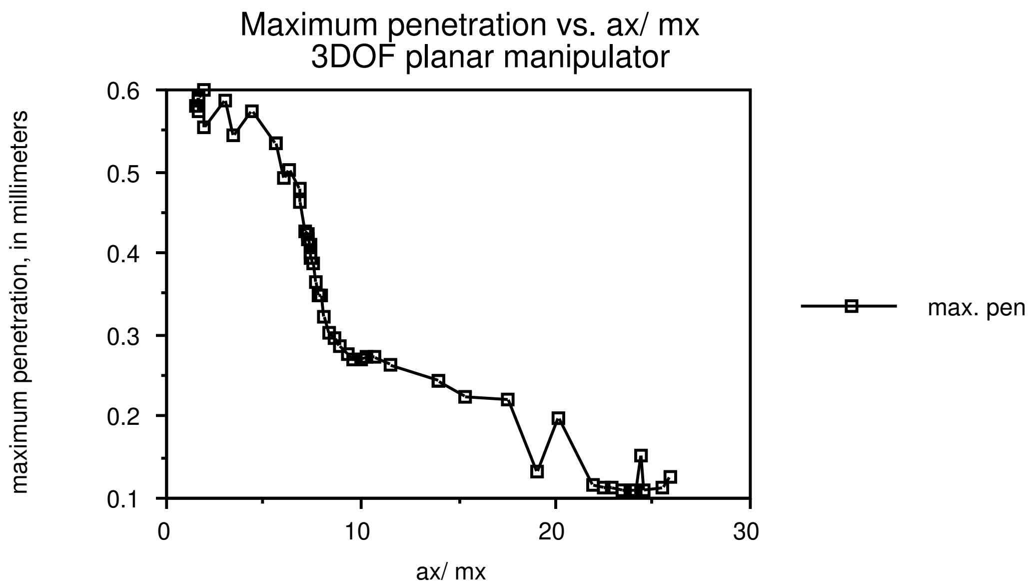
The value c is wholly dependent on the stiffness of the wall, since the matrix $\mathbf{K}_e$ is a constant property of the wall. Therefore, to maximize the weighted damping ratio, it is sufficient to maximize a_x while minimizing the square root of the effective mass.

Our experiments using the weighted damping ratio as a criterion for judging the effectiveness of the configurations of our simulated manipulators in reducing impact force show much better results than that of simply using the damping ratio. Figures 1.5.5(a) and (b) show the plots of penetration versus the weighted damping ratio for our simulations. Again, we feel that the quality of our simulations for these tests on the two manipulator types is good enough to suggest that we would generally find similar results using actual redundant manipulators.

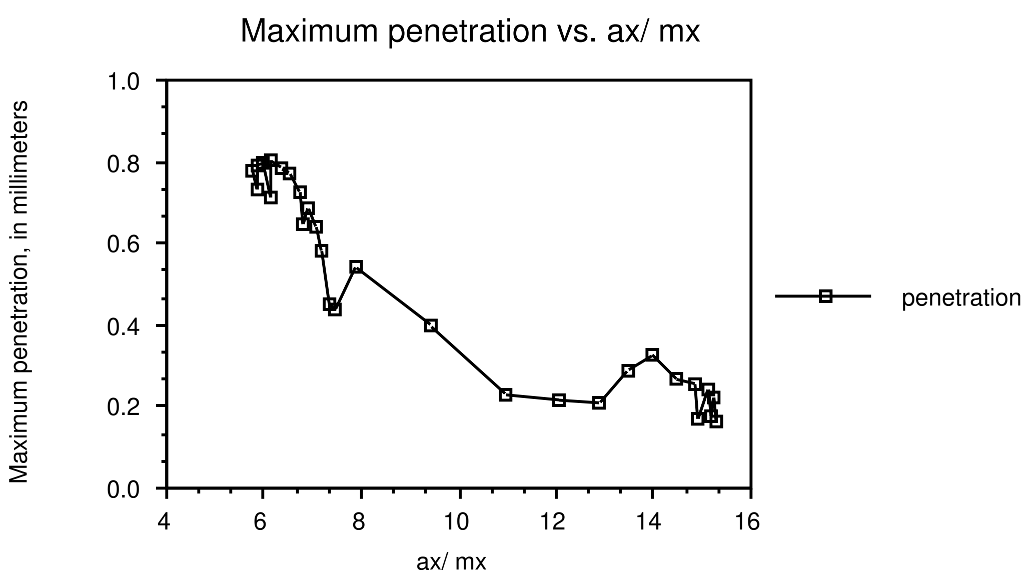
We have also found, however, that this weighted damping ratio criterion was not useful when the configuration was close to being singular. This was not readily apparent when the manipulator simulated was the three degrees-of-freedom planar manipulator (although the reader will note in Figure 1.5.5(a) that there are two slight disturbances in the plot, corresponding to locations where two joint frames "line up"), but it was quite apparent when the manipulator simulated was the seven degrees-of-freedom manipulator. Figure 1.5.6 shows a plot of the penetration vs. the weighted damping ratio, where the configurations have been contrived to have two or more joint frames "lined up" in the simulations.

The strategy breaks down for near-singular configurations because the joint velocities are no longer small, and thus our assumptions about Coriolis/centripetal torques, etc., being negligible are no longer valid. Furthermore, since in practice it is difficult to completely eliminate null motion, any error between the Jacobian needed to provide only net velocity and the calculated Jacobian (stemming from the limitations of the machine calculating the Jacobian) will increase from negligible to perhaps not-so-negligible, introducing a null velocity which cannot be ignored. Much has been written on how to "measure" how singular a configuration is, and how to avoid such configurations [5,6,12]. While it is not our intent to discuss singularity avoidance methods in this thesis, it is clear that such methods would need to be used to aid the success of this impact strategy if used on any actual manipulator that might impact an object in a near-singular configuration.

For our simulated ideal redundant manipulators, the strategy presented in this section allowed us to reduce the impact force between a manipulator and an object by changing the configuration of the manipulator in such a way that the effective mass is decreased and the effective damping is increased. From these results, we feel that it is safe to conclude that this strategy would be effective for reducing the impact force for an actual redundant manipulator.



(a)



(b)

Figure 1.5.5: Using the weighted damping ratio criterion (a_x/ m_x), a cleaner relationship is found with respect to the penetration (and impact force) experienced during an impact event Figure 1.5.5(a) shows the simulation results for the 3DOF planar manipulator, while 1.5.5(b) shows the results for the simulated 7DOF manipulator.

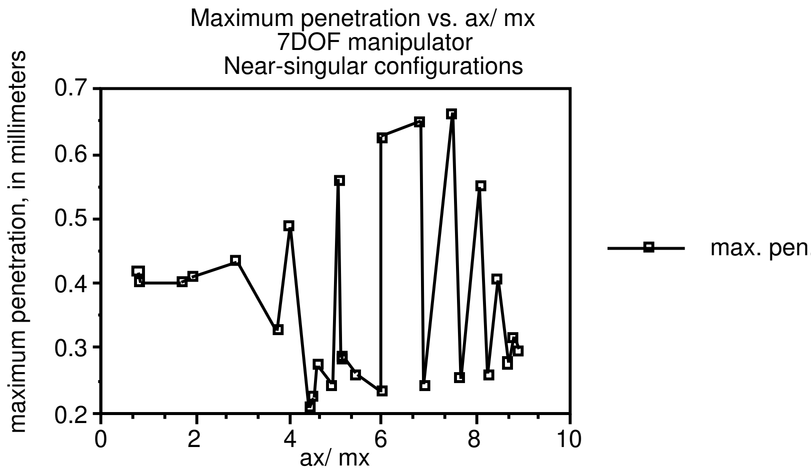


Figure 1.5.6: In this simulation, the weighted damping ratio criterion does not perform well for the 7DOF manipulator when the configurations tested are close to being singular, because our assumptions about small joint velocities are no longer valid. The increase in angular velocity also increases the joint torques, so it is best in any event to avoid configurations which are close to being singular, no matter what criterion is being used.

1.6. Conclusions

In this chapter, we have presented and discussed two strategies for minimizing the force of impact. When we applied these strategies to simulated redundant manipulators using ROBSIM, we were able to minimize the force of the impact between an end-effector and an object by applying a damping torque at the time of impact. Our results suggest that net motion should be heavily damped, while null motion should not, in order that the impact force should be decreased without affecting tasks which might require null motion. It was also seen in our simulations that certain configurations were better at minimizing impact force than others, and the response of a particular configuration could be gauged by examining a weighted version of the damping ratio of the idealized system, so long as the configuration was not close to being singular. Based on these simulation results, we feel it is safe to conclude that the strategies presented in this chapter would be effective at minimizing the force of impact on actual redundant manipulators.

2. Using ROBSIM to Simulate the Impact Problem

2.1. Introduction

In order to test our strategies for minimizing the impact force, it was necessary to use a simulation program, for several reasons. First, a simulator allowed us to run tests where the environment has zero gravity, something which is not possible on Earth when running tests using hardware. Second, running simulations minimized the chances of accidents, which might occur if, for instance, the configuration is near-singular. Third, using a simulator allowed us to use manipulators of varying geometries. Finally, and most importantly, using a simulator allowed us to run many tests in a shorter amount of time than possible with hardware, increasing the number of tests that could be run, allowing us to quickly verify the accuracy of any theory.

The simulator we chose to use is called the Robotic Systems Simulator (ROBSIM). The program is currently maintained by Langley Research Center. It was introduced in 1982, and over the past eight years has undergone several major evolutions as the needs of researchers change. ROBSIM produces outputs usable by such programs as Cricket Graph™ and MATRIXX™ to graph and analyze outputs, which enhanced its usefulness to us. It also possesses three dimensional graphic capabilities, which allow us to see the current condition of the manipulator as we might see it in a lab.

ROBSIM consists of two major subprograms. The first is called INITDRVR, and its purpose is to define a robotic system and its environment. The second is called SIMDRVR, and its purpose is to run the various simulations and report results. In this thesis, we will be dealing almost exclusively with the simulation aspects of ROBSIM.

Perhaps the most important feature of ROBSIM is its modularity. Every algorithm of ROBSIM is contained in its own subroutine. Thus, the code is easily updated and altered as necessary, and new capabilities can be grafted onto ROBSIM without affecting or changing code which should not be affected. This ease in reprogramming the simulator was of great benefit during our research, since many modifications to ROBSIM had to be made in order to properly research the impact problem.

ROBSIM was run on a VAXstation II/GPX. It runs under the VMS operating system, and the code is written mostly in FORTRAN.

Section 2.2 will discuss the modifications to ROBSIM that were needed to solve the problem. Section 2.3 will discuss the steps that were taken to test the damping torque strategy of Section 1.4.5. Section 2.4 will discuss the steps that were taken to test the configuration strategy of Section 1.5. Section 2.5 concludes this chapter of the thesis. Program listings are given in Appendix A, and the modifications to ROBSIM, along with a explanation for their use, are given in Appendix B.

2.2. Modifications to ROBSIM Used to Study Impact Phenomena

ROBSIM was written in FORTRAN, which the author finds to be a wholly unfriendly language. Furthermore, much of the code which is available for control is written in C. For instance, in The Robotics Institute at Carnegie Mellon University (of which the author is a member) the real-time system for controlling manipulators is written is C, and, wishing to use code from the existing control programs to increase compatibility, it became apparent that augmenting ROBSIM to run C-algorithms was a worthwhile task. ROBSIM had the capability, as every FORTRAN program does, to call C-executable files – what was missing was the ability to easily define variables in the C-routines so that information about the current state of the manipulator could be passed back and forth between ROBSIM and the C-algorithm. Additionally, much the code was written by other groups (Grumman Data Systems, Martin Marietta Aerospace) which had the NASA ROBSIM contract before Langley Research Center took direct control over the project, and this plurality of authors caused many of the routines and variables of ROBSIM to take on cryptic meanings.

To alleviate the situation, an interface between ROBSIM and the C-algorithms (hereafter referred to as *user-defined functions*, or *UDFs*) was created for the impact project, whereby the common-variable FORTRAN structures of the ROBSIM code were translated to C structures, for inclusion as appropriate into the UDFs. In all, two hundred and thirty-six matrices and variables relating to the control of robotic manipulators were made available to users wishing to write UDFs in C-language. Token functions were created to make access to the passed variables easy, and these were incorporated into what we call the ROBSIM Programming Token Language (RPTL).

As an example of the use of RPTL, consider the following piece of FORTRAN code:

```
        SUBROUTINE ABC123(AMAS_RCD)
        INCLUDE 'MXPRMS.TXT'
        INCLUDE 'AMASBK.STRUCTURE'
        RECORD /AMASBK/  AMAS_RCD

        WRITE(6,150) ((AMAS_RCD.ATMAS(I,A),I=1,MXLNKS+2), A=1,MXARMS)
150     FORMAT (F13.4)
```

This is the header for some user routine written in FORTRAN which for some reason requires information in the ROBSIM common record AMAS_RCD, the structure of which is stored in file *amasbk.structure*. The file *mxprms.txt* stores all of the ROBSIM system-wide constants, including MXLNKS, the maximum number of links a manipulator is allowed to have, and MXARMS, the maximum number of arms permitted to a system. In C, without RPTL and the C-interface, the above piece of code would look like the following code:

```
/*  note the indexing inversion between FORTRAN and C, and the offset
       of the indexing.  FORTRAN indexing starts at one by default and
       is column-major, while C indexing starts at 0 by default and is
       row-major */

#define MXARMS 5
#define MXLNKS 10
void abc123(amas_rcd)
struct {
       float
              atmas[MXARMS][MXLNKS+2];
              atcg[MXARMS][MXLNKS+2][3];
              ainmat[MXARMS][MXLNKS+2][3]
              effjt[MXARMS][MXLNKS];
              effkt[MXARMS][MXLNKS];
              fcoef[MXARMS][MXLNKS];
              dcoef[MXARMS][MXLNKS];
              cfcoef[MXARMS][MXLNKS];
       } *amasbk;

{
int i,a;

for (a=0;a<MXARMS;++a)
       for(i=0;i<MXLNKS+2;++i)
              printf("%g",amasbk->atmas[a][i]);
}
```

Several problems are evident with this code. First, if a user ever decides to change the values in the ROBSIM master value file *mxprms.txt*, then programmers will have to go through and change every C-program which includes values translated from that file (there are thirty-eight constants defined in *mxprms.txt*, and we find that, on average, about five need to be used in any particular UDF – in this case, MXARMS and MXLNKS). Second, for every program which uses AMAS_RCD, such as above, the entire definition of the structure pointed to by *amasbk* will have to be declared, even if only one variable or array is to be used from that structure. As many as forty such structures may be passed to a UDF (not merely one, as in this example), so this can create quite an overhead on memory space. Finally, the names of the ROBSIM variables are cryptic, and not helpful to understanding what the program does. By translating the FORTRAN structure and constant files to C, and adding in the token language RPTL, this same piece of code can be written as follows:

```
#include <mxprms.h>
#include <rptl.h>
void abc123(amas_rcd)
#include <amasbk.h>
{
int i,a;

for (a=0;a<MXARMS;++a)
       for(i=0;i<MXLNKS+2;++i)
              printf("%g",LINKMASS[a][i]);
}
```

The economy of this method of programming is clear. If any values in the ROBSIM master value file*mxprms.txt* are changed, then only the file *mxprms.h* needs to be changed to upgrade all of the C-routines, no matter how many the user has written. The space wasted by declaring the structure of the ROBSIM variable is regained, and it becomes clear that the information the user requires is the mass of the links of each arm. Programming UDFs in C is no longer as difficult, and thus the user now has the choice to program in either FORTRAN or C, whichever is more convenient.

These modifications, a guide to RPTL, and a more extensive explanation of how they work are included in Appendix B.

2.3. Simulating the Damping Torque Impact Problem

2.3.1. Introduction

Now that we have explained how routines on ROBSIM are created, we can discuss how the impact routines were simulated on ROBSIM. In Section 1.4.5, a strategy was given for minimizing the impact force by introducing a net damping torque, and results were shown that indicated that the strategy was successful under simulation. In this section, we will discuss how these results were obtained from ROBSIM for a simulated three degrees-of-freedom planar manipulator.

2.3.2. Programming the Strategy

The algorithm which calculates the dynamics of the robotic system is modular; that is, the routine DYNAM merely calls four other routines: CABSM, which calculates the absolute angular velocity, angular acceleration, and linear acceleration for each joint or link; FORCE, which calculates reaction forces at the joints due to inertial and gravity loads; TORQUE, which then calculates the joint reaction torques; and ACTORQ, which calculates the torque for each joint actuator. For the purposes of our strategy, the gear ratio of each actuator we tested was 1:1, so that the damping torque could be added directly to the actuator. (This was not necessary for testing the strategy, but is simply convenient, since adding torques in ROBSIM is easiest code-wise at the actuator.) DYNAM was altered to call a new function, COLLISION, after ACTORQ. COLLISION is a UDF written in C which is passed the inertia matrix and Jacobian of the system, and which returns the damping torque. [1] The matrix routines used in COLLISION were designed for use with CHIMERA II, a real-time operating system developed at The Robotics Institute, and have been optimized

1. In fact, DYNAM does not use the inertia matrix, since torques due to angular acceleration are calculated elsewhere in ROBSIM. The program that calls DYNAM had to be modified to pass the inertia matrix to DYNAM, and thence to COLLISION.

for speed. COLLISION uses static variables to increase the speed of the algorithm even more, since matrices are not created and destroyed on every call to the UDF.

The three-degrees-of-freedom manipulator used in the test is described in Table 1.1. It has been made as "ideal" as possible, and set in zero-gravity environment, in order to see the pure effects of the damping torque on the system.

Another test condition, as mentioned in Section 1.4.2, was the the manipulator should have zero null motion. That is, any motion in the joints should contribute solely to moving the end-effector. As implied in equation (12) and shown in [7,8,13], the joint velocity can be decomposed into net and null components:

$$\dot{\theta} = \dot{\theta}_{\pi} + \dot{\theta}_{\eta} \tag{41}$$

The null component is what we wish to suppress, so we choose a joint velocity coincident with the net velocity vector, which will have no null velocity, since net and null components are orthogonal. To determine what this joint velocity will be numerically, we arbitrarily selected some desired Cartesian velocity and solve for the joint velocity. The Premultiplier Diagram is especially useful here since the pseudoinverses it gives correspond the the pseudoinverses in [13], which Baillieul shows as having the property of projecting Cartesian velocities into net joint motion space:

$$\dot{\mathbf{x}} = \vartheta \dot{\theta}_{\pi} \tag{42}$$

$$\mathbf{J}^{-} \dot{\mathbf{x}} = \dot{\theta}_{\pi} \tag{43}$$

$$\dot{\theta}_{\pi} = \vartheta_{H}^{+} \dot{\xi} \tag{44}$$

To get the proper joint motion for the test, therefore, ROBSIM had to calculate the inertia matrix and Jacobian at the beginning of the trial, and from these calculate the initial conditions for the problem. In the tests for this impact strategy, the initial configuration of the manipulator was held constant in order to study the effects of damping torque in a controlled situation, so the initial joint velocities were the same in every case. In the configuration-based strategy (Sections 1.5 and 2.4), which also used velocities which were net-motion only for its trials, the initial configuration varied on every test, so the initial velocities had to be calculated previous to every simulation.

2.3.3. Results for the 3DOF Planar Manipulator, and Discussion

The damping torque strategy was tested on the planar manipulator against three different criteria: the value of h_1, the value of h_2, and the value of the stiffness of the environment. The configuration was initialized in every test to θ = (60, -60, -60) degrees (shown in Figure 1.2.2 (b)). The target point was 2.0 meters horizontal distance from the base of the manipulator, and the initial velocity was 0.1 m/sec in the horizontal direction. Each simulation began at the moment of contact.

Graphs showing the simulated response of the system versus h_1 have already been shown in Figures 1.4.2 and 1.4.4. As expected, as h_1 approached zero, the resulting decrease in net damping torque caused the manipulator to penetrate farther into the wall. At $h_1 = 0$, the system would be reduced to the mass-spring model of [4].

The strategy works well up to at least $h_1 = 18.0$. At this point, the initial joint torques at impact are (-181.7, 41.27, 132.2) n•m/rad, perhaps higher than the average manipulator is equipped to handle. While our experiments involving this strategy only involved the one simulated three degrees-of-freedom planar manipulator, the results agree with what we might expect should happen. It would not be unreasonable to assume that the strategy can be extended for use in various other redundant manipulators, choosing values of h_1 that do not saturate the actuators of the manipulators.

We can consider the system model given in Section 1.5.3 to see why this should be so. Recall that the the system was idealized to the differential equation (30), reprinted here as equation (45):

$$\ddot{\mathbf{x}} + \mathbf{J}\mathbf{H}^{-1}\mathbf{K}_D\mathbf{J}_H^{+}\dot{\mathbf{x}} + \mathbf{J}\mathbf{H}^{-1}\mathbf{J}^T\mathbf{K}_e\mathbf{x} = 0 \tag{45}$$

The damping torque is contained in this equation as:

$$K_\Delta\dot{\theta}_\pi \quad = K_\Delta\vartheta_H^{+}\dot{\xi} \tag{46}$$

We are simply multiplying the damping torque by a scalar, h_1. Equation (45) becomes:

$$\ddot{\mathbf{x}} + h_1\mathbf{J}\mathbf{H}^{-1}\mathbf{K}_D\mathbf{J}_H^{+}\dot{\mathbf{x}} + \mathbf{J}\mathbf{H}^{-1}\mathbf{J}^T\mathbf{K}_e\mathbf{x} = 0 \tag{47}$$

Finally, we reduce from matrix form to scalar form as we did in Section 1.5.3, by choosing our frames so that motion is along one axis only. This gives the equation:

(48)

$$\ddot{\mathbf{x}} + h_1 a \dot{\mathbf{x}} + b\mathbf{x} = 0$$

Unlike the situation of Section 1.5.3, the quantities *a* and *b* are constants, because they are configuration dependent, and we are using the same configuration for each test. If the stiffness of the wall is relatively high (like steel, as opposed to clay),then it is likely that $b > a^2/2$, and the system will be oscillatory. The damping ratio for the system is given by:

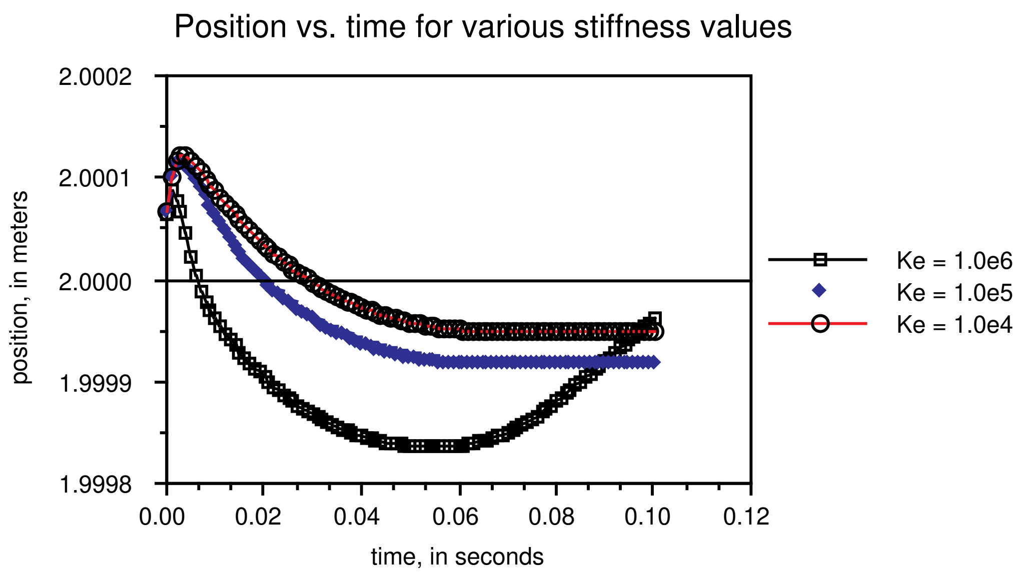


Figure 2.3.1: As the stiffness of the wall increases, the visible oscillation also increases. And, while the penetration decreases slightly when the stiffness increases, the damping ratio actually decreases as well, increasing the size of the undershoot from the wall.

(49)

$$\zeta = \frac{\eta_1 \alpha}{2\sqrt{\beta}}$$

It is clear from this that as h_1 increases, the damping ratio will increase, and the system will have less overshoot past the equilibrium point of **x**=2.0 m [11], thus incurring less impact force.

Since we confined ourselves to net motion only, and since null motion should not affect the position of the end-effector, the null portion of the damping torque had very little effect on the penetration, as shown in Figure 1.4.3. What little effect it did have is possibly attributable to machine error in calculating the net joint velocity, since h_2 had to be increased to an absurd value to cause the manipulator to change its impact penetration by even 0.1 millimeter. Thus we are justified in keeping the null component of the damping torque out of the characteristic equation of (47), since its effects were very small compared to the other terms.

As the stiffness of the environment changes, we would expect that the value *b* of equation

(48) would change, since $b = k/m$. Since the natural frequency of the system is given from [11] as:

$$\omega_\nu = \sqrt{\frac{\kappa}{\mu}} \qquad (50)$$

the rate of oscillation should increase as the stiffness of the environment increases. As Figure 2.3.1 shows, this was precisely the case in our simulations. For $h_1 = 10.0$, $h_2 = 0.01$, and the configuration set at $\theta = (60,-60,-60)$ degrees, the manipulator was impacted against three walls of

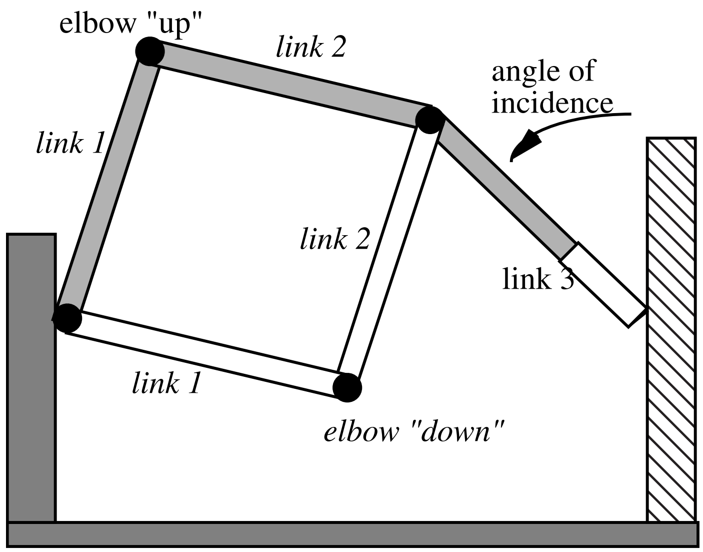


Figure 2.4.1: The 3DOF planar manipulator's configuration can be defined by two criteria: the angle of incidence and whether or not the configuration is "elbow up" or "elbow down."

varying magnitudes of stiffness. As the stiffness of the simulated wall was increased, the rate of oscillation increased. It can also be seen from our results that making a wall stiffer decreases the penetration into the wall only slightly so that, although penetration decreased, the stiffness of the wall actually caused the impact force to increase. The decrease in the damping ratio also caused more undershoot ("rebound") from the wall in our simulations.

The parameters for the simulated three degrees-of-freedom manipulator are listed in Table 1.1.

2.4. Simulating the Configuration Impact Problem

2.4.1. Introduction

The routines needed to test the configuration problem required little change from the code already developed for the damping torque problem of the previous section. The values of a_x and

m_x had to be calculated, but since the equations for those quantities have elements which are used to calculate the weighted inverse of the Jacobian (and hence, the net velocity vector), modifying the code to report these values was trivial.

Since the results of the experiment were fairly well documented in Section 1.5.3, this section will devote itself primarily to relating how those results were obtained, and how the theory was developed.

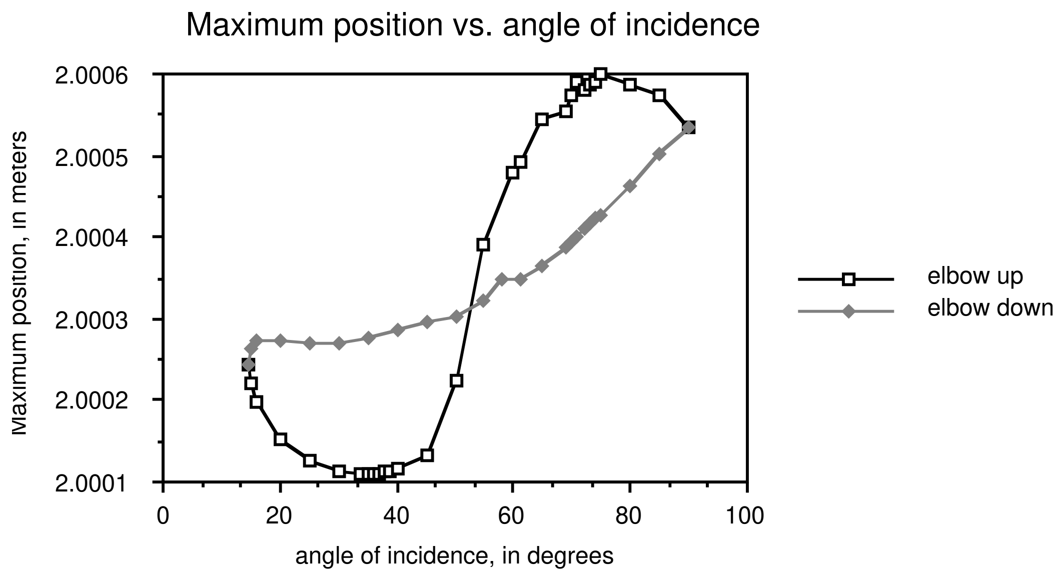


Figure 2.4.2: The angle of incidence was checked to see how it affected penetration. The lowest penetration occurs at approximately 38 degrees "elbow up," which is also where the maximum effective damping and minimum effective mass occur.

2.4.2. Trials with the 3DOF Planar Manipulator

The three degrees-of-freedom is redundant with respect to position, but not with respect to position and orientation combined. This fact makes it very easy to obtain a representative number of configurations for the trials. A configuration can be defined by its angle of incidence with the wall, and by whether its elbow is "up" (joint 2 away from the horizontal) or "down" (joint 2 near the horizontal), as shown in Figure 2.4.1. If the manipulator has three links of length one meter each, and the wall is two meters away from the base of the manipulator, then simple geometry reveals that for an angle of incidence with the wall, φ:

(51)

$$\theta_1 = \mathrm{Tan}_2^{-1}(\cos\varphi, 2 - \sin\varphi) \pm \mathrm{Cos}^{-1}(\xi)$$

$$\theta_2 = \pm\left(-\frac{\pi}{2} + \Sigma\iota\nu^{-1}(\xi) - \mathrm{X}o\sigma^{-1}(\xi)\right) \tag{52}$$

$$\theta_3 = -\frac{\pi}{2} - \theta_1 - \theta_2 + \varphi \tag{53}$$

where:

,

$$\xi = \frac{1}{2}\sqrt{\chi o\sigma^2\varphi + (2 - \sigma\iota\nu\varphi)^2} \tag{54}$$

θ_1 = angle from horizontal axis to link 1,
θ_2 = angle from extension of link 1 to link 2,
θ_3 = angle from extension of link 3 to link 3.

It can be shown that there are two solutions for every angle of incidence between (approx.) 14.5 and 165.5 (except in the singular case where links one and two are perfectly aligned, in which case there is only one configuration that is neither “elbow up” nor “elbow down”). Since we have been assuming that the gravity of the environment is zero, all configurations resulting from an angle of incidence of 90 + γ, γ being an arbitrary number of degrees less than 75.5, will react similarly to a configuration resulting from an angle of incidence of 90 - γ. Therefore, only configurations from 14.5 φ 90 need to be tested, for both “elbow up” and “elbow down” configurations, to cover all possible configurations for the three degrees-of-freedom planar manipulator.

Several variables were checked to see if a relationship existed between the configuration and the amount of penetration, including the effective damping (Figure 1.5.3), the effective mass (Figure 1.5.4), and even the angle of incidence (Figure 2.5.2). Theory and results indicated that a ratio of effective damping to the square root of the effective mass best described the system in simulation (as discussed in Section 1.5.3).

The parameters for the simulated three degrees-of-freedom manipulator are listed in Table 1.1.

2.4.3. Trials with the 7DOF Manipulator

No extra code was required to test the configuration strategy on the seven degrees-of-freedom manipulator. Unfortunately, though, this manipulator is redundant for tasks requiring both position and orientation control as well as those just requiring simple position control. This makes it very hard to get representative sample of the configurations possible to achieve a desired contact point, because orientation cannot be used to describe various configurations, as was done for the three degrees-of-freedom planar manipulator.

To get data for the experiment, a contact point was chosen, and a joint solution using ROBSIM was calculated, using a standard least-squares pseudoinverse technique. The program then executed motion along a trajectory created to reach the point with the desired configuration. There is a tendency for ROBSIM to generate trajectories which pass near singularities, so we modified the program in such a way that the end-effector was forced to pass through two other specified points before attempting to solve for the contact point. This gave a certain amount of control to how singular the final configuration would be, and also helped avoid passing near singularities to achieve the goal position.

To get a set of points, the contact point and two intermediate points were given to ROBSIM. Additionally, an orientation for the contact point was given. By altering the orientation (but not the position) of the contact point, different configurations could be found for the point. Specifically, in our trials, the contact point was rotated about the y-axis of the task frame from -90 to +90 degrees. The two intermediate points were the same in all trials. This generated the data shown in Figure 1.5.5(b). To generate the singular points of Figure 1.5.6, intermediate points were chosen such that the manipulator would be virtually guaranteed to have two joint frames aligned, and the contact point was rotated about the z-axis of the task frame to get different joint configuration solutions.

The results of Figure 1.5.5(b) tend to support the conclusions garnered from the three degrees-of-freedom experiment (see Sections 1.5.3 and 2.4.2), although, as mentioned, it is impossible to get a sampling of points representative of all configurations – the data would take more disk space than currently available on most systems. It is clear, though, that for the points gathered, the configuration with the largest weighted damping ratio was globally the best configuration to minimize the force of impact in our simulations.

The parameters for the simulated three degrees-of-freedom manipulator are listed in Table 1.1.

2.5. Conclusions

In this chapter, we discussed how the strategies to reduce impact force which were developed in Chapter 1 were implemented on the robotic simulator ROBSIM. An interface between ROBSIM and user-defined functions written in C was presented, and it was shown that this interface allowed control routines to be written in C in a manner that was non-cryptic, easily portable, and space-conservative. We also discussed how results from the simulations were obtained and interpreted, for both the three degrees-of-freedom planar manipulator and the seven degrees-of-freedom manipulator.

3. Conclusions and Future Research

This project introduced two strategies for minimizing the force of impact of redundant manipulators, and tested them using the simulator ROBSIM. The strategies were introduced theoretically, building on known control theory, and then tested. Code to test the strategies was written using a C-interface and a token language designed especially so that C-routines could run on ROBSIM.

Our strategies are based on a reduction of the impact system to a second order characteristic equation representing an impact of finite duration, with certain assumptions made about joint velocities and the operational frames. The first strategy relies on reducing the velocity into an object by applying a damping torque. This has the effect of reducing the momentum of the end-effector, and thereby reduces the impact force. The second strategy uses changes in the configuration to reduce the force of impact. Certain configurations have a lower effective mass and a higher effective damping than others, and it is shown in simulation that these configurations cause the lowest impact force. It is also shown that effective mass alone cannot pinpoint the best configuration, nor can effective damping alone.

The first strategy was tested on a simulated three degrees-of-freedom manipulator, and it was shown to be a robust algorithm. Damping torques were effective in reducing the force of impact for all reasonable values of torque in the simulations. The second strategy was tested on both a three degrees-of-freedom manipulator and a seven degrees-of-freedom manipulator. The strategy was shown to be effective in simulation for all configurations where the manipulator is not singular or near singular. Near a singularity, assumptions about joint velocities become invalid.

The code added to ROBSIM was able to successfully simulate the effects of an impact event. The C-interface allows code to be ported from C-libraries and used with very minor modifications, using the token language RPTL.

This thesis discusses the benefits of using redundant manipulators for an impact event. Also discussed are the benefits from using simulations to test theories and tasks.

There are several directions that future research might take. The most obvious is to test our strategies on a real manipulator, and compare the results to those garnered by simulation. To do this, one must be able to accurately build existing manipulators in ROBSIM. While ROBSIM does have this capability, it is still necessary to get the information about link lengths, masses, centers of gravity, inertia terms, and geometry, as well as actuator parameters.

Another area for research is the addition of discrete amounts of force to keep the manipulator from rebounding from the surface, as mentioned in Section 1.4.3 and in [10]. In this case, care must be taken to make certain that the total force at the end-effector is not too high, since this would negate our attempts to reduce impact force. Similarily, the addition of a stiffness torque as in Section 1.4.4 and [10] could also be tested.

Instead of using the inertia matrix for the weight in the pseudoinverses involved in damping, some other constant matrix might by used. This would end the interdependence of the effective mass and the effective damping of Section 1.5, and perhaps result in an even cleaner method of identifying the best configuration for impact.

Similarly, a strategy might be introduced which takes the Coriolis and centripetal forces into consideration, making the algorithm more robust around singularities. The inclusion of null motion, and the effects it might have on the strategies, should also be investigated.

Minimization of the duration of the surface compression time to reduce the length of time that the end-effector is endangered by excessive forces would also be a useful thing to do. Simply minimizing the definite integral of the pentration of the end-effector versus time would not be sufficient, since one could end up with a small compression duration but a large (if brief) penetration. Simultaneous reduction of the amount of penetration and reduction of contact time would have to be achieved for this to be worthwhile. Research into discovering which is more damaging to an end-effector – high force or prolonged compression duration – would give valuable clues as to how this simulataneous minimization should be balanced.

Finally, a great enhancement to the configuration theory would be to devise an algorithm to find the configuration with the lowest effective damping/effective mass ratio for a particular contact point. This would be fairly straightforward for a three degrees-of-freedom manipulator, and even for a six degrees-of-freedom manipulator (by taking advantage of the fact that these are redundant as concerns position, but nonredundant as concerns position and orientation together). However, for a seven degrees-of-freedom manipulator, this becomes more difficult. Besides the infinite number of configurations possible, one must also be concerned with the fact that it has six "elbows" which can be "up" or "down," introducing 64 possible types of configurations, irrespective of the number of trials needed to be taken for each possible type of configuration. If each joint is modified through 90 degrees (or whatever) for each type of configuration, even taking the trials every five degrees (for instance), the amount of computing needed to be done becomes prohibitive in size. Another concern is that the best configuration might be impossible to achieve; for instance, one of the joints might stray through a wall, in which case it is necessary to get "second best" configurations unrelated to the "best configuration," all of which increases the time the algorithm must run.

Appendix A. Program Listings

A.1. Introduction

Here follow the program listings for testing out the algorithms. Section A.1.1 lists the program *collision.c*, which calculates the damping torques applied during a collision, and adds the torque accrued from impacting with the wall. Section A.1.2 lists the program *collisionprep.c*, which was run before a test to calculate the effective mass, effective damping, and net joint speeds of the manipulator, based on the initial configuration.

Other programs were written for use with these tests. These include a program to calculate joints angles for the three degrees-of-freedom planar manipulator based on the angle of incidence, and a program to calculate and output the torques, position, and potential and kinetic energies of the manipulator. These programs are not listed here.

The matrix routines used in these programs are part of the FastMat package available with the CHIMERA II real-time operating system, and were adapted for use on the VMS operating system by the author. These routines are not listed in this paper.

A.1.1: collision.c

```
/* collision.c

       This routine simulates the collision of the end-effector with some semi-rigid
boundary.  Although the boundary does not "really" exist in ROBSIM, the force of such a
collision is counterbalanced so that the only motion in the direction of the barrier is
attributable to forces of compression.  That is, the force on the end-effector from the
collision with the wall is calculated based on the position of the end-effector relative
to the boundary, and is transformed into joint torques and fed back to the manipulator's
joints to simulate the presence of a wall.

       The matrix routines used in the following programs were written by Don Schmitz for
CHIMERA, and later amended by Dave Stewart for CHIMERA II. Further modifications to adapt
them to the VAX's microprocessor were made by Matt Gertz.  The name of the matrix routines
package is "fastmat," and the routines are designed for speed.  These matrix routines
can, of course, be replaced by other routines, if the user cares to write his or her own
code.


**********************************************************

THEORY:

       The state-space dynamic equation for any manipulator is:

              tor = M(th) thdd + V(th,thd) + G(th)(1)

       where:

              th = n x 1 vector of joint thetas
              thd = n x 1 vector of joint omegas
```

thdd = n x 1 vector of joint angular accelerations
M(th) = n x n mass matrix
V(th,thd) = n x 1 vector of centrifugal and Coriolis terms
G(th) = n x 1 vector of gravity terms
tor = n x 1 vector of torque at each joint

M, V, and G are based on physical properties of the manipulator, such as inertia, center of gravity, mass, and link length. It is assumed that both the theta and omega are known. Therefore, given the joint torque vector, one can find the angular acceleration of each joint; or, knowing the angular accelertion, one can find the torque vector. Typically, though, the torque is known and we wish to find the angular acceleration.

For this experiment, the state-space dynamics equation is altered to best utilize the redundancy of the manipulator. Although the primary function of this program is to calculate the torque at the joints, we are in fact most interested in the resulting change of position of the end-effector in Cartesian space. Since F = k dx, the change in position will be a good indicator as to whether or not our algorithm is reducing the force (and hence the energy) of the collision of the manipulator with the boundary. Certain parameters will be varied to try to discover the best way to preserve the end-effector.

The new state-space equation is (solved for angular acceleration):

```
thdd(t + dt) = inv(H)*(-C(th(t + dt),thd(t + dt)) - trans(J)*Fe - tor_d')  (2)

       where
              H = M,
              C = V { + G },
              J = the jacobian of the manipulator,
              Fe = the force at the end-effector, which is Ke*dxe (3)
                     if dxe > 0, and zero otherwise,
              tor_d' = h1*trans(J)*trans(JHp)*Kd*thd(t+dt)(4)
                     + h2*(Kd*thd(t+dt) - trans(J)*trans(JHp)*Kd*thd(t+dt)),
                     referred to as the damping torque
              h1 and h2 = known control variables, which control the amount of
                     damping torque applied in the net and null directions,
              Kd = a variable spring stiffness,
              Ke = end-effector and boundary stiffness,
              JHp = inv(H)*trans(J)*inv(J*inv(H)*trans(J))(5)
```

The program flow will be as follow:

1. Obtain H. I'll also create inv(H).

2. I will find the position of the end-effector, and compare it with the x-location of the “wall.” I can then solve for Fe. OUTPUT: dxe is our output.

3. Solve for JHp.

4. Solve for tor_d'.

5. Add tor_d' to the existing joint torques, and add JT*Fe.

****
WARNINGS:
****

NOTE that the following items are assumed:

-the joints are all ROBSIM hinge joints
-theta(1) is measured from the world "x" axis
-theta(i+1) is measured from a line extending from link i

```
-the gear ratios must be 1:1
Nothing is assumed about the mass of the links, their lengths, their moments of inertia,
their centers of gravity, or the gravity of the environment.

Another thing that the user should be aware of is that this algorithm will work well only
if the time frame in which it operates is "continuous."  The processing step size should
be small enough so that the joint position changes from one step to the next will be very
small.  For instance, if the initial velocity of the end-effector is 0.1 m/s, then a
processing step size of 0.01 seconds would be appropriate.  Obviously, choosing a too-
small processing step would eat up a lot of the programmer's time, while choosing a too-
large processing step would result in impossible forces being generated at the contact
site (because the end-effector will have moved impossibly far into the "wall").

RETURNS:
        Nothing (void).

SIDE EFFECTS:
        The record forcbk is updated with the correct joint torques.

NOTE:
        This routine must be called from DYNAM2 after ACTORQ is called.
************************
HISTORY:
        collision.c:   Written by Matthew Gertz
                            Advanced Manipulators Laboratory
                            The Robotics Institute
                            Dept. of Computer and Electrical Engineering
                            Carnegie Mellon University

        First editing:Monday, August 7, 1990
        Latest editing:Wednesday, September 5, 1990

        15-Aug-1990:  Changed it so that it only supplies the torque.
                      The rest (solution of thdd) is done in ROBSIM.

        30-Aug-1990:  Fixed numerous bugs.

        02-Sep-1990:  Changed it so that it adds the torques to ACTUATOR_TORQUE
                      (force->tac), not JOINT_TORQUE.  JOINT_TORQUE is in
                      world coordinates, and therefore not at all suitable
                      for my purposes.  This means that the gear ratio must
                      be 1:1, though.  Also changed calling from TORQUE to
                      DYNAM2.

        03-Sep-1990:  Changed it so tor_d' was added, not subtracted.

        05-Sep-1990:  Changed it so JT*Fe was added, not subtracted.  This
                      seems to have completely fixed the program.

*********************
REFERENCES:

Craig,John J.   _Introduction To Robotics: Mechanics and Control_,
Addison-Wesley Publishing Company, Inc.  c. 1989.  Relevant reading: Chapter
Six.

*********************
*/



#include <stdio.h>
#include <fastmat.h>/* the matrix routines */
#include "mxprms.h" /* ROBSIM parameters */
#include <math.h>
#include "robsim_tokens.h"/* RPTL for easy programming */
```

```
/* The following definitions are usually entered on the command line instead of in this
file, since they are changed so often */

/* #define H1 5.0*/        /* range: 0.1 ... 5 */
/* #define H2 0.01*//* range: 0.01... 1 */
/* #define KE 100000.0  */

/* the following definitions were used for the 7DOF manipulator*/
#define KD11 70.0
#define KD22 60.0
#define KD33 50.0
#define KD44 40.0
#define KD55 30.0
#define KD66 20.0
#define KD77 10.0
#define M 3
#define N 7
#define XWALL 2.1

/* the following definitions were used for the 3DOF manipulator */
/* #define KD11 200 */
/* #define KD22 150 */
/* #define KD33 100 */
/* #define M 2 */
/* #define N 3 */
/* #define XWALL 2.0 */

void jac_fort_to_c(jacobbk,jacob_c,karm)/* jacobian transfer */
#include "jacobbk.h"
Fmat *jacob_c;
int karm;

/* The Fortran matrices have elements of four-byte precision and are
   addressed in column-major form.  FastMat's are of eight-byte precision,
   and are addressed in row-major form. This function and the others like it
   non-destructively tranfer one to the other. */

{
int i,j;                    /* counters */
for (i=0;i<(jacob_c->m);++i)
    for (j=0;j<(jacob_c->n);++j)
       dereFmat(jacob_c,i,j) = JACOBIAN(karm,j,i);
}


void theta_fort_to_c(systbk,theta_c,omega_c,karm)
#include "systbk.h"
Fmat *theta_c, *omega_c;
int karm;

{
int i;                      /* counter */

for (i=0;i<(theta_c->m);++i) {
    dereFmat(theta_c,i,0) = LINK_THETA(karm,i);
    dereFmat(omega_c,i,0) = LINK_OMEGA(karm,i);
    }
}


void collision(forcbk,jacobbk,systbk,user_data,dynmbk)
#include "forcbk.h"
#include "jacobbk.h"
#include "systbk.h"
#include "user_data.h"
#include "dynmbk.h"
```

```
{
	static Fmat theta,/* joint position */
		omega, /* joint velocity */
		force, /* force at the end-effector */
		h,		/* the mass (inertia) matrix */
		hi,		/* the inverse of the mass matrix */
		j,		/* the jacobian */
		jt,		/* the transpose of the jacobian */
		kd,		/* the stiffness matrix */
		td,		/* tpr_d' */
		tdp,		/* the particular tor_d' */
		tdh,		/* the homogeneous tor_d' */
		jhp,		/* the mass-weighted pseudoinverse */
		scratch,/* scratch matrix used in calculations */
		scratch2,/* scratch matrix used in calculations */
		scratch3,/* scratch matrix used in calculations */
		jhpt;		/* jhp's transpose */

	double x_position,/* x-coordinate (world) of end-effector */
		 delta_x;/* the change in position of end-effector */
	int	karm,		/* the arm in use */
		i,k;		/* counters */

	static unsigned char first_pass = TRUE;

	karm = USER_INPUT_INT(0)-1;/* get arm number */
	if (first_pass){
		first_pass = FALSE;/* allocate matrices on first pass */
		newFmat(&theta,N,1);
		newFmat(&omega,N,1);
		newFmat(&h,N,N);
		newFmat(&hi,N,N);
		newFmat(&j,M,N);
		newFmat(&jt,N,M);
		newFmat(&force,M,1);
		newFmat(&kd,N,N);
		newFmat(&td,N,1);
		newFmat(&tdh,N,1);
		newFmat(&tdp,N,1);
		newFmat(&jhp,N,M);
		newFmat(&scratch,M,M);
		newFmat(&scratch2,N,M);
		newFmat(&scratch3,N,N);
		newFmat(&jhpt,M,N);
		for (i=0;i<N;++i)
			for (k=0;k<N;++k)
				dereFmat(&kd,i,k) = 0.0;
		dereFmat(&kd,0,0) = KD11;
		dereFmat(&kd,1,1) = KD22;
		dereFmat(&kd,2,2) = KD33;
		dereFmat(&kd,3,3) = KD44;
		dereFmat(&kd,4,4) = KD55;
		dereFmat(&kd,5,5) = KD66;
		dereFmat(&kd,6,6) = KD77;
		USER_INPUT_FLT(0)=KE;/* save the stiffness of the
						wall for ROBSIM to use in
						output calculations */
		}

	theta_fort_to_c(systbk,&theta,&omega,karm);/* get theta & omega */

	jac_fort_to_c(jacobbk,&j,karm);/* get jacobian */

	/* Step 1 - get H */

	for (i=0;i<N;++i)
	   for (k=0;k<N;++k)
		dereFmat(&h,i,k) = dynmbk->phi[k][i];
		/* actually, the FORTRAN/C transposition reversal isn't necessary
```

```
        in this case because h must be symmetric */

    /* step 2 - calculate dx and Fe */

    delta_x = LINK_POSITION(karm,N+2,0) - XWALL;
                        /* if dx is positive, we are compressing */
    if (delta_x > 0.0)
        dereFmat(&force,0,0) = KE * delta_x;
    else
        dereFmat(&force,0,0) = 0.0;
    for (i=1;i<M;++i)
        dereFmat(&force,i,0) = 0.0;



    /* Step 3 - calculate JHp */

    Fmatcopy(&h,&hi);          /* take the inertia matrix */
    Fmatinvert(&hi,FULL_PIVOTING);/* and invert it */
    Fmatranspose(&j,&jt);/* get the jacobian transpose */
    Fmatmul(&j,&hi,&jhpt);/* use jhpt as scratch to get jhp */
    Fmatmul(&jhpt,&jt,&scratch);
    Fmatinvert(&scratch,FULL_PIVOTING);
    Fmatmul(&jt,&scratch,&scratch2);
    Fmatmul(&hi,&scratch2,&jhp);/* this is the mass-weighted
                                           pseudoinverse of the jacobian */
    Fmatranspose(&jhp,&jhpt);/* and this is its transpose */

    /* Step 4 - calculating td */
    Fmatmul(&kd,&omega,&td);/* use td as scratch to get torque
                                           due to angular velocity */
    Fmatmul(&jt,&jhpt,&scratch3);
    Fmatmul(&scratch3,&td,&tdp);/* this is the net part */
    Fmatsub(&td,&tdp,&tdh);/* this is the null part */
    Fmatscalarmul(&tdp,H1);/* scale the net part */
    Fmatscalarmul(&tdh,H2);/* scale the null part */

    Fmatadd(&tdh,&tdp,&td);/* calculate torque due to impact */
    Fmatmul(&jt,&force,&tdp);/* use tdp as scratch to hold it */

    /* Step 5 - apply the calculated damping torque and torque due to
        impact to the joint actuators */
    for (i=0;i<N;++i)
        ACTUATOR_TORQUE(karm,i) += dereFmat(&td,i,0) + dereFmat(&tdp,i,0);
}
```

A.1.2. collisionprep.c

```
/* collisionprep.c

	This program calculates.

*/

#include <stdio.h>
#include <fastmat.h>
#include <math.h>
#include "mxprms.h"
#include "robsim_tokens.h"

/* the following definitions were used for the 7DOF manipulator */
#define M 3
#define N 7
#define KD11 70.0
#define KD22 60.0
#define KD33 50.0
#define KD44 40.0
#define KD55 30.0
#define KD66 20.0
#define KD77 10.0

/* the following definitions were used for the 3DOF manipulator */
/* #define KD11 200 */
/* #define KD22 150 */
/* #define KD33 100 */
/* #define M 2 */
/* #define N 3 */

#define XVEL 0.1/* m/sec cartesian */
#define YVEL 0.0/* m/sec cartesian */
#define ZVEL 0.0/* m/sec cartesian */

void damping_output(jacobbk,dynmbk,systbk)
#include "jacobbk.h"
#include "dynmbk.h"
#include "systbk"
{
	static Fmat j,/* jacobian */
			hi,		/* inverse joint mass (inertia) matrix */
			jt,		/* transpose(j) */
			kd,		/* inverse joint damping matrix */
			ax,		/* cartesian exponent matrix */
			scratch22,
			scratch23,
			scratch32a,
			scratch32,
			omega, /* the resulting joint velocities */
			dx,		/* the cartesian velocity */
			theta; /* joint configuration */

	int i,k,		/* counters */
		karm;		/* the arm number */
	static int counter;
	double mxx;		/* the effective mass */

	FILE *outfile;

	static unsigned char first_pass = TRUE;

	if (first_pass){
		first_pass = FALSE;
		counter = 1;
		newFmat(&kd,N,N);
		for (i=0;i<N;++i)
```

```
                for (k=0;k<N;++k)
                        dereFmat(&kd,i,k) = 0.0;
         dereFmat(&kd,0,0) = KD11;
    dereFmat(&kd,1,1) = KD22;
    dereFmat(&kd,2,2) = KD33;
    dereFmat(&kd,3,3) = KD44;
    dereFmat(&kd,4,4) = KD55;
    dereFmat(&kd,5,5) = KD66;
    dereFmat(&kd,5,5) = KD77;
          newFmat(&j,M,N);
          newFmat(&jt,N,M);
          newFmat(&hi,N,N);
          newFmat(&ax,M,M);
          newFmat(&scratch22,M,M);
          newFmat(&scratch23,M,N);
          newFmat(&scratch32,N,M);
          newFmat(&scratch32a,N,M);
          newFmat(&theta,N,1);
         newFmat(&omega,N,1);
         newFmat(&dx,M,1);
         dereFmat(&dx,0,0) = XVEL;/* the cartesian velocity */
         dereFmat(&dx,1,0) = YVEL;/* the cartesian velocity */
          if (M>2)
            dereFmat(&dx,2,0) = ZVEL;
          }

/* this program will be called three times by ROBSIM: beginning, end, and during the run
(which itself only lasts one time unit, as we are just getting the mass, etc., and not
running the actual impact simulation yet).  For brevity's sake, have it output the data
only once, and it will save time later (we won't have to clean up the output file) */

    if (counter==3){
      counter=0;

      karm = USER_INPUT_INT(0)-1;/* get arm number */

      for (i=0;i<M;++i)          /* get the jacobian */
           for (k=0;k<N;++k)
                dereFmat(&j,i,k) = JACOBIAN(karm,k,i);

      for (i=0;i<N;++i)          /* get the inertia matrix */
          for (k=0;k<N;++k)
           dereFmat(&hi,i,k) = dynmbk->phi[k][i];

      for (i=0;i<N;++i)
           dereFmat(&theta,i,0) = LINK_THETA(karm,i);
                                            /*get the configuration */

      Fmatranspose(&j,&jt);/* get the jacobian transpose */

      Fmatinvert(&hi,FULL_PIVOTING);/* invert the inertia matrix */

      Fmatmul(&j,&hi,&scratch23);/* mxx = inv(j*hi*jt) */
      Fmatmul(&scratch23,&jt,&scratch22);
      Fmatinvert(&scratch22,FULL_PIVOTING);
      mxx = dereFmat(&scratch22,0,0);/* this is the effective mass */

      Fmatmul(&jt,&scratch22,&scratch32);/* jhp = jt*hi*inv(j*hi*jt) */
      Fmatmul(&hi,&scratch32,&scratch32a);/* this is jhp */

      Fmatmul(&scratch32a,&dx,&omega);/* jhp*dx = net joint velocity */

      Fmatmul(&kd,&scratch32a,&scratch32);
      Fmatmul(&hi,&scratch32,&scratch32a);
      Fmatmul(&j,&scratch32a,&ax);/* j*kd*jhp = effective damping */

      /* create the joint velocity file, also output as comments the effective
```

```
          mass and the efective damping */

    outfile=fopen("joints2.com","w");
    fprintf(outfile," ! a11 = %18.16g \n",dereFmat(&ax,0,0));
    fprintf(outfile," ! mxx = %18.16g \n",mxx);
    for (i=0;i<N;++i){
        fprintf(outfile," %18.16g\n",dereFmat(&theta,i,0));
        fprintf(outfile," %18.16g\n",dereFmat(&omega,i,0));
        fprintf(outfile," 0.00000000e+00\n");
    }
    fclose(outfile);
  }
  counter+=1;
}
```

Appendix B. ROBSIM Preprocessor Token Language

B.1. How to Use RPTL and the C-interface

Not every user of ROBSIM will be familiar with the FORTRAN 77 programming language, and others prefer using a less-structured language. In order to ease the creation of C-routines for ROBSIM, a set of preprocessor tokens have been devised. Because these macros are interpreted during the compilation time, and not during run-time, the speed of the executable produced does not suffer.

The list of macros given is by no means complete. They only access the records typically used by user-defined functions, and do not include those records which are used solely by the ROBSIM core.

Because the macros reference variable locations, they can be used to store values as well as retrieve them. For instance,

```
var1=RJACOB(ARM1,0,1);
```

assigns the value of the element contained in the first row, second column of the first arm's jacobian to *var1*, while

```
RJACOB(ARM3,4,6)=var2;
```

assigns the value of *var2* to the element in the fifth row, seventh column of the jacobian of the third arm.

For the macros to work, at least two files must be included:

```
#include <mxprms.h>/* this one must precede all others */
#include <robsim_tokens.h>
```

These lines must appear before any C code in the file (i.e., put them in the same place that the *stdio.h* file is included). *mxprms.h* must be included even if you are not using the ROBSIM tokens defined below, as the information in it is needed to interface between FORTRAN and C programs.

There are also two other types of "include" files. The first type are called **bk** files (*amasbk.h*, *systbk.h*, etc.). These files each contain one structure definition, and are used as follows:

```
c_type a_function(amasbk,some_other_var)
#include <amasbk.h>
a_type some_other_var;
```

Note that *amasbk* must be called "amasbk" throughout your program, since it is named so in the "include" file. (You can, of course, change the **bk** files so that the structures are only defined but not named yet. You would then include the files the same place that mxprms.h is included, and the function call would say `struct whatever amasbk;` instead of `#include <amasbk.h>`)

The other type of file is *struct_defs.h*. Its file format is:

```
/* struct_defs.h */
#include <amasbk.h>
#include <basgmbk.h>
#include <chkvelbk.h>
```

(etc.)

This include file is used in connection with the defined token STRUCT_DEFS, which is mentioned in section B.2.1.

Section B.2 will provide a "user's manual" for the RPTL tokens. Section B.3 gives a listing of the C-interface files.

B.2. Macros Defined in robsim_tokens.h (RPTL)

B.2.1. Token Definitions in RPTL

These macros are used as constants, not pseudofunctions.

```
#define X_COORD 0
#define Y_COORD 1
#define Z_COORD 2
#define ROLL 5 /* rot(z) */
#define PITCH 4/* rot(y) */
#define YAW 3  /* rot(x) */
#define ARM1 0
#define ARM2 1
#define ARM3 2
```

```
#define ARM4 3
#define ARM5 4
```

For example, the line POS(ARM1,3,X_COORD) would return the x-coordinate of the position of link 3 of arm 1.

```
#define STRUCT_LIST amasbk, basgmbk, chkvelbk, cnstbk, cntrbk, \
 complbk, envtbk, forcbk, fortorbk, geombk, jacobbk, loadbk, \
 motorbk, movbasbk, outrefbk, segmbk, sensbk, sprgbk, systbk, \
 targbk, timebk, toolbk, user_data
```

This macro allows ease in passing ROBSIM structure pointers. It only works if *all* of the records/structures listed in the definition are passed to the C program (since pointers take up very little space, this is not a bad assumption). The resulting type declarations are held in the file `struct_defs.h` (actually, `struct_defs.h` simply holds more preprocessor commands which cause all of the relevant include files to be processed). The definition and file are used like so:

```
void my_function(STRUCT_LIST, var1,var2)
#include <struct_defs.h>
a_type var1;
b_type var2;
```

where *var1* and *var2* are non-ROBSIM variables.

The user is encouraged to alter the STRUCT_LIST to suit his or her own needs, or to make new macros including only the needed records. WARNING: any change in STRUCT_LIST must be reflected in struct_defs.h!

B.2.2. Pseudofunction Definitions in RPTL

Here follows a description of the pseudofunctions (token macro functions) defined in the file robsim_tokens.h. They are organized by structure type. A word of warning: if you decide to forego the use of the definitions X_COORD, ARM1, and so forth defined above, be aware that in FORTRAN, the first element in an array is element 1, whereas in C it is element 0. Thus, arm 1 is arm 0 in C. However, link i may be link i-1, link i, or link i+1 depending on the context (sometimes the base is counted as a link, sometimes not).

In all definitions, **n** refers to the number of links in the system (1..10) and **m** refers to rank of the jacobian (1..6).

All arguments to the macro functions are integer quantities. The arguments used in the documentation are, with their ranges:

a, the arm number: a=0 is the first arm, a=MXARMS-1 is the last arm;
c, the column number: c=0 is the first column; ***or***
c, the constraint number: c=0 is the first constraint;
d, the object number: d=0 is the first object;
e, the range end: e=0 is limit minimum, e=1 is limit maximum;
f, the link frame number: f=0 is the base, f=k, 1≤k≤n is link k, f=n+1 is the tool, and f=n+2 is the reference point of the end-effector;
i, the link number: i=0 is the base, i=k, 1≤k≤n is link k, and i=n+1 is the tool;
j, the joint number: j=0 is joint 1, j=n-1 is joint n;
p, the point-mass number: p=0 is the first point mass; ***or***
p, the component of a vector describing a plane: p=0 is the x component, p=1 is the y component, p=2 is the z component, and p=3 is the w scale value of the plane;
q, the order of the polynomial: q=k refers to the kth order coefficient;
r, the row number: r=0 is the first row;
t, the target number: t=0 is the first target;
s, the load number: s=0 is the first load;
u, the axial direction: generally u=0, 1, or 2 indicates translation along the x, y, or z axis respectively, while u=3, 4, or 5 indicates rotation around the x, y, or z axis, respectively. Occasionally, the function in question will be translational or rotational only, in which case u=0, 1, or 2 indicates translation or rotation along or around the x, y, or z axis, respectively, as appropriate;
x, an integer: see the function's description for the boundary.

amasbk

These functions are concerned with the mass, inertia, and friction of the current system.

LINK_MASS(a,i)
requires: a=(0..MXARMS-1) {arm number}
i=(0..n+1) {link number: 0=base, 1≤i≤n=link,n+1=tool}
references: float
The total mass of the link between joint i and joint i+1 of arm a is referenced.

LINK_CG(a,i,u)
requires: a=(0..MXARMS-1) {arm number}
    i=(0..n+1) {link number: 0=base, 1≤i≤n=link,n+1=tool}
    u=(0..2) {directional vector: x=0,y=1,z=2}
references: float
The component of the center of gravity in the direction of u between joint i and joint i+1 for arm a in joint i system is referenced.

LINK_INERTIA(a,i,r,c)
requires: a=(0..MXARMS-1) {arm number}
    i=(0..n+1) {link number: 0=base, 1≤i≤n=link,n+1=tool}
    r=(0..2) {row number: 0=x row,1=y row, 2=z row}
    c=(0..2) {column number: 0=x column,1=y column, 2=z column}
references: float
References the (r,c) element in the inertia matrix for arm a, link i, in local coordinates. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element Ixy of link i in the first arm is referenced as LINK_INERTIA(ARM1,i,0,1) and not LINK_INERTIA(ARM1,i,1,0).

EFFJT(a,i)
requires: a=(0..MXARMS-1) {arm number}
    j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the effective moment of inertia of the armature and motor gear for a particular joint in an arm.

EFFKT(a,i)
requires: a=(0..MXARMS-1) {arm number}
    j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
The effective viscous frictional coefficient of the armature and motor gear for a particular joint on an arm is referenced.

FRICTION(a,j)
requires: a=(0..MXARMS-1) {arm number}
    j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}

references: float
The friction coefficient for a joint in an arm is referenced.

VISCOUS(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: $0 \le j \le n-1$=joint j+1}
references: float
References the viscous friction coefficient for a joint on an arm.

COULOMB(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: $0 \le j \le n-1$=joint j+1}
references: float
References the coulomb friction coefficient for a joint in an arm.

basgmbk

These macros return values which have to do with the current status of the base of the manipulator. These variables are not well documented in the ROBSIM structures, so there may be some descriptive errors.

IBRATE(a)
requires: a=(0..MXARMS-1) {arm number}
references: int
This call references a flag which tells how the base on a manipulator is being controlled. If a 1 is referenced, the base is under rate control; if a 2 is referenced, the base is under position control.

BASE_SEGMENT_START_TIME(a)
requires: a=(0..MXARMS-1) {arm number}
references: float
The start time of the current segment of base motion for a manipulator is referenced from this macro. The time referenced is absolute with respect to the simulation start time, not relative.

BASE_MOTION(a,u,q)
requires: a=(0..MXARMS-1) {arm number}
u=(0..5) {direction}
q=(0..MAXORD) {order}

references: float
References the qth order coefficient of the polynomial describing the motion of the base in all six directions of an arm, where u=0,1, or 2 refers to a translation along the x, y, or z axis respectively, and u=3, 4, or 5 refers to a rotation around the x, y, or z axis respectively.

BASE_ATTRIBUTES(a,u)
requires: a=(0..MXARMS-1) {arm number}
u=(0..5) {direction}
references: float
References the current position/orientation of the base of a manipulator, where u=0, 1, or 2 refers to a translation along the x, y, or z axis respectively, and u=3, 4, or 5 refers to a rotation around the x, y, or z axis respectively.

BASE_ROTATION(a,r,c)
requires: a=(0..MXARMS-1) {arm number}
r=(0..2) {row number: 0=x row,1=y row, 2=z row}
c=(0..2) {column number: 0=x column,1=y column, 2=z column}
references: float
References the (r,c) element in the 3x3 orientation matrix of the base of a manipulator. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element Ayz in the first arm's orientation matrix is referenced as BASE_ROTATION(ARM1,1,2) and not BASE_ORIENTATION(ARM1,2,1).

BASE_DELPOS(a,u)
requires: a=(0..MXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the change in base position of a manipulator along axis u since the current base motion segment began.

BASE_DELORI(a,u)
requires: a=(0..MXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the change in base orientation of a manipulator around axis u since the current motion segment began.

BASE_INITPOS(a,u)
requires: a=(0..MXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the base position of a manipulator along axis u as it was at the start of the current base motion segment.

BASE_INITROT(a,r,c)
requires: a=(0..MXARMS-1) {arm number}
r=(0..2) {row number: 0=x row,1=y row, 2=z row}
c=(0..2) {column number: 0=x column,1=y column, 2=z column}
references: float
References the (r,c) element in the 3x3 orientation matrix of the base of a manipulator as it was at the beginning of the current base motion segment. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element Azx in the second arm's starting orientation matrix is referenced as BASE_INITROT(ARM2,2,0) and not BASE_INITROT(ARM2,0,2).

BPH(a)
requires: a=(0..MXARMS-1) {arm number}
references: float
No documentation available.

TBDR(a,u)
requires: a=(0..MXARMS-1) {arm number}
u=(0..5) {direction}
references: float
No documentation available.

BASE_VELOCITY(a,u)
requires:a=(0..MXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the component of the linear velocity vector for the base of a manipulator along the direction of the u axis.

BASE_OMEGA(a,u)
requires:a=(0..MXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the component of the absolute angular velocity vector for the base of a manipulator around the u axis.

BASE_ACCEL(a,u)
requires:a=(0..MXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the component of the linear acceleration vector for the base of a manipulator along the direction of the u axis.

BASE_OMEGA_DOT(a,u)
requires:a=(0..MXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the component of the absolute angular acceleration vector for the base of a manipulator around the u axis.

BASE_FORCE(a,u)
requires:a=(0..MXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the component of the linear force vector for the base of a manipulator along the direction of the u axis.

BASE_TORQUE(a,u)
requires:a=(0..MXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the component of the absolute torque vector for the base of a manipulator around the u axis.

chkvelbk

These two macros return information about solutions to manipulator tasks.

IAPR(a)
requires:a=(0..MXARMS-1) {arm number}
references: int
This macro references a flag indicating whether the equations were solved exactly or not. If a zero is referenced, the solution is exact. If a 1 is referenced, the solution is not in reachable space. If a 2 is referenced, the solution is in reachable, but could not be solved without exceeding certain limits.

SCALE_FACTOR(a)
requires:a=(0..MXARMS-1) {arm number}
references: float
References the scale factor which gives the proportion of the components of the actual velocity to the reachable components of the desired velocity.

cnstbk

These macro functions give us information about constraints applied to a manipulator.

ICNST(a)
requires:a=(0..MXARMS-1) {arm number}
references: int
References a flag indicating when the motion of an arm is constrained or not. If a zero is referenced, the motion of the arm is unconstrained. If a 1 is referenced, the motion is constrained.

NUM_CONSTRAINTS(a)
requires:a=(0..MXARMS-1) {arm number}
references: int
This references the number of constraint vectors of an arm, which will be less than or equal to six.

CONSTRAINT_TYPE(a,c)
requires:a=(0..MXARMS-1) {arm number}
c=(0..5) {constraint number}
references: int
References whether or not a constraint on an arm is single-acting or double-acting.

CONSTRAINT_SIGN(a,c)
requires:a=(0..MXARMS-1) {arm number}
c=(0..5) {constraint number}
references: int
References the direction sign of a double-acting constraint on an arm.

NUM_CONSTRAINT_PLANES(a)
requires:a=(0..MXARMS-1) {arm number}
c=(0..5) {constraint number}
references: int
References the number of constraint planes for an arm.

CONSTRAINT_POINT(a,c,u)
requires:a=(0..MXARMS-1) {arm number}
c=(0..5) {constraint number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u axis component of the location of a constraint point for a manipulator.

CONSTRAINT_VECTOR(a,c,u)
requires:a=(0..MXARMS-1) {arm number}
c=(0..5) {constraint number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u axis component of the location of a constraint vector indicating constrained motion direction for a manipulator.

CONSTRAINT_VELOCITY(a,c)
requires:a=(0..MXARMS-1) {arm number}
c=(0..5) {constraint number}
references: float
References the magnitude of the velocity in the direction of the constraint, for a particular arm.

CONSTRAINT_FORCE(a,c)
requires:a=(0..MXARMS-1) {arm number}
c=(0..5) {constraint number}

references: float
References the magnitude of the reaction force along a constraint direction, for a particular arm.

CONSTRAINT_PLANE(a,c,p)
requires:a=(0..MXARMS-1) {arm number}
c=(0..MXCONS-1) {constraint number}
p=(0..3) {plane vector component: 0=x axis, 1=y axis, 2=z axis,
3=scale factor w}
references: float
References the p component of the vector describing the constraint plane of a particular constraint for a particular arm.

cntrbk

These macro functions and definitions are concerned with control parameters such as gains, errors, and update intervals.

P_GAIN(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the proportional gain constant for a particular joint in an arm under positional control.

D_GAIN(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the derivative gain constant for a particular joint in an arm under positional control.

I_GAIN(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the integrator gain constant for a particular joint in an arm under positional control.

ERROR_INT(a,j)

requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the integrator error for a particular joint in an arm under positional control.

THETA_REF(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the joint angle reference signal for a particular joint in an arm under simulation control.

OMEGA_REF(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the joint angular velocity reference signals for a particular joint under simulation control.

FORCE_P_GAIN(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the proportional gain constant for a joint of an arm under force/torque control.

FORCE_D_GAIN(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the derivative gain constant for a joint of an arm under force/torque control.

FORCE_I_GAIN(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the integrator gain constant for a joint of an arm under force/torque control.

FORCE_ERROR_INT(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the integrator error used for a joint of an arm under force/torque control.

SIGNAL_INTERVAL
value: float
This is the value of the control signal update interval.

LAST_SIGNAL_TIME
value: float
This is the time the control signal was last updated.

complbk

These macros deal with values used during stiffness control.

Although there are only 4 functions under this heading, the structure "complbk" lists nine other member-arrays which have been commented out. It is not mentioned whether these arrays are obsolete or will be implemented in a future version of ROBSIM. I have not created macros for them, therefore, to guard against there being used with any version of ROBSIM where they might not be implemented. All nine are of the form FUNCTION(a,j), where **a** is the arm number and **j** is the joint number (again with the caveat that joint 1 is element 0, etc). If these arrays are ever uncommented in a future version, then their names (in some future release of this macro function package) will be:

CIGAIN(a,j) {integrator gain constant for stiffness control}
CDGAIN(a,j) {derivative gain constant for stiffness control}
CLLGAIN(a,j) {lead-lag filter gain constant}
TADV(a,j) {advance time constant}
TLAG(a,j) {lag time constant}
OLDDELT(a,j) {old joint torque deltas}
OLDDEL(a,j) {old joint thetas}
OLDDLO(a,j) {old lead-lag filter output}
SERRINT(a,j) {integrator error for stiffness control}

The following macros are currently defined:

STIFFNESS(a,r,c)
requires: a=(0..MXARMS-1) {arm number}
r=(0..5) {row number}
c=(0..5) {column number}
references: float
References the (r,c) element of the stiffness matrix at the end-effector reference point. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element S(2,4) of the first arm's stiffness matrix is referenced as END_STIFFNESS(ARM1,1,3) and not END_STIFFNESS(ARM1,3,1).

FORCE_BIAS(a,u)
requires: a=(0..MXARMS-1) {arm number}
u=(0..5) {direction}
references: float
References the end-effector bias force/torque for an arm, where u=0,1, or 2 refers to a translation (force) along the x, y, or z axis respectively, and u=3, 4, or 5 refers to a rotation (torque) around the x, y, or z axis respectively.

CONTROL_TORQUE(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the control torque calculated in ROBSIM executable CNTRSIG for a joint in an arm.

ISTF(a)
requires: a=(0..MXARMS-1) {arm number}
references: int
References a flag relating whether or not compliance control is active for a particular arm. If a zero is referenced, then compliance control is not active. If a 1 is referenced, then the arm is under compliance control.

envtbk

These macro functions and definitions return information about the environment through an arm or set of arms moves.

The structure **envtbk** also contains certain arrays and integers other than those referenced below. There is ienv[10*MXGRAFPT], which is a data array for the graphics description of the environment; ictenv, which contains the number of components in ienv; nstrte[MXENCMPS], which holds the location in ienv where each evironment component begins. It was not deemed necessary to create a macro to access the values in these elements.

NUM_ENVIRON
value: int
This definition holds the number of components in the environment.

GRAVITY(u)
requires: u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the component of gravity in the direction of u.

ENVIRON_SPAN
value: float
This definition holds the span for the environment. For instance, a span of 1.5 meters indicates that 1.5 meters is represented on either side of the center of the graphics area. If your program is relying on built-in graphics functions in ROBSIM (as it probably is), then this is important only if you wish to see if some part of your manipulator has left the environment represented on the screen.

forcbk

These function macros reference values having to do with the forces, torques, and energy encountered in joints, centroids, and the end-effector. These values are of actual forces/torques experienced, not applied. For force/torque application, see **fortorbk**, below.

JOINT_FORCE(a,i,u)
requires: a=(0..MXARMS-1) {arm number}

i=(0..n+1) {joint number: 0=base, 1≤i≤n=joint,n+1=tool}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the component of the force vector at a joint in an arm in the direction of u.

CG_FORCE(a,i,u)
requires: a=(0..MXARMS-1) {arm number}
i=(0..n+1) {joint number: 0=base, 1≤i≤n=joint,n+1=tool}

u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the component of the force vector at a link's centroid in an arm in the direction of u.

END_FORCE(a,u)
requires: a=(0..MXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the component of the force vector at the end-effector of an arm in the direction of u. Should be the same as JOINT_FORCE(a,n+1,u) if the end-effector location is specified as equivalent to the location of the origin of the tool frame. Do not confuse this command with EE_FORCE, in **fortorbk**.

JOINT_TORQUE(a,i,u)
requires: a=(0..MXARMS-1) {arm number}
i=(0..n+1) {joint number: 0=base, 1≤i≤n=joint,n+1=tool}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the component of the torque vector at a joint in an arm around the axis specified by u.

END_TORQUE(a,u)
requires: a=(0..MXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the component of the torque vector at the end-effector of an arm around the axis specified by u. Should be the same as JOINT_TORQUE(a,n+1,u) if the end-effector location is specified as equivalent to the location of the origin of the tool frame. Do not confuse this command with EE_TORQUE, in **fortorbk**.

ACTUATOR_TORQUE(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the torque at an actuator in an arm. (The documentation actually states "torque vector," but since TAC is defined in the original FORTRAN structure as tac(MXLNKS,MXARMS), there is no way that this can refer to a vector, and must instead refer to the magnitude of the torque at

any actuator.)

JOINT_FORCE_LMT(a,i,e)
requires: a=(0..MXARMS-1) {arm number}
i=(0..n+1) {joint number: 0=base, 1≤i≤n=joint,n+1=tool}
e=(0,1) {range end: 0=minimum, 1=maximum}
references: float
References the force limits of a joint in an arm.

JOINT_TORQUE_LMT(a,i,e)
requires: a=(0..MXARMS-1) {arm number}
i=(0..n+1) {joint number: 0=base, 1≤i≤n=joint,n+1=tool}
e=(0,1) {range end: 0=minimum, 1=maximum}
references: float
References the torque limits of a joint in an arm.

ACTUATOR_TORQUE_LMT(a,j,e)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
e=(0,1) {range end: 0=minimum, 1=maximum}
references: float
References the torque limits of an actuator in an arm.

RATE_LMT(a,j,e)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
e=(0,1) {range end: 0=minimum, 1=maximum}
references: float
References the rate limits of an joint in an arm.

DRIVING_TORQUE(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the driving torque for a joint in an arm.

STEP_ENERGY(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the energy used per time step for a joint in an arm.

TOTAL_ENERGY(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the total energy used by a joint in an arm.

fortorbk

These macro functions return values having to do with force/torque control. For actual forces/torques experienced during the simulation, see **forcbk**, above.

IFTCN(a)
requires:a=(0..MAXARMS-1) {arm number}
references: int
This references a flag which relates whether or not force/torque control is active for a particular arm. If a zero is referenced, then F/T control is not active; if a 1 is referenced then F/T control is in use.

FORCE_COMP_NUM(a)
requires:a=(0..MAXARMS-1) {arm number}
references: int
This references the number of force components to be controlled for an arm. The number referenced is between 0 and 3.

TORQUE_COMP_NUM(a)
requires:a=(0..MAXARMS-1) {arm number}
references: int
This references the number of torque components to be controlled for an arm. The number referenced is between 0 and 3.

FORCE_VECTOR(a,r,u)

requires:a=(0..MAXARMS-1) {arm number}
r=(0..2) {row number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
Vforc is an array made up of three vectors, one for each (potential) force component to be controlled, and it is referenced by FORCE_VECTOR(). This function references the value of the rth component along the direction of the u axis, for a particular arm. For instance, the call FORCE_VECTOR(ARM1,0,2) references the z portion of the 2nd component's unit vector of the first arm.

FORCE_MAG(a,r)
requires:a=(0..MAXARMS-1) {arm number}
r=(0..2) {row number}
references: float
This references the magnitude of the rth force vector of an arm.

TORQUE_VECTOR(a,r,u)
requires:a=(0..MAXARMS-1) {arm number}
r=(0..2) {row number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
Vtorq is an array made up of three vectors, one for each (potential) torque component to be controlled, and it is referenced by TORQUE_VECTOR(). This function references the value of the rth component around the direction of the u axis, for a particular arm. For instance, the call TORQUE_VECTOR(ARM1,0,2) references the z portion of the 2nd component's unit vector of the first arm.

TORQUE_MAG(a,r)
requires:a=(0..MAXARMS-1) {arm number}
r=(0..2) {row number}
references: float
This references the magnitude of the rth torque vector of an arm.

EE_FORCE(a,u)
requires:a=(0..MAXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}

references: float
References the value of the force at the end-effector along the u axis for an arm. Do not confuse this with END_FORCE, in **forcbk**.

EE_TORQUE(a,u)
requires:a=(0..MAXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the value of the torque at the end-effector around the u axis for an arm. -Do not confuse this with END_TORQUE, in **fortorbk**.

POSITION_REF(a,u)
requires:a=(0..MAXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the value of the reference position of the end-effector along the u axis for an arm.

ROTATION_REF(a,r,c)
requires: a=(0..MXARMS-1) {arm number}
r=(0..2) {row number}
c=(0..2) {column number}
references: float
References the (r,c) element of the orientation matrix of the end-effector reference point of a particular arm. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element S(2,3) of the first arm's orientation matrix is referenced as ROTATION_REF(ARM1,1,2) and not ROTATION_REF(ARM1,2,1).

ERROR_POS(a,u)
requires:a=(0..MAXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the difference between end-effector reference position and actual position along the direction of u for a particular arm.

ERROR_ORIENT(a,u)
requires:a=(0..MAXARMS-1) {arm number}

u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the difference between end-effector reference orientation and actual orientation around the direction of the u-axis for a particular arm.

DELTA_TORQUE(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the torque delta at each joint for an arm.

OLD_DELTA_TORQUE(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the old torque delta at each joint for an arm.

REFERENCE_FT(a,u)
requires: a=(0..MXARMS-1) {arm number}
u=(0..5) {direction}
references: float
References the user-specified reference force or torque along or around the u axis, as appropriate.

POS_DIFF(a,u)
requires: a=(0..MXARMS-1) {arm number}
u=(0..5) {direction}
references: float
References the error in position along or around the u axis, as appropriate.

FT_DIFF(a,u)
requires: a=(0..MXARMS-1) {arm number}
u=(0..5) {direction}
references: float
References the error in force or torque along or around the u axis, as appropriate.

geombk

This set of macro functions and definitions will almost certainly be necessary in any routine written for ROBSIM. They return information about the robot, such as the number of links, joints, and the limits on the robot.

NUM_ARMS
value: int
This definition contains the number of existing arms in the system.

NUM_JOINTS(a)
requires: a=(0..MXARMS-1) {arm number}
references: int
References the number of joints in an arm.

JOINT_TYPE(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: int
References the joint type for a joint in an arm. If a 1 is referenced, the joint is a hinge joint (rotates around the y-axis). If a 2 is referenced, then the joint is a swivel-type (rotates around the x-axis). If a 3 is referenced, then the joint is a prismatic (sliding) joint (translates along the x-axis).

TRAVEL_LMT(a,j,e)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
e=(0,1) {range end: 0=minimum, 1=maximum}
references: float
References the travel limit on a joint in an arm.

JOINT_LOCATION(a,i,u)
requires: a=(0..MXARMS-1) {arm number}
i=(0..n+1) {joint number: 0=base, 1≤i≤n=joint,n+1=tool}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the location of joint i in the joint i-1 coordinate system, for a particular arm.

JOINT_ROTATION(a,i,r,c)
requires: a=(0..MXARMS-1) {arm number}
    i=(0..n+1) {joint number: 0=base, 1≤i≤n=joint,n+1=tool}
    r=(0..2) {row number}
    c=(0..2) {column number}
references: float
References the (r,c) element of the orientation transformation matrix from joint i to the joint i-1 coordinate system of a particular arm. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element A(2,3) of the 2nd joint of the first arm's orientation matrix is referenced as JOINT_ROTATION(ARM1,2,1,2) and not JOINT_ROTATION(ARM1,2,2,1).

GRASP_POINT(a,u)
requires: a=(0..MXARMS-1) {arm number}
    u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the location of the grasp point of an arm.

APPROACH_VECTOR(a,u)
requires: a=(0..MXARMS-1) {arm number}
    u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the approach vector of an arm.

NEWFILE(a)
requires: a=(0..MXARMS-1) {arm number}
references: int
References a flag indicating whether or not there is a new file describing a particular arm (0=no, 1=yes). This is used primarily in INITDRVR and not in simulations.

ARM_ORDER(a)
requires: a=(0..MXARMS-1) {queue array position number}
references: int
References the ath arm to be processed. In this case, a does not refer to a particular arm, but a position in a queue. For instance, if (ARM_ORDER(0)==3) was **true**, then the first arm to be processed would be arm 3. **Caution:** the correct way to use this in a statement would be, for example, var1=NJ(ARM_ORDER(0)-1); which would return the number of joints in the first arm to be pro-

cessed, while ARM_ORDER(2)=4 causes the fifth arm to be the third arm in the processing queue. You must subtract 1 from the referenced value of ARM_ORDER() when using its value for any reason (other than simple output), because arm 1 corresponds to 0, arm 2 corresponds to 1, and so forth.

ARMHOLD(a)
requires: a=(0..MXARMS-1) {arm number}
references: int
The documentation states that armhold(MXARMS) is an "array containing arm# holding current arm." The meaning of this is not clear.

CURRENT_ARM
value: int
This definition contains the number of the arm currently being processed. **Caution:** the proper way to use CURRENT_ARM in a statement requiring an arm number is, for example, var2=NJ(CURRENT_ARM-1); which will return the number of joints in the arm currently being processed. You must subtract 1 from the value of CURRENT_ARM when using its value for any reason (other than simple output), because arm 1 corresponds to 0, arm 2 corresponds to 1, and so forth. See ARM_ORDER(), above, for further information.

ARM_NAME(a)
requires: a=(0..MXARMS-1) {arm number}
references: char *
References a pointer to a string of characters containing the name of an arm. The string is eight characters long.

jacobbk

These two macros return elements from the jacobian and its (pseudo)inverse. ROBSIM calculates the jacobian in an routine called JACOB, and the (least-squares pseudo)inverse is calculated in a routine called GIASOS. If the user wishes to write his or her own jacobian routines, then the results must be saved in jacobbk (if they will be used by other routines); if the user simply wishes to access the ROBSIM-calculated results then they must be called from jacobbk. Either way, these macros are of good use.

The row/column FORTRAN/C reversal is accounted for in these macros; thus, element J(2,3) of the first arm's jacobian matrix is referenced as JACOBIAN(ARM1,1,2) and not JACOBIAN(ARM1,2,1).

JACOBIAN(a,r,c)
requires: a=(0..MXARMS-1) {arm number}
r=(0..m-1) {row number}
c=(0..n-1) {column number}
references: float
References the (r,c) element of the current jacobian of an arm.

INVERSE_JACOBIAN(a,r,c)
requires: a=(0..MXARMS-1) {arm number}
r=(0..m-1) {row number}
c=(0..n-1) {column number}
references: float
References the (r,c) element of the current (pseudo)inverse jacobian of an arm.

loadbk

Values pertaining to the loads and objects in a system environment can be referenced by the use of these macros and definitions. The words "object" and "load" in the definitions are used interchangeably.

NUM_LOADS
value: int
This definition contains the number of loads in the system.

NUM_LOAD_PT_MASSES(d)
requires:d=(0..MXOBJS-1) {object number}
references: int
References the number of point masses assigned to an object.

LOAD_SEQ(d)
requires:d=(0..MXOBJS-1) {object queue position}
references: int
References the dth object in the object rotation sequence. For instance, if (LOAD_SEQ(0)==5) is **true**, then object 5 is the first object in the sequence. **Caution:** because object number referencing starts at zero and not one, the value referenced from LOAD_SEQ() must be diminished by one when used in other functions. For instance, LOAD_LENGTH(LOAD_SEQ(2)-1) references the

length of the third object in the rotation sequence, while LOAD_SEQ(0)=4 causes object 5 to be the first object in the rotation sequence.

LOAD_HOLDER(a,d)
requires:a=(0..MXARMS-1) {arm number}
d=(0..MXOBJS-1) {object number}
references: int
References the number of the arm holding the load: if a zero is referenced, then no arm is holding it. **Caution:** if you use this referenced value to reference some other value connected to the arm, be sure to subtract 1 first. For instance, LOAD_HOLDER(1)=4; causes ROBSIM to know that the second load is now held by arm 5, while TH(LOAD_HOLDER(3)-1,1); references the current theta of the second link of the arm holding the third load object.

LOAD_LENGTH(d)
requires:d=(0..MXOBJS-1) {object number}
references: float
References the length of an object.

LOAD_RADIUS(d)
requires:d=(0..MXOBJS-1) {object number}
references: float
References the radius of an object.

LOAD_MASS(d)
requires:d=(0..MXOBJS-1) {object number}
references: float
References the mass of an object.

LOAD_CG(d,u)
requires:d=(0..MXOBJS-1) {object number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the vector describing the center of gravity of an object, in local coordinates.

LOAD_INERTIA(d,r,c)

requires:d=(0..MXOBJS-1) {object number}
r=(0..2) {row number}
c=(0..2) {column number}
references: float
References the (r,c) element of the inertia matrix of an object, in local coordinates. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element S(2,3) of the first object's inertia matrix is referenced as LOAD_INERTIA(0,1,2) and not LOAD_INERTIA(0,2,1).

LOAD_GRIP_POINT(d,u)
requires:d=(0..MXOBJS-1) {object number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the relative grip point of the object.

LOAD_APPROACH_VEC(d,u)
requires:d=(0..MXOBJS-1) {object number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the approach vector of the object.

LOAD_PT_MASS(d,p)
requires:d=(0..MXOBJS-1) {object number}
p=(0..MXPTS-1) {point mass number}
references: float
References the mass of a point mass assigned to an object.

LOAD_PT_MASS_LOC(d,p,u)
requires:d=(0..MXOBJS-1) {object number}
p=(0..MXPTS-1) {point mass number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the location of a point mass assigned to an object, in local coordinates.

LOAD_ORIENTATION(d,u)
requires:d=(0..MXOBJS-1) {object number}

u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the orientation angle of an object around the u-axis, in world coordinates.

LOAD_POSITION(d,u)
requires:d=(0..MXOBJS-1) {object number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the location of a load in world coordinates.

LOAD_ROTATION(d,r,c)
requires:d=(0..MXOBJS-1) {object number}
r=(0..2) {row number}
c=(0..2) {column number}
references: float
References the (r,c) element of a load's orientation matrix in world coordinates. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element A(2,3) of the first object's orientation matrix is referenced as LOAD_ROTATION(0,1,2) and not LOAD_ROTATION(0,2,1).

LOAD_TOTAL_MASS(d)
requires:d=(0..MXOBJS-1) {object number}
references: float
References the total mass of an object, including point masses.

LOAD_TOTAL_CG(d,u)
requires:d=(0..MXOBJS-1) {object number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the vector describing the center of gravity of the object, including point masses, in local coordinates.

LOAD_TOTAL_INERTIA(d,r,c)
requires:d=(0..MXOBJS-1) {object number}
r=(0..2) {row number}
c=(0..2) {column number}
references: float

References the (r,c) element of a load's total inertia matrix, including point masses, in local coordinates. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element A(2,3) of the first object's inertia matrix is referenced as LOAD_TOTAL_INERTIA(0,1,2) and not LOAD_TOTAL_INERTIA(0,2,1).

LOAD_NAME(d)
requires:d=(0..MXOBJS-1) {object number}
references: char *
References a pointer to a string of characters containing the name of a load object. The name is eight characters long.

motorbk

These macro functions reference armature motor values.

MOTOR_TORQUE(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the torque constant of a joint motor on an arm.

MOTOR_RESISTANCE(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the armature resistance of a joint motor on an arm.

MOTOR_INDUCTANCE(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the armature inductance of a joint motor on an arm.

MOTOR_GAIN(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float

References the amplifier gain of a joint motor on an arm.

MOTOR_RATIO(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the gear ratio of a joint motor on an arm.

MOTOR_EMF(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the back EMF constant of a joint motor on an arm.

MOTOR_MOMENT(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the true moment of inertia of armature and motor gear of a joint on an arm.

MOTOR_VOLTAGE(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the amplifier output voltage of a joint motor on an arm.

MOTOR_CURRENT(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the armature current of a joint motor on an arm.

movbasbk

These definitions and macro reference certain flags having to do with the state of moving bases in a system. The documentation is not clear on several of the elements of movbasbk.

IMOVBAS
value: int
This definition is **true** ($\neq 0$) if the system has moving bases; otherwise, it is **false** (=0).

NUM_BASES
value: int
This definition references the number of moving bases in the system.

IBASSIM
value: int
No documentation available.

IBASTRQ
value: int
No documentation available.

IBASAV
value: int
No documentation available.

MOUNTED(a)
requires:a=(0..MXARMS-1) {arm number}
references: int
References the number of the base upon which the arm is mounted. A zero indicates that the arm is not mounted on a moving base. **Caution:** when using the referenced value, don't forget to subtract 1. For instance, MOUNTED(0)=4 tells ROBSIM that the first arm is mounted on base 5, not 4.

BTLAST
value: float
No documentation available.

BTNEXT
value: float
No documentation available.

outrefbk

These macro functions reference values having to do with the error in position, orientation, and the epsilon error tolerences.

REFERENCE_THETA(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the reference value of theta of a joint on an arm.

REFERENCE_JOINT_ERROR(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the reference joint error of a joint on an arm.

DESIRED_POSITION(a,u)
requires:a=(0..MAXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the next position point on the trajectory for the end-effector of an arm.

ARM_EPSILON(a,u)
requires:a=(0..MAXARMS-1) {arm number}
u=(0..5) {direction}
references: float
References the permissible error (epsilon) in position or orientation along or around the u-axis, whichever is applicable.

DESIRED_ROTATION(a,r,c)
requires: a=(0..MXARMS-1) {arm number}
r=(0..2) {row number}
c=(0..2) {column number}
references: float
References the (r,c) element of the next orientation matrix of the end-effector of a particular arm.

The row/column FORTRAN/C reversal is accounted for in the macro; thus, element E(2,3) of the first arm's orientation matrix is referenced as DESIRED_ROTATION(ARM1,1,2) and not DESIRED_ROTATION(ARM1,2,1).

segmbk

These macro functions and definitions deal with the simulation conditions during this time segment. For instance, if an AL command is given as drive joint(1) of arm 1 to 90 with duration=1.0; then arm 1 has a time segment existing from the time the command started until 1.0 seconds later.

Several of the structure elements are not defined in the documentation.

IMOT
value: int
This definition indicates whether or not this program step is motion-type. If it is 1, then it is a motion step; if it is 2, then it is not.

MOTION(a,j,q)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
q=(0..MAXORD) {order}
references: float
References the qth-order coefficient of the polynomial describing the motion time history of a joint of an arm.

ARM_SEGMENT_START_TIME(a)
requires: a=(0..MXARMS-1) {arm number}
references: float
References the time that the current segment started for an arm.

ARM_CONTROL(a)
requires: a=(0..MXARMS-1) {arm number}
references: int
References the segment joint or end-effector control flag for an arm. If a 1 is referenced, then the arm is under joint control. If a 2 is referenced, then the arm is under base end-effector motion control. If a 3 is referenced, then the arm is under end-effector control.

IRATE(a)

requires: a=(0..MXARMS-1) {arm number}
references: int
References the flag indicating what type on control an arm is under. If a 1 is referenced, then the arm is under rate control. If a 2 is referenced, then the arm is under position control. If a 3 is referenced, then the arm is under sensor control.

TOOLRP(a,u)
requires:a=(0..MAXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
No documentation available.

POSIT(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
No documentation available. The array this macro references is used cryptically in routine SGMNT.

POSOR(a,r,c)
requires: a=(0..MXARMS-1) {arm number}
r=(0..2) {row number}
c=(0..2) {column number}
references: float
No documentation available. The array this macro references is used cryptically in routine SGMNT. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element P(2,3) of the first arm's matrix is referenced as POSOR(ARM1,1,2) and not POSOR(ARM1,2,1).

SEGTH(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
No documentation available.

SEGPOS(a,u)
requires: a=(0..MXARMS-1) {arm number}

u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
No documentation available.

SEGOR(a,r,c)
requires: a=(0..MXARMS-1) {arm number}
r=(0..2) {row number}
c=(0..2) {column number}
references: float
No documentation available. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element P(2,3) of the first arm's matrix is referenced as SEGOR(ARM1,1,2) and not SEGOR(ARM1,2,1).

DELTH(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
No documentation available. A similar variable is declared internally in routine PIDCON.

DELPOS(a,u)
requires: a=(0..MXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
No documentation available.

DPHI(u)
requires: u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
May reference the u-axial component of a vector for changing orientation from OR to ROR. The array dphi is internal to routines BORERR, JTPOS, and ORERR.

PH
value: float
This definition may reference the angle of rotation for desired orientation change. This value is internal to routine BORERR.

IDCC
value: int
This definition references the dual arm coordination control flag. If a 1 is referenced, then the arms are independent. If a 2 is referenced, then coordinated control is in use. If a 3 is referenced, then multi-arm control is in use.

ARM_TASK(a)
requires: a=(0..MXARMS-1) {arm number}
references: int
References the arm's task currently in effect. These are the the possible return values:
1=pick up an object
2=place the object
3=move the arm
4=hold the arm at the current position
5=change the reference point of the arm
6=arm is under operator control
7=arm is under response control
8=arm is under end-effector position control
9=arm is under teleoperator joint control
10=arm is under teleoperator base control
11=arm is under teleoperator end-effector control

NT(a)
requires: a=(0..MXARMS-1) {arm number}
references: int
No documentation available.

SENSOR_VELOCITY(a)
requires: a=(0..MXARMS-1) {arm number}
references: float
References the maximum sensor to target velocity (sic) for an arm.

ARM_SEGMENT_START(a)
requires: a=(0..MXARMS-1) {arm number}
references: float
References the time that the current segment started for an arm. It is unclear how this differs from

ARM_SEGMENT_START_TIME(a).

ARM_SEGMENT_STOP(a)
requires: a=(0..MXARMS-1) {arm number}
references: float
References the time that the current segment will stop for an arm.

BASE_SEGMENT_START(a)
requires: a=(0..MXARMS-1) {arm (base) number}
references: float
References the time that the current time segment started for a base. It is unclear how this is different than BASE_SEGMENT_START_TIME.

BASE_SEGMENT_STOP(a)
requires: a=(0..MXARMS-1) {arm (base) number}
references: float
References the time that the current time segment will stop for a base.

sensbk

These macro functions reference the various components of the PID control signal, and the compliant feedback.

SENSED_THETA(a,j)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
Returns the theta PID control signal for a joint on an arm.

SENSED_OMEGA(a,j)
requires: a=(0..MXARMS-1) {arm number}

j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
Returns the theta-dot PID control signal for a joint on an arm.

SENSED_OMEGA_DOT(a,j)

requires: a=(0..MXARMS-1) {arm number}

j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}

references: float

Returns the theta-double-dot PID control signal for a joint on an arm.

OLD_SENSED_THETA(a,j,b)

requires: a=(0..MXARMS-1) {arm number}

j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}

b=(0,1) {iteration: 0=previous, 1=the time before previous}

references: float

Returns an old theta control signal for a joint on an arm. This is used primarily to calculate the new derivatives of the theta control signal.

SENSED_FORCES(a,u)

requires:a=(0..MAXARMS-1) {arm number}

u=(0..5) {direction}

references: float

References the compliant feedback along or around the u-axis, whichever is appropriate.

sprgbk

These macro functions return information regarding the spring of the end-effector.

SPRING_LOCATION(a,u)

requires: a=(0..MXARMS-1) {arm number}

u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}

references: float

References the u-axial component of the spring reference point (in world coordinates) when no force is present, for a particular spring on an arm..

SPRING_ROTATION(a,r,c)

requires: a=(0..MXARMS-1) {arm number}

r=(0..2) {row number}

c=(0..2) {column number}

references: float

Returns the (r,c) component of the spring reference orientation matrix when there is no force exerted on the spring, for a particular spring on an arm. The documentation notes that this should be the same as the end-effector orientation matrix as well. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element A(2,3) of the first arm's spring reference orientation matrix is referenced as SPRING_ROTATION(ARM1,1,2) and not SPRING_ROTATION(ARM1,2,1).

SPRING_CONSTANT(a,r,c)
requires: a=(0..MXARMS-1) {arm number}
r=(0..5) {row number}
c=(0..5) {column number}
references: float
References the (r,c) element of the spring constant matrix K of a particular spring on an arm. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element K(4,6) of the first arm's spring constant matrix is referenced as SPRING_CONSTANT(ARM1,3,5) and not SPRING_CONSTANT(ARM1,5,3).

SPRING_REST_POSITION(a,u)
requires: a=(0..MXARMS-1) {arm number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the spring reference point (in the end-effector coordinate system) when no force is present, for a particular spring on an arm..

SPRING_FORCE(a,u)
requires: a=(0..MXARMS-1) {arm number}
u=(0..5) {direction}
references: float
References the force or torque around or along the u-axis, which ever is appropriate.

ISPR(a)
requires: a=(0..MXARMS-1) {arm number}
references: int
References a flag indicating the spring type for an arm. If a 0 is referenced, then the arm has no spring. If a 1 is referenced, then the spring is translational only, and therefore only the upper left 3x3 elements of SPRK() (see above) are meaningful. If a 2 is referenced, then the spring is both translational and rotational, and its K matrix is 6x6.

systbk

These macro functions contain current information about the arms in the system: their positions, velocities, etc. Note that some (but not all) of these are updated by calling the routine SETUP2. Notable exceptions are LINK_THETA(), LINK_OMEGA(), and LINK_OMEGA_DOT(), which are updated elsewhere (for instance, by routine JTPOS, or by a position control routine which you might write). Be aware of where these variables are updated in your program flow so that the other routines that you don't control have complete and up-to-date information.

LINK_THETA(a,j)
requires: a=(0..MXARMS-1) {arm number}
    j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the current angular displacement of a link j with respect to link j-1 for an arm; that is, the current theta of a joint on an arm.

LINK_OMEGA(a,j)
requires: a=(0..MXARMS-1) {arm number}
    j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the current angular velocity of a link j with respect to link j-1 for an arm; that is, the current theta-dot of a joint on an arm.

LINK_OMEGA_DOT(a,j)
requires: a=(0..MXARMS-1) {arm number}
    j=(0..n-1) {joint number: 0≤j≤n-1=joint j+1}
references: float
References the current angular acceleration of a link j with respect to link j-1 for an arm; that is, the current theta-double-dot of a joint on an arm.

LINK_POSITION(a,f,u)
requires: a=(0..MXARMS-1) {arm number}
    f=(0..n+2) {joint number: 0=base, 1≤f≤n=joint,n+1=tool,n+2=end-effector
        reference point}
    u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the translational position vector for each link/frame in world

coordinates, for a particular arm.

LINK_VELOCITY(a,f,u)
requires: a=(0..MXARMS-1) {arm number}
f=(0..n+2) {joint number: 0=base, 1≤f≤n=joint,n+1=tool,n+2=end-effector reference point}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the linear velocity vector for each link/frame in world coordinates, for a particular arm.

LINK_ACCELERATION(a,f,u)
requires: a=(0..MXARMS-1) {arm number}
f=(0..n+2) {joint number: 0=base, 1≤f≤n=joint,n+1=tool,n+2=end-effector reference point}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the linear acceleration vector for each link/frame in world coordinates, for a particular arm.

LINK_ROTATION(a,i,r,c)
requires: a=(0..MXARMS-1) {arm number}
i=(0..n+1) {link number: 0=base, 1≤i≤n=link,n+1=tool}
r=(0..2) {row number: 0=x row,1=y row, 2=z row}
c=(0..2) {column number: 0=x column,1=y column, 2=z column}
references: float
References the (r,c) element of the rotation matrix from the joint i coordinate system to the inertial system, for a joint in an arm. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element T(3,2) of the first arm's rotation matrix is referenced as LINK_ROTATION(ARM1,2,1) and not LINK_ROTATION(ARM1,1,2).

LINK_WORLD_OMEGA(a,i,u)
requires: a=(0..MXARMS-1) {arm number}
i=(0..n+1) {link number: 0=base, 1≤i≤n=link,n+1=tool}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float

References the u-axial component of the absolute angular velocity vector for each link/frame in world coordinates, for a particular arm.

LINK_WORLD_OMEGA_DOT(a,i,u)
requires: a=(0..MXARMS-1) {arm number}
i=(0..n+1) {link number: 0=base, 1≤i≤n=link,n+1=tool}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the absolute angular acceleration vector for each link/frame in world coordinates, for a particular arm.

JOINT_VECTOR(a,j,u)
requires: a=(0..MXARMS-1) {arm number}
j=(0..n) {joint number: 0≤j≤n-1=joint j+1,n=tool}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the vector from joint/frame j-1 to joint/frame j, in world coordinates, for a joint on an arm. Note the slightly different definition of j in the "requires" section above.

ARM_CG(a,i,u)
requires: a=(0..MXARMS-1) {arm number}
i=(0..n+1) {link number: 0=base, 1≤i≤n=link,n+1=tool}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the vector from a joint/frame i to the center of gravity of its link, for a joint in an arm. This call is identical to LINK_CG(a,i,u) in **amasbk**.

ARM_INERTIA(a,i,r,c)
requires: a=(0..MXARMS-1) {arm number}
i=(0..n+1) {link number: 0=base, 1≤i≤n=link,n+1=tool}
r=(0..2) {row number: 0=x row,1=y row, 2=z row}
c=(0..2) {column number: 0=x column,1=y column, 2=z column}
references: float
References the (r,c) element of the inertia matrix of a link in an arm, in world coordinates. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element T(3,2) of the third

link of the first arm's inertia matrix is referenced as ARM_INERTIA(ARM1,3,2,1) and not ARM_INERTIA(ARM1,1,2). This call is identical to LINK_INERTIA(a,i,r,c) in **amasbk**.

VECTOR_CG(a,i,j,u)
requires: a=(0..MXARMS-1) {arm number}
i=(0..n) {joint number: 0≤i≤n-1=joint i+1,n=tool}
j=(0..n) {link number: 0≤j≤n-1=link j+1,n=tool}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of a vector from joint/frame i to the center of gravity of link j, in world coordinates, for a particular arm.

DOUBLE_ARM_FORCE(u)
requires: u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the component of the constraint force due to the use of dual-arm coordination around or along the u-axis (whichever is appropriate) .

targbk

These macro functions and definitions are concerned with target objects in the environment.

NUM_TARGETS
value: int
This definition references the number of target objects in the environment.

ITRGSEQ(t)
requires:t=(0..MXTRGS-1) {target number}
references: int
References the number of rotation axes used for orienting target.

NUM_TARGET_PT_MASSES(t)
requires:t=(0..MXTRGS-1) {target number}
references: int
References the number of point masses in a target.

TARGET_HOLDER(t)

requires:t=(0..MXTRGS-1) {target number}
references: int
References the number of the arm holding the target load: if a zero is referenced, then no arm is holding it. **Caution:** if you use this referenced value to reference some other value connected to the arm, be sure to subtract 1 first. For instance, TARGET_HOLDER(1)=4; causes ROBSIM to know that the second target is now held by arm 5, while TH(TARGET_HOLDER(3)-1,1); references the current theta of the second link of the arm holding the third target object.

TARGET_LENGTH(t)
requires:t=(0..MXTRGS-1) {target number}
references: float
References the length of a target object.

TARGET_RADIUS(t)
requires:t=(0..MXTRGS-1) {target number}
references: float
References the radius of a target object.

TARGET_MASS(t)
requires:t=(0..MXTRGS-1) {target number}
references: float
References the mass of a target object.

TARGET_CG(t,u)
requires:t=(0..MXTRGS-1) {target number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the vector describing the center of gravity of a target object, in local coordinates.

TARGET_INERTIA(t,r,c)
requires:t=(0..MXTRGS-1) {target number}
r=(0..2) {row number: 0=x row,1=y row, 2=z row}
c=(0..2) {column number: 0=x column,1=y column, 2=z column}
references: float
References the (r,c) element of the inertia matrix of a target, in local coordinates. The row/column

FORTRAN/C reversal is accounted for in the macro; thus, element T(3,2) of the first arm's target inertia matrix is referenced as TARGET_INERTIA(ARM1,2,1) and not TARGET_INERTIA(ARM1,1,2).

TARGET_GRIP_POINT(s,u)
requires:s=(0..MXLODS-1) {load number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
Not currently used in ROBSIM. References the u-axial component of the relative grip point of a target load.

TARGET_APPROACH_VEC(s,u)
requires:s=(0..MXLODS-1) {load number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
Not currently used in ROBSIM. References the u-axial component of the approach vector to a target load.

TARGET_PT_MASS(t,p)
requires:t=(0..MXTRGS-1) {target number}
p=(0..MXPTS-1) {point mass number}
references: float
References the mass of a point mass assigned to a target object.

TARGET_PT_MASS_LOC(t,p,u)
requires:t=(0..MXTRGS-1) {target number}
p=(0..MXPTS-1) {point mass number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the location of a point mass assigned to a target object, in local coordinates.

TARGET_ORIENTATION(t,u)
requires:t=(0..MXTRGS-1) {target number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float

References the orientation angle of a target object around the u-axis, in world coordinates.

TARGET_POSITION(t,u)
requires:t=(0..MXTRGS-1) {target number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the location of a target object in world coordinates.

TARGET_ROTATION(t,r,c)
requires:t=(0..MXTRGS-1) {target number}
r=(0..2) {row number}
c=(0..2) {column number}
references: float
References the (r,c) element of a target object's orientation matrix in world coordinates. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element A(2,3) of the first object's orientation matrix is referenced as TARGET_ROTATION(0,1,2) and not TARGET_ROTATION(0,2,1).

TARGET_TOTAL_MASS(t)
requires:t=(0..MXTRGS-1) {target number}
references: float
References the total mass of a target object, including point masses.

TARGET_TOTAL_CG(t,u)
requires:t=(0..MXTRGS-1) {target number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the vector describing the center of gravity of the object, including point masses, in local coordinates.

TARGET_TOTAL_INERTIA(t,r,c)
requires:t=(0..MXTRGS-1) {target number}
r=(0..2) {row number}
c=(0..2) {column number}
references: float
References the (r,c) element of a target object's total inertia matrix, including point masses, in local

coordinates. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element A(2,3) of the first object's inertia matrix is referenced as TARGET_TOTAL_INERTIA(0,1,2) and not TARGET_TOTAL_INERTIA(0,2,1).

TARGET_NAME(t)
requires:t=(0..MXTRGS-1) {target number}
references: char *
References a pointer to a string of characters containing the name of a target object object. The name is eight characters long.

timebk

These definitions reference start, stop, and step times for the simulation.

CURRENT_TIME
value: float
This definition references the current time in the simulation. It is set to some value (STRTTM) at the beginning of the simulation, and is updated by STPPRO every processing step.

START_TIME
value: float
This definition references the value that CURR_TIME was set to at the beginning of the simulation.

STOP_TIME
value: float
This definition references the time at which the simulation will stop. The simulation will stop when CURR_TIME is greater than STOPTM.

PROCESS_STEP
value: float
This definition references the simulation processing time step.

SOF_STEP
value: float
This definition references the simulation output time step for E&S graphics playback.

PRT_STEP
value: float
This definition references the printed output time step to be used.

AVT_STEP
value: float
This definition references the acc-vel-theta file time step to be used.

PLOT_STEP
value: float
This definition references the plot data output step size to be used.

TORQUE_STEP
value: float
This definition references the torque data output step size to be used.

GRAPHICS_STEP
value: float
This definition references the graphics output step size to be used (at what interval will the graphics be redrawn).

toolbk

These macro functions reference information about the tool on the end-effector.

GRASPING(a,s)
requires: a=(0..MXARMS-1) {arm (tool) number}
s=(0..MXLODS-1) {load number}
references: int
The documentation states that this would reference "the number of load (sic) a tool is holding... =0, no load." This sounds suspect, because only an a-sized array would be required, not a by s.

TOOL_MASS(a)
requires: a=(0..MXARMS-1) {arm (tool) number}
references: float
References the mass of a tool on an arm.

TOOL_CG(a,u)
requires: a=(0..MXARMS-1) {arm (tool) number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the vector describing the center of gravity of the tool. The coordinate system is not specified if the documentation.

TOOL_INERTIA(a,r,c)
requires: a=(0..MXARMS-1) {arm (tool) number}
r=(0..2) {row number: 0=x row,1=y row, 2=z row}
c=(0..2) {column number: 0=x column,1=y column, 2=z column}
references: float
References the (r,c) element of the inertia matrix of a arm's tool. The coordinate system is not specified in the documentation. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element T(3,2) of the first arm's matrix is referenced as TOOL_INERTIA(ARM1,2,1) and not TOOL_INERTIA(ARM1,1,2).

TOOL_TO_LOAD_TRANS(a,s,u)
requires: a=(0..MXARMS-1) {arm (tool) number}
s=(0..MXLODS-1) {load number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
References the u-axial component of the vector from the tool origin to the load origin.

TOOL_TO_LOAD_ROT(a,s,r,c)
requires: a=(0..MXARMS-1) {arm (tool) number}
s=(0..MXLODS-1) {load number}
r=(0..2) {row number: 0=x row,1=y row, 2=z row}
c=(0..2) {column number: 0=x column,1=y column, 2=z column}
references: float
References the (r,c) element of the rotation matrix from the tool coordinate system to the load coordinate system. The row/column FORTRAN/C reversal is accounted for in the macro; thus, element T(3,2) of the first arm's matrix is referenced as TOOL_TO_LOAD_ROT(ARM1,2,1) and not TOOL_TO_LOAD_ROT(ARM1,1,2).

TTOBSO(a,u)

requires: a=(0..MXARMS-1) {arm (tool) number}
u=(0..2) {direction: 0=x axis, 1=y axis, 2=z axis}
references: float
No documentation available.

TTULBS(a,r,c)
requires: a=(0..MXARMS-1) {arm (tool) number}
r=(0..2) {row number: 0=x row,1=y row, 2=z row}
c=(0..2) {column number: 0=x column,1=y column, 2=z column}
references: float
No documentation available. The row/column FORTRAN/C reversal is accounted for in the macro.

user_data

These macro functions reference values in the user_data structure. This structure is a handy place to store integers, floats, and strings when passing between ROBSIM routines and user-defined routines. No ROBSIM routine uses them, so the user is free to put any values in the structure. There are 5 ints, 5 floats, and 5 strings in a bundle, and there are two bundles, one called "input," and one called "output," for the user's convenience. Of course, the user is not restricted to just input or just output for either bundle.

USER_INPUT_INT(x)
requires: x=(0..NUM_INTS)
references: int
References the xth input integer stored in the structure.

USER_INPUT_FLT(x)
requires: x=(0..NUM_FLOATS)
references: float
References the xth input floating-point number stored in the structure.

USER_INPUT_CHR(x)
requires: x=(0..NUM_CHAR_STR)
references: int
References the xth input string stored in the structure. The string is of length LINE_LENGTH.

USER_OUTPUT_INT(x)
requires: x=(0..NUM_INTS)
references: int
References the xth output integer stored in the structure.

USER_OUTPUT_FLT(x)
requires: x=(0..NUM_FLOATS)
references: float
References the xth output floating-point number stored in the structure.

USER_OUTPUT_CHR(x)
requires: x=(0..NUM_CHAR_STR)
references: int
References the xth output string stored in the structure. The string is of length LINE_LENGTH.

B.3. The C-interface and RPTL files

The following appendix lists all of the include files necessary to create user-defined functions, and to use the ROBSIM Preprocessor Token Language (RPTL). The file *mxprms.h* **must** be included before any other include file is referenced, even if RPTL is not used. The file *robsim_tokens.h* must be included next if RPTL is to be used. *struct_defs.h* may be included in a function declaration, if all of the structures in **STRUCT_LIST** (which is defined in *robsim_tokens.h*) are being passed to the function. The other files are included in function declarations as necessary.

```
/* mxprms.h */

/*      THE #defines IN THIS INCLUDE FILE ARE CONSTANT VALUES REQUIRED
     BY THE PROGRAM.
*/
#defineISEGCAD 1
#defineIFACTCAD 1
#defineMAXSPN 500
#defineMXLNKS 10
#defineMXARMS 5
#defineMXSPJT 3
#defineMXPLTS 31
#defineMAXORD 3
#defineMAXSEG 20
#defineMXPTS 10
#defineMXGRCMP 40
#defineMXENCMPS 30
#defineMXLDCMPS 10
#defineMXLODS 120
#defineMXTRGCMPS 10
#defineMXTRGS 10
#defineMXOBJS MXLODS+MXTRGS
#defineMXGRAFPT 5000
#defineMAXTRAN 50
#defineMAXLIN 20000
#defineMAXPNT 40
#defineMAXARC 300
#defineMAXSPL 200
#defineMAX_DRIVERS 50
#defineMAX_MODES 10
#defineIFLGEND 5557
#defineIDIV 10
#defineITLNGTH 4
#defineMAXFLD 30000
#defineMAXBRK 4
#defineCURRENT_ARM_VERSION 1
#defineCURRENT_SYSTEM_VERSION 1
#defineFILE_LENGTH 255
#defineLINE_LENGTH 255
#define RAD_TO_DEG 57.2957795130823208768/* radians to degrees */
#defineDEG_TO_RAD .01745329251994329576924/* degrees to radians */
#define ON      1                                        /* on    true */
#defineOFF      0                                                /* off   false */
/*
C
C       DEFINITIONS
C
C       SYMBOL  TYPEDIMENSIONDEFINITION
C
C     MXARMS    I*4     1           MAXIMUM NUMBER OF ARMS
C     MXLNKS    I*4     1           MAXIMUM NUMBER LINKS ALLOWABLE
C     MXSPJT    I*4     1           MAXIMUM NUMBER SPECIAL JOINTS
C     MXPLTS    I*4     1           MAXIMUM NUMBER OF Y ARRAY DATA
```

```
C                              PARAMETERS WHICH MAY BE
C                              WRITTEN TO PLOT FILE
C      MAXORD    I*4     1           ORDER OF THE POLYNOMIAL DESCRIBING
C                              THE MOTION TIME HISTORY
C      MAXSEG    I*4     1           MAXIMUM NUMBER OF TIME SEGMENTS
C                              ALLOWED TO DESCRIBE THE MOTION
C                              TIME HISTORY
C      MXPTS     I*4     1           MAXIMUM NUMBER OF POINT MASSES
C                                            IN EACH LINK OR LOAD
C      MXGRCMP   I*4     1           MAXIMUM NUMBER OF GRAPHIX COMPONENTS
C                              ALLOWED PER LINK
C        MXENCMPS  I*4     1MAXIMUM NUMBER OF GRAPHICS COMPONENTS
C                                            IN ENVIRONMENT
C      MXLDCMPS  I*4     1           MAXIMUM NUMBER OF GRAPHICS COMPONENTS
C                              IN EACH LOAD OBJECT
C      MXLODS    I*4     1           MAXIMUM NUMBER OF LOAD OBJECTS ALLOWED
C      MXTRGCMPS I*4     1           MAXIMUM NUMBER OF GRAPHICS COMPONENTS
C                              IN EACH TARGET OBJECT
C      MXTRGS    I*4     1           MAXIMUM NUMBER OF TARGET OBJECTS
C                              ALLOWED
C      MXGRAFPT  I*4     1           MAXIMUM NUMBER OF GRAFIX POINTS ALLOWED
C                              IN EACH COMPONENT
C
C      ISEGCAD   I*2     1           NUMBER OF SEGMENTS FOR THE CAD/CAM
C                              GRAPHICS
C      IFACTCAD  I*2     1           SCALE FACTOR FOR THE CAD/CAM GRAPHICS
C      MAXSPN    I*2     1           MAXIMUM SPAN OF THE ARM DURING CAD/CAM
C                              GRAPHICS
C      ITLNGTH   I*4     1           LENGTH OF THE CAD/CAM FILE TITLE
C      IFLGEND   I*4     1           CAD/CAM FILE FLAG SPECIFYING END OF
C                              DIRECTORY DATA SECTION IN FILE
C      MAXFLD    I*4     1           MAXIMUM NUMBER OF LINES IN THE CAD/CAM
C                              DATA FILE
C      MAXARC    I*4     1           MAXIMUM NUMBER ALLOWABLE CAD/CAM
C                              ARC ENTITIES
C      IDIV      I*4     1           NUMBER OF STRAIGHT LINE SEGMENTS
C                              INTO WHICH CURVE ENTITY IS DIVIDED
C                              FOR CAD/CAM GRAPHICS DISPLAY
C      MAXLIN    I*4     1           MAXIMUM NUMBER ALLOWABLE CAD/CAM
C                              LINE ENTITIES
C      MAXPNT    I*4     1           MAXIMUM NUMBER ALLOWABLE CAD/CAM
C                              POINT ENTITIES
C      MAXSPL    I*4     1           MAXIMUM NUMBER ALLOWABLE CAD/CAM
C                              B-SPLINE ENTITIES
C      MAXBRK    I*4     1           MAXIMUM ORDER ALLOWED FOR B-SPLINE
C                              DEFINING EQUATION DURING CAD/CAM
C      MAXTRAN   I*4     1           MAXIMUM NUMBER ALLOWABLE CAD/CAM
C                              TRANSFORMATION ENTITIES
C
*/
```

```
/* robsim_tokens.h */

/* These token macros are for aiding the creation of user-defined functions */

/*     <amasbk.h>*/
#define LINK_MASS(a,i) amasbk->atmas[a][i]
#define LINK_CG(a,i,u) amasbk->atcg[a][i][u]
#define LINK_INERTIA(a,i,r,c) amasbk->ainmat[a][i][c][r]
#define EFFJT(a,i) amasbk->effjt[a][i]
#define EFFKT(a,i) amasbk->effkt[a][i]
#define FRICTION(a,j) amasbk->fcoef[a][j]
#define VISCOUS(a,j) amasbk->dcoef[a][j]
#define COULOMB(a,j) amasbk->cfcoef[a][j]

/*     <basgmbk.h>*/
#define IBRATE(a) basgmbk->ibrate[a]
#define BASE_SEGMENT_START_TIME(a) basgmbk->btimseg[a]
#define BASE_MOTION(a,u,q) basgmbk->bacoef[a][u][q]
#define BASE_ATTRIBUTES(a,u) basgmbk->baposit[a][u]
#define BASE_ROTATION(a,r,c) basgmbk->baposor[a][c][r]
#define BASE_DELPOS(a,u) basgmbk->bdelpos[a][u]
#define BASE_DELORI(a,u) basgmbk->bdphi[a][u]
#define BASE_INITPOS(a,u) basgmbk->bsegpos[a][u]
#define BASE_INITROT(a,r,c) basgmbk->bsegor[a][c][r]
#define BPH(a) basgmbk->bph[a]
#define TBDR(a,u) basgmbk->tbdr[a][u]
#define BASE_VELOCITY(a,u) basgmbk->basvel[a][u]
#define BASE_OMEGA(a,u) basgmbk->basom[a][u]
#define BASE_ACCEL(a,u) basgmbk->basacc[a][u]
#define BASE_OMEGA_DOT(a,u) basgmbk->basomd[a][u]
#define BASE_FORCE(a,u) basgmbk->basfor[a][u]
#define BASE_TORQUE(a,u) basgmbk->bastor[a][u]

/*     <chkvelbk.h>*/
#define IAPR(a) chkvelbk->iapr[a]
#define SCALE_FACTOR(a) chkvelbk->iapr[a]

/*     <cnstbk.h>*/
#define ICNST(a) cnstbk->icnst[a]
#define NUM_CONSTRAINTS(a) cnstbk->ncnst[a]
#define CONSTRAINT_TYPE(a,c) cnstbk->idcnst[a][c]
#define CONSTRAINT_SIGN(a,c) cnstbk->isdcnst[a][c]
#define NUM_CONSTRAINT_PLANES(a) cnstbk->npcns[a]
#define CONSTRAINT_POINT(a,c,u) cnstbk->ptcnst[a][c][u]
#define CONSTRAINT_VECTOR(a,c,u) cnstbk->vcnst[a][c][u]
#define CONSTRAINT_VELOCITY(a,c) cnstbk->scnst[a][c]
#define CONSTRAINT_FORCE(a,c) cnstbk->fcnst[a][c]
#define CONSTRAINT_PLANE(a,c,p) cnstbk->pcnst[a][c][p]

/*     <cntrbk.h>*/

#define P_GAIN(a,j) cntrbk->pgain[a][j]
#define D_GAIN(a,j) cntrbk->dgain[a][j]
```

```
#define I_GAIN(a,j) cntrbk->sigain[a][j]
#define ERROR_INT(a,j) cntrbk->errint[a][j]
#define THETA_REF(a,j) cntrbk->thref[a][j]
#define OMEGA_REF(a,j) cntrbk->thdref[a][j]
#define FORCE_P_GAIN(a,j) cntrbk->fpgain[a][j]
#define FORCE_D_GAIN(a,j) cntrbk->fdgain[a][j]
#define FORCE_I_GAIN(a,j) cntrbk->figain[a][j]
#define FORCE_ERROR_INT(a,j) cntrbk->ferrint[a][j]
#define SIGNAL_INTERVAL cntrbk->signew
#define LAST_SIGNAL_TIME cntrbk->siglast

/*	<complbk.h>*/
#define STIFFNESS(a,r,c) complbk->rkend[a][c][r]
#define FORCE_BIAS(a,u) complbk->fbias[a][u]
#define CONTROL_TORQUE(a,j) complbk->control_torque[a][j]
#define ISTF(a) complbk->istf[a]

/*	<envtbk.h>*/
#define NUM_ENVIRON envtbk->nume
#define GRAVITY(u) envtbk->grav[u]
#define ENVIRON_SPAN envtbk->envspn

/*	<forcbk.h>*/
#define JOINT_FORCE(a,i,u) forcbk->fjt[a][i][u]
#define CG_FORCE(a,i,u) forcbk->fcg[a][i][u]
#define END_FORCE(a,u) forcbk->fend[a][u]
#define JOINT_TORQUE(a,i,u) forcbk->tjt[a][i][u]
#define END_TORQUE(a,u) forcbk->tend[a][u]
#define ACTUATOR_TORQUE(a,j) forcbk->tac[a][j]
#define JOINT_FORCE_LMT(a,i,e) forcbk->fjtlmt[a][i][e]
#define JOINT_TORQUE_LMT(a,i,e) forcbk->tjtlmt[a][i][e]
#define ACTUATOR_TORQUE_LMT(a,j,e) forcbk->taclmt[a][j][e]
#define RATE_LMT(a,j,e) forcbk->ratlmt[a][j][e]
#define DRIVING_TORQUE(a,j) forcbk->tdr[a][j]
#define STEP_ENERGY(a,j) forcbk->energstp[a][j]
#define TOTAL_ENERGY(a,j) forcbk->energtot[a][j]

/*	<fortorbk.h>*/
#define IFTCN(a) fortorbk->iftcn[a]
#define FORCE_COMP_NUM(a) fortorbk->nforc[a]
#define TORQUE_COMP_NUM(a) fortorbk->ntorq[a]
#define FORCE_VECTOR(a,r,u) fortorbk->vforc[a][r][u]
#define FORCE_MAG(a,r) fortorbk->sforc[a][r]
#define TORQUE_VECTOR(a,r,u) fortorbk->vtorq[a][r][u]
#define TORQUE_MAG(a,r) fortorbk->storq[a][r]
#define EE_FORCE(a,u) fortorbk->eefor[a][u]
#define EE_TORQUE(a,u) fortorbk->eetor[a][u]
#define POSITION_REF(a,u) fortorbk->posref[a][u]
#define ROTATION_REF(a,r,c) fortorbk->orref[a][c][r]
#define ERROR_POS(a,u) fortorbk->erpos[a][u]
#define ERROR_ORIENT(a,u) fortorbk->eror[a][u]
#define DELTA_TORQUE(a,j) fortorbk->torjnt[a][j]
#define OLD_DELTA_TORQUE(a,j) fortorbk->oldtjt[a][j]
#define REFERENCE_FT(a,u) fortorbk->ftref[a][u]
```

```
#define POS_DIFF(a,u) fortorbk->delp[a][u]
#define FT_DIFF(a,u) fortorbk->delft[a][u]

/*      <geombk.h>*/
#define NUM_ARMS geombk->narm
#define NUM_JOINTS(a) geombk->nj[a]
#define JOINT_TYPE(a,j) geombk->jttyp[a][j]
#define TRAVEL_LMT(a,j,e) geombk->trvlmt[a][j][e]
#define JOINT_LOCATION(a,i,u) geombk->ajtloc[a][i][u]
#define JOINT_ROTATION(a,i,r,c) geombk->or[a][i][c][r]
#define GRASP_POINT(a,u) geombk->agp[a][u]
#define APPROACH_VECTOR(a,u) geombk->aapvec[a][u]
#define NEWFILE(a) geombk->newfile[a]
#define ARM_ORDER(a) geombk->armord[a]
#define ARMHOLD(a) geombk->armhold[a]
#define CURRENT_ARM geombk->armh
#define ARM_NAME(a) geombk->armnam[a]

/*      <jacobbk.h>*/
#define JACOBIAN(a,r,c) jacobbk->rjacob[a][c][r]
#define INVERSE_JACOBIAN(a,r,c) jacobbk->ijacob[a][c][r]

/*      <loadbk.h>*/

#define NUM_LOADS loadbk->nload
#define NUM_LOAD_PT_MASSES(d) loadbk->nptmas[d]
#define LOAD_SEQ(d) loadbk->ldseq[d]
#define LOAD_HOLDER(a,obj) loadbk->held_by[a][d]
#define LOAD_LENGTH(d) loadbk->aldlen[d]
#define LOAD_RADIUS(d) loadbk->aldrad[d]
#define LOAD_MASS(d) loadbk->aldmas[d]
#define LOAD_CG(d,u) loadbk->aldcg[d][u]
#define LOAD_INERTIA(d,r,c) loadbk->aldin[d][c][r]
#define LOAD_GRIP_POINT(d,u) loadbk->rlgp[d][u]
#define LOAD_APPROACH_VEC(d,u) loadbk->apvec[d][u]
#define LOAD_PT_MASS(d,p) loadbk->aldpms[d][p]
#define LOAD_PT_MASS_LOC(d,p,u) loadbk->aldpml[d][p][u]
#define LOAD_ORIENTATION(d,u) loadbk->angld[d][u]
#define LOAD_POSITION(d,u) loadbk->aldloc[d][u]
#define LOAD_ROTATION(d,r,c) loadbk->aldor[d][c][r]
#define LOAD_TOTAL_MASS(d) loadbk->atldms[d]
#define LOAD_TOTAL_CG(d,u) loadbk->atldcg[d][u]
#define LOAD_TOTAL_INERTIA(d,r,c) loadbk->atldin[d][c][r]
#define LOAD_NAME(d) loadbk->ldnam[d]

/*      <motorbk.h>*/
#define MOTOR_TORQUE(a,j) motorbk->torki[a][j]
#define MOTOR_RESISTANCE(a,j) motorbk->armra[a][j]
#define MOTOR_INDUCTANCE(a,j) motorbk->armla[a][j]
#define MOTOR_GAIN(a,j) motorbk->ampga[a][j]
#define MOTOR_RATIO(a,j) motorbk->gratn[a][j]
#define MOTOR_EMF(a,j) motorbk->emfkb[a][j]
#define MOTOR_MOMENT(a,j) motorbk->motin[a][j]
#define MOTOR_VOLTAGE(a,j) motorbk->ampve[a][j]
```

```
#define MOTOR_CURRENT(a,j) motorbk->armcu[a][j]

/*      <movbasbk.h>*/
#define IMOVBAS movbasbk->imovbas
#define NUM_BASES movbasbk->nbas
#define IBASSIM movbasbk->ibassim
#define IBASTRQ movbasbk->ibastrq
#define IBSAV movbasbk->ibsav
#define MOUNTED(a) movbasbk->kbasnum[a]
#define BTLAST movbasbk->btlast
#define BTNEXT movbasbk->btnext

/*      <outrefbk.h>*/
#define REFERENCE_THETA(a,j) outrefbk->reftht[a][j],
#define REFERENCE_JOINT_ERROR(a,j) outrefbk->rjterr[a][j],
#define DESIRED_POSITION(a,u) outrefbk->epos[a][u],
#define ARM_EPSILON(a,u) outrefbk->epserr[a][u],
#define DESIRED_ROTATION(a,r,c) outrefbk->eor[a][c][r];

/*      <segmbk.h>*/
#define IMOT segmbk->imot
#define MOTION(a,j,q) segmbk->pcoef[a][j][q]
#define ARM_SEGMENT_START_TIME(a) segmbk->timseg[a]
#define ARM_CONTROL(a) segmbk->jnttul[a]
#define IRATE(a) segmbk->irate[a]
#define TOOLRP(a,u) segmbk->toolrp[a][u]
#define POSIT(a,j) segmbk->posit[a][j]
#define POSOR(a,r,c) segmbk->posor[a][c][r]
#define SEGTH(a,j) segmbk->segth[a][j]
#define SEGPOS(a,u) segmbk->segpos[a][u]
#define SEGOR(a,r,c) segmbk->segor[a][c][r]
#define DELTH(a,j) segmbk->delth[a][j]
#define DELPOS(a,u) segmbk->delpos[a][u]
#define DPHI(u) segmbk->dphi[u]
#define PH segmbk->ph
#define IDCC segmbk->idcc
#define ARM_TASK(a) segmbk->itask[a]
#define NT(a) segmbk->nt[a]
#define SENSOR_VELOCITY(a) segmbk->senvel[a]
#define ARM_SEGMENT_START(a) segmbk->sgstrt[a]
#define ARM_SEGMENT_STOP(a) segmbk->sgstop[a]
#define BASE_SEGMENT_START(a) segmbk->bsgstrt[a]
#define BASE_SEGMENT_STOP(a) segmbk->bsgstop[a]

/*      <sensbk.h>*/
#define SENSED_THETA(a,j) sensbk->sth[a][j]
#define SENSED_OMEGA(a,j) sensbk->sthd[a][j]
#define SENSED_OMEGA_DOT(a,j) sensbk->sthdd[a][j]
#define OLD_SENSED_THETA(a,j,b) sensbk->oldsth[a][j][b]
#define SENSED_FORCES(a,j) sensbk->sensed_forces[a][j]

/*      <sprgbk.h>*/
#define SPRING_LOCATION(a,u) sprgbk->sprp[a][u]
#define SPRING_ROTATION(a,r,c) sprgbk->spror[a][u][c][r]
```

```
#define SPRING_CONSTANTK(a,r,c) sprgbk->sprk[a][u][c][r]
#define SPRING_REST_POSITION(a,u) sprgbk->splc[a][u]
#define SPRING_FORCE(a,u) sprgbk->sprf[a][u]
#define ISPR(a) sprgbk->ispr[a]

/*      <systbk.h>*/
#define LINK_THETA(a,j) systbk->th[a][j]
#define LINK_OMEGA(a,j) systbk->thd[a][j]
#define LINK_OMEGA_DOT(a,j) systbk->thdd[a][j]
#define LINK_POSITION(a,f,u) systbk->pos[a][f][u]
#define LINK_VELOCITY(a,f,u) systbk->vel[a][f][u]
#define LINK_ACCELERATION(a,f,u) systbk->acc[a][f][u]
#define LINK_ROTATION(a,i,r,c) systbk->rot[a][i][c][r]
#define LINK_WORLD_OMEGA(a,i,u) systbk->om[a][i][u]
#define LINK_WORLD_OMEGA_DOT(a,i,u) systbk->omd[a][i][u]
#define JOINT_VECTOR(a,j,u) systbk->hij[a][j][u]
#define ARM_CG(a,i,u) systbk->hcg[a][i][u]
#define ARM_INERTIA(a,i,r,c) systbk->ainw[a][i][c][r]
#define VECTOR_CG(a,i,j,u) systbk->systr[a][j][i][u]
#define DOUBLE_ARM_FORCE(u) systbk->fdcc[u]

/*      <targbk.h>*/
#define NUM_TARGETS targbk->ntarg
#define ITRGSEQ(t) targbk->itrgseq[t]
#define NUM_TARGET_PT_MASSES(t) targbk->nptrmas[t]
#define TARGET_HOLDER(t) targbk->atrgheld[t]
#define TARGET_LENGTH(t) targbk->atrglen[t]
#define TARGET_RADIUS(t) targbk->atrgrad[t]
#define TARGET_MASS(t) targbk->atrgmas[t]
#define TARGET_CG(t,u) targbk->atrgcg[t][u]
#define TARGET_INERTIA(t,r,c) targbk->atrgin[t][c][r]
#define TARGET_GRIP_POINT(s,u) targbk->atrgrlgp[s][u]
#define TARGET_APPROACH_VEC(s,u) targbk->atrgapvec[s][u]
#define TARGET_PT_MASS(t,p) targbk->atrgpms[t][p]
#define TARGET_PT_MASS_LOC(t,p,u) targbk->atrgpml[t][p][u]
#define TARGET_ORIENTATION(t,u) targbk->angtrg[t][u]
#define TARGET_POSITION(t,u) targbk->atrgloc[t][u]
#define TARGET_ROTATION(t,r,c) targbk->atrgor[t][c][r]
#define TARGET_TOTAL_MASS(t) targbk->attrgms[t]
#define TARGET_TOTAL_CG(t,u) targbk->attrgcg[t][u]
#define TARGET_TOTAL_INERTIA(t,r,c) targbk->attrgin[t][c][r]
#define TARGET_NAME(t) targbk->targnam[t]

/*      <timebk.h>*/
#define CURRENT_TIME timebk->time
#define START_TIME timebk->time
#define STOP_TIME timebk->time
#define PROCESS_STEP timebk->time
#define SOF_STEP timebk->time
#define PRT_STEP timebk->time
#define AVT_STEP timebk->time
#define PLOT_STEP timebk->time
#define TORQUE_STEP timebk->time
#define GRAPHICS_STEP timebk->time
```

```
/*      <toolbk.h>*/
#define GRASPING(a,ld) toolbk->grasping[a][ld]
#define TOOL_MASS(a) toolbk->tulmas[a]
#define TOOL_CG(a,u) toolbk->tulcg[a][u]
#define TOOL_INERTIA(a,r,c) toolbk->tulin[a][c][r]
#define TOOL_TO_LOAD_TRANS(a,ld,u) toolbk->ttoldo[a][ld][u]
#define TOOL_TO_LOAD_ROT(a,r,c) toolbk->ttulld[a][ld][c][r]
#define TTOBSO(a,u) toolbk->ttobso[a][u]
#define TTULBS(a,r,c) toolbk->ttulbs[a][c][r]

/*      <user_data.h>*/
#define USER_INPUT_INT(x) user_data->input.ints[x]
#define USER_INPUT_FLT(x) user_data->input.floats[x]
#define USER_INPUT_CHR(x) user_data->input.chars[x]
#define USER_OUTPUT_INT(x) user_data->output.ints[x]
#define USER_OUTPUT_FLT(x) user_data->output.floats[x]
#define USER_OUTPUT_CHR(x) user_data->output.chars[x]

/* This next bit defines the parameter list used to pass the structure
   pointers around the C functions:  */
#define STRUCT_LIST amasbk,basgmbk,chkvelbk,cnstbk,cntrbk,complbk,envtbk,\
forcbk,fortorbk,geombk,jacobbk,loadbk,motorbk,movbasbk,outrefbk,segmbk,\
sensbk,sprgbk,systbk,targbk,timebk,toolbk,user_data

#define X_COORD 0
#define Y_COORD 1
#define Z_COORD 2
#define ROLL 5
#define PITCH 4
#define YAW 3
#define ARM1 0
#define ARM2 1
#define ARM3 2
#define ARM4 3
#define ARM5 4
```

```
/* struct_defs.h */
/* Use this file as follows:
   a_type a_function(STRUCT_LIST)
   #include struct_defs.h

   This will define all of the structures in STRUCT_LIST.
*/

#include <amasbk.h>
#include <basgmbk.h>
#include <chkvelbk.h>
#include <cnstbk.h>
#include <cntrbk.h>
#include <complbk.h>
#include <envtbk.h>
#include <forcbk.h>
#include <fortorbk.h>
#include <geombk.h>
#include <jacobbk.h>
#include <loadbk.h>
#include <motorbk.h>
#include <movbasbk.h>
#include <outrefbk.h>
#include <segmbk.h>
#include <sensbk.h>
#include <sprgbk.h>
#include <systbk.h>
#include <targbk.h>
#include <timebk.h>
#include <toolbk.h>
#include <user_data.h>
```

```
/* amasbk.h */
struct {
   float
         atmas[MXARMS][MXLNKS+2],/* total mass of link between joint i and i+1 */
         atcg[MXARMS][MXLNKS+2][3],/* resultant location of center of mass of link
                                          between joint i and joint i+1 in joint i system
*/
         ainmat[MXARMS][MXLNKS+2][3][3],/* total inertia matrix for each link, in local
                                          coordinates */
         effjt[MXARMS][MXLNKS],/* effective moment of inertia of armature &
                                          motor gear,
                                          for each joint */
         effkt[MXARMS][MXLNKS],/* effective viscous frictional coefficient of
                                          armature and motor gear, for each joint */
         fcoef[MXARMS][MXLNKS],/* friction coefficient for each joint */
         dcoef[MXARMS][MXLNKS],/* viscous friction coefficient for each joint */
         cfcoef[MXARMS][MXLNKS];/* coulomb friction coefficient for each joint
                                          */
         } *amasbk;



/* basgmbk.h */

struct {
   int
         ibrate[MXARMS];          /* Base rate or position control flag
                                          1=Rate, 2=Position */
   float
         btimseg[MXARMS],         /* Segment start time array */
         bacoef[MXARMS][6][MAXORD+1],
         baposit[MXARMS][6],/* Base position */
         baposor[MXARMS][3][3],
         bdelpos[MXARMS][3],
         bdphi[MXARMS][3],
         bsegpos[MXARMS][3],
         bsegor[MXARMS][3][3],
         bph[MXARMS],
         tbdr[MXARMS][6],
         basvel[MXARMS][3],
         basom[MXARMS][3],
         basacc[MXARMS][3],
         basomd[MXARMS][3],
         basfor[MXARMS][3],
         bastor[MXARMS][3];
         } *basgmbk;




/* bldabk.h */
```

```
struct {
   float
         ratlmt[MXLNKS][2],          /* Joint rate limit minimum and maximum */
         torki[MXLNKS],              /* Actuator torque constant per joint */
         armra[MXLNKS],              /* Actuator armature resistance per joint */
         armla[MXLNKS],              /* Actuator armature inductance per joint */
         ampga[MXLNKS],              /* Actuator amplifier gain per joint */
         gratn[MXLNKS],              /* Actuator gear ratio per joint */
         emfkb[MXLNKS];              /* Actuator back EMF constant per joint */
         } *bldabk;



/* bldgbk.structure */


#define   KARM  1
#define      FIXED  0
#define      HINGE  1
#define      SWIVEL  2
#define      PRISMATIC  3
#define      REVOLUTE  4

struct {
   int
         narm ,                              /* Number of arms = 1 for arm definition */
                                             /* the assignment narm = 1 was cancelled out for */
                                             /* the purposes of this structure, which is concerned
*/
                                             /* with only the location of the variables */
         nj[MXARMS],          /* Number of joints in arm */
         jnttyp[MXARMS][MXLNKS];/* Joint type: fixed = 0
                                                   hinge = 1
                                                   swivel = 2
                                                   prismatic = 3
                                                   revolute = 4 */
   float
         trvlmt[MXARMS][MXLNKS][2],/* Rotation sequence for base, joints, & tool */
         ajtvar[MXARMS][MXLNKS],/* Initial joint variable */
         dh[MXARMS][MXLNKS+2][4][4];/* Orientation matrix for each arm component */

   } *bldgbk;



/* bldmbk.h */

struct   {
   int
         ipntms[MXLNKS+2];           /* number of point masses (base, joints, tool)
                                                                  */
   float
         alkin[MXLNKS+2][3][3],/* inertia matrix (base, joints, tool) */
```

```
        alcpms[MXLNKS+2][MXPTS][3],/* point masses CG */
        alkcg[MXLNKS+2][3],/* center of mass location (base, joints, tool)
        alkmas[MXLNKS+2],/* mass of base, joints, tool */
        aptmas[MXLNKS+2][MXPTS],/* mass of point masses */
        atmas[MXLNKS+2],          /* total mass including point masses */
        atcg[MXLNKS+2][3],/* effective cg including point masses */
        ainmat[MXLNKS+2][3][3],/* effective inertia matrix including point
                                                 masses */
        effjt[MXLNKS],            /* effective actuator inertia per joint */
        effkt[MXLNKS],            /* effective actuator viscous friction */
        fcoef[MXLNKS],            /* dry friction per joint */
        dcoef[MXLNKS],            /* not used or referenced */
        cfcoef[MXLNKS];           /* coulomb friction per joint */
        } *bldmbk;



/* bldsjbk.h */
struct {
   int
        nspjt,                                  /* number of special joints */
        isptyp[MXSPJT],           /* special joint types */
        njtsp[MXSPJT];            /* which joint is a special joint */
        } *bldsjbk;



/* cadarybk.h */

struct {
   float
        trans[MAXTRAN][4][3],
        alines[MAXLIN][2][3],
        pnts[MAXPNT][3],
        arcpnts[MAXARC*(IDIV+1)][3],
        arcpt1[3],
        arcs[MAXARC][7],
        splins[MAXSPL][30];
        } *cadarybk;


/* cadflgbk.h */

struct {
   short int
        idrew,
        idrawl,
        idrawc,
        idrawp,
        idraws;
        } *cadflgbk;
/* cadiarybk.h */

struct {
```

```
   short int
         ipnts[MAXPNT][3],
         ilines[2*MAXLIN][3],
         nncrv[MAXARC],
         iarcs[MAXARC*(IDIV+1)][3],
         iarcs2[MAXARC][IDIV+1][3],
         nnspl[MAXSPL],
         isplins[MAXSPL*(IDIV+1)][MAXBRK*12];
         } *cadiarybk;



/* cadinprbk.h */

struct {
   short int
         itype[MAXFLD],
         istrtl[MAXFLD],
         ivers[MAXFLD],
         ifont[MAXFLD],
         ilevl[MAXFLD],
         matptr[MAXFLD],
         numfld[MAXFLD],
         itranl[MAXFLD];
         } *cadinprbk;



/* cadivdbk.h */

struct {
   float
         salines[MAXLIN][2][3];
   int
         numrd;
   short int
         numsal,
         idivsv,
         idivsp;
         } *cadivdbk;



/* cadrepbk.h */

struct {
   short int
         nnl,
         l1l,
         l2l,
         l1c,
         l2c,
         l1s,
         l2s;
```

```
   int
         numt,
         numl,
         nump,
         numc,
         nums,
         icircl;
         } *cadrepbk;



/* chkvelbk.h */

struct {
   int
         iapr[MXARMS];              /* flag indicating whether equations
                                            were solved exactly:
                                             = 0, exact solution,
                                             = 1, B not in reachable space,
                                             = 2, B in reachable space but
                                                   could not be solved without
                                                   exceeding limits on X. */
   float
         sf[MXARMS];                /* scale factor which gives the
                                            proportion of the actual velocity
                                            to the reachable components of the
                                            desired velocity */
         } *chkvelbk;



/* cnstbk.h */

#define MXCONS  4

struct {
   int
         icnst[MXARMS],             /* flag, motion of arm is constrained
                                             = 0, unconstrained
                                             = 1, constrained */
         ncnst[MXARMS],             /* number of constraint vectors (<= 6) */
         idcnst[MXARMS][6],         /* single- or double-acting constraint,
                                             = 0, single-acting
                                             = 1, double-acting */
         isdcnst[MXARMS][6],/* sign indicating direction of double-acting
                                              constraint */
         npcns[MXARMS];             /* number of constraint planes */
   float
         ptcnst[MXARMS][6][3],/* location (world coordinates) of each
                                              constraint point */
         vcnst[MXARMS][6][3],/* constraint vector indicating constrained
                                              motion direction */
         scnst[MXARMS][6],          /* magnitude of velocity in constraint
                                              direction */
```

```
        fcnst[MXARMS][6],           /* magnitude of reaction force along each
                                                  constraint direction */
        pcnst[MXARMS][MXCONS][4];/* coefficients defining constraint plane */
        } *cnstbk;



/* cntrbk.h */

struct {
   float
        pgain[MXARMS][MXLNKS],/* proportional gain constant for joints
                                                 when in position control */
        dgain[MXARMS][MXLNKS],/* derivative gain constant for joints
                                                 when in position control */
        sigain[MXARMS][MXLNKS],/* integrating gain constant for joints
                                                 when in position control */
        errint[MXARMS][MXLNKS],/* integrating error for position control */
        thref[MXARMS][MXLNKS],/* joint angle reference signals for
                                                 simulation control */
        thdref[MXARMS][MXLNKS],/* joint angular velocity ref signals for
                                                 simulation control */
        fpgain[MXARMS][MXLNKS],/* proportional gain constant for joints
                                                 when in F/T control */
        fdgain[MXARMS][MXLNKS],/* derivative gain constant for joints
                                                 when in F/T control */
        figain[MXARMS][MXLNKS],/* integrating gain constant for joints
                                                 when in F/T control */
        ferrint[MXARMS][MXLNKS],/* integrating error used in force/torque
                                                control */
        signew,                           /* control signal update interval */
        siglast;                          /* time that control signal was last updated */
        } *cntrbk;



/* complbk.h */

struct {
   float
        rkend[MXARMS][6][6],/* Stiffness matrix at
                                                 end-effector reference point */

/*
        cigain[MXARMS][MXLNKS], Integrator gain constant
                                                 for stiffness control
        cdgain[MXARMS][MXLNKS],Derivative gain constant
                                                for stiffness control
        cllgain[MXARMS][MXLNKS], lead-lag filter gain constant
        tadv[MXARMS][MXLNKS], advance time constant
        tlag[MXARMS][MXLNKS], lag time constant
        olddelt[MXARMS][MXLNKS], old joint torque deltas
        olddel[MXARMS][MXLNKS], old joint thetas
        olddlo[MXARMS][MXLNKS], old lead-lag filter output
```

```
        serrint[MXARMS][MXLNKS], integrator error for
                                                    stiffness control
*/

        fbias[MXARMS][6],           /* end effector bias force */
        control_torque[MXARMS][MXLNKS];/* control torque calculated in CNTRSIG */
   int
        istf[MXARMS];               /* Flag for active compliance control
                                                    = OFF, no compliance control
                                                    = ON, use compliance control */
        } *complbk;



/* dynmbk.h */

struct {
   float
        phi[6+MXARMS*MXLNKS][6+MXARMS*MXLNKS];/* total system inertia array */
        } *dynmbk;



/* envtbk.h */

struct {
   short int
        ienv[10*MXGRAFPT],/* data array for graphics description
                                                    of environment */
        nume,                                      /* number of components in environment */
        ictenv,                                    /* number of components in ienv */
        nstrte[MXENCMPS];           /* location in ienv where each environment
                                                    component begins */
   float
        grav[3],                         /* gravity vector */
        envspn;                          /* span for environment */
        } *envtbk;



/* flnambk.h */

struct {
   char
        file1[FILE_LENGTH],/* (r, ) interactive input device */
        file2[FILE_LENGTH],/* ( ,w) interactive output device */
        file3[FILE_LENGTH],/* ( ,w) print device */
        file4[FILE_LENGTH],/* (r,w) arm data file or iges cad/cam file */
        file5[FILE_LENGTH],/* (r,w) environment data file */
        file6[FILE_LENGTH],/* ( ,w) simulation output file/postprocessing
                                                     input */
        file7[FILE_LENGTH],/* (r, ) actuator parameter file */
        file8[FILE_LENGTH],/* (r, ) time history file */
        file9[FILE_LENGTH],/* ( , ) no references */
```

```
        file10[FILE_LENGTH],/* (r,w) load data file */
        file11[FILE_LENGTH],/* (r, ) error message file */
        file12[FILE_LENGTH],/* (r, ) PID control file */
        file13[FILE_LENGTH],/* (r,w) torque i/o file */
        file14[FILE_LENGTH],/* (r,w) acc-vel-th file */
        file15[FILE_LENGTH],/* (r,w) requirement option file */
        file16[FILE_LENGTH],/* (r,w) plot data file */
        file17[MXARMS][FILE_LENGTH],/* (r,w) time history file */
        file18[FILE_LENGTH],/* (r, ) hardware control signal file */
        file19[FILE_LENGTH],/* ( , ) no reference */
        file20[FILE_LENGTH],/* ( ,w) arm geometry print file */
        file27[FILE_LENGTH],/* (r,w) target data file */
        file30[MXARMS][FILE_LENGTH],/* (r,w) base time history file */
        file40[FILE_LENGTH],/* (r, ) base acc-vel-pos file */
        file42[FILE_LENGTH],/* (r, ) constraint file */
        file43[FILE_LENGTH],/* ( ,w) base torque data file */
        file44[FILE_LENGTH],/* (r,w) detailed component definition data
                                                  file */
        file45[FILE_LENGTH];/* (r,w) AL program name */
        } *flnambk;
```

```
/* forcbk.h */

struct {
   float
        fjt[MXARMS][MXLNKS+2][3],/* force vector at each joint of each arm */
        fcg[MXARMS][MXLNKS+2][3],/* force vector at centroid of each arm */
        fend[MXARMS][3],            /* force vector at end effector */
        tjt[MXARMS][MXLNKS+2][3],/* torque vector at each joint of each arm */
        tend[MXARMS][3],            /* torque vector at end effector */
        tac[MXARMS][MXLNKS],/* torque vector at each actuator */
        fjtlmt[MXARMS][MXLNKS+2][2],/* force limit of each joint */
        tjtlmt[MXARMS][MXLNKS+2][2],/* torque limit of each joint */
        taclmt[MXARMS][MXLNKS][2],/* torque limit of each actuator */
        ratlmt[MXARMS][MXLNKS][2],/* rate limit of each joint */
        tdr[MXARMS][MXLNKS],/* driving torque for each joint */
        energstp[MXARMS][MXLNKS],/* energy used per time step */
        energtot[MXARMS][MXLNKS];/* total energy used */
        } *forcbk;
```

```
/* fortorbk.h */

struct {
   int
        iftcn[MXARMS],              /* flag for force/torque control on or off */
        nforc[MXARMS],              /* number of force components to be
                                                  controlled (0 to 3) */
        ntorq[MXARMS];              /* number of torque components to be
                                                  controlled (0 to 3) */
   float
```

```
        vforc[MXARMS][3][3],/* unit vectors in direction of forces to be
                                        controlled */
        sforc[MXARMS][3],       /* magnitude of each force vector */
        vtorq[MXARMS][3][3],/* unit vectors in direction of torques to be
                                        controlled */
        storq[MXARMS][3],       /* magnitude of each torque vector */
        eefor[MXARMS][3],       /* force at the end-effector */
        eetor[MXARMS][3],       /* torque at the end-effector */
        posref[MXARMS][3],/* reference position of end-effector ref point
                                            */
        orref[MXARMS][3][3],/* reference orientation of end-effector */
        erpos[MXARMS][3],       /* difference between end-effector reference
                                        position and actual position */
        eror[MXARMS][3],        /* vector denoting difference between end-
                                        effector reference orientation and actual
                                        orientation */
        torjnt[MXARMS][MXLNKS],/* torque deltas at each joint (used in
                                        force/torque control) */
        oldtjt[MXARMS][MXLNKS],/* torque deltas from previous update time */
        ftref[MXARMS][6],       /* user-specified reference force */
        delp[MXARMS][6],        /* error in position */
        delft[MXARMS][6];       /* error in force/torque */
        } *fortorbk;



/* fricbk.h */

struct {
   float
        cfric[MXARMS][6],
        vfric[MXARMS][6][3],
        ffric[MXARMS][6];
        } *fricbk;



/* geombk.h */

struct {
   int
        narm,                           /* number of exisiting arms */
        nj[MXARMS],             /* number of joints in each arm */
        jttyp[MXARMS][MXLNKS];/* number denoting joint type of each joint in
                                        each arm:
                                        = 1, hinge joint (y-axis rot)
                                        = 2, swivel joint (x-axis rot)
                                        = 3,   sliding joint (x-axis trans) */
   float
        trvlmt[MXARMS][MXLNKS][2],/* travel limits of each joint */
        ajtloc[MXARMS][MXLNKS+2][3],/* location of joint i in joint i-1 coord. system */
        or[MXARMS][MXLNKS+2][3][3],/* transformation from joint i to joint i-1 coord. system
*/
        agp[MXARMS][3],         /* arm grasp point */
```

```
        aapvec[MXARMS][3];/* arm approach vector */
    int
        newfile[MXARMS],            /* flag indicating new arm file */
        armord[MXARMS],             /* array containing order to process arms */
        armhold[MXARMS],            /* array containing arm# holding current arm */
        armh;                                       /* arm number */
    char
        armnam[MXARMS][8];/* arm names */
        } *geombk;



/* getlib.h */

struct {
    char
        uinput[LINE_LENGTH],
        prompt[LINE_LENGTH],
        nth[20][5];
        } *get;



/* grafbk.h */

    /* these parameters are used to make the
      graphics code more readable */
#define  ARM_CREATE  1
#define  ARM_MODIFY  2

struct {

    int
        nsdlnk[MXLNKS+2];
    short int
        iarmobj[MXLNKS*MXGRAFPT+MXLNKS*MXGRAFPT],
        /* (why not just 2*MXLNKS*MXGRAFPT?) --Matt Gertz */
        icntarm,
        numarm[MXLNKS+2],
        nstarm[MXLNKS+2][MXGRCMP];
    float
        armspn,
        endpts[MXLNKS][2],
        alkrad[MXLNKS+2];
        } *grafbk;




/* interpbk.h */

#define MAXSTREAMS  MXARMS*2/* max. number of instruction streams  */

struct {
```

```
   int first_write;                  /* the Fortran type was a LOGICAL, which
                                        is 4 bytes in size -- I simply needed
                                        to allocate 4 bytes of space and this
                                        seemed the easiest way.  This logical
                                        tells whether we've written a THP file
                                        yet. */
   char prog_name[120];       /* Al program name */

   float
         tstop,                              /* Current movement stop time */
         tstart;                             /* Current movement start time */

   int
         tidcc[MXARMS],               /* Temporary IDCC value */
         timot[MXARMS],               /* Temporary IMOT value */
         tiop[MXARMS],                /* Temporary IOP value */
         tlod[MXARMS],                /* Temporary LOD value */
         init_flag[MAXSTREAMS],/* Has this stream been initialized */
         cimode;                            /* Current mode */
         } *interpbk;



/* intoptbk.h */

struct {
   int
         irdgrfil,
         iwrgrfil,
         igraf;                              /* determines if graphics is enabled */
         } *intoptbk;



/* iobjbk.h */

struct {
   float
         sysspn;                            /* system span or reach */
   short int
         iobj[20*MXGRAFPT],/* array to hold graphics representation */
         icount,                             /* last location in iobj */
         num[MXARMS][MXLNKS+2],/* number of components in current link */
         nstrt[MXARMS][MXLNKS+2][MXGRCMP];/* iobj start location for each
                                                   component */
         } *iobjbk;
```

```c
/* ioptbk.h */

#define I__NONE 0
#define I__WARNINGS 1
#define I__ERRORS 2
#define I__SEVERE_ERRORS 3

#define I__NOGRAPHICS 0
#define I__ROBSIM_GRAPHICS 1
#define I__USER_GRAPHICS 2

#define I__NOPRINT 0
#define I__FULL_PRINT 1
#define I__CONDENSED_PRINT 2
#define I__USER_PRINT 3

#define I__NOPLOT 0
#define I__FULL_PLOT 1
#define I__CONDENSED_PLOT 2
#define I__USER_PLOT 3
#define I__CRICKET_PLOT 4

#define I__NOAVT 0
#define I__ROBSIM_AVT 1
#define I__USER_AVT 2

#define I__NOSOF 0
#define I__ROBSIM_SOF 1
#define I__USER_SOF 2

#define I__USER_MOTION 0
#define I__AVT 1
#define I__THP 2
#define I__G_INPUT 3
#define I__INTERPRETER 4
#define I__BMP 5

#define I__INPUT_TORQUE 0
#define I__HARDWARE_VOLTAGE 1
#define I__PID_CONTROL 2
#define I__LEAD_LAG_TOR_CONTROL 3
#define I__USER_MOTOR_CONTROL 4

#define I__CARTESIAN_NONE 0
#define I__USER_CARTESIAN_CONTROL 1
#define I__FORCE_TORQUE 2
#define I__COMPLIANCE 3

#define I__ROBSIM_RATE 0
#define I__USER_RATE 1

#define I__ROBSIM_POSITION 0
#define I__USER_POSITION 1
```

```
#define I__ROBSIM_TRACKING 0
#define I__USER_TRACKING 1

#define I__ROBSIM_TRAJECTORY 0
#define I__USER_TRAJECTORY 1

#define I__ROBSIM_DYNAMICS 0
#define I__NO_DYNAMICS 1
#define I__USER_DYNAMICS 2

#define I__NO_CONSTRAINT 0
#define I__DEFINE_CONSTRAINT 1

#define I__ROBSIM_PLANT_SENSING 0
#define I__USER_PLANT_SENSING 1

struct {
   int
        idynm,                  /* flag to perform dynamic calculations */
/*
        ifric[MXARMS],
        icoul[MXARMS],
        idamp[MXARMS],
*/
        iprint,                 /* flag to write output data */
        isimo,                  /* flag to write simulation data */
        ibdata,                 /* If ibdata=1, the data is read from an
                                   existing acc-vel-pos file.  If ibdata=2, the
                                   values are calculated from the base motion
                                   profiles. */
        idata,                  /* determines how we read or calculate
                                   positions, velocities, and accelerations. */
        motor_control_mode,     /* respon method for controlling the motion
                                   at th  motor level */
        cartesian_control_mode,/* respon method for controlling the motion
                                   in Cartesian space */
        igraf,                  /* graphics mode switch */
        iplot,                  /* flag to write plot data */
        itorq,                  /* flag to write joint torque data */
        ibtrq,                  /* flag to write base torque data */
        iavt,                   /* flag to write acc-vel-theta file */
/*
        mbtyp[MXARMS],
*/
        iact[MXLNKS],           /* Option flag for actuator definition:
                                  = 1, input interactively actuator parameter
                                        values for joint
                                  = 2, read from a file the actuator parameter
                                        values for joint
                                  = 3, do not include actuator for joint */
/*
        mbflg[MXARMS],
        irdcntl,                  Voltage input data file option flag
```

```
        iwrcntl,
        irdthet,
*/
        rate_selection,                 /* flag specifying which rate algorithm to use
                                                      */
        position_selection,         /* flag specifying which position algorithm to
                                                     use */
        tracking_selection,          /* flag specifying which tracking algorithm to
                                                     use */
        trajectory_selection,       /* flag specifying which trajectory algorithm
                                                     to use */
        pid_control_law,              /* flag specifying which PID control
                                                     algorithm to use in respon loops */
        force_torque_control_law,/* flag specifying which force/torque control
                                                     algorithm to use in respon loops */
        compliance_control_law,/* flag specifying which compliance control
                                                     algorithm to use in respon loops */
        lead_lag_control_law,/* flag specifying which lead-lag control algorithm
                                                    to use in respon loops */
        plant_sensing;                /* flag specifying which plant-sensing algorithm
                                                    to use in respon loops */
   short int
        idebug;                           /* debug printout level */

/* For the following set of char arrays, I have commented out
   the initialization of the arrays, because we are only concerned
   about determining the size and setup of the structure -- mwg */


   char
        cdebug[4][30]                /* debug level options */
/*              = {{"None"},{"Warnings"},{"Errors"},{"Severe_errors"}} */

        ,cgraf[3][30]                    /* graphics mode options */
/*              = {{"No_graphics_output"},{"ROBSIM_graphics"},{"User_function"}} */

        ,cplot[5][30]                    /* plot data option */
/*              = {{"No_plot_file"},{"Full_MatrixX_plot_file"},{"Brief_MatrixX_plot_file"},
                   {"User_function"},{"Cricket_plot_file"}} */

        ,cprint[4][30]                   /* print data options */
/*              = {{"No_printed_output"},{"Verbose_printed_output"},{"Brief_output"},
                   {"User_function"}} */

        ,cavt[3][30]                     /* acceleration velocity theta options */
/*              = {{"No_acceleration_velocity_theta_file"},{"ROBSIM_avt_output"},
                   {"User_function"}} */

        ,csof[3][30]                     /* simulation output options */
/*              = {{"No_simulation_output_file"},{"ROBSIM_sof_output"},{"User_function"}}
*/

        ,cdata[6][30]                    /* Motion specification options */
/*              = {{"User_function"},{"Acc-vel-
```

```
theta_data_file"},{"Polynomial_time_histories"},
                   {"E_and_S_function_keys"},{"Interpret-
er"},{"Read_from_base_motion_profile"}}  */

        ,cmotor_control_mode[5][30]
/*              =
{{"Input_torque"},{"Hardware_voltage"},{"Pid_control"},{"Lead_lag_control_tor"},
                   {"User_function"}}  */

        ,ccartesian_control_mode[4][30]
/*              = {{"None"},{"User_function"},{"Force_Torque"},{"Compliance"}} */

        ,crate[2][30]                       /* Rate control algorithm options */
/*              = {{"ROBSIM_Rate_Control"},{"User_function"}} */

        ,cposition[2][30]           /* Position control algorithm options */
/*              = {{"ROBSIM_Position_Control"},{"User_function"}} */

        ,ctracking[2][30]           /* Tracking control algorithm options */
/*              = {{"ROBSIM_Tracking"},{"User_function"}} */

        ,ctrajectory[2][30]         /* Trajectory generation algorithm options */
/*              = {{"ROBSIM_Trajectory"},{"User_function"}} */

        ,cdynamic[3][30]            /* Dynamic algorithm options */
/*              =
{{"ROBSIM_Dynamic_Computations"},{"No_Dynamic_Computations"},{"User_function"}}
*/

        ,ccnst[2][30]                       /* Constraint options */
/*              = {{"No_constraint"},{"Define_constraint"}} */

        ,cpid_computations[3][30]/* PID control options */
/*              = {{"ROBSIM_PID_formulation"},{"PID_with data_aquisition_delay"},
                   {"User_function"}} */

        ,cft_computations[2][30]/* Force/Torque control options */
/*              = {{"ROBSIM_FORCE_TORQUE_formulation"},{"User_function"}}  */

        ,ccomp_computations[2][30]/* compliance control options */
/*              = {{"ROBSIM_COMPLIANCE_formulation"},{"User_function"}}  */

        ,clead_lag[2][30]           /* lead/lag control options */
/*              = {{"ROBSIM_LEAD_LAG_formulation"},{"User_function"}}  */

        ,cplant_sensing_computations[2][30]/* Plant sensing options */
/*              = {{"ROBSIM_PLANT_SENSING"},{"User_function"}}  */

        ;} *ioptbk;
```

```
/* jacobbk.h */

struct {
   float
         rjacob[MXARMS][MXLNKS][6],/* Jacobian matrix */
         ijacob[MXARMS][6][MXLNKS];/* Inverse of Jacobian matrix */
         } *jacobbk;



/* lgrafbk.h */

struct {
   float
         aldspn;
   short int
         ilodobj[10*MXGRAFPT],
         icntlod,
         numlod[MXLODS],
         nstlod[MXLODS][MXLDCMPS];
         } *lgrafbk;



/* loadbk.h */

struct {
   int
         nload,                          /* number of loads */
         nptmas[MXOBJS],                 /* number of point masses */
         ldseq[MXOBJS],                  /* rotation sequence */
         held_by[MXARMS][MXOBJS];/* arm number holding load
                                            = 0, not held */
   float
         aldlen[MXOBJS],                 /* length of object */
         aldrad[MXOBJS],                 /* radius of object */
         aldmas[MXOBJS],                 /* mass of object */
         aldcg[MXOBJS][3],               /* CG vector in local coordinates */
         aldin[MXOBJS][3][3],/* inertia matrix in local coordinates */
         rlgp[MXOBJS][3],                /* relative grip point */
         apvec[MXOBJS][3],               /* approach vector */
         aldpms[MXOBJS][MXPTS],/* mass of each point mass */
         aldpml[MXOBJS][MXPTS][3],/* location of point masses in local
                                            coordinates */
         angld[MXOBJS][3],               /* orientation angles wrt world coordinates */
         aldloc[MXOBJS][3],              /* load location wrt world coordinates */
         aldor[MXOBJS][3][3],/* load orientation wrt world coordinates */
         atldms[MXOBJS],                 /* total mass including point masses */
         atldcg[MXOBJS][3],              /* CG vector including point masses in local
                                            coordinates */
         atldin[MXOBJS][3][3];/* total inertia matrix including point masses
                                            in local coordinates */
```

```
   char
         ldnam[MXOBJS][8];            /* load names */
         } *loadbk;



/*  lunitbk.h */

struct {
   int
         lu1                                    /* (r, ) interactive input device */
         ,lu2                                   /* ( ,w) interactive output device */
         ,lu3                                   /* ( ,w) print device */
         ,lu4                                   /* (r,w) arm data file or iges cad/cam file */
         ,lu5                                   /* (r,w) environment data file */
         ,lu6                                   /* ( ,w) simulation output file/postprocessing
                                                   input */
         ,lu7                                   /* (r, ) actuator parameter file  */
         ,lu8                                   /* (r, ) time history file */
         ,lu9                                   /* ( , ) no references */
         ,lu10                                  /* (r,w) load data file */
         ,lu11                                  /* (r, ) error message file */
         ,lu12                                  /* (r, ) PID control file */
         ,lu13                                  /* (r,w) torque i/o file */
         ,lu14                                  /* (r,w) acc-vel-th file */
         ,lu15                                  /* (r,w) requirement option file */
         ,lu16                                  /* (r,w) plot data file */
         ,lu17[MXARMS]                /* (r,w) time history file */
         ,lu18                                  /* (r, ) hardware control signal file */
         ,lu19                                  /* ( , ) no reference */
         ,lu20                                  /* ( ,w) arm geometry print file */
         ,lu27                                  /* (r,w) target data file */
         ,lu30[MXARMS]                /* (r,w) base time history file */
         ,lu40                                  /* (r, ) base acc-vel-pos file */
         ,lu42                                  /* (r, ) constraint file */
         ,lu43                                  /* ( ,w) base torque data file */
         ,lu44                                  /* (r,w) detailed component definition data
                                                   file */
/*       ,lu45                                     (r,w) AL program (NOT USED)    */
         ,iopn1
         ,iopn2
         ,iopn3
         ,iopn4
         ,iopn5
         ,iopn6
         ,iopn7
         ,iopn8
         ,iopn9
         ,iopn10
         ,iopn11
         ,iopn12
         ,iopn13
         ,iopn14
         ,iopn15
```

```
        ,iopn16
        ,iopn17[MXARMS]
        ,iopn18
        ,iopn19
        ,iopn20
        ,iopn27
        ,iopn30[MXARMS]
        ,iopn40
        ,iopn43
        ,iopn44
        ,iopn45;
        } *lunitbk;



/* motorbk.h */

struct {
   float
        torki[MXARMS][MXLNKS],/* torque constant */
        armra[MXARMS][MXLNKS],/* armature resistance */
        armla[MXARMS][MXLNKS],/* armature inductance */
        ampga[MXARMS][MXLNKS],/* amplifier gain */
        gratn[MXARMS][MXLNKS],/* gear ratio */
        emfkb[MXARMS][MXLNKS],/* back emf constant */
        motin[MXARMS][MXLNKS],/* true moment of inertia of armature
                                      and motor gear, for each joint */
        ampve[MXARMS][MXLNKS],/* amplifier output voltage for each joint */
        armcu[MXARMS][MXLNKS];/* armature current for each joint of each arm
                                      */
        } *motorbk;



/* movbasbk.h */

struct {
   int
        imovbas,                    /* logical Y if system has moving bases */
        nbas,                            /* number of moving bases in system */
        ibassim,
        ibastrq,
        ibsav,
        kbasnum[MXARMS];/* number of base on which arm is mounted
                                      0 ==> no mount */
   float
        btlast,
        btnext;
        } *movbasbk;
```

```
/* outrefbk.h */

struct {
   float
         reftht[MXARMS][MXLNKS],
         rjterr[MXARMS][MXLNKS],
         epos[MXARMS][3],
         epserr[MXARMS][6],
         eor[MXARMS][3][3];
         } *outrefbk;



/* pegbk.h */

/* No documentation on these variables was found, thus
  these are the best guesses for their use. */

#define MXHOLES 4           /* compile time variable, should be in mxprms.txt */

struct {
   int
         ipih[MXARMS],
         ihole[MXARMS],
         nholes;                              /* number of holes */

   float
         holfric[MXHOLES],            /* hole friction */
         holloc[MXHOLES][3],/* hole location */
         holdir[MXHOLES][3],/* unit vector along hole axis
                                                    pointing out of hole */
         holdpth[MXHOLES], /* depth of hole */
         holrad[MXHOLES],             /* hole radius */
         pegloc[MXARMS][3],/* peg location */
         pegdir[MXARMS][3],/* unit vector along peg axis
                                                    pointing into hole */
         dpthins[MXARMS];
         } *pegbk;



/* segmbk.h */

#define PICK_UP_OBJECT 1
#definePLACE_OBJECT  2
#defineMOVE_ARM  3
#defineHOLD_CURRENT_POSITION  4
#defineCHANGE_REFERENCE_POINT  5
#defineOPERATOR_CONTROL  6
#defineRESPONSE_CONTROL  7
#defineEE_POSITION_CONTROL  8
#defineTELEOP_JOINT_CONTROL  9
#defineTELEOP_BASE_CONTROL   10
#defineTELEOP_EE_CONTROL   11
```

```
#defineI__RATE   1                     /* Rate control */
#defineI__POSITION   2       /* Position control */
#defineI__SENSOR 3                     /* Sensor control */
#defineI__MOTION 1                     /* Motion step */
#defineI__NOMOTION    2   /* No motion step */
#defineI__JOINT 1                      /* Joint control */
#defineI__BASE_EE 2                    /* Base end effector motion */
#defineI__EE            3                        /* End effector control */
#defineI__INDEPENDENT 1/* Single arm control */
#defineI__DUAL       2                 /* Dual arm control */
#defineI__NARM       3                 /* Multi arm control in use */

struct {
   int
        imot;                                          /* Flag indicating motion-type program step
                                                          */
                                                       /*   = i__motion,motion */
                                                       /*   = i__nomotion,no-motion step */
   float
        pcoef[MXARMS][MXLNKS][MAXORD+1]/* Coefficients of the polynomial  */
                                                             /* describing the motion time
                                                                histories */
        ,timseg[MXARMS];             /* Segment start time array */
   int
        jnttul[MXARMS]               /* Segment joint or end effector control flag
                                                          */
                                                       /*   = i__joint, joint control */
                                                       /*   = i__base_ee,base end effector
                                                          motion */
                                                       /*   = i__ee      end effector control */
        ,irate[MXARMS];              /* Segment rate or position control flag */
                                                       /*   = I__RATE, Rate control */
                                                       /*   = I__POSITION, Position control */
                                                       /*   = I__SENSOR, Sensor control */
   float
        toolrp[MXARMS][3]
        ,posit[MXARMS][MXLNKS]/* No definition found, but in sgmnt.for */
        ,posor[MXARMS][3][3]/* these two variables were stored in in */
                                                       /*        temporary reals */
        ,segth[MXARMS][MXLNKS]
        ,segpos[MXARMS][3]
        ,segor[MXARMS][3][3]
        ,delth[MXARMS][MXLNKS]/* ???a local variable is defined in  */
                                                       /*  pidcon.for, but is it the same */
        ,delpos[MXARMS][3]
        ,dphi[3]                           /* ?Vector for changing orientation from  */
                                                       /* ?  OR to ROR */
                                                       /*  -- definitions from borerr.for */
                                                       /*  -- also internal variable in jtpos.for */
                                                       /*  -- also internal variable in orerr.for */
        ,ph;                                           /* ?Angle of rotation for desired  */
                                                       /* ?  orientation change */
                                                       /*  -- definitions from borerr.for */
```

```
int
        idcc                                            /* Dual arm coordinated control flag */
                                                        /*   = I__INDEPENDENT, not in use */
                                                        /*   = I__DUAL, coordinated control in use
                                                             */
                                                        /*   = I__NARM, multi arm control in use */
        ,itask[MXARMS]                 /* Task is currently in effect */
                                                        /*   = Pick_up_object = 1 */
                                                        /*   = Place_object = 2 */
                                                        /*   = Move_arm = 3 */
                                                        /*   = Hold_current_position = 4 */
                                                        /*   = Change_reference_point = 5 */
                                                        /*   = Operator_control = 6 */
                                                        /*   = Response_control = 7 */
                                                        /*   = EE_position_control = 8 */
                                                        /*   = Teleop_Joint_Control = 9 */
                                                        /*   = Teleop_Base_control  = 10 */
                                                        /*   = Teleop_EE_control  = 11 */
        ,nt[MXARMS];
   float
        senvel[MXARMS]                 /* the max sensor to target velocity */
        ,sgstrt[MXARMS]                /* Arm segment start time */
        ,sgstop[MXARMS]                /* Arm segment stop time */
        ,bsgstrt[MXARMS]               /* Base segment start time */
        ,bsgstop[MXARMS];              /* Base stop time */
        } *segmbk;



/* sensbk.h */

#define NUMBER_OF_BITS 12
#define MAX_ANALOG 3.14159
#define min_analog -3.14159

struct {
   float
        sth[MXARMS][MXLNKS],/* th PID control signal */
        sthd[MXARMS][MXLNKS],/* thd PID control signal */
        sthdd[MXARMS][MXLNKS],/* thdd PID control signal */
        oldsth[MXARMS][MXLNKS][2],/* last th control signal */
        sensed_forces[MXARMS][6];/* compliant feedback */
        } *sensbk;



/* spjtbk.h */

struct {
   int
        nspjt[MXARMS],                 /* number of special joints (max. = 3) */
        isptyp[MXARMS][3],/* special joint type */
        njtsp[MXARMS][3];              /* special joint location */
        } *spjtbk;
```

```
/* sprgbk.h */

struct {
   float
        sprp[MXARMS][3],           /* no force position of spring ref point (world
                                            coord) */
        spror[MXARMS][3][3],/* spring reference (no force) orientation ( =
                                            end-eff. ori.) */
        sprk[MXARMS][6][6],/* spring constant matrix */
        splc[MXARMS][3],           /* spring reference point (end-eff. coord.
                                            system) */
        sprf[MXARMS][6];           /* spring forces and torques */
   int
        ispr[MXARMS];              /* flag denoting spring type:
                                           = 0, no spring
                                           = 1, 3x3 spring (no rot.)
                                           = 2, 6x6 spring */
        } *sprgbk;



/* systbk.h */

struct {
   float
        th[MXARMS][MXLNKS],/* angular displacement of link i wrt link i-1
                                            */
        thd[MXARMS][MXLNKS],/* angular velocity of link i wrt link i-1 */
        thdd[MXARMS][MXLNKS],/* angular acceleration of link i wrt link i-1 */
        pos[MXARMS][MXLNKS+3][3],/* translational position vector for each link */
        vel[MXARMS][MXLNKS+3][3],/* linear velocity vector for each link */
        acc[MXARMS][MXLNKS+3][3],/* linear acceleration vector for each link */
        rot[MXARMS][MXLNKS+2][3][3],/* rotation matrices from joint coordinate
                                           systems to inertial system for each joint of
                                            each arm */
        om[MXARMS][MXLNKS+2][3],/* absolute angular velocity vector for each
                                            link */
        omd[MXARMS][MXLNKS+2][3],/* absolute angular acceleration vector for
                                            each link */
        hij[MXARMS][MXLNKS+1][3],/* vector from joint i-1 to joint i in world
                                            system */
        hcg[MXARMS][MXLNKS+2][3],/* vector from joint i to cg(i) in world
                                            coordinates */
        ainw[MXARMS][MXLNKS+2][3][3],/* inertia matrices for each link in world
                                            coordinates */
        r[MXARMS][MXLNKS+1][MXLNKS+1][3],/* vector from joint i to cg(j) in world
                                                    coordinates */
        fdcc[6];                          /* constraint forces due to use of dual arm
                                            coordination */
        } *systbk;
```

```
/* targbk.h */

struct {
   int
         ntarg,                                    /* number of target objects */
         itrgseq[MXTRGS],              /* rotation axes for orienting target */
         nptrmas[MXTRGS],              /* number of point masses in target */
         atrgheld[MXTRGS];             /* arm# holding load: 0 ==> not held */
   float
         atrglen[MXTRGS],              /* length of each target object */
         atrgrad[MXTRGS],              /* radius of each target object */
         atrgmas[MXTRGS],              /* mass of each target object */
         atrgcg[MXTRGS][3],            /* cg vector of each target (local coordinates)
                                                 */
         atrgin[MXTRGS][3][3],/* inertia matrix for each target (local
                                                 coordinates) */
         atrgrlgp[MXLODS][3],/* relative grip point (not currently used) */
         atrgapvec[MXLODS][3],/* approach vector (not currently used) */
         atrgpms[MXTRGS][MXPTS],/* mass of each point mass */
         atrgpml[MXTRGS][MXPTS][3],/* location of each point mass (local
                                                 coordinates) */
         angtrg[MXTRGS][3],            /* angles for orienting target wrt world */
         atrgloc[MXTRGS][3],/* location of target wrt world */
         atrgor[MXTRGS][3][3],/* target to world transformation matrix */
         attrgms[MXTRGS],              /* total mass of target object including point
                                                 masses */
         attrgcg[MXTRGS][3],/* cg vector of target plus point masses (local
                                                 coordinates) */
         attrgin[MXTRGS][3][3];/* inertia matrix for target plus point masses
                                                 (local coordinates) */
   char
         targnam[MXTRGS][8];/* target name */
         } *targbk;



/* tgrafbk.h */

struct {
   float
         atrgspn;
   short int
         itrgobj[10*MXGRAFPT],
         icntrg,
         numtrg[MXTRGS],
         nstrg[MXTRGS][MXTRGCMPS];
         } *tgrafbk;
```

```
/* timebk.h */

struct {
   float
         time,                          /* current simulation times */
         strttm,                        /* simulation start time */
         stoptm,                   /* simulation stop time */
         stppro,                   /* simulation processing time step */
         stpsof,                   /* simulation output time step for E&S playback */
         stpprt,                        /* printed output time step */
         stpavt,                   /* acc-vel-theta file step size */
         stplot,                        /* plot data output step size */
         stptrq,                        /* torque data output step size */
         stpgraf;                  /* graphice output step size */
         } *timebk;



/* toolbk.h */

struct {
   int
         grasping[MXARMS][MXLODS];/* # of load a tool is holding
                                              = 0, no load */
   float
         tulmas[MXARMS],           /* mass of each tool */
         tulcg[MXARMS][3],         /* center of gravity */
         tulin[MXARMS][3][3],/* tool inertia matrix */
         ttoldo[MXARMS][MXLODS][3],/* tool origin to load origin */
         ttulld[MXARMS][MXLODS][3][3],/* tool coordinate system to load
                                              coordinate system transformation */
         ttobso[MXARMS][3],/* find out from Grumman */
         ttulbs[MXARMS][3][3];/* find out from Grumman */
         } *toolbk;



/* unitbk.h */

struct {
   float
         flunit,                             /* type of internal units to be used */
         conunit[7];                    /* conversion factors between metric and
                                              English */
   char
         lisunit[7][12];                /* Character string for units type */
         } *unitbk;
```

```
/* user_data.h */

#define NUM_INTS 5
#define NUM_FLOATS 5
#define NUM_CHAR_STR 5

struct {
  struct {
    int
        ints[NUM_INTS];
    float
        floats[NUM_FLOATS];
    char
        chars[NUM_CHAR_STR][LINE_LENGTH];/* correction:
                                                    NUM_FLOATS -->
                                                    NUM_CHAR_STR
                                                    --Matt Gertz-- */
        } input,output;
    } *user_data;



/* user_function_support.h */

/*      names to reference drivers*/
#define CONTROL_REQ1
#defineCONTROL_SIM      2
#defineOUTPUT_REQ       3
#defineOUTPUT_SIM       4
#defineMOTION_SPECIFICATION5
#defineREQUIRE_PLOT_OUTPUT6
#defineGRAPHICS_OUTPUT7
#defineRATE_ALGORITHM8
#definePOSITION_ALGORITHM9
#defineTRACKING_ALGORITHM10
#defineDYNAMICS_ALGORITHM11
#defineOUTPUT_AVT       12      /* acceleration velocity theta output algorithm
                                                */
#defineOUTPUT_SOF       13      /* simulation output algorithm */
#defineRESPON_PLOT_OUTPUT14
#defineTRAJECTORY_ALGORITHM15
#defineERROR_SIGNAL_ALGORITHM16/* no support */
#defineMOTOR_CONTROL_ALGORITHM 17/* current algorithm being used during
                                                    any specific time interval */
#defineMOTOR_SENSING_ALGORITHM18/* current algorithm being used during
                                                    any specific time interval */
                                                  /* not supported */

#defineUSER_MOTOR_CONTROL_ALGORITHM 19
#defineUSER_MOTOR_SENSING_ALGORITHM 20
#definePID_CONTROL_ALGORITHM 21
#defineLEAD_LAG_CONTROL_ALGORITHM 22
#defineINPUT_TORQUE_CONTROL_ALGORITHM 23/* NO SUPPORT */
```

```
#defineHARDWARE_VOLT_CONTROL_ALGORITHM 24/* NO SUPPORT */
#definePID_SENSING_ALGORITHM 25
#defineLEAD_LAG_SENSING_ALGORITHM 26
#defineINPUT_TORQUE_SENSING_ALGORITHM 27/* NO SUPPORT */
#defineHARDWARE_VOLT_SENSING_ALGORITHM 28/* NO SUPPORT */
#defineCARTESIAN_SENSING_ALGORITHM 29
#defineCARTESIAN_CONTROL_ALGORITHM 30

                              /*    The above algorithms are initialized to one of the */
                              /*    following control algorithms below in: */
                              /*        read motion step(when using thp files) */
                              /*        modify_resopt(when selecting end effector
                                          control mode) */

#defineNULL_ALGORITHM          31
#defineUSER_CART_SENSING_ALGORITHM 32
#defineUSER_CART_CONTROL_ALGORITHM 33
#defineCOMPLIANCE_CONTROL_ALGORITHM  34/* NO SUPPORT */
#defineCOMPLIANCE_SENSING_ALGORITHM 35/* NO SUPPORT */
#defineFORCE_TORQUE_CONTROL_ALGORITHM 36
#defineFORCE_TORQUE_SENSING_ALGORITHM 37

/*      driver modes  */
#defineINITIALIZE_USER_FUNCTION1
#defineRUN_USER_FUNCTION2
#defineSHOW_USER_FUNCTION3
#defineSAVE_USER_FUNCTION4
#defineCLEAN_UP_USER_FUNCTION5
#defineFIRST_PASS_USER_FUNCTION6

struct {
   int
        driver_entry_point[MAX_MODES][MAX_DRIVERS];/* entry point returned by */
                                                    /*  lib$find_image_symbol */
   char
        driver_loaded[MAX_MODES][MAX_DRIVERS];/* signals driver has been */
                                                    /*  loaded */
   char
        driver_name[MAX_MODES][MAX_DRIVERS][FILE_LENGTH],
        exe_name[MAX_DRIVERS][FILE_LENGTH];
        } *dri_blk;
```

References:

[1] Walker, I.D. "The Use of Kinematic Redundancy in Reducing Impact and Contact Effects in Manipulation", 1990 IEEE Conference on Robotics and Automation, Cincinnati, OH, pp. 434-439, May 1990.

[2] Zheng, Y. and Hemami, H. "Mathematical Modeling of a Robot Collision with its Environment," Journal of Robotic Systems, pp. 289-307, ©1985.

[3] Cai, C. *Instantaneous Robot Motion with Contact between Surface*, Ch. 7, "Impact Impulse and Link-Mass Distribution," pp. 141-157, Ph.D. Thesis, Dept. of Computer Science, Stanford University, 1988.

[4] Khatib, O. and Burdick, J. "Motion and Force Control of Robot Manipulators," 1986 IEEE Conference on Robotics and Automation, San Francisco, CA, pp. 1381-1386, 1986.

[5] Yoshikawa, T. "Analysis and Control of Redundant Manipulators with Redundancy," Robotics Research: The first int. symp., ed. Brady and Paul, MIT press, pp. 735-747, 1984.

[6] Nenchev, D. N. "Redundancy Resolution through Local Optimization: A Review," Journal of Robotic Systems 6(6), pp. 769-798 (1989), John-Wiley & Sons, Inc.,©1989.

[7] Kim, J., Khosla, P.K., and Chung, W. "Static Behaviors of Redundant Manipulators," Technical Report CMU-RI-AML-89-01, Advanced Manipulators Laboratory, The Robotics Institute, Carnegie Mellon University, Sept. 1989.

[8] Kim, J., Khosla, P.K., and Chung, W. "Modeling Static Behavior of Redundant Manipulators," *3rd ISRAM Conference*, 1990.

[9] Craig, J.J. *Introduction to Robotics: Mechanics and Control*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, ©1989.

[10] Cai, C. *Instantaneous Robot Motion with Contact between Surface*, Ch. 5, "A Control Formulation for Contact," pp. 104-123, Ph.D. Thesis, Dept. of Computer Science, Stanford University, 1988.

[11] Franklin, G.F., Powell, J. D., and Emami-Naeini, A. *Feedback Control of Dynamic Systems*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, ©1986.

[12] Kelmar, L. and Khosla, P.K. "Automatic Generation of Kinematics for a Reconfigurable Modular Manipulator System," Proceedings of the 1988 IEEE International Conference on Robotics and Automation, Philadelphia April 1988.

[13] Baillieul, J. "Kinematic Progamming Alternatives for Redundant Manipulators," 1985 IEEE Conference on Robotics and Automation, pp. 722-728, 1985.

[14] Lowrie, J., Fermelia, A., Haley, D., Gremban, K.,Van Baalen, J., and Walsh, R. "Evaluation of Automated Decisionmaking Methodologies and Development of an Integrated Robotic System Simulation: Study Results," NASA Contractor Report 165975, Martin Marietta Aerospace, Contract NAS1-16759, September 1982.

[15] Khatib, O. "Dynamic Control of Manipulators in Operational Space," Sixth CISM-IFToMM Congress on Theory of Machines and Mechanisms, Dec. 1983, pp. 1128-1131, New Delhi, India (Wiley, New Delhi).