Prototyping a Robotic Manipulator and Controller

Tarek M. Sobh, Mohamed Dekhil, and Thomas C. Henderson[1]

UUCS-93-013

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

July 6, 1993

Abstract

Building a robot and its environment (control, software, hardware, simulation, etc) is a complex task that requires the efforts of an experienced engineering team. Once a robot model has been chosen and a design has been agreed upon, it becomes difficult to make design changes without affecting the manufactured parts, actuators and sensors. Therefore, developing an environment that enables flexible design and reconfigurable links, joints, actuators, and sensors would be an essential step for efficient prototyping. Such an environment should have the right "mix" of software and hardware components for designing the physical parts and controllers and for the algorithmic control and specifications of the kinematics, inverse kinematics, dynamics, trajectory planning, analog control and computer (digital) control of the manipulator. Specifying object-based communications and catalog mechanisms between the software (control, simulation, and monitoring) modules, PPL hardware controllers, physical parts' CAD designs, and actuator and sensor components is a necessary step in the prototyping activities. We discuss and present a framework and intermediate results in the process of prototyping an experimental reconfigurable 3-link robot in this report.

[1]This work was supported in part by DARPA grant N00014-91-J-4123, NSF grant CDA 9024721, and a University of Utah Research Committee grant. All opinions, findings, conclusions or recommendations expressed in this document are those of the author and do not necessarily reflect the views of the sponsoring agencies.

Contents

1 Introduction **3**

2 Objectives **3**

3 Phases of Building a Robot **4**

3.1 Design Phase 4
3.2 Simulation Phase 5
3.3 Prototyping and testing phase 5
3.4 Manufacturing Phase 5

4 Robot Modules and Parameters **6**

4.1 Forward Kinematics 6
4.2 Inverse Kinematics 6
4.3 Dynamics 7
4.4 Trajectory Generation 8
4.5 Linear Feedback Control 10

5 Hardware vs. Software modules **10**

5.1 Types of Inputs 11
5.2 Desired Frequency of the control system 12
5.3 Error Analysis 13

6 One Link Manipulator **13**

6.1 The experiment 13
6.2 Results and analysis 17

7 Three link manipulator **17**

7.1 Controller Design 17
7.2 Simulation 23
7.3 Benchmarking 24

8 Current Developments **28**

8.1 Sensor and Actuator Interface 29
8.2 Database and Spreadsheet Interface 29
8.3 Building the Robot 29

9 Bibliography **30**

10 Appendix A **32**

11 Appendix B **53**

1 Introduction

The concept of *Reconfigurable Modular Manipulator System* (RMMS) was proposed by Khosla, Kanade, Hoffman, Schmitz, and Delouis [8] at Carnegie Mellon University. Their goal is to create a complete manipulator system, including mechanical and control hardware, and control algorithms that are automatically and easily reconfigurable.

Several research groups are jointly developing the *Palo Alto Collaborative Testbed* (PACT) [3], which is a concurrent engineering infrastructure that encompasses multiple sites, subsystems, and disciplines. Their project involves the design of a large system that consists of smaller sub-systems developed in different sites. To organize and synchronize the work and efforts of the teams involved in developing these subsystems, it is important to have some kind of a knowledge representation scheme and a shared database that relates the common parts in each subsystem and maintains the integrity of the whole system.

Our goal is to develop a design and rapid prototyping environment that allows the specification of simulation, software control and hardware control (if necessary), communication, interfaces, CAD, and manufacturing modules for building a robotic manipulator. The end product of the first "pass" at this ambitious project is an experimental robotic kit that will be useful for teaching control and robotic classes at all levels, and will be useful for researchers experimenting with reconfiguring the physical design of robots and also the sensors, actuators, control modules, etc.

2 Objectives

The objective of this research project is to explore the basis for a consistent software and hardware environment, and a flexible system that enables easy and fast modifications, and online simulation for the chosen manipulator parameters.

Another goal we aim for is to create a design package that can automatically choose the optimal design parameters for a certain task and performance requirements (those parameters would include choice of links, joints, configuration, motor parameters, required sensors, and the necessary control algorithms).

The current aim for this project, is to build a prototype three link robot with two revolute, and one prismatic link, then the final controller will be implemented in software and (possibly) hardware for the prototype robot, which will be connected to a monitor program as well to study the behavior and performance of this design.

The importance of this project arises from several points:

- It will facilitate and speed the design process of robots.
- This project will improve the cooperation of several team groups in the department (VLSI group, Robotics group, Graphics, CAD, and Manufacturing group,) and the cooperation of the department with other departments and units (Mechanical and Electrical Engineering, CED).
- The prototype robot will be used as an educational tool in robotics and automatic control classes.
- This project will establish the bases and the framework for design automation in the department.

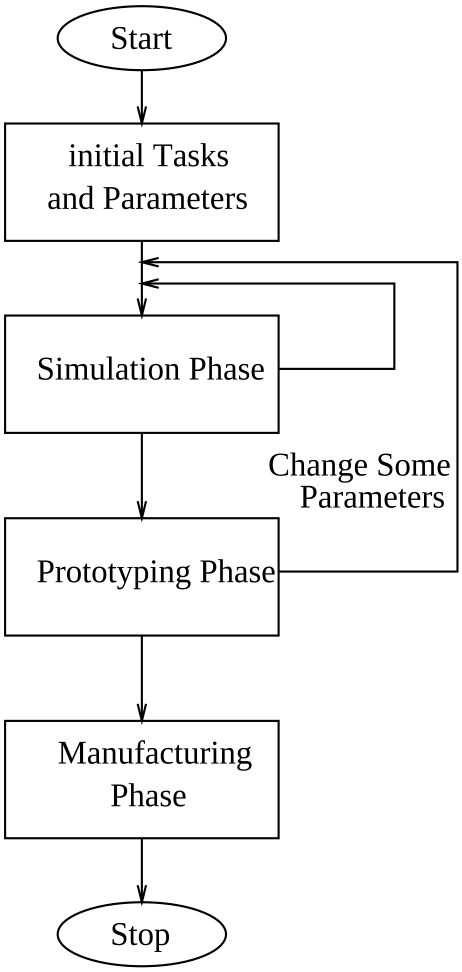


Figure 1: Robot design Phases

- The interdisciplinary nature of the proposed research will provide an exceptional educational environment for those involved in the work.

3 Phases of Building a Robot

We can divide the process of building a robot into several phases. In the following sections, we will describe each phase, and the tasks involved in each of them. (See Figure 1 for details.)

3.1 Design Phase

This phase can be named (for the time being) the *pencil and paper* phase. This phase includes the following tasks:

- Specify the required robot tasks. This includes: the workspace of the manipulator's end effector, the type of each link (revolute or prismatic), the shape of the links (cylindrical, rectangular, etc.)

- Choose the robot parameters such as the length and mass of each link, the friction at each joint, the position and motor parameters for each joint. This choice depends on the task specifications from the first phase.
- Setting the control equation and the trajectory planning strategy. The control equation deals with the error in position and velocity measured at each joint and tries to minimize that error using a feedback control. Trajectory planning is the position and velocity equations to move from one point to another.
- Study the singular points and set some rules and decisions to deal with them. This will affect the trajectory planning algorithm so that it avoids such points.

As mentioned before, the *designer package* we eventually intend to produce will be a valuable complement to the *pencil and paper* phase.

3.2 Simulation Phase

After choosing the required manipulator and motor parameters, and setting the control equation and trajectory planning method, the design can be tested using a simulation program to test the behavior and the performance of the chosen manipulator. This simulation program should allow us to change any of the parameters and see the variations in behavior with respect to each parameter. One of the most important factors in this simulation program is the update rate, that's the frequency in which the calculations are done and a new value for the torque (or the voltage) that drives the motor is generated. We will talk about that in more detail in a later section. The simulation module is to be also used while the physical design is being performed in order to verify the ranges for torques and sensor parameters and thus help in choosing the actuator, sensor, and physical design parameters that would be suitable to the expected kind of application for the robot.

3.3 Prototyping and testing phase

Once the design is agreed upon, a prototype model for the manipulator is built using the final parameters, then a monitor program is used to test the behavior of the robot, and compare it with the simulated results. In this phase, a testing procedure should be performed to test the behavior and the performance under different kinds of movements, and study the error patterns generated to give an overall evaluation of the design. This phase will be the last chance to change any of the parameters before building the real manipulator. In this phase, a software interface should be implemented to enable controlling and monitoring the prototype robot.

3.4 Manufacturing Phase

This is the final phase of building a robot. After testing the prototype version, and getting satisfactory performance, the actual robot can be manufactured; then, using the monitor program again, the robot can be tested and its performance can be measured. Once the actual robot is built, it is difficult (and costly) to change some of the parameters, thus the testing procedure in the prototyping phase is very important. On the other hand, the robot we would like to design should be reconfigurable, in the sense of being able to exchange links, actuators, and sensors.

4 Robot Modules and Parameters

Controlling and simulating a robot is a process that involves a large number of mathematical equations, formulae, and calculations. To be able to deal with such a number of equations, it is better to divide them into modules, where each module accomplishes a certain task. The most important modules, as described in [2], are: kinematics, inverse kinematics, dynamics, trajectory generation, and linear feedback control. In the following sections we will describe briefly each of these modules, and the parameters involved in each.

4.1 Forward Kinematics

Forward kinematics is the study of the motion without regard to the forces that cause it. Sometimes it is called *direct kinematics*. In our case, we are only concerned with the static position and orientation of the manipulator linkages. There are two different ways to express the position of any link: using the *Cartesian space*, which consists of position (x, y, z), and orientation, which can be represented by a 3×3 matrix called the rotation matrix; or using the *joint space*, representing the position by the angles of the manipulator's links. In our case, forward kinematics is the transformation from joint space to Cartesian space.

This transformation depends on the configuration of the robot, (i.e., link lengths, joint positions, type of each link, etc.) In order to describe the location of each link relative to its neighbors, a frame is attached to each link, we then specify a set of parameters that characterizes this frame. This representation is called *Denavit-Hartenberg notation.* (See [2] for more details.) Forward kinematics is used for simulation and computing the actual position in Cartesian space from sensors outputs at the joints.

4.2 Inverse Kinematics

This module solves for the joint angles given the desired position and orientation in the Cartesian space. This is a more complicated problem than the forward kinematics. The complexity of this problem arises from the nature of the transformation equations which are nonlinear. There are two issues in solving these equations: *existence of solutions* and *multiple solutions.* A solution can exist only if the given position and orientation lies within the workspace of the manipulator's end-effector. By workspace, we mean the volume of space which the end-effector of the manipulator can reach. On the other hand, the problem of multiple solutions forces the designer to set criteria for choosing one solution. A good choice would be the solution that minimizes the distance that each joint is required to move.

There are two methods for solving the inverse kinematics problem: *closed form solution* and *numerical solutions.* Numerical solutions are much slower than closed form solutions, but, sometimes it is too difficult to find the closed form solution for some configurations. In our case, we will use closed form solutions, since our models are three link manipulators with an easy closed form formula. Chang [1], has proposed a method to find the closed-form solution for the inverse kinematics of robot manipulators with redundancy. Tourassis and Ang, Jr. [16], proposed a modular architecture for inverse robot kinematics using numerical solutions.

4.3 Dynamics

Dynamics is the study of the forces required to cause the motion. In other words, it is concerned with the way in which motion of the manipulator arises from torques applied by the actuators, or from external forces applied to the manipulator. The serial chain nature of manipulators makes it easy to use simple methods in dynamic analysis that exploits of this nature.

For most manipulators, the parameters that can be directly controlled are the joint forces and/or torques. However, the variables to be controlled are the position and the orientation of the end-effector. Thus, the study of dynamics leads to finding the forces and/or torques that should be applied at each joint to produce the desired trajectory of the end-effector.

There are two problems related to the dynamics of a manipulator: *controlling* the manipulator, and *simulating* the motion of the manipulator. In the first problem, we have a set of required positions for each link, and we want to calculate the required torques to be applied at each joint. This is called *inverse dynamics*. In the second problem, we are given set of torques applied to each link, and it is required to calculate the new position and the velocities during the motion of each link. The latter is used to simulate a manipulator mathematical model before building the physical model, which gives the chance to update and modify the design without the cost of changing or replacing any physical part.

The dynamics equations for any manipulator depend on the following parameters:

- The mass of each link.
- The mass distribution for each link, which is called *the inertia tensor*, which can be thought of as a generalization of the scalar moment of inertia of an object.
- Length of each link.
- Link type (revolute or prismatic.)
- Manipulator configuration and joint locations.

The dynamics model we are using to control the manipulator is of the form:

$$\tau = M(\theta)\ddot{\theta} + V(\theta, \dot{\theta}) + G(\theta) + F(\theta, \dot{\theta})$$

Where $M(\theta)$ is the *mass matrix* of the manipulator, $V(\theta, \dot{\theta})$ is the vector of centrifugal and Coriolis terms, and τ is the required torque. G and F represents the gravity and friction terms respectively. To simulate the motion of a manipulator we must make use of the same model we have used to *control* that manipulator. The model for simulation will be of the form:

$$\ddot{\theta} = M^{-1}(\theta)[\tau - V(\theta, \dot{\theta}) - G(\theta) - F(\theta, \dot{\theta})]$$

Once we have the acceleration $\ddot{\theta}$, we can calculate the velocity $\dot{\theta}$ and the position θ using simple numerical integration techniques. Given initial conditions for the position and velocity, usually in the form:

$$\theta(0) = \theta_0,$$

$$\dot{\theta}(0) = 0,$$

we can integrate forward in time by steps of size Δt. Using *Euler integration*, starting with $t = 0$, we can compute θ and $\dot{\theta}$ iteratively as follows:

$$\dot{\theta}(t + \Delta t) = \dot{\theta}(t) + \ddot{\theta}(t)\Delta t,$$

$$\theta(t + \Delta t) = \theta(t) + \dot{\theta}(t)\Delta t + \frac{1}{2}\ddot{\theta}(t)\Delta t^2$$

Choosing a value for Δt depends on several factors such as:

- The required frequency for simulating the robot.
- The natural frequency of the manipulator. From sampling theory, the sampling frequency should be at least twice the natural frequency of the system.
- The maximum speed available to do the required calculations. This is the most important issue when choosing between software and hardware solutions. We will discuss this point in detail in a later section.

The dynamics module is the most time consuming part among the manipulator modules. This is because of the tremendous amount of calculation involved in the dynamics equations. This fact makes the dynamics module a good candidate for hardwiring, to enhance the performance of the control and/or the simulation system. This issue will be discussed in more detail in a later section.

There are some parallel algorithms to calculate the dynamics of a manipulator. One approach described in [14], is to use multiple microprocessor systems, each one is assigned to a manipulator link. Using a method called *branch-and-bound*, a schedule of the subtasks of calculating the input torque for each link is obtained. The problem with this method is that the scheduling algorithm itself was the bottleneck, thus it limits the total performance. Several other approaches were suggested [9, 11, 15] which are based on multiprocessor controllers. Hashimoto and Kimura [4] proposed a new algorithm called the *resolved Newton-Euler algorithm*, which is based on a new description of the Newton-Euler formulation for manipulator dynamics. Another approach was proposed by Li, Hemami, and Sankar [13] to drive linearized dynamic models about a nominal trajectory for the manipulator using a straightforward Lagrangian formulation. An efficient structure for real-time computation of the manipulators dynamics was proposed by Izaguirre, Hashimoto, Paul and Hayward [6]. The fundamental characteristic of that structure was the division of the computation into high-priority synchronous tasks and low-priority background tasks, possibly sharing the resources of a conventional computing unit based on commercial microprocessors.

4.4 Trajectory Generation

This module is concerned with computing a trajectory in multidimensional space which describes the motion of the manipulator. *Trajectory*, is the time history of position, velocity, and acceleration for each of the manipulator's links. This module includes the human interface problem of describing the desired behavior of the manipulator. The complexity of this problem arises from the wide meaning of a *manipulator's behavior.* In some applications we might only need to specify the goal position, while in some other applications, we might need to specify the velocity with which the end effector should move. Since trajectory generation occurs at run time on a digital computer, the trajectory points are calculated at a certain rate, called the *path update rate.* We will talk about this point when we talk about the speed considerations.

There are several strategies to calculate trajectory points which generate a smooth motion for the manipulator. It's important to guarantee this smoothness of the motion because of some physical considerations such as: the required torque that causes this motion, the friction at the joints, and the frequency of update required to minimize the sampling error.

One of the simplest methods is using *cubic polynomials*, which assumes a cubic function for the angle of each link, then by differentiating this equation, the velocity and acceleration are computed. Linear acceleration is achieved using this method (double differentiation of the distance function.) The problem is to calculate the coefficients of the cubic polynomial equation. A cubic equation has the form:

$$\theta(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3,$$

and so the joint velocity and acceleration along this path are:

$$\dot{\theta}(t) = a_1 + 2a_2 t + 3a_3 t^2,$$
$$\ddot{\theta}(t) = 2a_2 + 6a_3 t.$$

By stating the initial and final positions and velocities, we can calculate the required coefficients as follows:

$$\theta(0) = \theta_0,$$
$$\theta(t_f) = \theta_f,$$
$$\dot{\theta}(t_f) = 0,$$
$$\dot{\theta}(0) = 0.$$

where, t_f is the time to move from θ_0 to θ_f.

Substituting these values and solving for the a_i we obtain:

$$a_0 = \theta_0,$$
$$a_1 = 0,$$
$$a_2 = \frac{3}{t_f^2}(\theta_f - \theta_0),$$
$$a_3 = -\frac{2}{t_f^3}(\theta_f - \theta_0).$$

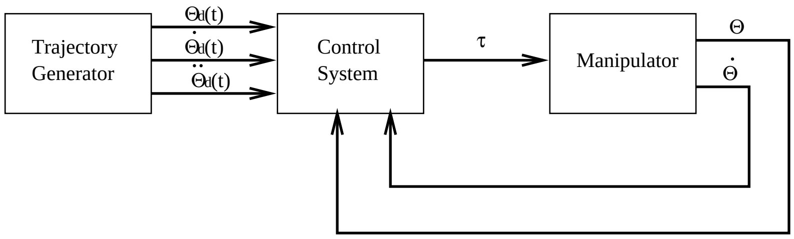


Figure 2: High-level block diagram of a robot control system.

4.5 Linear Feedback Control

We will use a linear control system in our design, which is an approximation of the nonlinear nature of the dynamics equations of the system, which are more properly represented by nonlinear differential equations. This is a reasonable approximation, and it is used in the current industrial practice.

We will assume that, there are sensors at each joint to measure the joint angle and velocity, and there is an actuator at each joint to apply a torque on the neighboring link. Our goal is to cause the manipulator joints to follow a desired trajectory. The readings from the sensors will constitute the feedback of the control system. By choosing appropriate gains we can control the behavior of the output function representing the actual trajectory generated. Minimizing the error between the desired and actual trajectories is our main concern. Figure 2 shows a high level block diagram of a robot control system.

When we talk about control systems, we should consider several issues related to that field, such as: *stability*, *controllability*, and *observability*. For any control system to be stable, its poles should be negative, since the output equation contains terms of the form $k_i e^{p_i}$, if p_i is positive, the system is said to be unstable. We can guarantee the stability of the system by choosing certain values for the feedback gains.

We will assume a second order control system of the form:

$$m\ddot{\theta} + b\dot{\theta} + k\theta.$$

Another desired behavior of the control system is to be *critically damped*, which means that, the output will reach the desired position at minimum time without overshooting. This can be accomplished by making $b^2 = 4mk$. Figure 3 shows the three types of damping: *underdamped*, *critically damped*, and *overdamped*. Figure 4 depicts the robot control loop including the modules described in this section.

5 Hardware vs. Software modules

When we design real time systems that involve a large amount of floating point calculations, speed becomes an important issue. In such situations, a certain question usually arises: do we need to hardwire the system, or at least portions of the system that contain costly calculations, or will software give us the desired speed?

To answer this question we have to consider several factors that affect the desired speed (frequency of calculations). The experiments to compute the maximum speed we can get using software solutions, and

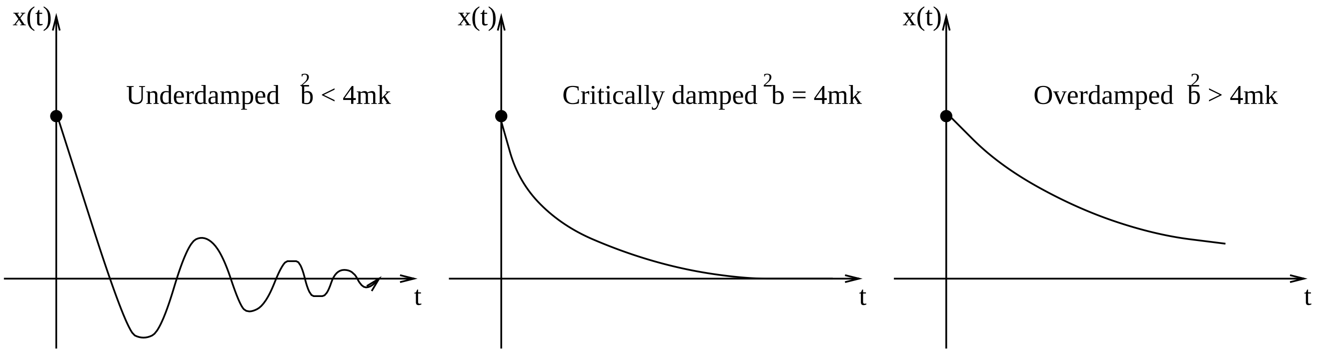


Figure 3: Different types of damping in a second order control system.

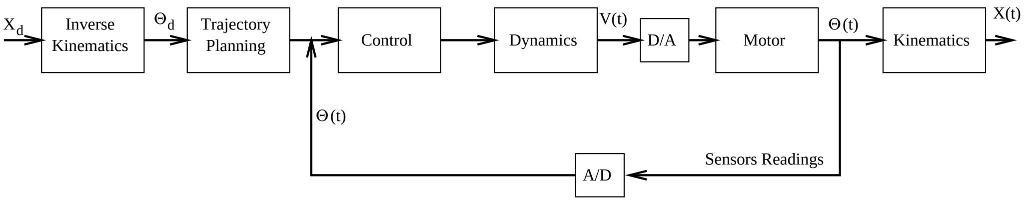


Figure 4: Overall Robot Control Loop.

deciding on the required hardware we need to build if we are to use hardware solution are the minimal analysis to be performed. The desired frequency of calculations depends on the type and frequency of inputs, the noise in the system, and the required output accuracy. In the following sections we will discuss some of these points in more detail.

5.1 Types of Inputs

The user interface to the system should allow the user to specify the desired motion of the manipulator with different ways depending on the nature of the job that the manipulator is designed to do. The following are some of the possible input types the user can use:

- Move from point x_0, y_0, z_0 to point x_d, y_d, z_d in Cartesian space.
- Move in a pre-defined position trajectory $[x_i, y_i, z_i]$. This is called *position planning.*
- Move in a pre-defined velocity trajectory $[\dot{x}_i, \dot{y}_i, \dot{z}_i]$. This is called *velocity planning.*
- Move in a pre-defined acceleration trajectory $[\ddot{x}_i, \ddot{y}_i, \ddot{z}_i]$. This is called *force control.*

This will affect the place in which the inverse kinematics module should be implemented; outside the update loop, as in the first two cases, or inside the update loop, as in the last three cases. For the last three cases we have two possible solutions: we can include the inverse kinematics module in the main update loop, as we mentioned before, or we can plan ahead in the joint space before we start the update loop. We should calculate the time required for each case plus the time required to make the decision.

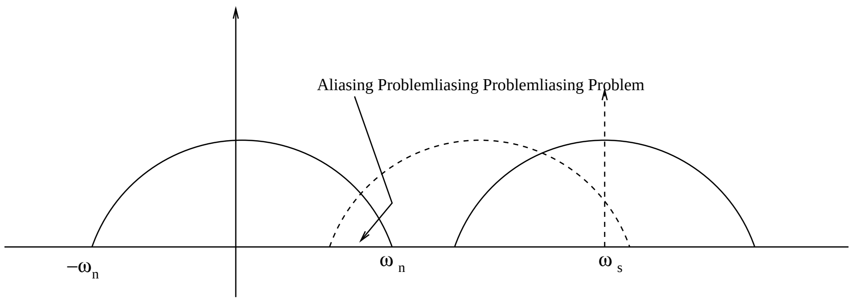


Figure 5: Sampling Theory and the Aliasing Problem.

5.2 Desired Frequency of the control system

We have to decide on the required frequency of the system. In this system we have four frequencies to be considered:

- Input frequency, which represents the frequency of changes to the manipulator status (position, velocity, and acceleration).
- Update frequency, representing the speed of calculation involved.
- Sensing frequency, which depends on the A/D converters that feed the control system with the actual positions and velocities of the manipulator links.
- Noise frequency: since we are dealing with a real-time control system, we have to consider different types of noise affecting the system such as: input noise, system noise, and output noise (from the sensors).

From *digital control theory*, to be able to restore the original function after sampling, the sampling frequency should be at least twice as much as the highest frequency in the function. So, if we assume that the system can be approximated by the second order differential equation:

$$s^2 + 2\omega_n + \omega_n^2,$$

where ω_n is the natural frequency of the system, then, the sampling frequency ω_s should be:

$$\omega_s \geq \omega_n.$$

Figure 5 shows the function in the frequency domain, and the problem when ω_s is less than $2\omega_n$, *the aliasing problem.*

The problem is to calculate the natural frequency of our control system to be able to decide on the value of the minimum update frequency required for that system. This problem can be solved analytically by using the finite element theory for mechanical structures, but this is beyond the scope of our discussion here. Another approach is to select values for the update frequency, and measure the error generated using each value, and choose the value that causes an acceptable error. This approach is not very accurate, but for large systems, using the analytical method is very complicated and sometimes is impossible to solve.

5.3 Error Analysis

The error is the difference between the desired and actual behavior of the manipulator. In any physical real-time control system, there is always a certain amount of error resulting from modeling error or different types of noise. One of the designer decisions is; what is the maximum allowable error. This decision depends on the nature of the tasks this manipulator is designed to accomplish. For example, in the medical field the amount of error allowed is much less than in a simple experimental manipulator. The update frequency is the most dominant factor in minimizing the error. Figure 6 shows the error in position of a three link manipulator using two different update frequencies.

It is clear that increasing the update frequency results in decreasing the error, however, the update frequency is limited by the speed of the machine used to run the system. Even if we use a hardware piece to speedup the calculations, we still have a limit on the update frequency, but, as we mentioned in the previous section, it is sufficient to have the update rate more than double the natural frequency of the system.

6 One Link Manipulator

Controlling a one-link robot in a real-time manner is not too difficult; on the other hand, it is not a trivial task. This is the basis of controlling multi-link manipulators, and it gives an indication of the type of problems and difficulties that might arise in a larger environment. The idea here is to establish a complete model for controlling and simulating a one-link robot, starting from the analysis and design phases and ending with simulation and error analysis.

6.1 The experiment

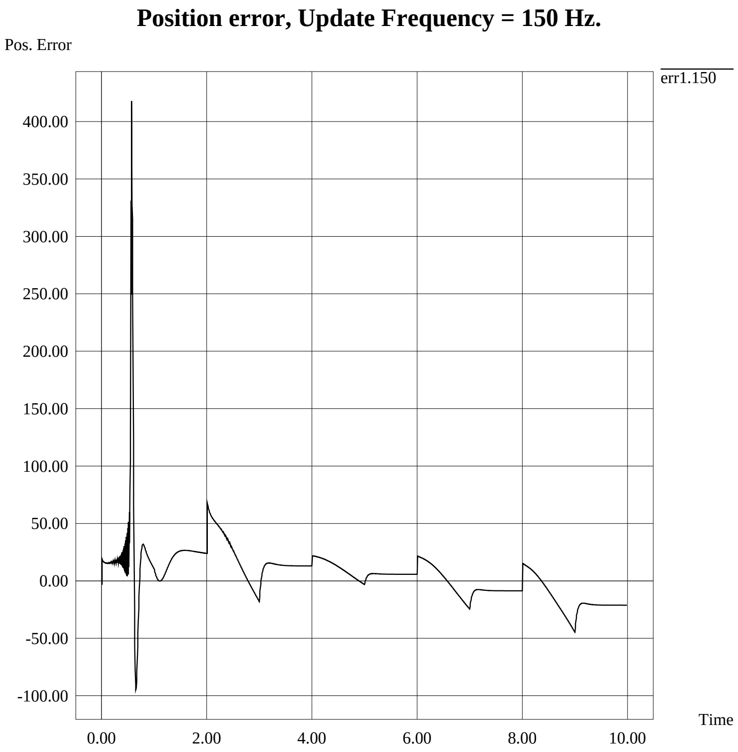
The first experiment in this project was to design a control system for a one link revolute manipulator which is physically represented by a motor with a load. The kinematics and inverse kinematics for this model is trivial. The dynamics is a single equation of the form:

$$\tau = I_{zz}\ddot{\theta} + f\dot{\theta}$$

where, Izz is the inertia tensor in the z direction, and f is the friction.

We used a motor from the Mechanical Engineering control lab which is controlled by a PID controller. We also used an analog I/O card, named PC-30D, connected to a Hewlett Packard PC. This card has sixteen 12-bit A/D input channels and two 12-bit D/A output channels. There are card interface drivers with a Quick BASIC program that uses the card drivers to control the DC motor. Figure 7 shows the setup for our experiment.

The maximum scanning rate of this I/O board and the PC combination are greatly influenced by the amount of code being processed. This means that the amount of calculation should be minimized to obtain the best performance. The problem here is that the maximum frequency that can be obtained is 1000 Hz, and when we implemented a complete on-line controller for a one-link robot using this configuration, the



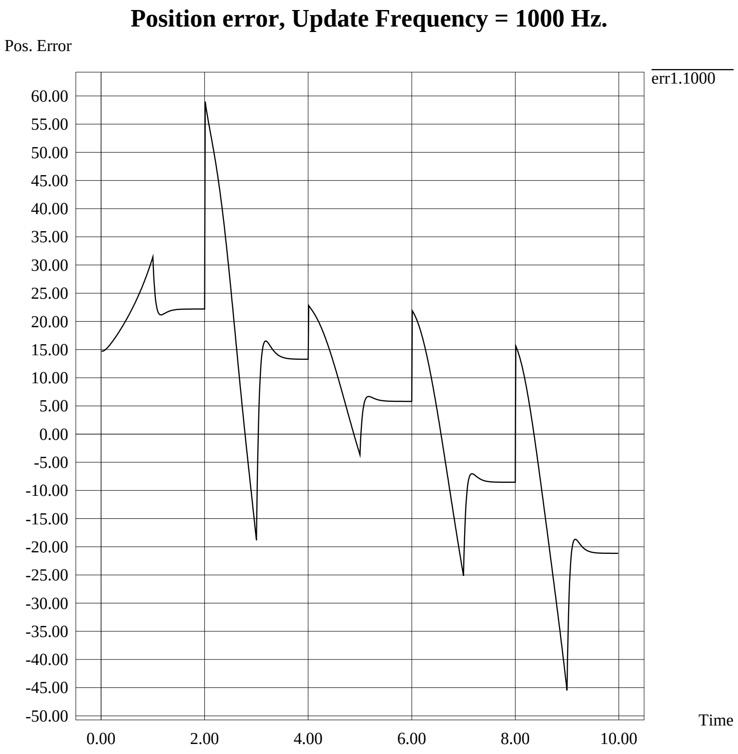


Figure 6: Errors using 2 update frequencies.

Figure 7: Snap shot of the Mechanical Engineering Control lab. and the motor setup.

performance was much degraded such that we were not able to control the error any more, and the system became unstable.

To overcome this problem, we implemented another program on a SUN workstation to simulate the one-link robot and the motor as well, and we generated a file that contains the required voltages to drive the motor. In the simulation program we used a frequency of 1000 Hz, which is the maximum scanning speed for the A/D card on the PC, and a simulation interval of 10 seconds. We tried three different sequences of desired positions, we then transferred the output voltage file to the PC and applied these voltages to the motor using the I/O card. In a sense, the setup can be considered as an *off-line* control system.

One of the problems we faced in this process was to establish the transfer function between the torque and the voltage. We used the motor parameters to form this function by making some simplifications, since some of the motor parameters have nonlinear components which make it too difficult to make an exact model. Figure 9 shows the circuit diagram of the motor and its parameters. The transfer function is of the form:

$$v(t) = \frac{T_m}{K_T} R + \frac{\dot{T}_m}{K_T} L + K_E \dot{\theta},$$

where,

$V(t)$ is the voltage at time t.
K_T is the torque from the motor.
$L = 13.4 \times 10^{-3} H$.
$R = 4.96\Omega$.
$K_T = 20.8 oz.in.sec^2$.
$K_E = 0.147 v/rad/sec$.

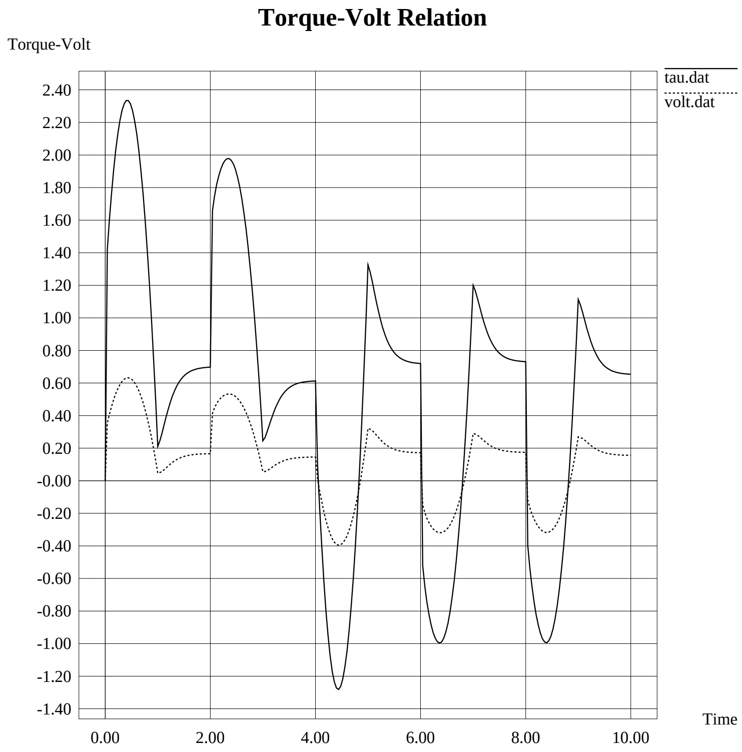


Figure 8: The Relation Between the Torque and the Voltage.

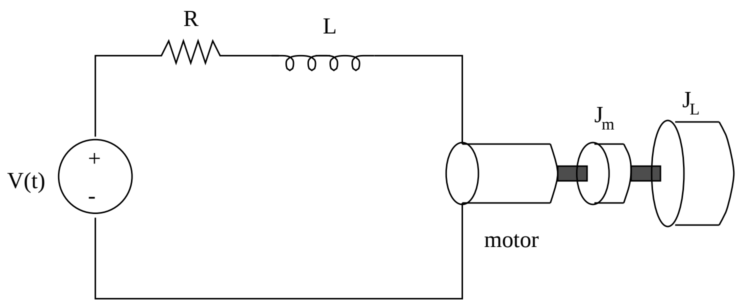


Figure 9: Circuit Diagram of the DC-motor Used in the Experiment

6.2 Results and analysis

As mentioned before, we have used three input sequences for the desired positions, the move from one position to another was done in one second then the new position is kept for another second. After applying the voltage files to the motor using the I/O card, the actual positions and velocities are measured using a potentiometer for the position, and a tachometer for the angular velocity. These measured values are saved in other files, then a graphical simulation program is used to display the movement of the link, the desired and actual positions, and the desired and actual velocity. The error in both, position and velocity, is displayed. Figures 10, 11, and 12 show the output window displaying the link and graphs for the position and the velocity.

From these figures we notice that the error in position is much lower than the error in velocity; that's due to the simplification we made for the motor model in the simulation program, and the low accuracy of the tachometer we used to get these measurements. There is also a small error between the actual and simulated positions and velocity, This is also due to the approximations made in the motor model. Figure 13 shows the difference between the actual and desired position and velocity in each of the three sequences. The presence of steady state error in these graphs is because we are using off-line control, so the actual behavior of the motor might be slightly different from the simulations, but the system does not recognize that.

In general, This experiment gave us an indication of the feasibility of our project, and good practical insight. It also helped us determine some of the technical problems that we might face in building and controlling the three-link robot. For example, we decided to use torque controllers for the joint actuators instead of position controllers. This decision came as a result of the problem we have encountered in finding an accurate transfer function from torque to voltage. Using a torque controller will save time and accuracy, since we will be able to supply the motors with the torque which is the output of the dynamic equations.

7 Three link manipulator

Our goal in this project is to prototype and create a design environment for a three-link robot manipulator with its controller and simulator, in a modular manner that enables fast and easy modification. There are several teams involved in this work: one team for the analysis and design of the robot control and simulation modules, another team designing and building the manipulator, ordering the required parts, and assembling the mechanical and electrical components, and a third team for the A/D and D/A interface between the manipulator and the workstation. The coordination between these teams is handled by a team that organizes the tasks, sets deadlines, and arranges regular meetings with each team. In the following sections, we will describe what we have done so far, and the steps we have gone through to accomplish this task.

7.1 Controller Design

The first step in designing a controller for a robot manipulator is to solve for its kinematics, inverse kinematics, dynamics, and the feedback control equation that will be used. Also the type of input and the user interface is determined at this stage. We should also know the parameters of that robot such as: link

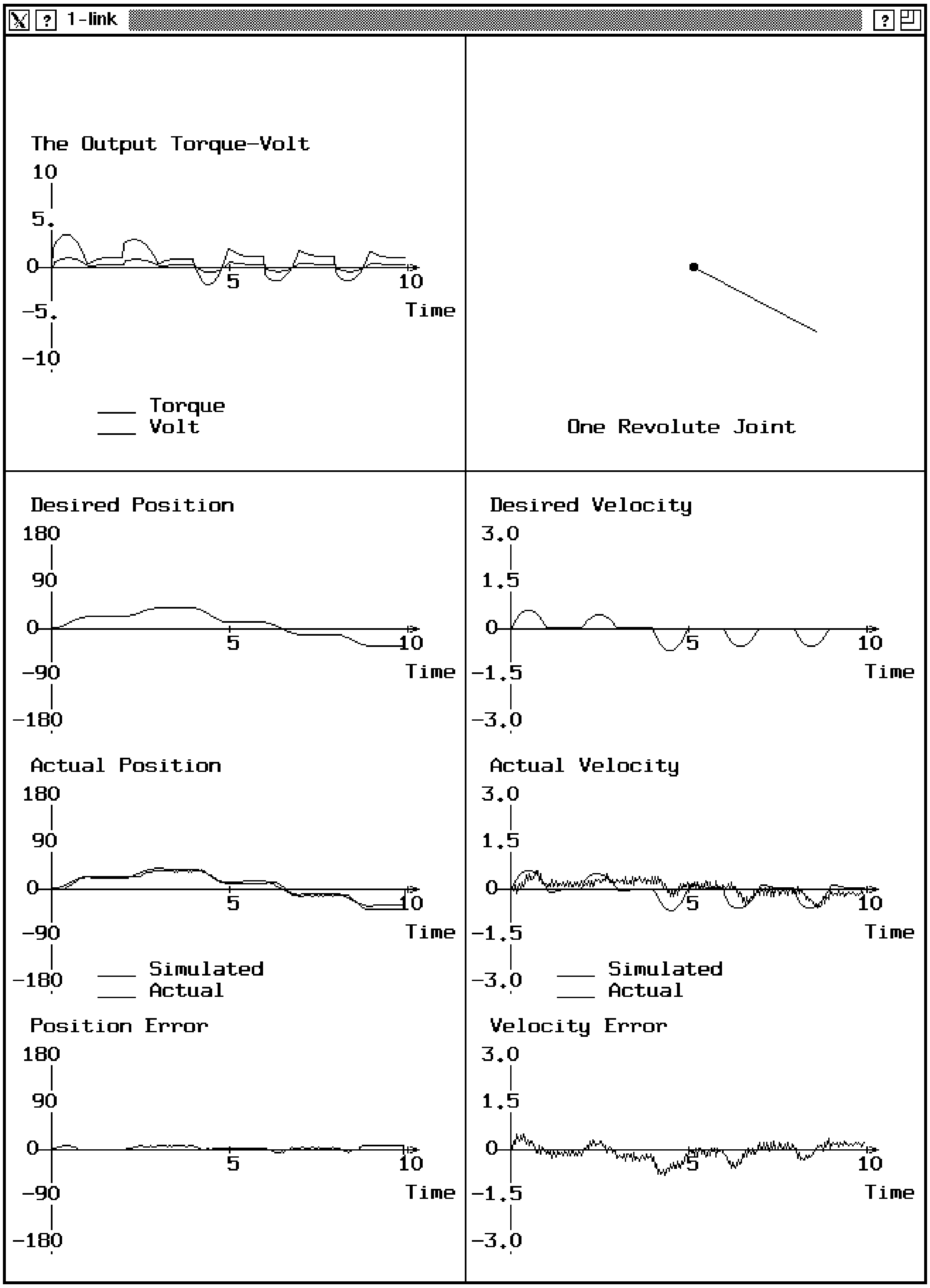


Figure 10: The Output Window of the Simulation Program for Sequence One.

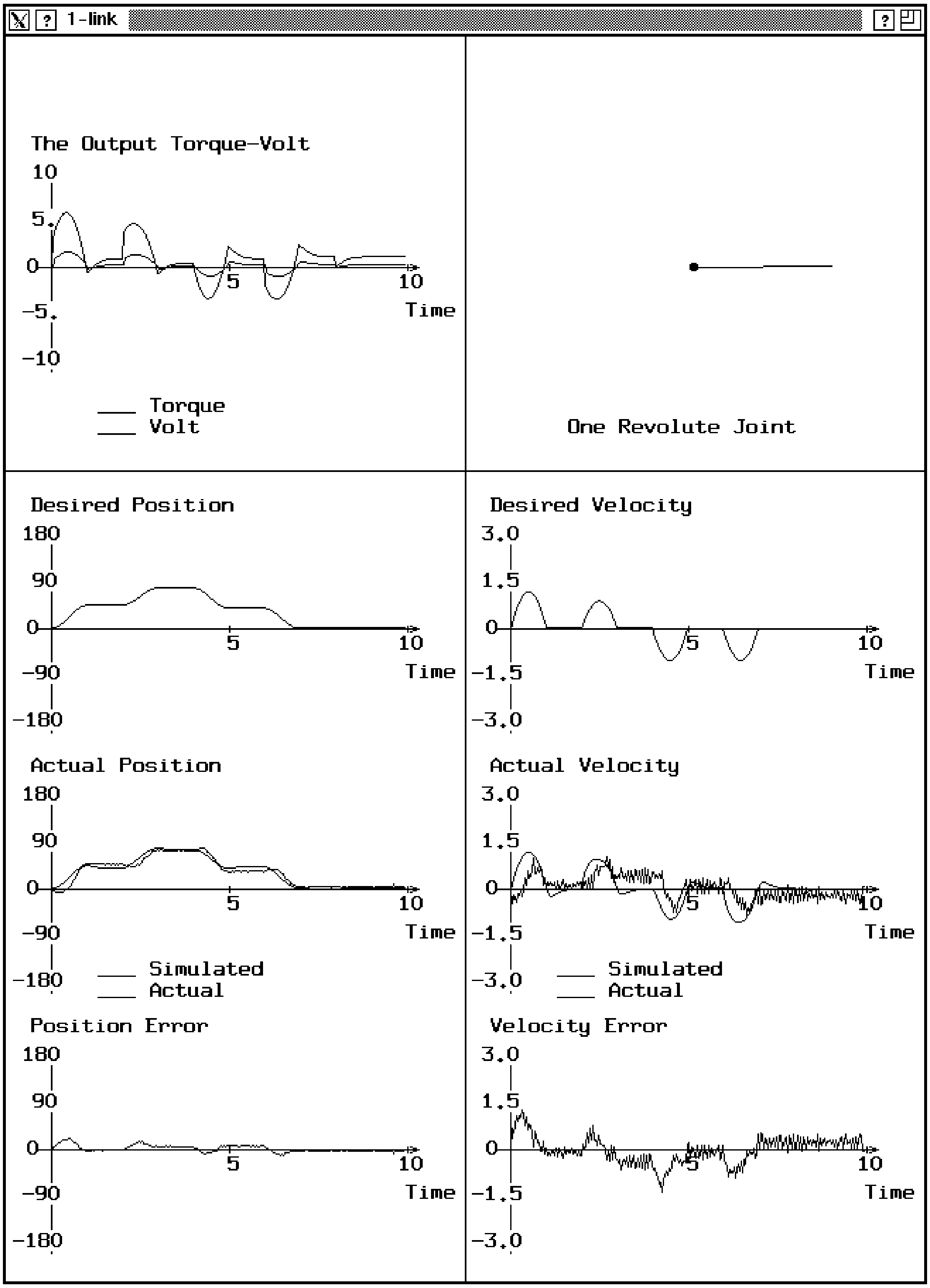


Figure 11: The Output Window of the Simulation Program for Sequence Two.

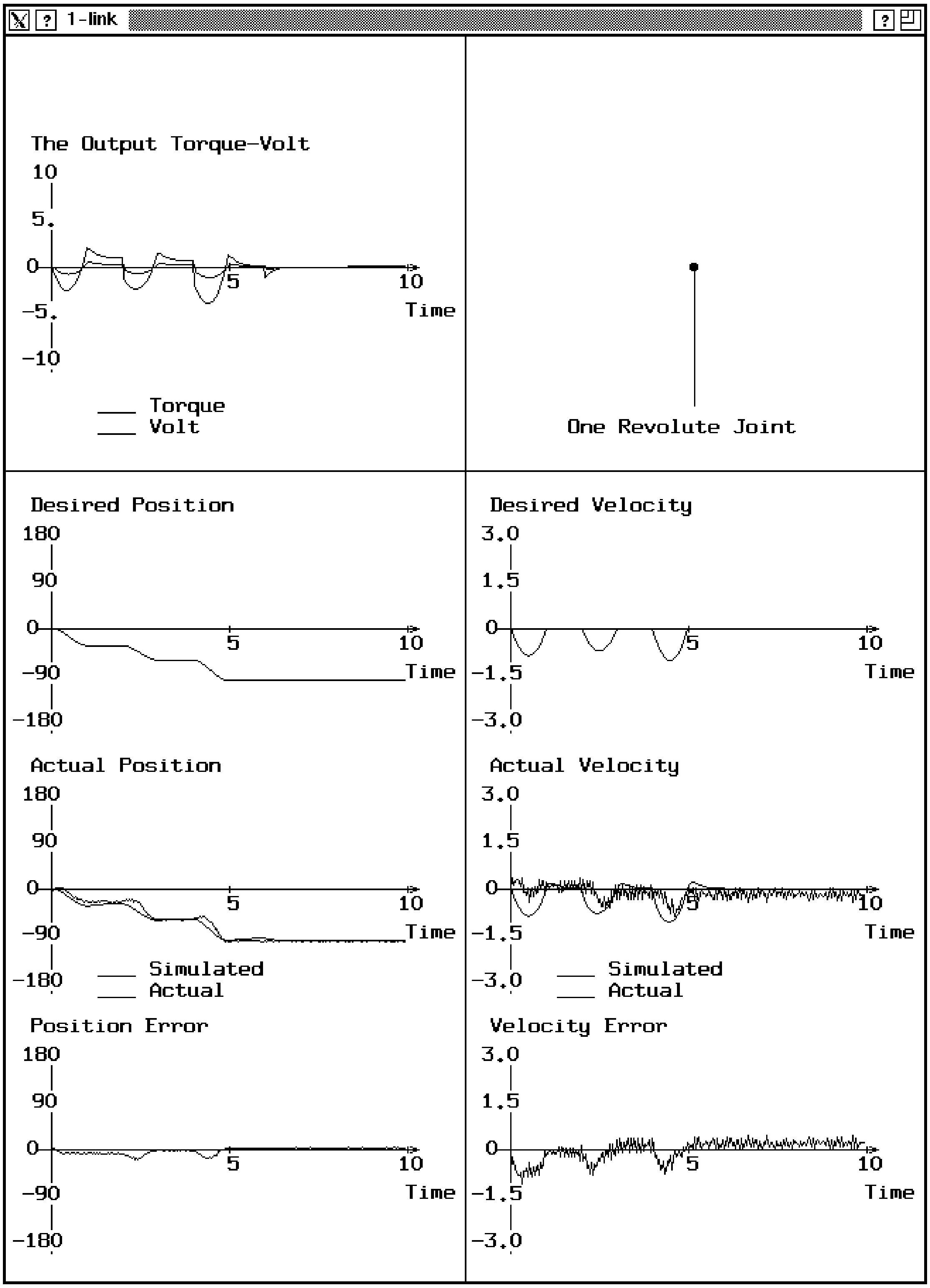


Figure 12: The Output Window of the Simulation Program for Sequence Three.

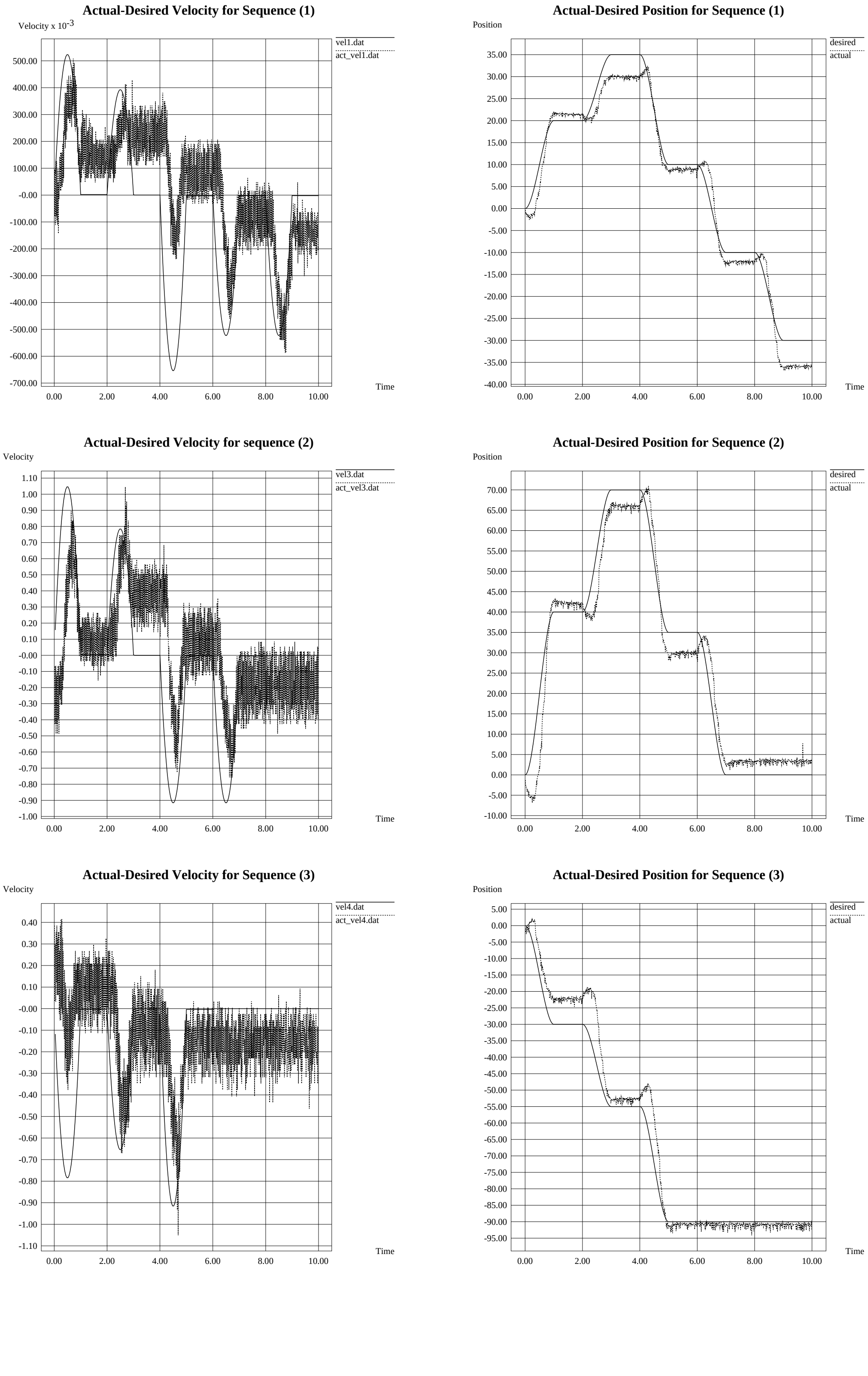


Figure 13: The Difference Between the Actual and the Desired Behavior.

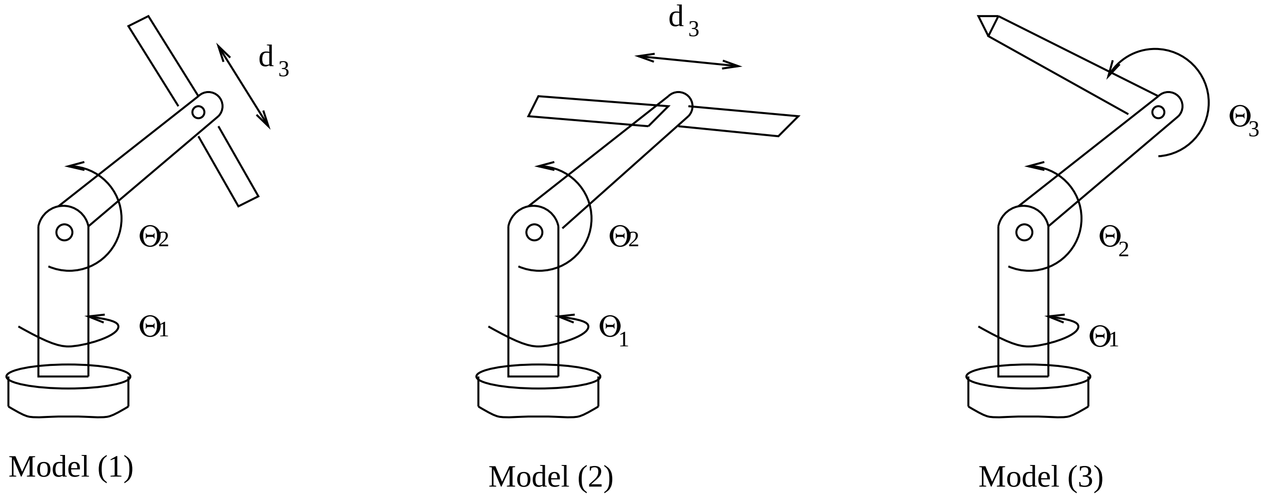


Figure 14: The three different configurations of the robot manipulator.

lengths, masses, inertia tensor, distance between each joint, the configuration of the robot, and the type of each link (revolute or prismatic). To make a modular and flexible design, we use variable parameters that can be fed to the system at run-time, so that we can use this controller for different configurations without any changes. We call this method the "Symbolic Robot" approach in design.

We have chosen three different configurations to be studied and developed. The first configuration is revolute-revolute-prismatic with the prismatic link parallel to the second link. The second configuration is also revolute-revolute-prismatic with the prismatic link perpendicular to the second link. The last configuration is three revolute joints. (See figure 14.)

We generated the kinematics and the dynamics of the three models using some software tools in the department called *genkin* and *gendyn* that take the configuration of the manipulator in a certain format and generate the corresponding kinematics and dynamics for that manipulator[5]. The kinematics and the dynamics of the three models are shown in Appendix A. One problem with the resultant equations is that they are not simplified at all, so, we simplify the results using the mathematical package *Mathematica*, which gives more simplified results, but still, not totally factorized. The comparison between the number of calculations before and after simplification will be discussed in the benchmarking section. Appendix B shows the dynamics after simplifying the equations using *Mathematica*.

For the trajectory generation, we used the cubic polynomials method that was described before in the trajectory generation section. This method is easy to implement and does not require a lot of calculations. It generates a cubic function that describes the motion from a starting point to a goal point in a certain time. Thus, this module will give us the desired trajectory to be followed, and this trajectory will serve as the input to the control module.

As we mentioned before, we will be using a linear feedback system to control our manipulator. Figure 15 shows a block diagram of a trajectory following controller. We seek a critically damped behavior for the output of that system, which means, we have to choose the feedback gains K_p and K_v such that:

$$K_p = 2\sqrt{K_v}$$

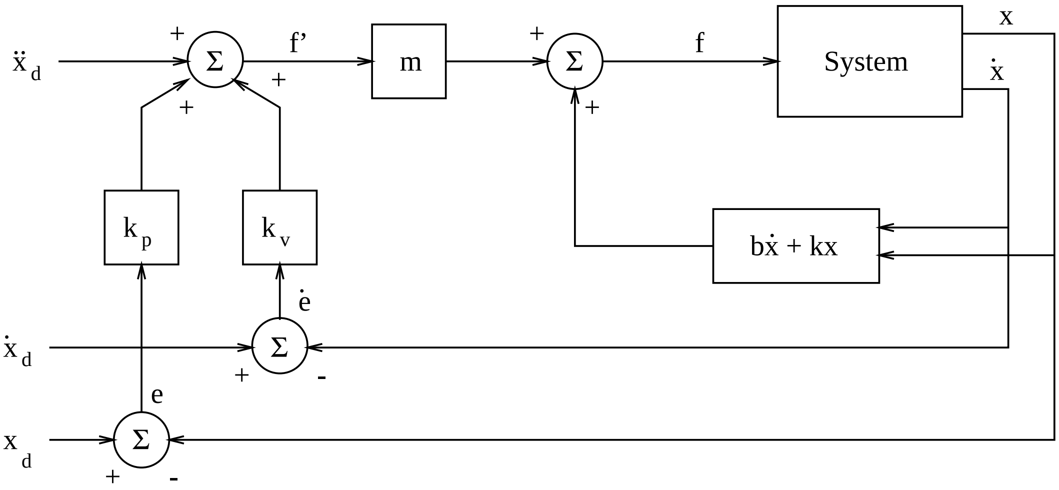


Figure 15: A trajectory follower controller.

The error in position and velocity is calculated using the readings of the actual position and velocity from the sensors at each joint. Our control module will simulate a PD controller to minimize that error. The error will depend on several factors such as, the frequency of update, the frequency of reading from the sensors, and the desired trajectory (for example, if we want to move with a big angle in a very small time interval, the error will be large).

The number of parameters in each module is an important factor if we decide to hardwire any of these modules. This number determines the type of chips that can be used; the range of each parameter determines the number of bits required per register in the chip. In our design we have forty-one fixed parameters (excluding the actuators' electrical parameters) which are:

- Link lengths, 3 parameters.
- Link masses, 3 parameters.
- The inertia tensor, we have three 3×3 matrixes, 27 parameters.
- The friction at each joint, 3 parameters.
- Configuration parameters, such as, the distance between any two links axis in a certain direction, 4 parameters.
- A selector to select between the three models.

By *fixed parameters* we mean those parameters that are fixed for a certain robot. Besides these parameters, we have 9 variable parameters representing position, velocity, and acceleration for each link.

7.2 Simulation

A simulation program has been implemented to study the performance of each manipulator and the effect of varying the update frequency on the system. It also helps to find approximate ranges for the required torque and/or voltage, determining the maximum velocity to know the type of sensors and A/D conversion needed. To make the benchmarks, as described in the next section, we didn't use a graphical interface to that simulator, since the drawing routines are very time consuming, and that will not help provide real figures for the speed.

In this simulator, we choose some reasonable parameters for the manipulator. The user can select the length of the simulation, and the update frequency. We used the third model for testing and benchmarking because its dynamics is the most difficult and time consuming. The following table shows the number of calculations and the dynamics module for each model.

	Addition	Multiplication	Division
Model 1	89	271	13
Model 2	85	307	0
Model 3	195	576	22

The next step is to implement a complete simulation for the three models with graphical interface similar to that of the one-link simulator described before. There is also a simulation package that was developed in the department last year, it is called *Z-Infinity*, which simulates multi-link manipulators, and gives a complete analysis of the trajectories, errors, and torques [5]. There are some parameters the user can change through the interface, such as the friction, the update frequency, single stepping, etc. *Z-Infinity* generates graphs showing the position velocity and acceleration which is generated on-line. The problem with the *Z-Infinity* is that the control part is hardwired in the system, so to change the control, you have to change the code and recompile. This limitation will be removed in the near future, so that we can use this system as a simulator for our models.

7.3 Benchmarking

One important decision that had to be made was the necessity to hardwire some or all of the controller module, and in that case, which modules, or even parts of the modules, should be hardwired? To answer these questions requires approximate figures for the required speed to achieve certain performance, the available machines we can use to run the controller, the available hardware that can be used to build such modules, and a time chart for each module in the system to figure out the bottlenecks. This also involves calculating the number of operations involved in each module to have a rough estimate for the time taken by each module.

We used the simulator described in the previous section to generate time charts for each module, and to compare the execution time on different machines. The machine used in this benchmarking are: SUN SparcStation-2, SUN SparcStation-10 model 30, SUN SparcStation-10 model 41, and HP-730. The following table shows the configurations of the machines used in this benchmarking, with the type, the clock cycle rate, the MIPS and MFLOPS for each of them.

	SPARC-2	SPARC-10 (30)	SPARC-10 (41)	HP-730
Clock Rate	40.0	36.0	40.0	66.0
MIPS	28.5	101.6	109.5	76.0
MFLOPS	4.3	20.5	22.4	23.0

To generate time charts for the execution time of each module, we used a program called *gprof* which produces an execution profile of C, Pascal, or Fortran77 programs. It give the execution time for each routine in the program, and the accumulated time for all the routines. Then we used *xgraph* to draw charts showing these time profiles. The following is part of the output of the gprof program for each machine.

```
Execution profile on Sun SPARC-2
================================

   %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 43.5       1.11     1.11    10001     0.11     0.11  _calc_dyn_var [5]
 12.5       1.43     0.32    10000     0.03     0.03  _calc_theta_dot_dot [6]
 11.0       1.71     0.28                            _sincos [7]
 10.6       1.98     0.27    10001     0.03     0.03  _trig [8]
  6.3       2.14     0.16    10000     0.02     0.02  _calc_tau [9]
  5.5       2.28     0.14    10000     0.01     0.01  _spline1 [10]
  4.7       2.40     0.12                            mcount (70)
  3.1       2.48     0.08    10000     0.01     0.01  _servo [11]
  2.0       2.53     0.05    10000     0.01     0.17  _update [4]
  0.4       2.54     0.01        4     2.50     2.50  _getfreehdr [15]
  0.4       2.55     0.01        3     3.33     3.33  _ioctl [16]

================================================================================

Execution profile on Sun SPARC-1- model 30
==========================================

   %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 39.2       0.40     0.40    10001     0.04     0.04  _calc_dyn_var [5]
 18.6       0.59     0.19    10000     0.02     0.02  _calc_theta_dot_dot [6]
 10.8       0.70     0.11    10001     0.01     0.01  _trig [7]
  7.8       0.78     0.08                            _sincos [8]
  7.4       0.86     0.08    10000     0.01     0.01  _calc_tau [9]
  4.4       0.90     0.05    10000     0.00     0.07  _update [4]
  3.9       0.94     0.04    10000     0.00     0.00  _spline1 [10]
  3.9       0.98     0.04                            mcount (51)
  2.0       1.00     0.02    10000     0.00     0.00  _servo [11]
  2.0       1.02     0.02        1    20.00   900.04  _three_link_robot [3]

================================================================================
```

```
Execution profile on Sun SPARC-10 model 41
==========================================

  %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 46.7       0.43     0.43    10001     0.04     0.04  _calc_dyn_var [5]
 16.3       0.58     0.15    10000     0.02     0.02  _calc_theta_dot_dot [6]
  9.8       0.67     0.09    10001     0.01     0.01  _trig [7]
  8.7       0.75     0.08                            _sincos [8]
  8.7       0.83     0.08    10000     0.01     0.01  _calc_tau [9]
  3.3       0.86     0.03    10000     0.00     0.00  _spline1 [10]
  3.3       0.89     0.03    10000     0.00     0.07  _update [4]
  2.2       0.91     0.02    10000     0.00     0.00  _servo [11]
  1.1       0.92     0.01                            mcount (51)

================================================================================

Execution profile on HP-730
===========================

%time cumsecs seconds   calls  msec/call name
 23.2    0.13    0.13   10001       0.01 calc_dyn_var
 12.5    0.20    0.07   10000       0.01 calc_tau
 10.7    0.26    0.06   10000       0.01 update
 10.7    0.32    0.06                    cos
  8.9    0.37    0.05   10001       0.00 trig
  8.9    0.42    0.05   10000       0.01 calc_theta_dot_dot
  8.9    0.47    0.05                    _mcount
  7.1    0.51    0.04                    sin
  3.6    0.53    0.02   10000       0.00 spline1
  1.8    0.54    0.01   10000       0.00 servo
  1.8    0.55    0.01      11       0.91 round64
  1.8    0.56    0.01       1      10.00 three_link_robot

================================================================================
```

We ran the simulation program with update frequency of 1000 Hz for 10 seconds, which means that each routine will be called 10,000 times. From this output, it is obvious that the bottleneck is the dynamics routine and usually it takes between 25% to 50% of the total execution time. Figure 16 shows these results graphically for each machine.

From these results we found that the HP-730 was the fastest of all, followed by the SPARC-10 machines. One thing we noticed is that after simplification using Mathematica, the execution time increases, but that

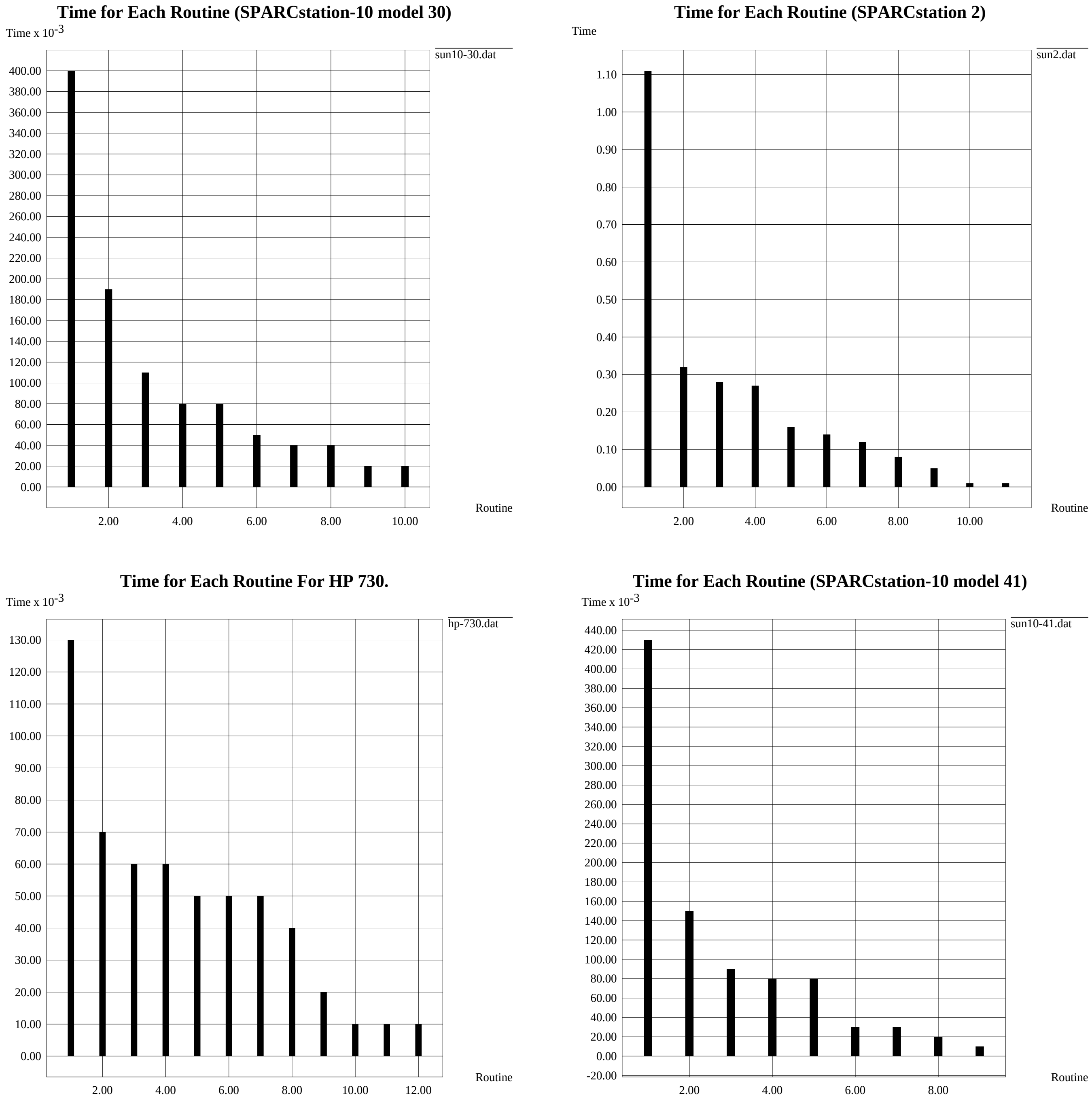


Figure 16: Execution Time for Each routine in the controller on different platforms.

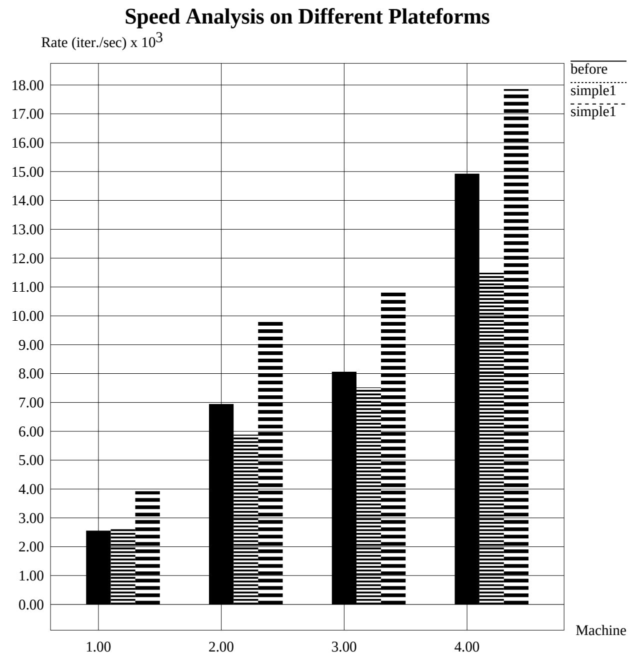


Figure 17: Performance comparison for different platforms: The machines from left to right are: SPARC 2, SPARC 10-30, SPARC 10-41, and HP 730. For each machine, there are three columns; before simplification, after using Mathematica, and after simplifying the trig. functions.

is because the results contain a lot of different trig functions, and it seems that the compilers on those machines do not use lookup tables for such functions (they probably use series expansions) so, we rewrote all non-basic trigonometric functions, such as $\sin 2\theta$ in terms of basic trigonometric functions as $2 \sin \theta \cos \theta$. Using this conversion, the performance was much better. Figure 17 shows a speed comparison between the machines. The graph represents the speed of each machine in terms of iteration per second. The machines are SPARC-2, SPARC-10-30, SPARC-10-41, and HP-730, respectively. For each machine, the first column is the speed before any simplification, the second column is the speed after using Mathematica (notice the performance degradation here), and the third column after simplifying the trig functions.

These benchmarks helped us decide that a software solution on a machine like the Sun SPARC-10 will be enough for our models, and there is no need for a special hardware solution. However, for more links, the decision might be different.

8 Current Developments

Several parts of this project are currently under development. In the following sections we will give a brief description for each and its role in this project. There will be another technical report in the near future to follow-up this one, and describes these developments in detail.

8.1 Sensor and Actuator Interface

This is an essential part of the project. It is concerned with the communication between the manipulator and the workstation used to control it. Basically, we have three D/A lines that transmit the required torque (or voltage) from the workstation to the three manipulator's actuators, and we have six A/D lines that transmit the sensors readings at each joint to the workstation (three for the position, and three for the velocity). These readings are used in the controller for the feedback information.

Right now, we are on the phase of controlling one-link robot using this interface, but this is still under debugging and testing.

8.2 Database and Spreadsheet Interface

In the beginning we talked about modularity and ease of changing and modifying. Since we have a large system that consists of several sub-systems, such as the controller, the simulator, the design, and the graphical interface, it becomes too difficult to keep track of the required changes in some sub-systems as a consequence of changing one or more of the parameters in one of the sub-systems. One tool that was developed to achieve this goal is a database interface system that builds (using a graphical interface) relations between the parameters of the system, so that any change in any of the parameters will automatically perform a set of modifications to the related parameters on the same system, and to the corresponding parameters in the other parts of the system. For example, link length is one of the parameters that has relations with other parameters such as masses and inertia tensors, and it also appears in the design, control, and simulation subsystems. If we change the length of one of the links, we would want the corresponding mass and inertia tensor to change with pre-specified functions that relate the length to each of them, we would also want the lengths in the other sub-systems to change as well.

This system has been implemented but it is still in the testing stage. It will be discussed in more detail in the next technical report.

8.3 Building the Robot

The design team has already completed a large portion of the robot physical design and modeling, some of the the required parts has already been ordered, such as motors, actuators, and sensors. The assembly process of the mechanical and electrical parts will start soon.

In this design the last link will be movable, so that the robot can be set in different configurations. The details of this process will be discussed in the next technical report.

9 Bibliography

References

[1] P. H. Chang, *A Closed-form Solution for Inverse Kinematics of Robot Manipulators with Redundancy*, IEEE Journal of Robotics and Automation, Vol. 3, No. 5, pp. 393-403, October 1987.

[2] J. Craig, *Introduction To Robotics*, Addison-Wesley, 1989.

[3] M. R. Cutkosky, R. S. Engelmore, R. E. Fikes, M. R. Genesereth, T. R. Gruber, W. S. Mark, J. M. Tenenbaum, and J, C. Weber, *PACT: An Experiment in Integrating Concurrent Engineering Systems*, IEEE Computer, January 1993.

[4] K. Hashimoto, and H. Kimura, *A New Parallel Algorithm for Inverse Dynamics*, The International Journal of Robotics Research, Vol. 8, No. 1, February 1989.

[5] T. C. Henderson, P. Dalton, and J. Zachary,*A Research Program for Autonomous Agent Behavior Specification and Analysis*, Proceeding of the IEEE International Symposium on Intelligent Control, Washington, D.C, 1991.

[6] A. Izaguirre, M. Hashimoto, R. P. Paul, and V. Hayward, *A New Computational Structure for Real-Time Dynamics*, The International Journal of Robotics Research, Vol. 8, No. 1, February 1989.

[7] L. Kelmar, P. K. Khosla, *Automatic Generation of Forward and Inverse Kinematics for a Reconfigurable Manipulator System*, Journal of Robotics Systems, Vol. 7, No. 4, pp. 599-619, 1990.

[8] P. Khosla, T. Kanade, R. Hoffman, D. Schmitz, and M. Delouis, *The Carnegie Mellon Reconfigurable Modular Manipulator System Project*, Technical Report, Carnegie Mellon University, 1992.

[9] R. H. Lathrop, *Parallelism in Manipulator Dynamics*, The International Journal of Robotics Research, Vol. 11, No. 4, pp. 346-361, August 1992.

[10] C. S. G. Lee, *Robot Arm Kinematics, Dynamics, and Control*, IEEE Computer, Vol. 15, No. 12, pp. 62-80, 1982.

[11] C. S. G. Lee, and P. R. Chang, *Efficient Parallel Algorithm for Robot Inverse Dynamics Computation*, IEEE Trans. Syst., Man, Cybern. SMC Vol. 16, No. 4, pp. 532-542, 1986.

[12] F. L. Lewis, C. T. Abdallah, D.M. Dawson, *Control of Robot Manipulator*, Macmillan, 1993.

[13] C. Li, A. Hemami, and T. S. Sankar, *A New Computational Method for Linearized Dynamics Models for Robot Manipulators*, The International Journal of Robotics Research, Vol. 9, No. 1, pp. 134-146, February 1990.

[14] J. Y. S Luh, and C. S. Lin, *Scheduling of Parallel Computation for a Computer-controlled Mechanical Manipulator*, IEEE Trans. Syst., Man, Cybern. SMC Vol. 12, No. 2, pp. 214-234, 1984.

[15] R. Nigam, and C. S. G. Lee, *A Multiprocessor-based Controller for Mechanical Manipulators*, IEEE Journal of Robotics and Automation, Vol. 1, No. 4, pp. 173-182, 1985.

[16] V. D. Tourassis, and M. H. A. Jr, *A Modular Architecture for Inverse Robot Kinematics*, IEEE Trans. on Robotics and Automation, Vol. 5, No. 5, pp. 555-568, October 1989.

[17] Digital Control Lab, ME522, *PC Analog I/O Users Guide*, Department of Mechanical Engineering, University of Utah.

10 Appendix A

The following is the dynamics and kinematics of the three robot models which was generated form the *gendyn* program. These dynamics equations are not simplified.

```
/* Dynamics equations for the first model */

#include <math.h>
#include "dyn1.h"

void dyn1Dyn (M, V, G, F, J_pos, J_vel, B_acc, External_F, External_M)
double **M, *V, *G, *F, *J_pos, *J_vel, *B_acc, *External_F, *External_M;

{
  double external_force_x = External_F[ 0];
  double external_force_y = External_F[ 1];
  double external_force_z = External_F[ 2];
  double external_moment_x = External_M[ 0];
  double external_moment_y = External_M[ 1];
  double external_moment_z = External_M[ 2];
  double base_x = B_acc[ 0];
  double base_y = B_acc[ 1];
  double base_z = B_acc[ 2];
  double T1 = J_pos[ 0];
  double T2 = J_pos[ 1];
  double D3 = J_pos[ 2];
  double vel_T1 = J_vel[ 0];
  double vel_T2 = J_vel[ 1];
  double vel_D3 = J_vel[ 2];
  double sinT1 = sin(T1);
  double cosT1 = cos(T1);
  double sinT2 = sin(T2);
  double cosT2 = cos(T2);

  M[ 0][ 0] = 2 * -JXY * cosT2+90 * sinT2+90
             - 2.0 * -KXZ * cosT2+90 * sinT2+90
             + 0.5*L2 * 0.5*L2 * M2 * cosT2+90 * cosT2+90
             + A2 * A2 * M3 * cosT2+90 * cosT2+90
             + A2 * C3+D3 * M3 * cosT2+90 * sinT2+90
             + 2 * A2 * C4-0.5*L3 * M3 * cosT2+90 * sinT2+90
             + 2.0 * C3+D3 * C4-0.5*L3 * M3 * sinT2+90 * sinT2+90
             + C4-0.5*L3 * C4-0.5*L3 * M3 * sinT2+90 * sinT2+90 + IZZ
```

```
            + JXX * sinT2+90 * sinT2+90 + JYY * cosT2+90 * cosT2+90
            + KXX * sinT2+90 * sinT2+90 + KZZ * cosT2+90 * cosT2+90  ;

M[ 0][ 1] = -JXZ * sinT2+90 + -JYZ * cosT2+90 + -KXY * sinT2+90
            - 1.0 * -KYZ * cosT2+90  ;

M[ 0][ 2] = 0  ;

M[ 1][ 0] = -JXZ * sinT2+90 + -JYZ * cosT2+90 + -KXY * sinT2+90
            - 1.0 * -KYZ * cosT2+90  ;

M[ 1][ 1] = 0.5*L2 * 0.5*L2 * M2 + A2 * A2 * M3
            + 2.0 * C3+D3 * C4-0.5*L3 * M3 + C4-0.5*L3 * C4-0.5*L3 * M3
            + JZZ + KYY  ;

M[ 1][ 2] = -1.0 * A2 * M3  ;

M[ 2][ 0] = 0  ;

M[ 2][ 1] = -1.0 * A2 * M3  ;

M[ 2][ 2] = M3  ;

V[ 0] = 2 * -JXY * cosT2+90 * cosT2+90 * vel_T1 * vel_T2
         - 2 * -JXY * sinT2+90 * sinT2+90 * vel_T1 * vel_T2
         + -JXZ * cosT2+90 * vel_T2 * vel_T2
         - 1 * -JYZ * sinT2+90 * vel_T2 * vel_T2
         + -KXY * cosT2+90 * vel_T2 * vel_T2
         - 2.0 * -KXZ * cosT2+90 * cosT2+90 * vel_T1 * vel_T2
         + 2.0 * -KXZ * sinT2+90 * sinT2+90 * vel_T1 * vel_T2
         + -KYZ * sinT2+90 * vel_T2 * vel_T2
         - 2 * 0.5*L2 * 0.5*L2 * M2 * cosT2+90 * sinT2+90 * vel_T1 * vel_T2
         - 2 * A2 * A2 * M3 * cosT2+90 * sinT2+90 * vel_T1 * vel_T2
         + 2.0 * A2 * C3+D3 * M3 * cosT2+90 * cosT2+90 * vel_T1 * vel_T2
         + 2.0 * A2 * C4-0.5*L3 * M3 * cosT2+90 * cosT2+90 * vel_T1 * vel_T2
         - 2 * A2 * C4-0.5*L3 * M3 * sinT2+90 * sinT2+90 * vel_T1 * vel_T2
         + 2 * A2 * M3 * cosT2+90 * sinT2+90 * vel_D3 * vel_T1
         + 3.0 * C3+D3 * C4-0.5*L3 * M3 * cosT2+90 * sinT2+90 * vel_T1 * vel_T2
         + 2.0 * C4-0.5*L3 * C4-0.5*L3 * M3 * cosT2+90 * sinT2+90 * vel_T1 * vel_T2
         + 2 * C4-0.5*L3 * M3 * sinT2+90 * sinT2+90 * vel_D3 * vel_T1
         + 2 * JXX * cosT2+90 * sinT2+90 * vel_T1 * vel_T2
         - 2 * JYY * cosT2+90 * sinT2+90 * vel_T1 * vel_T2
```

```
          + 2.0 * KXX * cosT2+90 * sinT2+90 * vel_T1 * vel_T2
          - 2.0 * KZZ * cosT2+90 * sinT2+90 * vel_T1 * vel_T2  ;

V[ 1] = -1 * -JXY * cosT2+90 * cosT2+90 * vel_T1 * vel_T1
          + -JXY * sinT2+90 * sinT2+90 * vel_T1 * vel_T1
          + -KXZ * cosT2+90 * cosT2+90 * vel_T1 * vel_T1
          - 1 * -KXZ * sinT2+90 * sinT2+90 * vel_T1 * vel_T1
          + 0.5*L2 * 0.5*L2 * M2 * cosT2+90 * sinT2+90 * vel_T1 * vel_T1
          + A2 * A2 * M3 * cosT2+90 * sinT2+90 * vel_T1 * vel_T1
          + A2 * C3+D3 * M3 * sinT2+90 * sinT2+90 * vel_T1 * vel_T1
          + A2 * C3+D3 * M3 * vel_T2 * vel_T2
          - 1 * A2 * C4-0.5*L3 * M3 * cosT2+90 * cosT2+90 * vel_T1 * vel_T1
          + A2 * C4-0.5*L3 * M3 * sinT2+90 * sinT2+90 * vel_T1 * vel_T1
          - 1.0 * C3+D3 * C4-0.5*L3 * M3 * cosT2+90 * sinT2+90 * vel_T1 * vel_T1
          - 1.0 * C4-0.5*L3 * C4-0.5*L3 * M3 * cosT2+90 * sinT2+90 * vel_T1 * vel_T1
          + 2 * C4-0.5*L3 * M3 * cosT2+90 * vel_D3 * vel_T1
          - 1 * JXX * cosT2+90 * sinT2+90 * vel_T1 * vel_T1
          + JYY * cosT2+90 * sinT2+90 * vel_T1 * vel_T1
          - 1.0 * KXX * cosT2+90 * sinT2+90 * vel_T1 * vel_T1
          + KZZ * cosT2+90 * sinT2+90 * vel_T1 * vel_T1  ;

V[ 2] = -1.0 * A2 * M3 * cosT2+90 * sinT2+90 * vel_T1 * vel_T1
          - 1.0 * C3+D3 * M3 * sinT2+90 * sinT2+90 * vel_T1 * vel_T1
          - 1.0 * C3+D3 * M3 * vel_T2 * vel_T2
          - 1 * C4-0.5*L3 * M3 * sinT2+90 * sinT2+90 * vel_T1 * vel_T1
          - 1 * C4-0.5*L3 * M3 * vel_T2 * vel_T2  ;

G[ 0] = 0.5*L2 * GRAVITY * M2 * cosT1 * cosT2+90
          + A2 * GRAVITY * M3 * cosT1 * cosT2+90
          + C4-0.5*L3 * GRAVITY * M3 * cosT1 * sinT2+90  ;

G[ 1] = -1 * 0.5*L2 * GRAVITY * M2 * sinT1 * sinT2+90
          - 1.0 * A2 * GRAVITY * M3 * sinT1 * sinT2+90
          + C4-0.5*L3 * GRAVITY * M3 * cosT2+90 * sinT1  ;

G[ 2] = GRAVITY * M3 * sinT1 * sinT2+90  ;

/* Torque due to external moments */

F[ 0] = -1.0 * cosT2+90 * external_moment_z
          + external_moment_x * sinT2+90  ;
```

```
  F[ 1] = external_moment_y  ;

  F[ 2] = 0  ;

/* Torque due to external forces */

  F[ 0] += -1 * A2 * cosT2+90 * external_force_y
          - 1.0 * C3+D3 * external_force_y * sinT2+90
          - 1 * C4 * external_force_y * sinT2+90  ;

  F[ 1] += -1.0 * A2 * external_force_z + C3+D3 * external_force_x
          + C4 * external_force_x  ;

  F[ 2] += external_force_z  ;

/* Torque due to base movement */

  F[ 0] += 0.5*L2 * M2 * base_y * cosT1 * cosT2+90
          + A2 * M3 * base_y * cosT1 * cosT2+90
          + C4-0.5*L3 * M3 * base_y * cosT1 * sinT2+90  ;

  F[ 1] += -1 * 0.5*L2 * M2 * base_x * cosT1 * sinT2+90
          - 1 * 0.5*L2 * M2 * base_y * sinT1 * sinT2+90
          + 0.5*L2 * M2 * base_z * cosT2+90
          - 1.0 * A2 * M3 * base_x * cosT1 * sinT2+90
          - 1.0 * A2 * M3 * base_y * sinT1 * sinT2+90
          + A2 * M3 * base_z * cosT2+90
          + C4-0.5*L3 * M3 * base_x * cosT1 * cosT2+90
          + C4-0.5*L3 * M3 * base_y * cosT2+90 * sinT1
          + C4-0.5*L3 * M3 * base_z * sinT2+90  ;

  F[ 2] += M3 * base_x * cosT1 * sinT2+90 + M3 * base_y * sinT1 * sinT2+90

          - 1.0 * M3 * base_z * cosT2+90  ;

/* Force due to friction forces */

  F[ 0] += 0.0  ;
  F[ 1] += 0.0  ;
  F[ 2] += 0.0  ;
}
```

```
void dyn1Frm (TransformList, J_pos)
double ***TransformList, *J_pos;

{
  double T1 = J_pos[ 0];
  double T2 = J_pos[ 1];
  double D3 = J_pos[ 2];
  double sinT1 = sin(T1);
  double cosT1 = cos(T1);
  double sinT2 = sin(T2);
  double cosT2 = cos(T2);
  double **T;

  T = TransformList[ 0];
  T[ 0][ 0] = cosT1  ;
  T[ 0][ 1] = -1 * sinT1  ;
  T[ 0][ 2] = 0  ;
  T[ 0][ 3] = 0  ;
  T[ 1][ 0] = sinT1  ;
  T[ 1][ 1] = cosT1  ;
  T[ 1][ 2] = 0  ;
  T[ 1][ 3] = 0  ;
  T[ 2][ 0] = 0  ;
  T[ 2][ 1] = 0  ;
  T[ 2][ 2] = 1  ;
  T[ 2][ 3] = 0  ;
  T[ 3][ 0] = 0  ;
  T[ 3][ 1] = 0  ;
  T[ 3][ 2] = 0  ;
  T[ 3][ 3] = 1  ;

  T = TransformList[ 1];
  T[ 0][ 0] = cosT1 * cosT2+90  ;
  T[ 0][ 1] = -1 * cosT1 * sinT2+90  ;
  T[ 0][ 2] = sinT1  ;
  T[ 0][ 3] = 0  ;
  T[ 1][ 0] = cosT2+90 * sinT1  ;
  T[ 1][ 1] = -1 * sinT1 * sinT2+90  ;
  T[ 1][ 2] = -1.0 * cosT1  ;
  T[ 1][ 3] = 0  ;
  T[ 2][ 0] = sinT2+90  ;
```

```
T[ 2][ 1] = cosT2+90  ;
T[ 2][ 2] = 0  ;
T[ 2][ 3] = 0  ;
T[ 3][ 0] = 0  ;
T[ 3][ 1] = 0  ;
T[ 3][ 2] = 0  ;
T[ 3][ 3] = 1  ;

T = TransformList[ 2];
T[ 0][ 0] = cosT1 * cosT2+90  ;
T[ 0][ 1] = sinT1  ;
T[ 0][ 2] = cosT1 * sinT2+90  ;
T[ 0][ 3] = A2 * cosT1 * cosT2+90 + C3+D3 * cosT1 * sinT2+90  ;
T[ 1][ 0] = cosT2+90 * sinT1  ;
T[ 1][ 1] = -1.0 * cosT1  ;
T[ 1][ 2] = sinT1 * sinT2+90  ;
T[ 1][ 3] = A2 * cosT2+90 * sinT1 + C3+D3 * sinT1 * sinT2+90  ;
T[ 2][ 0] = sinT2+90  ;
T[ 2][ 1] = 0  ;
T[ 2][ 2] = -1.0 * cosT2+90  ;
T[ 2][ 3] = A2 * sinT2+90 - 1.0 * C3+D3 * cosT2+90  ;
T[ 3][ 0] = 0  ;
T[ 3][ 1] = 0  ;
T[ 3][ 2] = 0  ;
T[ 3][ 3] = 1  ;

T = TransformList[ 3];
T[ 0][ 0] = cosT1 * cosT2+90  ;
T[ 0][ 1] = sinT1  ;
T[ 0][ 2] = cosT1 * sinT2+90  ;
T[ 0][ 3] = C4 * cosT1 * sinT2+90 + A2 * cosT1 * cosT2+90
             + C3+D3 * cosT1 * sinT2+90  ;

T[ 1][ 0] = cosT2+90 * sinT1  ;
T[ 1][ 1] = -1.0 * cosT1  ;
T[ 1][ 2] = sinT1 * sinT2+90  ;
T[ 1][ 3] = C4 * sinT1 * sinT2+90 + A2 * cosT2+90 * sinT1
             + C3+D3 * sinT1 * sinT2+90  ;

T[ 2][ 0] = sinT2+90  ;
T[ 2][ 1] = 0  ;
T[ 2][ 2] = -1.0 * cosT2+90  ;
```

```
  T[ 2][ 3] = -1.0 * C4 * cosT2+90 + A2 * sinT2+90 - 1.0 * C3+D3 * cosT2+90  ;

  T[ 3][ 0] = 0  ;
  T[ 3][ 1] = 0  ;
  T[ 3][ 2] = 0  ;
  T[ 3][ 3] = 1  ;

}
/*****************************************************************************/

/* Dynamics equations for the second model */

#include <math.h>
#include "dyn2.h"

void dyn2Dyn (M, V, G, F, J_pos, J_vel, B_acc, External_F, External_M)
double **M, *V, *G, *F, *J_pos, *J_vel, *B_acc, *External_F, *External_M;

{
  double external_force_x = External_F[ 0];
  double external_force_y = External_F[ 1];
  double external_force_z = External_F[ 2];
  double external_moment_x = External_M[ 0];
  double external_moment_y = External_M[ 1];
  double external_moment_z = External_M[ 2];
  double base_x = B_acc[ 0];
  double base_y = B_acc[ 1];
  double base_z = B_acc[ 2];
  double T1 = J_pos[ 0];
  double T2 = J_pos[ 1];
  double D3 = J_pos[ 2];
  double vel_T1 = J_vel[ 0];
  double vel_T2 = J_vel[ 1];
  double vel_D3 = J_vel[ 2];
  double sinT1 = sin(T1);
  double cosT1 = cos(T1);
  double sinT2 = sin(T2);
  double cosT2 = cos(T2);

  M[ 0][ 0] = 2 * -JXY * cosT2 * sinT2 + 2 * -KXZ * cosT2 * sinT2
              + 0.5*L2 * 0.5*L2 * M2 * cosT2 * cosT2
              + A2 * A2 * M3 * cosT2 * cosT2
```

```
             - 2.0 * A2 * C4-0.5*L3 * M3 * cosT2 * sinT2
             - 2.0 * A2 * D3 * M3 * cosT2 * sinT2
             + C4-0.5*L3 * C4-0.5*L3 * M3 * sinT2 * sinT2
             + 2.0 * C4-0.5*L3 * D3 * M3 * sinT2 * sinT2
             + D3 * D3 * M3 * sinT2 * sinT2 + IZZ + JXX * sinT2 * sinT2
             + JYY * cosT2 * cosT2 + KXX * sinT2 * sinT2
             + KZZ * cosT2 * cosT2  ;

M[ 0][ 1] = -JXZ * sinT2 + -JYZ * cosT2 - 1.0 * -KXY * sinT2
             - 1.0 * -KYZ * cosT2  ;

M[ 0][ 2] = 0  ;

M[ 1][ 0] = -JXZ * sinT2 + -JYZ * cosT2 - 1.0 * -KXY * sinT2
             - 1.0 * -KYZ * cosT2  ;

M[ 1][ 1] = 0.5*L2 * 0.5*L2 * M2 + A2 * A2 * M3
             + C4-0.5*L3 * C4-0.5*L3 * M3 + 2.0 * C4-0.5*L3 * D3 * M3
             + D3 * D3 * M3 + JZZ + KYY  ;

M[ 1][ 2] = A2 * M3  ;

M[ 2][ 0] = 0  ;

M[ 2][ 1] = A2 * M3  ;

M[ 2][ 2] = M3  ;

V[ 0] = 2 * -JXY * cosT2 * cosT2 * vel_T1 * vel_T2
          - 2 * -JXY * sinT2 * sinT2 * vel_T1 * vel_T2
          + -JXZ * cosT2 * vel_T2 * vel_T2
          - 1 * -JYZ * sinT2 * vel_T2 * vel_T2
          - 1.0 * -KXY * cosT2 * vel_T2 * vel_T2
          + 2.0 * -KXZ * cosT2 * cosT2 * vel_T1 * vel_T2
          - 2.0 * -KXZ * sinT2 * sinT2 * vel_T1 * vel_T2
          + -KYZ * sinT2 * vel_T2 * vel_T2
          - 2 * 0.5*L2 * 0.5*L2 * M2 * cosT2 * sinT2 * vel_T1 * vel_T2
          - 2.0 * A2 * A2 * M3 * cosT2 * sinT2 * vel_T1 * vel_T2
          - 2.0 * A2 * C4-0.5*L3 * M3 * cosT2 * cosT2 * vel_T1 * vel_T2
          + 2.0 * A2 * C4-0.5*L3 * M3 * sinT2 * sinT2 * vel_T1 * vel_T2
          - 2.0 * A2 * D3 * M3 * cosT2 * cosT2 * vel_T1 * vel_T2
          + 2.0 * A2 * D3 * M3 * sinT2 * sinT2 * vel_T1 * vel_T2
```

```
          - 2.0 * A2 * M3 * cosT2 * sinT2 * vel_D3 * vel_T1
          + 2.0 * C4-0.5*L3 * C4-0.5*L3 * M3 * cosT2 * sinT2 * vel_T1 * vel_T2
          + 4.0 * C4-0.5*L3 * D3 * M3 * cosT2 * sinT2 * vel_T1 * vel_T2
          + 2 * C4-0.5*L3 * M3 * sinT2 * sinT2 * vel_D3 * vel_T1
          + 2.0 * D3 * D3 * M3 * cosT2 * sinT2 * vel_T1 * vel_T2
          + 2.0 * D3 * M3 * sinT2 * sinT2 * vel_D3 * vel_T1
          + 2 * JXX * cosT2 * sinT2 * vel_T1 * vel_T2
          - 2 * JYY * cosT2 * sinT2 * vel_T1 * vel_T2
          + 2.0 * KXX * cosT2 * sinT2 * vel_T1 * vel_T2
          - 2.0 * KZZ * cosT2 * sinT2 * vel_T1 * vel_T2  ;

V[ 1] = -1 * -JXY * cosT2 * cosT2 * vel_T1 * vel_T1
          + -JXY * sinT2 * sinT2 * vel_T1 * vel_T1
          - 1.0 * -KXZ * cosT2 * cosT2 * vel_T1 * vel_T1
          + -KXZ * sinT2 * sinT2 * vel_T1 * vel_T1
          + 0.5*L2 * 0.5*L2 * M2 * cosT2 * sinT2 * vel_T1 * vel_T1
          + A2 * A2 * M3 * cosT2 * sinT2 * vel_T1 * vel_T1
          + A2 * C4-0.5*L3 * M3 * cosT2 * cosT2 * vel_T1 * vel_T1
          - 1 * A2 * C4-0.5*L3 * M3 * sinT2 * sinT2 * vel_T1 * vel_T1
          + A2 * D3 * M3 * cosT2 * cosT2 * vel_T1 * vel_T1
          - 1 * A2 * D3 * M3 * sinT2 * sinT2 * vel_T1 * vel_T1
          - 1.0 * C4-0.5*L3 * C4-0.5*L3 * M3 * cosT2 * sinT2 * vel_T1 * vel_T1
          - 2.0 * C4-0.5*L3 * D3 * M3 * cosT2 * sinT2 * vel_T1 * vel_T1
          - 2.0 * C4-0.5*L3 * M3 * cosT2 * vel_D3 * vel_T1
          - 1 * D3 * D3 * M3 * cosT2 * sinT2 * vel_T1 * vel_T1
          - 2 * D3 * M3 * cosT2 * vel_D3 * vel_T1
          - 1 * JXX * cosT2 * sinT2 * vel_T1 * vel_T1
          + JYY * cosT2 * sinT2 * vel_T1 * vel_T1
          - 1.0 * KXX * cosT2 * sinT2 * vel_T1 * vel_T1
          + KZZ * cosT2 * sinT2 * vel_T1 * vel_T1  ;

V[ 2] = A2 * M3 * cosT2 * sinT2 * vel_T1 * vel_T1
          - 1 * C4-0.5*L3 * M3 * sinT2 * sinT2 * vel_T1 * vel_T1
          - 1.0 * C4-0.5*L3 * M3 * vel_T2 * vel_T2
          - 1 * D3 * M3 * sinT2 * sinT2 * vel_T1 * vel_T1
          - 1 * D3 * M3 * vel_T2 * vel_T2  ;

G[ 0] = 0.5*L2 * GRAVITY * M2 * cosT1 * cosT2
          + A2 * GRAVITY * M3 * cosT1 * cosT2
          - 1.0 * C4-0.5*L3 * GRAVITY * M3 * cosT1 * sinT2
          - 1.0 * D3 * GRAVITY * M3 * cosT1 * sinT2  ;
```

```
  G[ 1] = -1 * 0.5*L2 * GRAVITY * M2 * sinT1 * sinT2
          - 1 * A2 * GRAVITY * M3 * sinT1 * sinT2
          - 1.0 * C4-0.5*L3 * GRAVITY * M3 * cosT2 * sinT1
          - 1 * D3 * GRAVITY * M3 * cosT2 * sinT1  ;

  G[ 2] = -1 * GRAVITY * M3 * sinT1 * sinT2  ;

/* Torque due to external moments */

  F[ 0] = cosT2 * external_moment_z + external_moment_x * sinT2  ;
  F[ 1] = -1.0 * external_moment_y  ;
  F[ 2] = 0  ;

/* Torque due to external forces */

  F[ 0] += A2 * cosT2 * external_force_y - 1 * C4 * external_force_y * sinT2
          - 1.0 * D3 * external_force_y * sinT2  ;

  F[ 1] += A2 * external_force_z - 1.0 * C4 * external_force_x
          - 1 * D3 * external_force_x  ;

  F[ 2] += external_force_z  ;

/* Torque due to base movement */

  F[ 0] += 0.5*L2 * M2 * base_y * cosT1 * cosT2
          + A2 * M3 * base_y * cosT1 * cosT2
          - 1.0 * C4-0.5*L3 * M3 * base_y * cosT1 * sinT2
          - 1.0 * D3 * M3 * base_y * cosT1 * sinT2  ;

  F[ 1] += -1 * 0.5*L2 * M2 * base_x * cosT1 * sinT2
          - 1 * 0.5*L2 * M2 * base_y * sinT1 * sinT2
          + 0.5*L2 * M2 * base_z * cosT2
          - 1 * A2 * M3 * base_x * cosT1 * sinT2
          - 1 * A2 * M3 * base_y * sinT1 * sinT2
          + A2 * M3 * base_z * cosT2
          - 1.0 * C4-0.5*L3 * M3 * base_x * cosT1 * cosT2
          - 1.0 * C4-0.5*L3 * M3 * base_y * cosT2 * sinT1
          - 1.0 * C4-0.5*L3 * M3 * base_z * sinT2
          - 1 * D3 * M3 * base_x * cosT1 * cosT2
          - 1 * D3 * M3 * base_y * cosT2 * sinT1
          - 1 * D3 * M3 * base_z * sinT2  ;
```

```
  F[ 2] += -1 * M3 * base_x * cosT1 * sinT2
           - 1 * M3 * base_y * sinT1 * sinT2 + M3 * base_z * cosT2  ;

/* Force due to friction forces */

  F[ 0] += 0.0  ;
  F[ 1] += 0.0  ;
  F[ 2] += 0.0  ;
}


void dyn2Frm (TransformList, J_pos)
double ***TransformList, *J_pos;

{
  double T1 = J_pos[ 0];
  double T2 = J_pos[ 1];
  double D3 = J_pos[ 2];
  double sinT1 = sin(T1);
  double cosT1 = cos(T1);
  double sinT2 = sin(T2);
  double cosT2 = cos(T2);
  double **T;

  T = TransformList[ 0];
  T[ 0][ 0] = cosT1  ;
  T[ 0][ 1] = -1 * sinT1  ;
  T[ 0][ 2] = 0  ;
  T[ 0][ 3] = 0  ;
  T[ 1][ 0] = sinT1  ;
  T[ 1][ 1] = cosT1  ;
  T[ 1][ 2] = 0  ;
  T[ 1][ 3] = 0  ;
  T[ 2][ 0] = 0  ;
  T[ 2][ 1] = 0  ;
  T[ 2][ 2] = 1  ;
  T[ 2][ 3] = 0  ;
  T[ 3][ 0] = 0  ;
  T[ 3][ 1] = 0  ;
  T[ 3][ 2] = 0  ;
  T[ 3][ 3] = 1  ;
```

```
T = TransformList[ 1];
T[ 0][ 0] = cosT1 * cosT2  ;
T[ 0][ 1] = -1 * cosT1 * sinT2  ;
T[ 0][ 2] = sinT1  ;
T[ 0][ 3] = 0  ;
T[ 1][ 0] = cosT2 * sinT1  ;
T[ 1][ 1] = -1 * sinT1 * sinT2  ;
T[ 1][ 2] = -1.0 * cosT1  ;
T[ 1][ 3] = 0  ;
T[ 2][ 0] = sinT2  ;
T[ 2][ 1] = cosT2  ;
T[ 2][ 2] = 0  ;
T[ 2][ 3] = 0  ;
T[ 3][ 0] = 0  ;
T[ 3][ 1] = 0  ;
T[ 3][ 2] = 0  ;
T[ 3][ 3] = 1  ;

T = TransformList[ 2];
T[ 0][ 0] = cosT1 * cosT2  ;
T[ 0][ 1] = -1.0 * sinT1  ;
T[ 0][ 2] = -1 * cosT1 * sinT2  ;
T[ 0][ 3] = A2 * cosT1 * cosT2 - 1 * D3 * cosT1 * sinT2  ;
T[ 1][ 0] = cosT2 * sinT1  ;
T[ 1][ 1] = cosT1  ;
T[ 1][ 2] = -1 * sinT1 * sinT2  ;
T[ 1][ 3] = A2 * cosT2 * sinT1 - 1 * D3 * sinT1 * sinT2  ;
T[ 2][ 0] = sinT2  ;
T[ 2][ 1] = 0  ;
T[ 2][ 2] = cosT2  ;
T[ 2][ 3] = A2 * sinT2 + D3 * cosT2  ;
T[ 3][ 0] = 0  ;
T[ 3][ 1] = 0  ;
T[ 3][ 2] = 0  ;
T[ 3][ 3] = 1  ;

T = TransformList[ 3];
T[ 0][ 0] = cosT1 * cosT2  ;
T[ 0][ 1] = -1.0 * sinT1  ;
T[ 0][ 2] = -1 * cosT1 * sinT2  ;
T[ 0][ 3] = -1 * C4 * cosT1 * sinT2 + A2 * cosT1 * cosT2
```

```
              - 1 * D3 * cosT1 * sinT2  ;

  T[ 1][ 0] = cosT2 * sinT1  ;
  T[ 1][ 1] = cosT1  ;
  T[ 1][ 2] = -1 * sinT1 * sinT2  ;
  T[ 1][ 3] = -1 * C4 * sinT1 * sinT2 + A2 * cosT2 * sinT1
              - 1 * D3 * sinT1 * sinT2  ;

  T[ 2][ 0] = sinT2  ;
  T[ 2][ 1] = 0  ;
  T[ 2][ 2] = cosT2  ;
  T[ 2][ 3] = C4 * cosT2 + A2 * sinT2 + D3 * cosT2  ;

  T[ 3][ 0] = 0  ;
  T[ 3][ 1] = 0  ;
  T[ 3][ 2] = 0  ;
  T[ 3][ 3] = 1  ;

}
/****************************************************************************/

/* Dynamics equations for the third model */

#include <math.h>
#include "dyn3.h"

void dyn3Dyn (M, V, G, F, J_pos, J_vel, B_acc, External_F, External_M)
double **M, *V, *G, *F, *J_pos, *J_vel, *B_acc, *External_F, *External_M;

{
  double external_force_x = External_F[ 0];
  double external_force_y = External_F[ 1];
  double external_force_z = External_F[ 2];
  double external_moment_x = External_M[ 0];
  double external_moment_y = External_M[ 1];
  double external_moment_z = External_M[ 2];
  double base_x = B_acc[ 0];
  double base_y = B_acc[ 1];
  double base_z = B_acc[ 2];
  double T1 = J_pos[ 0];
  double T2 = J_pos[ 1];
  double T3 = J_pos[ 2];
```

```
double vel_T1 = J_vel[ 0];
double vel_T2 = J_vel[ 1];
double vel_T3 = J_vel[ 2];
double sinT1 = sin(T1);
double cosT1 = cos(T1);
double sinT2 = sin(T2);
double cosT2 = cos(T2);
double sinT3 = sin(T3);
double cosT3 = cos(T3);

M[ 0][ 0] = 2 * -JXY * cosT2 * sinT2
             + 2 * -KXY * cosT2 * cosT2 * cosT3 * sinT3
             + 2 * -KXY * cosT2 * cosT3 * cosT3 * sinT2
             - 2 * -KXY * cosT2 * sinT2 * sinT3 * sinT3
             - 2 * -KXY * cosT3 * sinT2 * sinT2 * sinT3
             + 0.5*L2 * 0.5*L2 * M2 * cosT2 * cosT2
             + 0.5*L3 * 0.5*L3 * M3 * cosT2 * cosT2 * cosT3 * cosT3
             - 1 * 0.5*L3 * 0.5*L3 * M3 * cosT2 * cosT3 * sinT2 * sinT3
             + 2 * 0.5*L3 * A2 * M3 * cosT2 * cosT2 * cosT3
             - 1 * 0.5*L3 * A2 * M3 * cosT2 * sinT2 * sinT3
             + A2 * A2 * M3 * cosT2 * cosT2 + IZZ + JXX * sinT2 * sinT2
             + JYY * cosT2 * cosT2 + KXX * cosT2 * cosT2 * sinT3 * sinT3
             + 2 * KXX * cosT2 * cosT3 * sinT2 * sinT3
             + KXX * cosT3 * cosT3 * sinT2 * sinT2
             + KYY * cosT2 * cosT2 * cosT3 * cosT3
             - 2 * KYY * cosT2 * cosT3 * sinT2 * sinT3
             + KYY * sinT2 * sinT2 * sinT3 * sinT3  ;

M[ 0][ 1] = -JXZ * sinT2 + -JYZ * cosT2 + -KXZ * cosT2 * sinT3
             + -KXZ * cosT3 * sinT2 + -KYZ * cosT2 * cosT3
             - 1 * -KYZ * sinT2 * sinT3  ;

M[ 0][ 2] = -KXZ * cosT2 * sinT3 + -KXZ * cosT3 * sinT2
             + -KYZ * cosT2 * cosT3  ;

M[ 1][ 0] = -JXZ * sinT2 + -JYZ * cosT2 + -KXZ * cosT2 * sinT3
             + -KXZ * cosT3 * sinT2 + -KYZ * cosT2 * cosT3
             - 1 * -KYZ * sinT2 * sinT3  ;

M[ 1][ 1] = 0.5*L2 * 0.5*L2 * M2 + 0.5*L3 * 0.5*L3 * M3
             + 2 * 0.5*L3 * A2 * M3 * cosT3
             + A2 * A2 * M3 * cosT3 * cosT3
```

```
            + A2 * A2 * M3 * sinT3 * sinT3 + JZZ + KZZ  ;

M[ 1][ 2] = 0.5*L3 * 0.5*L3 * M3 + 0.5*L3 * A2 * M3 * cosT3 + KZZ  ;

M[ 2][ 0] = -KXZ * cosT2 * sinT3 + -KXZ * cosT3 * sinT2
            + -KYZ * cosT2 * cosT3 - 1 * -KYZ * sinT2 * sinT3  ;

M[ 2][ 1] = 0.5*L3 * 0.5*L3 * M3 + 0.5*L3 * A2 * M3 * cosT3 + KZZ  ;

M[ 2][ 2] = 0.5*L3 * 0.5*L3 * M3 + KZZ  ;

V[ 0] = 2 * -JXY * cosT2 * cosT2 * vel_T1 * vel_T2
        - 2 * -JXY * sinT2 * sinT2 * vel_T1 * vel_T2
        + -JXZ * cosT2 * vel_T2 * vel_T2
        - 1 * -JYZ * sinT2 * vel_T2 * vel_T2
        + 2 * -KXY * cosT2 * cosT2 * cosT3 * cosT3 * vel_T1 * vel_T2
        + 2 * -KXY * cosT2 * cosT2 * cosT3 * cosT3 * vel_T1 * vel_T3
        - 2 * -KXY * cosT2 * cosT2 * sinT3 * sinT3 * vel_T1 * vel_T2
        - 2 * -KXY * cosT2 * cosT2 * sinT3 * sinT3 * vel_T1 * vel_T3
        - 7 * -KXY * cosT2 * cosT3 * sinT2 * sinT3 * vel_T1 * vel_T2
        - 7 * -KXY * cosT2 * cosT3 * sinT2 * sinT3 * vel_T1 * vel_T3
        - 2 * -KXY * cosT3 * cosT3 * sinT2 * sinT2 * vel_T1 * vel_T2
        - 2 * -KXY * cosT3 * cosT3 * sinT2 * sinT2 * vel_T1 * vel_T3
        + -KXY * sinT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T2
        + -KXY * sinT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T3
        - 1 * -KXZ * cosT2 * cosT2 * sinT2 * sinT3 * sinT3 * sinT3 * vel_T1 * vel_T1
        - 2 * -KXZ * cosT2 * cosT3 * sinT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T1
        + -KXZ * cosT2 * cosT3 * vel_T2 * vel_T2
        + 2 * -KXZ * cosT2 * cosT3 * vel_T2 * vel_T3
        + -KXZ * cosT2 * cosT3 * vel_T3 * vel_T3
        - 1 * -KXZ * cosT3 * cosT3 * sinT2 * sinT2 * sinT2 * sinT3 * vel_T1 * vel_T1
        - 1 * -KYZ * cosT2 * cosT2 * cosT3 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T1
        - 1 * -KYZ * cosT2 * cosT3 * cosT3 * sinT2 * sinT2 * sinT3 * vel_T1 * vel_T1
        + -KYZ * cosT2 * sinT2 * sinT2 * sinT3 * sinT3 * sinT3 * vel_T1 * vel_T1
        - 1 * -KYZ * cosT2 * sinT3 * vel_T2 * vel_T2
        - 2 * -KYZ * cosT2 * sinT3 * vel_T2 * vel_T3
        - 1 * -KYZ * cosT2 * sinT3 * vel_T3 * vel_T3
        + -KYZ * cosT3 * sinT2 * sinT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T1
        - 1 * -KYZ * cosT3 * sinT2 * vel_T2 * vel_T2
        - 2 * -KYZ * cosT3 * sinT2 * vel_T2 * vel_T3
        - 1 * -KYZ * cosT3 * sinT2 * vel_T3 * vel_T3
        - 2 * 0.5*L2 * 0.5*L2 * M2 * cosT2 * sinT2 * vel_T1 * vel_T2
```

```
         - 2 * 0.5*L3 * 0.5*L3 * M3 * cosT2 * cosT2 * cosT3 * sinT3 * vel_T1 * vel_T2
         - 2 * 0.5*L3 * 0.5*L3 * M3 * cosT2 * cosT2 * cosT3 * sinT3 * vel_T1 * vel_T3
         - 2 * 0.5*L3 * 0.5*L3 * M3 * cosT2 * cosT3 * cosT3 * sinT2 * vel_T1 * vel_T2
         - 2 * 0.5*L3 * 0.5*L3 * M3 * cosT2 * cosT3 * cosT3 * sinT2 * vel_T1 * vel_T3
         - 2 * 0.5*L3 * A2 * M3 * cosT2 * cosT2 * sinT3 * vel_T1 * vel_T2
         - 2 * 0.5*L3 * A2 * M3 * cosT2 * cosT2 * sinT3 * vel_T1 * vel_T3
         - 4 * 0.5*L3 * A2 * M3 * cosT2 * cosT3 * sinT2 * vel_T1 * vel_T2
         - 2 * 0.5*L3 * A2 * M3 * cosT2 * cosT3 * sinT2 * vel_T1 * vel_T3
         - 2 * A2 * A2 * M3 * cosT2 * sinT2 * vel_T1 * vel_T2
         + 2 * JXX * cosT2 * sinT2 * vel_T1 * vel_T2
         - 2 * JYY * cosT2 * sinT2 * vel_T1 * vel_T2
         + 2 * KXX * cosT2 * cosT2 * cosT3 * sinT3 * vel_T1 * vel_T2
         + 2 * KXX * cosT2 * cosT2 * cosT3 * sinT3 * vel_T1 * vel_T3
         + 2 * KXX * cosT2 * cosT3 * cosT3 * sinT2 * vel_T1 * vel_T2
         + 2 * KXX * cosT2 * cosT3 * cosT3 * sinT2 * vel_T1 * vel_T3
         - 1 * KXX * cosT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T2
         - 1 * KXX * cosT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T3
         - 1 * KXX * cosT3 * sinT2 * sinT2 * sinT3 * vel_T1 * vel_T2
         - 1 * KXX * cosT3 * sinT2 * sinT2 * sinT3 * vel_T1 * vel_T3
         - 2 * KYY * cosT2 * cosT2 * cosT3 * sinT3 * vel_T1 * vel_T2
         - 2 * KYY * cosT2 * cosT2 * cosT3 * sinT3 * vel_T1 * vel_T3
         - 2 * KYY * cosT2 * cosT3 * cosT3 * sinT2 * vel_T1 * vel_T2
         - 2 * KYY * cosT2 * cosT3 * cosT3 * sinT2 * vel_T1 * vel_T3
         + 2 * KYY * cosT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T2
         + 2 * KYY * cosT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T3
         + 2 * KYY * cosT3 * sinT2 * sinT2 * sinT3 * vel_T1 * vel_T2
         + 2 * KYY * cosT3 * sinT2 * sinT2 * sinT3 * vel_T1 * vel_T3
         - 1 * KZZ * cosT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T2
         - 1 * KZZ * cosT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T3
         - 1 * KZZ * cosT3 * sinT2 * sinT2 * sinT3 * vel_T1 * vel_T2
         - 1 * KZZ * cosT3 * sinT2 * sinT2 * sinT3 * vel_T1 * vel_T3  ;

V[ 1] = -1 * -JXY * cosT2 * cosT2 * vel_T1 * vel_T1
         + -JXY * sinT2 * sinT2 * vel_T1 * vel_T1
         - 1 * -KXY * cosT2 * cosT2 * cosT3 * cosT3 * vel_T1 * vel_T1
         + -KXY * cosT2 * cosT2 * sinT3 * sinT3 * vel_T1 * vel_T1
         + 4 * -KXY * cosT2 * cosT3 * sinT2 * sinT3 * vel_T1 * vel_T1
         + -KXY * cosT3 * cosT3 * sinT2 * sinT2 * vel_T1 * vel_T1
         - 1 * -KXY * sinT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T1
         + 0.5*L2 * 0.5*L2 * M2 * cosT2 * sinT2 * vel_T1 * vel_T1
         + 0.5*L3 * 0.5*L3 * M3 * cosT2 * cosT2 * cosT3 * sinT3 * vel_T1 * vel_T1
         + 0.5*L3 * 0.5*L3 * M3 * cosT2 * cosT3 * cosT3 * sinT2 * vel_T1 * vel_T1
```

```
         - 1 * 0.5*L3 * 0.5*L3 * M3 * cosT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T1
         - 1 * 0.5*L3 * 0.5*L3 * M3 * cosT3 * sinT2 * sinT2 * sinT3 * vel_T1 * vel_T1
         + 0.5*L3 * A2 * M3 * cosT2 * cosT2 * sinT3 * vel_T1 * vel_T1
         + 0.5*L3 * A2 * M3 * cosT2 * cosT3 * cosT3 * cosT3 * sinT2 * vel_T1 * vel_T1
         + 0.5*L3 * A2 * M3 * cosT2 * cosT3 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T1
         + 0.5*L3 * A2 * M3 * cosT2 * cosT3 * sinT2 * vel_T1 * vel_T1
         - 1 * 0.5*L3 * A2 * M3 * cosT3 * cosT3 * sinT2 * sinT2 * sinT3 * vel_T1 * vel_T1
         - 1 * 0.5*L3 * A2 * M3 * sinT2 * sinT2 * sinT3 * sinT3 * sinT3 * vel_T1 * vel_T1
         - 2 * 0.5*L3 * A2 * M3 * sinT3 * vel_T2 * vel_T3
         - 1 * 0.5*L3 * A2 * M3 * sinT3 * vel_T3 * vel_T3
         + A2 * A2 * M3 * cosT2 * cosT3 * cosT3 * sinT2 * vel_T1 * vel_T1
         + A2 * A2 * M3 * cosT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T1
         - 1 * JXX * cosT2 * sinT2 * vel_T1 * vel_T1
         + JYY * cosT2 * sinT2 * vel_T1 * vel_T1
         - 1 * KXX * cosT2 * cosT2 * cosT3 * sinT3 * vel_T1 * vel_T1
         - 1 * KXX * cosT2 * cosT3 * cosT3 * sinT2 * vel_T1 * vel_T1
         + KXX * cosT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T1
         + KXX * cosT3 * sinT2 * sinT2 * sinT3 * vel_T1 * vel_T1
         + KYY * cosT2 * cosT2 * cosT3 * sinT3 * vel_T1 * vel_T1
         + KYY * cosT2 * cosT3 * cosT3 * sinT2 * vel_T1 * vel_T1
         - 1 * KYY * cosT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T1
         - 1 * KYY * cosT3 * sinT2 * sinT2 * sinT3 * vel_T1 * vel_T1  ;

V[ 2] = -1 * -KXY * cosT2 * cosT2 * cosT3 * cosT3 * vel_T1 * vel_T1
         + -KXY * cosT2 * cosT2 * sinT3 * sinT3 * vel_T1 * vel_T1
         + 4 * -KXY * cosT2 * cosT3 * sinT2 * sinT3 * vel_T1 * vel_T1
         + -KXY * cosT3 * cosT3 * sinT2 * sinT2 * vel_T1 * vel_T1
         - 1 * -KXY * sinT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T1
         + 0.5*L3 * 0.5*L3 * M3 * cosT2 * cosT2 * cosT3 * sinT3 * vel_T1 * vel_T1
         + 0.5*L3 * 0.5*L3 * M3 * cosT2 * cosT3 * cosT3 * sinT2 * vel_T1 * vel_T1
         - 1 * 0.5*L3 * 0.5*L3 * M3 * cosT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T1
         - 1 * 0.5*L3 * 0.5*L3 * M3 * cosT3 * sinT2 * sinT2 * sinT3 * vel_T1 * vel_T1
         + 0.5*L3 * A2 * M3 * cosT2 * cosT2 * sinT3 * vel_T1 * vel_T1
         + 0.5*L3 * A2 * M3 * cosT2 * cosT3 * sinT2 * vel_T1 * vel_T1
         + 0.5*L3 * A2 * M3 * sinT3 * vel_T2 * vel_T2
         - 1 * KXX * cosT2 * cosT2 * cosT3 * sinT3 * vel_T1 * vel_T1
         - 1 * KXX * cosT2 * cosT3 * cosT3 * sinT2 * vel_T1 * vel_T1
         + KXX * cosT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T1
         + KXX * cosT3 * sinT2 * sinT2 * sinT3 * vel_T1 * vel_T1
         + KYY * cosT2 * cosT2 * cosT3 * sinT3 * vel_T1 * vel_T1
         + KYY * cosT2 * cosT3 * cosT3 * sinT2 * vel_T1 * vel_T1
         - 1 * KYY * cosT2 * sinT2 * sinT3 * sinT3 * vel_T1 * vel_T1
```

```
          - 1 * KYY * cosT3 * sinT2 * sinT2 * sinT3 * vel_T1 * vel_T1  ;

  G[ 0] = 0.5*L2 * GRAVITY * M2 * cosT1 * cosT2
          + 0.5*L3 * GRAVITY * M3 * cosT1 * cosT2 * cosT3
          + A2 * GRAVITY * M3 * cosT1 * cosT2  ;

  G[ 1] = -1 * 0.5*L2 * GRAVITY * M2 * sinT1 * sinT2
          - 1 * 0.5*L3 * GRAVITY * M3 * cosT3 * sinT1 * sinT2
          + A2 * GRAVITY * M3 * cosT2 * cosT3 * sinT1 * sinT3
          - 1 * A2 * GRAVITY * M3 * cosT3 * cosT3 * sinT1 * sinT2
          - 1 * A2 * GRAVITY * M3 * sinT1 * sinT2 * sinT3 * sinT3  ;

  G[ 2] = -1 * 0.5*L3 * GRAVITY * M3 * cosT3 * sinT1 * sinT2  ;

/* Torque due to external moments */

  F[ 0] = cosT2 * cosT3 * external_moment_y
          + cosT2 * external_moment_x * sinT3
          + cosT3 * external_moment_x * sinT2  ;

  F[ 1] = external_moment_z  ;

  F[ 2] = external_moment_z  ;

/* Torque due to external forces */

  F[ 0] += -1 * A2 * cosT2 * external_force_z
          - 1 * A3 * cosT2 * cosT3 * external_force_z  ;

  F[ 1] += A2 * cosT3 * external_force_y + A2 * external_force_x * sinT3
          + A3 * external_force_y  ;

  F[ 2] += A3 * external_force_y  ;

/* Torque due to base movement */

  F[ 0] += 0.5*L2 * M2 * base_y * cosT1 * cosT2
          + 0.5*L3 * M3 * base_y * cosT1 * cosT2 * cosT3
          + A2 * M3 * base_y * cosT1 * cosT2  ;

  F[ 1] += -1 * 0.5*L2 * M2 * base_x * cosT1 * sinT2
          - 1 * 0.5*L2 * M2 * base_y * sinT1 * sinT2
```

```
            + 0.5*L2 * M2 * base_z * cosT2
            - 1 * 0.5*L3 * M3 * base_x * cosT1 * cosT3 * sinT2
            - 1 * 0.5*L3 * M3 * base_y * cosT3 * sinT1 * sinT2
            + 0.5*L3 * M3 * base_z * cosT2 * cosT3
            + A2 * M3 * base_x * cosT1 * cosT2 * cosT3 * sinT3
            - 1 * A2 * M3 * base_x * cosT1 * cosT3 * cosT3 * sinT2
            - 1 * A2 * M3 * base_x * cosT1 * sinT2 * sinT3 * sinT3
            + A2 * M3 * base_y * cosT2 * cosT3 * sinT1 * sinT3
            - 1 * A2 * M3 * base_y * cosT3 * cosT3 * sinT1 * sinT2
            - 1 * A2 * M3 * base_y * sinT1 * sinT2 * sinT3 * sinT3
            + A2 * M3 * base_z * cosT2 * cosT3 * cosT3
            + A2 * M3 * base_z * cosT2 * sinT3 * sinT3
            + A2 * M3 * base_z * cosT3 * sinT2 * sinT3  ;

  F[ 2] += -1 * 0.5*L3 * M3 * base_x * cosT1 * cosT3 * sinT2
            - 1 * 0.5*L3 * M3 * base_y * cosT3 * sinT1 * sinT2
            + 0.5*L3 * M3 * base_z * cosT2 * cosT3  ;

/* Force due to friction forces */

  F[ 0] += 0.0  ;
  F[ 1] += 0.0  ;
  F[ 2] += 0.0  ;
}


void dyn3Frm (TransformList, J_pos)
double ***TransformList, *J_pos;

{
  double T1 = J_pos[ 0];
  double T2 = J_pos[ 1];
  double T3 = J_pos[ 2];
  double sinT1 = sin(T1);
  double cosT1 = cos(T1);
  double sinT2 = sin(T2);
  double cosT2 = cos(T2);
  double sinT3 = sin(T3);
  double cosT3 = cos(T3);
  double **T;

  T = TransformList[ 0];
```

```
T[ 0][ 0] = cosT1  ;
T[ 0][ 1] = -1 * sinT1  ;
T[ 0][ 2] = 0  ;
T[ 0][ 3] = 0  ;
T[ 1][ 0] = sinT1  ;
T[ 1][ 1] = cosT1  ;
T[ 1][ 2] = 0  ;
T[ 1][ 3] = 0  ;
T[ 2][ 0] = 0  ;
T[ 2][ 1] = 0  ;
T[ 2][ 2] = 1  ;
T[ 2][ 3] = 0  ;
T[ 3][ 0] = 0  ;
T[ 3][ 1] = 0  ;
T[ 3][ 2] = 0  ;
T[ 3][ 3] = 1  ;

T = TransformList[ 1];
T[ 0][ 0] = cosT1 * cosT2  ;
T[ 0][ 1] = -1 * cosT1 * sinT2  ;
T[ 0][ 2] = sinT1  ;
T[ 0][ 3] = 0  ;
T[ 1][ 0] = cosT2 * sinT1  ;
T[ 1][ 1] = -1 * sinT1 * sinT2  ;
T[ 1][ 2] = -1.0 * cosT1  ;
T[ 1][ 3] = 0  ;
T[ 2][ 0] = sinT2  ;
T[ 2][ 1] = cosT2  ;
T[ 2][ 2] = 0  ;
T[ 2][ 3] = 0  ;
T[ 3][ 0] = 0  ;
T[ 3][ 1] = 0  ;
T[ 3][ 2] = 0  ;
T[ 3][ 3] = 1  ;

T = TransformList[ 2];
T[ 0][ 0] = cosT1 * cosT2 * cosT3 - 1 * cosT1 * sinT2 * sinT3  ;
T[ 0][ 1] = -1 * cosT1 * cosT2 * sinT3 - 1 * cosT1 * cosT3 * sinT2  ;
T[ 0][ 2] = sinT1  ;
T[ 0][ 3] = A2 * cosT1 * cosT2  ;
T[ 1][ 0] = cosT2 * cosT3 * sinT1 - 1 * sinT1 * sinT2 * sinT3  ;
T[ 1][ 1] = -1 * cosT2 * sinT1 * sinT3 - 1 * cosT3 * sinT1 * sinT2  ;
```

```
  T[ 1][ 2] = -1.0 * cosT1  ;
  T[ 1][ 3] = A2 * cosT2 * sinT1  ;
  T[ 2][ 0] = cosT3 * sinT2 + cosT2 * sinT3  ;
  T[ 2][ 1] = -1 * sinT2 * sinT3 + cosT2 * cosT3  ;
  T[ 2][ 2] = 0  ;
  T[ 2][ 3] = A2 * sinT2  ;
  T[ 3][ 0] = 0  ;
  T[ 3][ 1] = 0  ;
  T[ 3][ 2] = 0  ;
  T[ 3][ 3] = 1  ;

  T = TransformList[ 3];
  T[ 0][ 0] = cosT1 * cosT2 * cosT3 - 1 * cosT1 * sinT2 * sinT3  ;
  T[ 0][ 1] = -1 * cosT1 * cosT2 * sinT3 - 1 * cosT1 * cosT3 * sinT2  ;
  T[ 0][ 2] = sinT1  ;
  T[ 0][ 3] = A3 * cosT1 * cosT2 * cosT3 - 1 * A3 * cosT1 * sinT2 * sinT3
              + A2 * cosT1 * cosT2  ;

  T[ 1][ 0] = cosT2 * cosT3 * sinT1 - 1 * sinT1 * sinT2 * sinT3  ;
  T[ 1][ 1] = -1 * cosT2 * sinT1 * sinT3 - 1 * cosT3 * sinT1 * sinT2  ;
  T[ 1][ 2] = -1.0 * cosT1  ;
  T[ 1][ 3] = A3 * cosT2 * cosT3 * sinT1 - 1 * A3 * sinT1 * sinT2 * sinT3
              + A2 * cosT2 * sinT1  ;

  T[ 2][ 0] = cosT3 * sinT2 + cosT2 * sinT3  ;
  T[ 2][ 1] = -1 * sinT2 * sinT3 + cosT2 * cosT3  ;
  T[ 2][ 2] = 0  ;
  T[ 2][ 3] = A3 * cosT3 * sinT2 + A3 * cosT2 * sinT3 + A2 * sinT2  ;

  T[ 3][ 0] = 0  ;
  T[ 3][ 1] = 0  ;
  T[ 3][ 2] = 0  ;
  T[ 3][ 3] = 1  ;

}
```

11 Appendix B

The following are the simplified dynamics using *Mathematica* and also using some manual simplifications for the trig functions.

```
/* Simplified dynamics equations for the first model */


M[1][1] =  2.*K3 + A2*K3 + K4 + 2*A2*K4 + K4*D3 + IZZ - 0.5*K4*L3 +
           JYY*s2*s2 + KZZ*s2*s2 + 0.25*L2*L2*m2*s2*s2 -
           A2*A2*m3*s2*s2 + JXX*c2*c2 + KXX c2*c2  -
           L3*m3*c2 + JXY*sin(2*T2) + 0.5*KXZ*sin(2*T2) -
           0.5*D3*m3*sin(2*T2) + 0.25*L3*m3*sin(2*T2) ;

M[1][2] = -1. + JYZ*s2 + KYZ*s2 - JXZ*c2 - KXY*c2 ;

M[1][3] = 0 ;

M[2][1] =  -1. + JYZ*s2 + KYZ*s2 - JXZ*c2 - KXY*c2 ;

M[2][2] = 2.*K3 + K4 + K4*D3 + JZZ + KYY - 0.5*K4*L3 + 0.25*L2*m2 + A2*m3 - L3*m3 ;

M[2][3] = - A2*m3 ;

M[3][1] = 0 ;

M[3][2] = -A2*m3 ;

M[3][3] = m3 ;

=============================================================================

V[1] = -1. + 3.*K3 + 2.A2*K3 + 4.*K4 + K4*D3 - 0.5*K4*L3 -
       0.25*L3*m3*vel_D3*vel_T1 +
       0.25*L3*m3*cos(180+2*T2)*vel_D3*vel_T1 -
       JXY*vel_T1*vel_T2 - KXZ*vel_T1*vel_T2 +
       0.5*D3*m3*vel_T1*vel_T2 - 0.5*L3*m3*vel_T1*vel_T2 +
       0.5*D3*m3*cos(180+2*T2)*vel_T1*vel_T2 +
       JXZ*s2*vel_T2*vel_T2 + KXY*s2*vel_T2*vel_T2 -
       JYZ*vel_T2*vel_T2*c2 - KYZ*vel_T2*vel_T2*c2 +
       A2*m3*vel_D3*vel_T1*sin(180+2*T2) +
       JXX*vel_T1*vel_T2*sin(180+2*T2) -
```

```
       JYY*vel_T1*vel_T2*sin(180+2*T2) +
       KXX*vel_T1*vel_T2*sin(180+2*T2) -
       KZZ*vel_T1*vel_T2*sin(180+2*T2) -
       0.25*L2*L2*m2*vel_T1*vel_T2*sin(180+2*T2) -
       A2*A2*m3*vel_T1*vel_T2*sin(180+2*T2) -
       0.5*L3*m3*vel_T1*vel_T2*sin(180+2*T2) ;

V[2] = -2. - K3 + 2.*A2*K3 + K4 + K4*D3 - 0.5 K4*L3 +
       0.5*L3*m3*s2*vel_D3*vel_T1 - JXY*vel_T1*vel_T1
       KXZ*vel_T1*vel_T1 + 0.5*D3*m3*vel_T1*vel_T1 -
       0.5*L3*m3*vel_T1*vel_T1  -
       0.5*D3*m3*cos(180+2*T2)*vel_T1*vel_T1 + D3*m3*vel_T2*vel_T2 -
       0.5*JXX*vel_T1*vel_T1*sin(180+2*T2) + 0.5*JYY*vel_T1*vel_T1*sin(180+2 T2)
       0.5*KXX*vel_T1*vel_T1*sin(180+2*T2) + 0.5*KZZ*vel_T1*vel_T1*sin(180+2*T2)
       0.125*L2*L2*m2*vel_T1*vel_T1*sin(180+2*T2) +
       0.5*A2*A2*m3*vel_T1*vel_T1*sin(180+2*T2) -
       0.5*L3*m3*vel_T1*vel_T1*sin(180+2*T2) ;

V[3] = -2.*K3 - 2.*K4 + D3*m3*vel_T2*vel_T2 - 0.5*L3*m3*vel_T2*vel_T2 +
       D3*m3*vel_T1*vel_T1*c2*c2  -
       0.5*L3*m3*vel_T1*vel_T1*c2*c2 -
       0.5*A2*m3*vel_T1*vel_T1*sin(180+2*T2) ;

============================================================================

G[1] = -2.*K3 - 2.*K4 + D3*m3*vel_T2*vel_T2 - 0.5*L3*m3*vel_T2*vel_T2 +
       D3*m3*vel_T1*vel_T1*c2*c2 -
       0.5*L3*m3*vel_T1*vel_T1*c2*c2 -
       0.5*A2*m3*vel_T1*vel_T1*sin(180+2*T2) ;

G[2] = K4 + 0.5*g*L3*m3*s2*s1 -
       0.5*g*L2*m2*s1*c2 - A2*g*m3*s1*c2 ;

G[3] =  g*m3*s1*c2 ;

=============================================================================

/* Simplified dynamics equations for the second model */

M[1][1] =  4. + 3.*C4 - 2.*A2*C4 + IZZ - 0.5*C4*L3 +
           JYY*cosT2*cosT2  + KZZ*cosT2*cosT2  + 0.25*L2*L2*m2*cosT2*cosT2  +
           A2*A2*m3*cosT2*cosT2 + JXX*sinT2*sinT2 + KXX*sinT2*sinT2  +
```

```
          D3*D3*m3*sinT2*sinT2 - 0.5*L3*m3*sinT2*sinT2 - 0.5*D3*L3*m3*sinT2*sinT2 -
          0.5*JXY*sin2T2 - 0.5*KXZ*sin2T2 - A2*D3*m3*sin2T2 -
          0.25*L3*m3*sin2T2) ;

M[1][2] =  -2. - JYZ*cosT2 - KYZ*cosT2 - JXZ*sinT2 - KXY*sinT2 ;

M[1][3] = 0 ;

M[2][1] = -2. - JYZ*cosT2 - KYZ*cosT2 - JXZ*sinT2 - KXY*sinT2 ;

M[2][2] =  0.5*(6.*C4 + 2.*JZZ + 2.*KYY - C4*L3 + 0.5*L2*m2 + 2.*A2*m3 +
          2.*D3*D3*m3 - L3*m3 - D3*L3*m3) ;

M[2][3] = A2*m3 ;

M[3][1] = 0 ;

M[3][2] = A2*m3 ;

M[3][3] = m3 ;

===========================================================================

V[1] = 0.5*(-4. + 16.*C4 - C4*L3 + 2.*D3*m3*(vel_D3)*(vel_T1) -
       0.5*L3*m3*(vel_D3)*(vel_T1) - 2.*D3*m3*cos2T2*(vel_D3)*(vel_T1) +
       0.5*L3*m3*cos2T2*(vel_D3)*(vel_T1) - 2.*JXY*(vel_T1)*(vel_T2) -
       2.*KXZ*(vel_T1)*(vel_T2) - L3*m3*(vel_T1)*(vel_T2) -
       4.*A2*D3*m3*cos2T2*(vel_T1)*(vel_T2) - 2.*JXZ*cosT2*(vel_T2)*(vel_T2)  -
       2.*KXY*cosT2*(vel_T2)*(vel_T2) - 2.*JYZ*(vel_T2)*(vel_T2)*sinT2 -
       2.*KYZ*(vel_T2)*(vel_T2)*sinT2 - 2.*A2*m3*(vel_D3)*(vel_T1)*sin2T2 +
       2.*JXX*(vel_T1)*(vel_T2)*sin2T2 -
       2.*JYY*(vel_T1)*(vel_T2)*sin2T2 +
       2.*KXX*(vel_T1)*(vel_T2)*sin2T2 -
       2.*KZZ*(vel_T1)*(vel_T2)*sin2T2 -
       0.5*L2*L2*m2*(vel_T1)*(vel_T2)*sin2T2 -
       2.*A2*A2*m3*(vel_T1)*(vel_T2)*sin2T2 +
       2.*D3*D3*m3*(vel_T1)*(vel_T2)*sin2T2 -
       0.5*L3*m3*(vel_T1)*(vel_T2)*sin2T2 -
       0.5*D3*L3*m3*(vel_T1)*(vel_T2)*sin2T2) ;

V[2] = 0.5*(-4. - 10.*C4 - C4*L3 - 4.*D3*m3*cosT2*(vel_D3)*(vel_T1) -
       L3*m3*cosT2*(vel_D3)*(vel_T1) - 2.*JXY*(vel_T1)*(vel_T1) -
```

```
       2.*KXZ*(vel_T1)*(vel_T1) - L3*m3*(vel_T1)*(vel_T1) +
       2.*A2*D3*m3*cos2T2*(vel_T1)*(vel_T1) - JXX*(vel_T1)*(vel_T1)*sin2T2 +
       JYY*(vel_T1)*(vel_T1)*sin2T2 - KXX*(vel_T1)*(vel_T1)*sin2T2 +
       KZZ*(vel_T1)*(vel_T1)*sin2T2 + 0.25*L2*L2*m2*(vel_T1)*(vel_T1)*sin2T2 +
       A2*A2*m3*(vel_T1)*sin2T2 - D3*D3*m3*(vel_T1)*(vel_T1)*sin2T2 -
       0.5*L3*m3*(vel_T1)*(vel_T1)*sin2T2 -
       0.5*D3*L3*m3*(vel_T1)*(vel_T1)*sin2T2) ;

V[3] = 2.*(-C4 - 0.5*D3*m3*(vel_T2)*(vel_T2) -
       0.25*L3*m3*(vel_T2)*(vel_T2) -
       0.5*D3*m3*(vel_T1)*(vel_T1)*sinT2*sinT2  -
       0.25*L3*m3*(vel_T1)*(vel_T1)*sinT2*sinT2 +
       0.25*A2*m3*(vel_T1)*(vel_T1)*sin2T2) ;

=============================================================================

G[1] = -C4 + 0.5*g*L2*m2*cosT1*cosT2 +
       A2*g*m3*cosT1*cosT2 - D3*g*m3*cosT1*sinT2 -
       0.5*g*L3*m3*cosT1*sinT2 ;

G[2] =  -C4 - D3*g*m3*cosT2*sinT1 -
       0.5*g*L3*m3*cosT2*sinT1 -
       0.5*g*L2*m2*sinT1*sinT2 - A2*g*m3*sinT1*sinT2 ;

G[3] =  -g*m3*sinT1*sinT2 ;

=========================================================================

/* Simplified dynamics equations for the third model */

M[1][1] =  2 + IZZ + (JXX+JYY+KXX+KYY)/2.0 + 0.125*L2*L2*m2 +
           A2*A2*m3/2.0 + 0.0625*L3*L3*m3 -
           (JXX*cos2T2+JYY*cos2T2)/2.0 + 0.125*L2*L2*m2*cos2T2 +
           A2*A2*m3*cos2T2/2.0 + 0.0625*L3*L3*m3*cos2T2 +
           0.125*A2*L3*m3*cos2T2mT3 + 0.5*A2*L3*m3*cosT3 +
           0.0625*L3*L3*m3*cos2T3 - (KXX cos2_T2pT3 +
           KYY cos2_T2pT3)/2.0 + 0.0625*L3*L3*m3*cos2_T2pT3 +
           0.375*A2*L3*m3*cos2T2pT3 -
           (JXY*sin2T2 - KXY*sin2T2 - KXY*sin2T3)/2.0 ;

M[1][2] =  -1 - JYZ*cosT2 - KYZ*cosT2mT3 - JXZ*sinT2 - KXZ*sinT2pT3 ;
```

```
M[1][3] = -(KYZ*cosT2*cosT3) - KXZ*cosT3*sinT2 - KXZ*cosT2*sinT3 ;

M[2][1] = -1 - JYZ*cosT2 - KYZ*cosT2mT3 - JXZ*sinT2 - KXZ*sinT2pT3

M[2][2] = JZZ + KZZ + 0.25*L2*L2*m2 + A2*A2*m3 + 0.25*L3*L3*m3 + *A2*L3*m3*cosT3 ;

M[2][3] = KZZ + 0.25*L3*L3*m3 + 0.5*A2*L3*m3*cosT3 ;

M[3][1] = -1 - KYZ*cosT2mT3 - KXZ*sinT2pT3 ;

M[3][2] = JZZ + KZZ + 0.25*L2*L2*m2 + A2*A2*m3 + 0.25*L3*L3*m3 + A2*L3*m3*cosT3 ;

M[3][3] = KZZ + 0.25*L3*L3*m3 ;

/* ============================================================================ */


V[1] = (-992 - KXZ*cosT2m3T3*theta_dot[1]*theta_dot[1] -
       7*KXZ*cosT2mT3*theta_dot[1]*theta_dot[1] +
       KXZ*cos3_T2mT3*theta_dot[1]*theta_dot[1] -
       KXZ*cos3T2mT3*theta_dot[1]*theta_dot[1] +
       5*KXZ*cosT2pT3*theta_dot[1]*theta_dot[1] -
       3*KXZ*cos3_T2pT3*theta_dot[1]*theta_dot[1] +
       3*KXZ*cos3T3pT3*theta_dot[1]*theta_dot[1]  +
       3*KXZ*cosT2p3T3*theta_dot[1]*theta_dot[1] -
       32*JXY*theta_dot[1]*theta_dot[2] -
       32*KXY*theta_dot[1]*theta_dot[2] -
       4*KXY*cos2_T2mT3*theta_dot[1]*theta_dot[2] +
       4*KXY*cos2_T2pT3*theta_dot[1]*theta_dot[2] -
       32*JXZ*cosT2*theta_dot[2]*theta_dot[2]  -
       16*KXZ*cosT2mT3*theta_dot[2]*theta_dot[2] -
       16*KXZ*cosT2pT3*theta_dot[2]*theta_dot[2]  -
       32*KXY*theta_dot[1]*theta_dot[3] -
       4*KXY*cos2_T2mT3*theta_dot[1]*theta_dot[3] +
       4*KXY*cos2_T2pT3*theta_dot[1]*theta_dot[3] -
       16*KXZ*cosT2mT3*theta_dot[2]*theta_dot[3] -
       16*KXZ*cosT2pT3*theta_dot[2]*theta_dot[3] -
       16*KXZ*cosT2mT3*theta_dot[3]*theta_dot[3] -
       16*KXZ*cosT2pT3*theta_dot[3]*theta_dot[3] -
       32*JYZ*theta_dot[2]*theta_dot[2]*sinT2 +
       32*JXX*theta_dot[1]*theta_dot[2]*sin2T2 -
       32*JYY*theta_dot[1]*theta_dot[2]*sin2T2 +
```

```
        8*KXX*theta_dot[1]*theta_dot[2]*sin2T2 -
        8*KZZ*theta_dot[1]*theta_dot[2]*sin2T2 -
        8.*L2*L2*m2*theta_dot[1]*theta_dot[2]*sin2T2 -
        32*A2*A2*m3*theta_dot[1]*theta_dot[2]*sin2T2 -
        4.*L3*L3*m3*theta_dot[1]*theta_dot[2]*sin2T2 +
        8*KXX*theta_dot[1]*theta_dot[3]*sin2T2 -
        8*KZZ*theta_dot[1]*theta_dot[3]*sin2T2 -
        4.*L3*L3*m3*theta_dot[1]*theta_dot[3]*sin2T2 +
        4*KYZ*theta_dot[1]*theta_dot[1]*sinT2m3T3 -
        8.*A2*L3*m3*theta_dot[1]*theta_dot[2]*sin2T2mT3 -
        4*KYZ*theta_dot[1]*theta_dot[1]*sin3T2mT3 -
        16.*A2*L3*m3*theta_dot[1]*theta_dot[2]*sinT3 -
        16.*A2*L3*m3*theta_dot[1]*theta_dot[3]*sinT3 +
        8*KXX*theta_dot[1]*theta_dot[2]*sin2T3 -
        8*KZZ*theta_dot[1]*theta_dot[2]*sin2T3 -
        4.*L3*L3*m3*theta_dot[1]*theta_dot[2]*sin2T3 +
        8*KXX*theta_dot[1]*theta_dot[3]*sin2T3 -
        8*KZZ*theta_dot[1]*theta_dot[3]*sin2T3 -
        4.*L3*L3*m3*theta_dot[1]*theta_dot[3]*sin2T3 -
        8*KYZ*theta_dot[1]*theta_dot[1]*sinT2pT3 -
        32*KYZ*theta_dot[2]*theta_dot[2]*sinT2pT3 -
        32*KYZ*theta_dot[2]*theta_dot[3]*sinT2pT3 -
        32*KYZ*theta_dot[3]*theta_dot[3]*sinT2pT3 +
        24*KXX*theta_dot[1]*theta_dot[2]*sin2_T2pT3 -
        32*KYY*theta_dot[1]*theta_dot[2]*sin2_T2pT3 +
        8*KZZ*theta_dot[1]*theta_dot[2]*sin2_T2pT3 -
        4.*L3*L3*m3*theta_dot[1]*theta_dot[2]*sin2_T2pT3 +
        24*KXX*theta_dot[1]*theta_dot[3]*sin2_T2pT3 -
        32*KYY*theta_dot[1]*theta_dot[3]*sin2_T2pT3 +
        8*KZZ*theta_dot[1]*theta_dot[3]*sin2_T2pT3 -
        4.*L3*L3*m3*theta_dot[1]*theta_dot[3]*sin2_T2pT3 -
        24.*A2*L3*m3*theta_dot[1]*theta_dot[2]*sin2T2pT3 -
        16.*A2*L3*m3*theta_dot[1]*theta_dot[3]*sin2T2pT3 +
        4*KYZ*theta_dot[1]*theta_dot[1]*sin3T3pT3 +
        4*KYZ*theta_dot[1]*theta_dot[1]*sinT2p3T3) / 32 ;

V[2] = 1 - JXY*theta_dot[1]*theta_dot[1] -
       KXY*theta_dot[1]*theta_dot[1] -
       (KXY*cos2_T2mT3*theta_dot[1]*theta_dot[1])/8 +
       KXY*cos2_T2pT3*theta_dot[1]*theta_dot[1]/8 -
       JXX*theta_dot[1]*theta_dot[1]*sin2T2/2
       JYY*theta_dot[1]*theta_dot[1]*sin2T2/2 +
```

```
      0.125 L2*L2  m2 (theta_dot[1]*theta_dot[1]*sin2T2 +
      A2*A2*m3*theta_dot[1]*theta_dot[1]*sin2T2/2 -
      A2*L3*m3*theta_dot[2]*theta_dot[3]*sinT3 -
      0.5*A2*L3*m3*theta_dot[3]*theta_dot[3]*sinT3 -
      KXX*theta_dot[1]*theta_dot[1]*sin2_T2pT3/2 +
      KYY*theta_dot[1]*theta_dot[1]*sin2_T2pT3/2 +
      0.125*L3*L3*m3*theta_dot[1]*theta_dot[1]*sin2_T2pT3 +
      0.5*A2*L3*m3*theta_dot[1]*theta_dot[1]*sin2T2pT3 ;

V[3] = (16 - 8*KXY*theta_dot[1]*theta_dot[1] -
       KXY*cos2_T2mT3*theta_dot[1]*theta_dot[1]  +
       KXY*cos2_T2pT3*theta_dot[1]*theta_dot[1] +
       2.*A2*L3*m3*theta_dot[1]*theta_dot[1]*sinT3 +
       4.*A2*L3*m3*theta_dot[2]*theta_dot[2]*sinT3 -
       4*KXX*theta_dot[1]*theta_dot[1]*sin2_T2pT3 +
       4*KYY*theta_dot[1]*theta_dot[1]*sin2_T2pT3 +
       L3*L3*m3*theta_dot[1]*theta_dot[1]*sin2_T2pT3 +
       2.*A2*L3*m3*theta_dot[1]*theta_dot[1]*sin2T2pT3) / 8 ;

============================================================================

G[1] = 0.5*(g*L2*m2*cosT1*cosT2 +
      2.*A2*g*m3*cosT1*cosT2 +
      g*L3*m3*cosT1*cosT2*cosT3) ;

G[2] = 0.5*(-1. g*L2*m2*sinT1*sinT2 -
      g*L3*m3*cosT3*sinT1*sinT2 -
      2.*A2*g*m3*cosT3*cosT3*sinT1*sinT2 -
      2.*A2*g*m3*sinT1*sinT2*sinT3*sinT3 +
      A2*g*m3*cosT2*sinT1*sin2T3) ;

G[3] = -0.5*g*L3*m3*cosT3*sinT1*sinT2 ;

==============================================================================
```