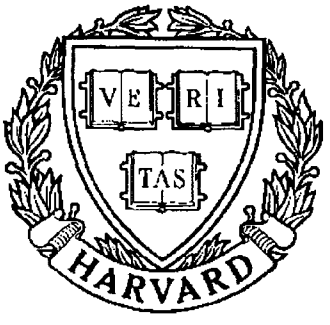
THESIS REPORT
Master's Degree

S Y S T E M S
R E S E A R C H
C E N T E R



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), the University of Maryland, Harvard University, and Industry

Object-Oriented Manipulator Design Exerciser

by X. Chen
Advisor: P.S. Krishnaprasad

M.S. 87-5
Formerly TR 87-161

OBJECT-ORIENTED MANIPULATOR DESIGN EXERCISER (OOMANEX)

by

Xin Chen

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
1987

Advisory Committee:

Professor P. S. Krishnaprasad
Associate Professor Joseph JaJa
Associate Professor Andre Tits

ACKNOWLEDGEMENTS

I wish to thank Prof. P. S. Krishnaprasad for his guidance, encouragement and support during my research work related to this thesis.

I acknowledge the support of the System Research Center through a Research Fellowship and Research Assistantship.

I also wish to thank Dr. Michael K. H. Fan for his help and encouragement during the development of the optimization method and its application in OOMANEX. With pleasure I acknowledge the help, critiques and suggestions I received from my friends Zidu Ma and Chii-ren Tsai during numerous discussions. I also appreciate the help from Dr. Narasingarao Sreenath.

My warmest thanks and gratitude go to my husband, Yuangeng Huang, for his love, inspiration and encouragement.

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER
ONE

Introduction

The design of a robotic manipulator system involves complex computations for solving nonlinear, coupled dynamical equations. Our goal to resolve the design difficulties was to have a general program package combining the following capabilities:

(i.) a high-level input environment with computer aided-design(CAD) features for constructing, and modifying models of a robot manipulator and its environment;

(ii.) automatic formulation of kinematic and dynamical equations;

(iii.) dynamical joint torque (force) calculation;

(iv.) an interactive environment for simulation, and task planning; a high-level language for robot programming, and an interactive editor for program debugging.

A program with the above capabilities would be attractive because it would allow the user to write and debug robot programs and to plan, simulate, and modify robot manipulator designs in a single programming environment.

The Object-oriented Manipulator Design Exerciser (OOMANEX), a software package for modeling, design and optimization of a robotic manipulator system has been developed to realize the above goals. One main purpose of

OOMANEX is to demonstrate the possibilities of using object-oriented programming (OOP) technique in the process of developing a tool for robotic manipulator design. From the system design point of view, object-oriented programming helps the designer understand the system structure, making full use of already existing software packages for the design purpose. It also couples symbolic computation and numerical computation , which are necessary for modeling the robot manipulator and forms the basis for simulation and optimization of the designed robot manipulator system. Not only does OOMANEX offer the possibility of automating the design procedure to a certain extent, but also helps realize the goal of designing robotic manipulators on the basis of their exact dynamic equations. Utilizing the dynamic graphics capabilities and programming environment provided by the Symbolics 3600 system, OOMANEX allows the simulated robot manipulator to be smoothly animated on the terminal screen, informs the user on the screen in an explanatory way, either by text or by graphics, and exploits its interactive capabilities by taking advantage of a window system, mouse button and menu box. As a final result, OOMANEX presents an intuitive and friendly user program interface.

One feature of OOMANEX is its integration of multiple software tools. OOMANEX provides a way to solve the robot manipulator design problem by using different software packages (e.g., DYNAMAN, CONSOLE, etc.) originated on different systems in a single programming environment.

Another feature of OOMANEX is its graphic representation of the manipulator system simulation. Built on top of a Symbolics 3600 simultaneously, OOMANEX displays information about the current simulation in alpha-numerics and portrays the animation of the manipulator on screen.

The following is a brief summary of the contents of this thesis.

Chapter Two presents an overview of the structure and functionalities of OOMANEX. The concept of *object−oriented programming* is introduced in this chapter. Each module constituting OOMANEX is described in detail. Several figures also illustrate the structure of OOMANEX.

Chapter Three describes the dynamics of a robot manipulator and reviews the control theory used in OOMANEX. The control scheme with the *partitioned control law* used to articulate a manipulator system is discussed. This chapter also considers the problem of optimization of controller parameters.

Chapter Four discusses how to plan straight line trajectories for a manipulator with two or three links. The method can be used to plan trajectories for a space robot to accomplish certain service tasks in the space environment.

Chapter Five presents a detailed design example using OOMANEX. The control system for a planar manipulator with three links is established with optimized controller parameters.

Chapter Six summarizes the ideas introduced in the previous chapters and makes suggestions for future development of OOMANEX.

CHAPTER
TWO

System Description

2.1 Architecture

As mentioned at the beginning of the previous chapter, a general program package with the properties (i)-(iv) is highly desirable for developing a robot manipulator system. To make OOMANEX have those capabilities, following segments are included in OOMANEX:

1. a mathematical segment for dynamic model building;
2. a segment for kinematic and dynamic calculation as well as parameter optimization;
3. a trajectory planner for synthesizing trajectory primitives;
4. an interactive graphics user interface;

Based on the segments above, OOMANEX has an architecture as shown in Figure 1.

As can be seen from Fig.1, OOMANEX consists of five functional modules controlled by the OOMANEX's Manager. Symbolic Processor obtains the description of a robot manipulator from the user, returns the corresponding dynamical equations symbolically and a numerical simulation program upon request. Numeric Simulator integrates those simulation equations obtained from Symbolic Processor by specifying various simulation data. Gain Optimizer chooses

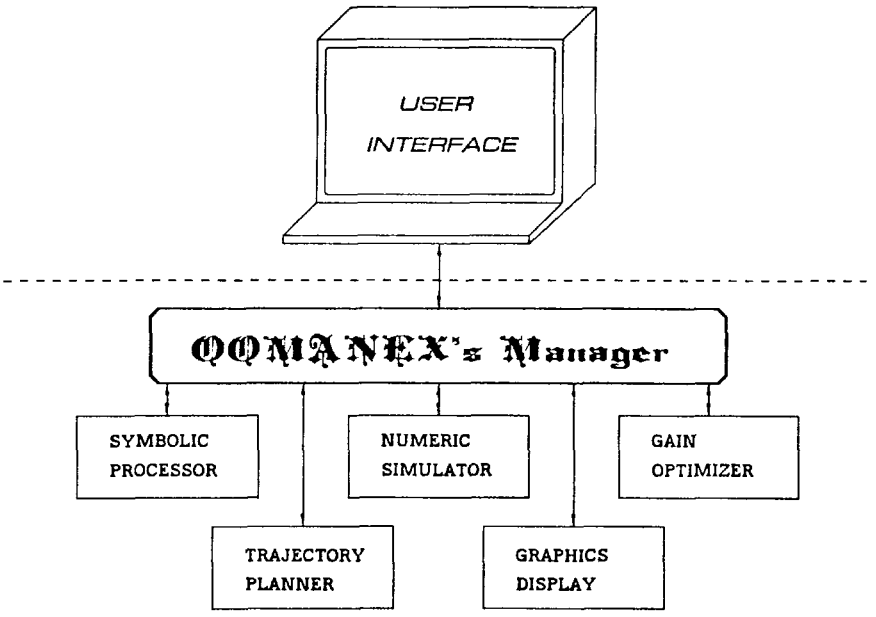


Figure 1. Architecture of OOMANEX

optimal control feedback gains for the robot manipulator system under suitable constraints. With the current implementation, Trajectory Planner is able to solve corresponding joint angles for a straight line trajectory specified by the user. Finally, Graphic Displayer presents on the screen the results from the above modules.

In OOMANEX, each module is written in the language most natural to it. For example, OOMANEX's top-level Manager is written in Zeta-Lisp; Symbolic Processor is based on DYNAMAN, a software package written in MACSYMA, a symbolic algebraic manipulation language; Numeric Simulator is written in FORTRAN so as to take advantage of standard numerical routines; and Gain Optimizer uses CONSOLE, an optimization package written in C language and FORTRAN under 4.3BSD Unix on a VAX 11/785.

The user interaction is as follows: when the user invokes OOMANEX, he sees a graphics display including many "command buttons" on screen , which queries

him about the robot manipulator system to be modeled and the simulation to be performed. After obtaining sufficient information to specify a complete simulation process, OOMANEX then generates the corresponding dynamical system symbolically, runs the numerical simulation and gain optimization, and checks for run-time error messages from the operating system. Finally, OOMANEX examines the outputs from the processes above and interprets them for the user in light of his request set up at the beginning. This invokes a dynamical graphics display for the robot manipulator, presents plots of time responses from controlled variables and makes comments on the proposed robot manipulator system design.

2.2 Description of the modules

(i) Symbolic Processor

Symbolic Processor is based on DYNAMAN, a software package to automatically formulate the dynamic equations for a robot manipulator of N links and with either revolute or prismatic joints using a Newton-Euler formulation [Sreenath-Krishnaprasad 86]. This package is written in MACSYMA, a symbolic algebraic manipulation language available on both VAX and Symbolics 3600. The original version of DYNAMAN is capable of

(1) generating dynamic models of a robot manipulator given the number of links N and the types (revolute or prismatic) of the manipulator joints;

(2) generating automatically upon request the requisite FORTRAN code for numerical simulation of the dynamic model generated;

(3) generating symbolically the manipulator Jacobian.

Symbolic Processor extends DYNAMAN so that it can not only generate the FORTRAN code of the dynamic equations for a N-link robot manipulator but also include a closed loop control system with a controller using a

partitioned control law (more details about the controller is given in Chapter 3). The simulator and the closed loop control system thus generated can be used to demonstrate the dynamic behavior of the robot manipulator and to design the controller parameters such as the position and velocity feedback gain matrices under some given engineering constraints.

Symbolic Processor has following basic blocks:

(1) A menu to show user the functions that Symbolic Processor can perform, and query the user for the necessary data, either symbolically or numerically, to specify the robot manipulator configuration the user wants to work on later;

(2) A block to generate the dynamic equation symbolically;

(3) A block to generate the Jacobian matrix;

(4) A block to generate the FORTRAN simulation program for the manipulator and the closed loop system.

The results from each block can be stored in files specified by the user for later use.

On being invoked by OOMANEX's Manager with a *message*, Symbolic Processor shows a list of questions on the screen to query the user. This can be shown in Figure 2.

(ii) Numeric Simulator

Numeric Simulator is used to simulate the dynamic behavior of a real robot manipulator. After obtaining the dynamic model of the manipulator generated by Symbolic Processor, Numeric Simulator connects the model with a controller and regulates the closed loop using a control law such as the partitioned control

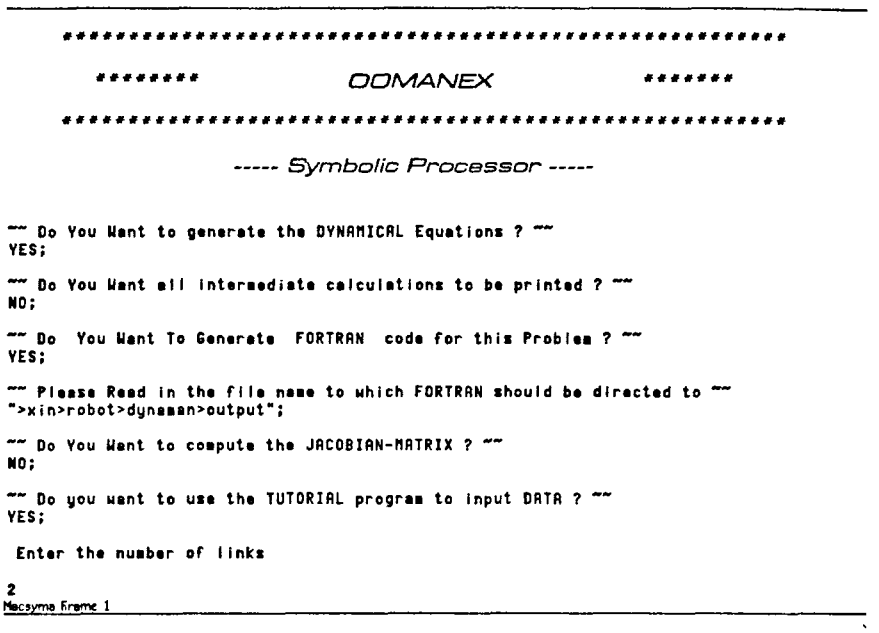


Figure 2. Part of the inquiring list of Symbolic Processor

law, in order to satisfy certain engineering constraints. More details about the control system can be found in Chapter Three.

Currently Numeric Simulator uses an exact dynamic model without concern for disturbances, and the controller consists of two parts: feed-forward and feedback. The feed-forward part of the controller uses the exact dynamic model of the manipulator. The feedback part of the controller is of P (proportional) plus D (differential) type. The input command contains a set of angular positions, velocities and accelerations. A fourth order Runge-Kutta method is used to perform the numerical integration of the dynamic equations. This simplified model system is shown in Figure 3.

Numeric Simulator begins its process by first asking the user to provide initial values for some parameters and obtains others by asking questions from

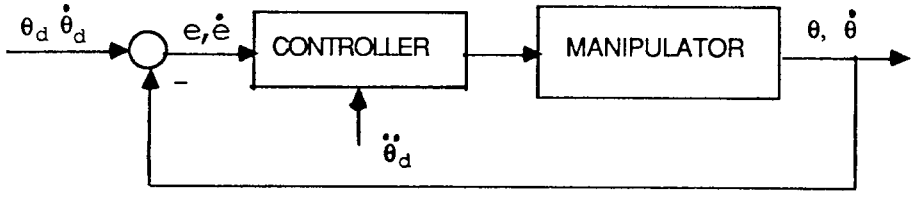


Figure 3. The manipulator control system

which it can infer them. This is done in a systematic manner as follows:

(1) Numeric Simulator gathers all the data required to define the system to be simulated. This includes getting values for the controller matrices.

(2) Numeric Simulator asks for the control goal or the desired (angular) trajectory, usually multi-step curves.

(3) Numeric Simulator then runs the simulation program and displays the corresponding time response.

(iii) Gain Optimizer

Under certain engineering constraints, usually a set of multi-step curves, Gain Optimizer determines the best parameters for the controller in the simulation system shown in Figure 3. At present time, we only consider the feedback part of the controller is of P and D type. Hence, only the *position* and *velocity* feedback gain matrices are to be optimized. Gain Optimizer takes the simulation system in (ii) as a simulator and then uses CONSOLE, an optimization-based design package, to determine the feedback gain matrices.

CONSOLE is an optimization-based design tool for a very broad of engineering systems provided that simulators are given. It is written in C programming language running on UNIX operating system and contains two distinct elements, CONVERT and SOLVE. CONVERT processes the problem description file given

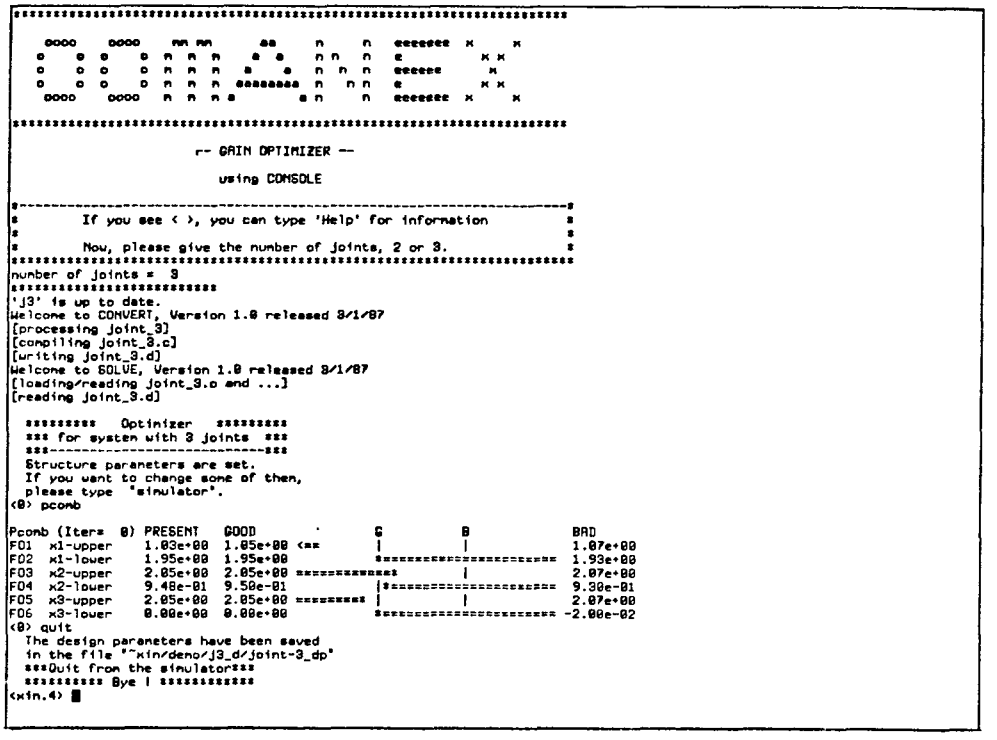


Figure 4. Example of Gain Optimizer

by the user and translates to an equivalent C program. SOLVE then links the C program, simulators (if any) and performs optimization interactively. If the design problem is well-posted, a local optimal solution can usually be obtained after sufficiently many iterations have been performed. The linking process of SOLVE, the C program and simulators is done by using a *dynamic loading technique*, which is a distinguishing feature of CONSOLE that makes it possible and simple for the design of robot manipulator controller by using advanced optimization algorithms.

Actually, Gain Optimizer first formulates the simulator obtained from Symbolic Processor into a problem description file which can be understood by CONSOLE. It then invokes CONSOLE to find the optimal control parameters. More details will be given in §3.4. An example is shown in Figure 4 where Gain Optimizer is called by the Manager sending an *invoke* message.

(iv) Trajectory Planner

Trajectory Panner devotes itself to plan a straight line trajectory for a planar robot manipulator. Usually, trajectory planning involves solving complicated inverse kinematic problems and making all kinds of trade-offs because the solutions to the inverse kinematic equations are not unique. However, solving for a straight line trajectory is the simplest and basic approach to planning, since any smooth trajectory can be approximated by piecewise linear trajectories.

In OOMANEX, the straight line trajectories can be planned for a manipulator of two or three links, and for a free flying space robot, which is expected to accomplish the task of mating electrical connectors. A detailed discussion is given in Chapter Four.

(v) Graphic Displayer

Graphic Displayer is the key to the task of integrating and displaying all information from other modules about the manipulator system design and simulation process. Its functions are used repeatedly as a graphics library by all other modules. In other words, its functions are just like $Software - ICs$ [Cox 86] which can be used to assemble a variety of graphical displays.

Graphic Displayer was written in Zeta-lisp on a Symbolics 3600. Zeta-lisp is a programming language that supports object-oriented programming. In Zeta-Lisp, a conceptual class of objects (definition see §2.3.1) and its operations are realized by the *Flavor system*, where part of its implementation is simply a convention in procedure calling style; another part is a powerful language feature, called **Flavors**, for defining classes of abstract objects. Here *Flavors* are the abstract types of object class; *methods* are the generic operations. We can define an object by making it to be an instance of a Flavor. Then the object can be

manipulated by sending *messages*, which are requests for specific operations. In other words, a conceptual class of objects is modeled by a single Lisp object, which represents the specified structure and has a list of state variables, i.e., the instance variables, while the operations are functions of the instance variables of the object.

For simulation of a robot manipulator on the screen, one of the basic requirements is to be able to draw various parts or objects constructing the manipulator and move them along the trajectories resulting from the simulation process or along certain preplanned trajectories. The display needs sometimes moving one part of the manipulator and sometimes the manipulator as a whole. To accomplish the above goal, Graphic Displayer has a set of functions to realize each animation, i.e., change the current graphics display to a new one. Each function can erase the parts needed to be changed and draw new parts.

To provide user information about simulation as much as possible, Graphic Displayer uses *frames* of Zeta-Lisp's window system to present the graphics and text. A *frame* is a window that is divided into subwindows, called *panes*, using the hierarchical structure of the window system. The panes are the inferiors of the frame, and the frame is the superior of each pane. By using a frame, Graphic Displayer is able to put many different things on the screen, each in its own place. In general, Graphic Displayer splits up the frame into areas in which graphics and text are displayed, areas where menus and mouse-sensitive input can be put, and areas to be intacted with, in which keyboard input is echoed or otherwise acknowledged. Figure 5 shows an example where Graphic Displayer not only illustrates the animation of the manipulator movement but also provides the information about the current simulation in the form of text, number and

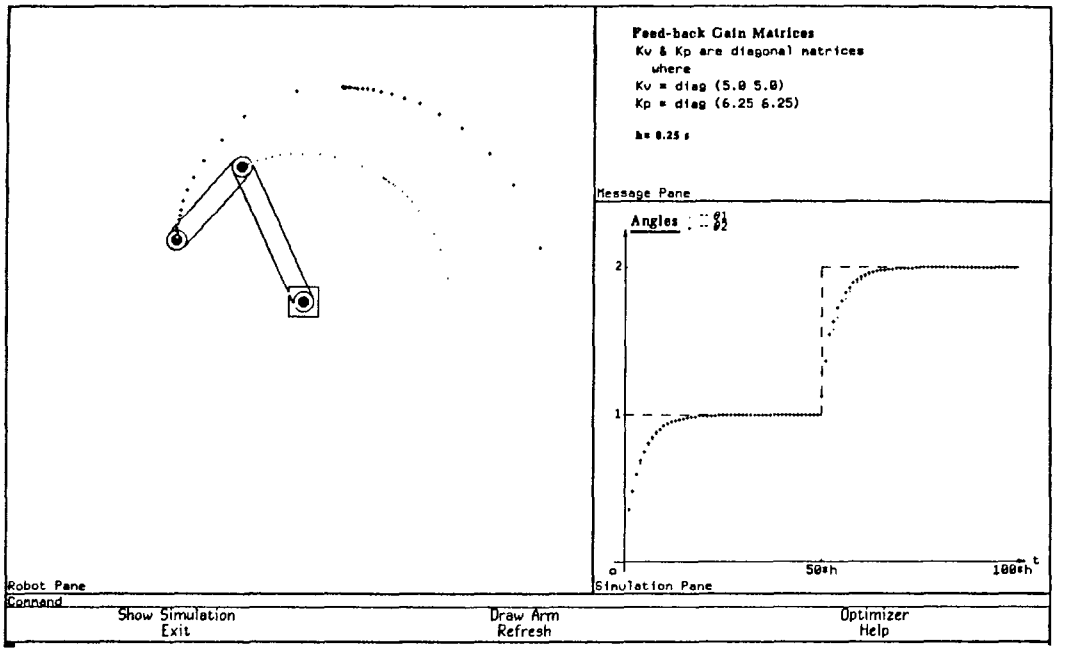


Figure 5. Example of Graphic Displayer

curves.

(vi) The Manager

The Manager is the top-level control element and the communication executor in OOMANEX. It interprets the commands given by the user via mouse or keyboard into function calls in OOMANEX's internal form; invokes the corresponding modules; finally executes certain processes according to the commands. More specifically, the Manager treats those modules as objects with associated *frames* and processes. After taking a command from the user, it brings up a corresponding frame and waits for the next command. It then executes the process specified by the command given by the user and calls Graphic Displayer to show the result from the process being executed in both graphics and text.

2.3 Implementation

The basic methodology used for designing OOMANEX is the object-oriented programming technique.

2.3.1 Object-oriented Programming Technique used in OOMANEX

Object-oriented programming (OOP) is a technique of organizing and building a software system using the concept of "object" . One feature of object-oriented programming is that it provides a way for a code designer to *encapsulate* functionality, which makes object-oriented programming one of the excellent code packaging techniques; another feature is *inheritance* which provides a means to link concepts into a related whole so that the changes in a higher level can be broadcast automatically throughout. Both features of object-oriented programming enable the software designer to produce reusable software components sometimes called $Software - ICs$; instead of building an entire program from raw materials, the bare statements and expressions of a programming language. Here, $Software - IC$ is used to emphasize the similarity of a *reusable* software component with the integrated silicon chip, an innovation that has revolutionized the computer hardware industry over the past twenty years or so [Cox 86].

The core concept of object-oriented programming is the notion of "object". An *object* is a conceptual entity that is an abstract data structure used frequently in a program and sometimes can be linked to real-world things. The program is built around a set of objects, each of which has a set of operations that can be performed on it. From the view point of the programmer, an *object* is a package of data and procedures that belong together. Roughly speaking, dividing a problem into objects is just a process of putting things where they belong [Kaehler 86].

An object is requested to perform one of its operations by sending a *message*

telling the object what to do. The receiver responds to the message by tracking its internal state, which can be examined and then altered by the operations available for that type of object upon receipt of the message.

OOMANEX has been designed by using above concept of object in an essential way. This naturally follows from Figure 1, in which six functional modules constitute OOMANEX. Each module has its own structure and uses a different programming language; each module is an independent unit but at the same time can communicate with other modules in some way. For example, Numeric Simulator has to use the results from Symbolic Processor, i.e., the symbolic dynamic equations for a user specified robot manipulator; the numeric data from Numeric Simulator must be used in order to obtain the optimal control feedback gains by Gain Optimizer. This reveals the fact that each one of the modules can be treated as an *object* and can be called when needed. Thus, another way to make the concept clear is that an *object* is an assembly of some private data and a collection of procedures that can access that data. The data is private to the object, and cannot be accessed without help from one of that object's procedures. The procedures are 'public' in the sense that it can be accessed by being called with message expressions, which tell an object what it should do. The object responds to a message by selecting and then performing a procedure. This procedure is called a *method.*

We point out that object-oriented programming can not only *encapsulate* objects but also create *inheritance* structures. This can be best shown by an example: A robot manipulator arm usually has two or three links and an end-effector (gripper) connected by some joints of certain type (rotational or prismatic). And the state of the arm can described by the position and orientation

of its links and end-effector (gripper). Thus, 'link' is one of the basic units to construct the arm. So, 'link' can be taken as an object and associated it with variables necessary for describing the link, such as link length, link angle and link position with respect to certain coordinate system. By measuring and changing the state variables of the links and gripper, the position and orientation of the arm can be measured and changed. On the other hand, each link has state variables such as length, angle and position. The links differ only in the values of their state variables. Hence, all links can be treated as members of a *class* of objects where objects have similar data structures and can be manipulated by using same operations. These operations are *generic* operations since they can be applied to all the objects in the same class. In other words, each object in a class can *inherit* those operations.

One simple example of generic operation is to rotate a link, which can be done by changing its angle with respect to certain coordinate system. After defining this operation to be "$rotate - link$", any link of a specific arm can be rotated by simply sending this message.

2.3.2 Object-oriented programming in User Interface Design

As mentioned in the previous section, OOMANEX consists of a Manager and five modules, each of which can be treated as an object. The user interacts with OOMANEX via the Manager, which is designed to be an *iconic* user interface using Zeta-Lisp, a programming language capable of object-oriented programming. One reason for the use of object-oriented programming in the iconic user interface is to *hide* information so as to limit the number of things that must be understood and properly dealt with . Another reason is that object-oriented programming is unusually suitable for the kind of programming involved in build-

ing iconic interfaces, since object-oriented programming provides such feature as *inheritance* which allows the basic abstraction of a class of objects to be implemented by using generic operations called *method.* A final reason is that iconic user interfaces are far more complex than *command – oriented* interfaces, making it more important than ever to control complexity, provide reusability, and encapsulate change [Cox 86].

Since Zeta-Lisp provides a number of functions or *special forms* of object-oriented programming capability, it makes it easier to build OOMANEX's user interface in much the same way that artists build *animated moves*, by drawing images onto pieces of transparent acetate and positioning the layers with respect to one another. In other words, by using Zeta-Lisp, the user interface can be designed so that it is able to project an intermediate world onto the screen, and reveal the physical laws of geometry, time, and all kinds of relationship in the present application in an intuitive way. OOMANEX's user interface was built on top of the graphics terminal of a Symbolics 3600 and is shown in Figure 6.

OOMANEX's user interface, which appears to be a house on the screen, is built by using Zeta-Lisp's window system. In OOMANEX's house, each window represents an access to a functional module, i.e., an object set in Figure 1, and from each window one can invoke the corresponding module and do the required jobs. Object-oriented programming is the background of OOMANEX's house, and all the programs for modules constitute its wall. This house structure represents OOMANEX's architecture (Figure 1), and involves the symbolics of object-oriented programming as well as the usage of Zeta-Lisp's window system.

As shown in Figure 6, it is expected that the interface can present information in pictures as well as alpha-numerics. Obviously, this requires that the

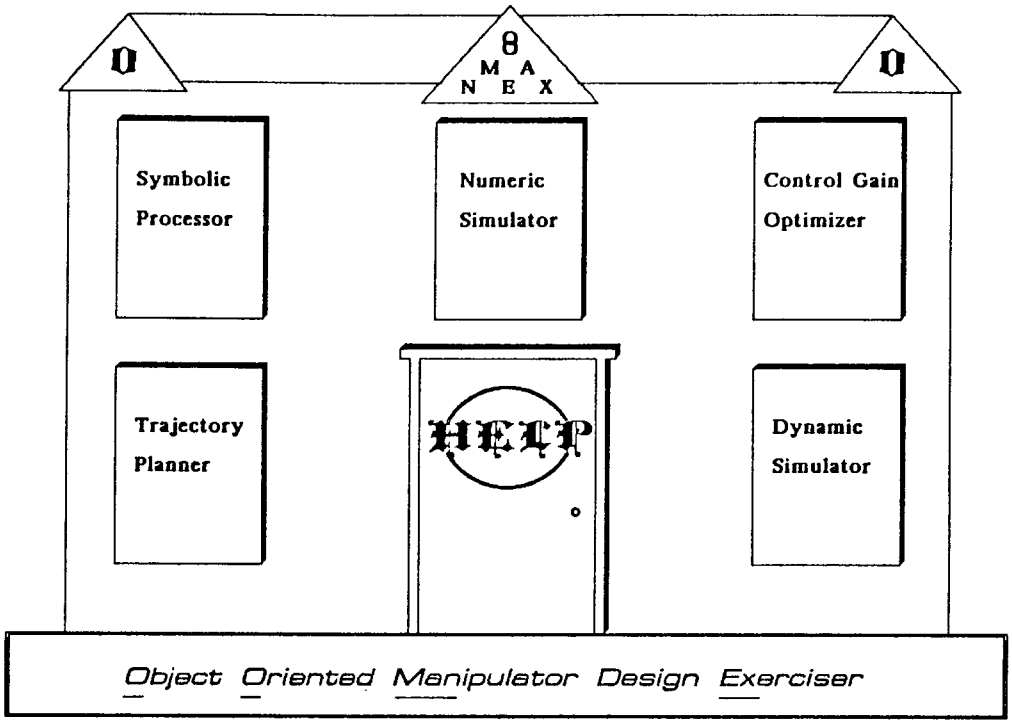


Figure 6. OOMANEX's User Interface

iconic or *object – oriented* user interface built for OOMANEX not only do all that conventional programs do, but also present information and their workings as pictures instead of the usual words and numbers. Of course, to implement the above features of the iconic user interface, support is needed from the terminal with graphic ability and graphic input device like a mouse rather than the usual keyboard for sending commands. Fortunately, these devices are available on Symbolics 3600. As a result, the windows and the door were designed to be mouse sensitive by using Zeta-Lisp's mouse system, so that user only needs to move the mouse over the windows or the door to give commands or to inquire information instead of typing on keyboard. This is very convenient and user-friendly.

Since there are five modules in OOMANEX, each of which is written in a different language, it is required that each module be supplied with a suitable

data when called by the Manager. In other words, each module requires a set of suitable initial data either symbolically or numerically and a set of goals that the module is expected to reach. Thus, it is very essential to provide appropriate symbolical or numerical data for each module when calling them. Here, "data" not only means numerical or symbolic data in the usual sense but also indicates the programs to be used in some modules. Roughly speaking, the above task involves associating with each module or object a good set of "conversion" processes which need to be supplied a list of data types *coming in* and *going out*.

For example, the output from Symbolic Processor, the dynamic equations, will be used by Numeric Simulator written in FORTRAN. In this case, care must be taken to ensure that Symbolic Processor produces a correct FORTRAN simulation program, with which Numeric Simulator can combine its own library of programs to obtain the required simulation results.

Since every module of OOMANEX is treated as an object, it is possible to make full use of already existing software for other approaches. This is indeed the fundamental justification for using object-oriented programming. Therefore, a vast resource of numerical software packages can be reused when solving the robot manipulator design problem.

In conclusion, in OOMANEX all the functional modules are "integrated" into one single programming environment through creative use of Zeta-Lisp's window system, package system and flavor system as are specified in Chapter One.

CHAPTER
THREE
CONTROLLER DESIGN

3.1 Introduction

Present day manipulators are commonly electrically actuated mechanical systems that consist of n rigid bodies connected by joints. One motor (usually a DC motor) is mounted on each joint.

In general, the manipulator control system has a hierarchical structure, where the following four control levels are encounted most often and listed from highest to lowest level [Vukobrato 85]:

- the highest which recognizes the obstacles in the operating space and the conditions under which a task is being performed and makes decisions on how the specified task imposed is to be accomplished;
- the strategic level which divides the specified operation into elementary movements;
- the trajectory planning level which performs the resolution of an elementary movement into the motion of each degree of freedom of the manipulator;
- and the executive level which executes the imposed motion of each degree of freedom.

A higher level communicates with a lower level giving it instructions and receiving from it relevant information required for decision-making. After obtaining information from a lower level, each level makes decisions taking into

account general decisions obtained from a higher level and forwards them to a lower level for execution.

Here we only treat the manipulator system as composed of the two lowest levels from the above structure under the assumption that, at the higher control levels, the task (the motion to be realized by the manipulator) has been defined. Also, to keep the discussion simple, the manipulator dynamics will be considered without the action of external contact forces, so that the system behaves as an open kinematic chain. Such a system can be illustrated as in Figure 7, where the trajectory planning level generates the trajectories of each degree of freedom which perform the desired functional movement, and the executive level executes these trajectories by means of appropriate actuators incorporated in each joint.

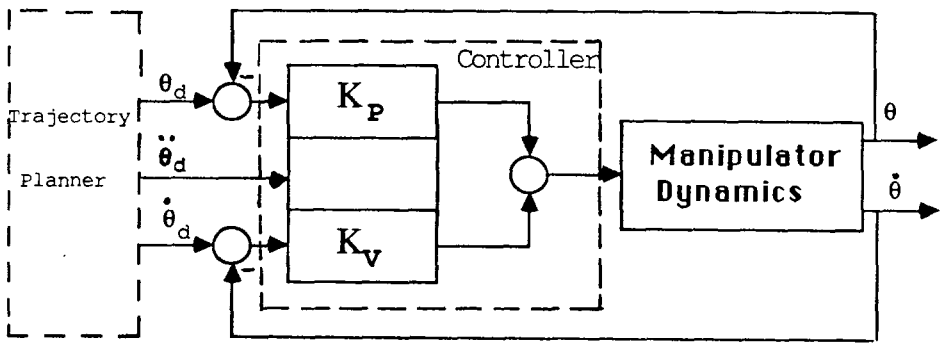


Figure 7. Block diagram of the control system

The unit which calculates the required driving torques according to some control law is the Controller; θ_d, $\dot{\theta}_d$ and $\ddot{\theta}_d$ are desired position, velocity and acceleration of the joints specified by trajectory planning; θ, $\dot{\theta}$ are joint position and motor shaft velocity measurement.

Manipulator systems are usually complicated dynamical systems since they are highly nonlinear and coupled multivariable systems. During the past decade,

many control schemes have been proposed. The control schemes suggested include classical controllers as well as optimal controllers. Among the classical design approaches, the calculation of the torques for a nominal trajectory is presented by Paul [Paul 72], which he called "inverse problem" technique (Bejczy named it "computed torque" technique [Bejczy 74]; we call it "partitioned control law"; others such as Furuta call it "nonlinear feedback" method [Furuta 84]). In the "inverse problem" technique or "partitioned control law", the input torques necessary to drive the manipulator end-effector along a preplanned path are computed on-line as functions of desired joint accelerations, velocities and positions and their actual counterparts. This method is simple conceptually and requires relative less computation time. Hence it is adequate for real-time control.

3.2 Dynamic Model of a Manipulator

In general, a manipulator is considered as a coupled electro-mechanical system. As shown in Figure 7, the inputs to the control system shown in Figure 3 are torques generated by motors driving the joints. The outputs are joint position and motor shaft velocity measurements. This input/output description forms the definition of the manipulator as a dynamical system. To make this definition of dynamical system (or dynamical model) quantitative, mathematical relations are required which relates input to output. The mathematical relation between input (torque) and output (position and velocity) is obtained by the specification of state equations (differential equations) governing the manipulator motion.

Generally, the dynamical model of a manipulator with n degrees of freedom can be described by one of two approaches. One approach is based on Lagrangian mechanics, and the other is based on the Newton-Euler method. The Lagrangian dynamics formulation is often utilized for modeling of such a mechanical system

as a manipulator, because the method is simple and systematic and the model can be driven by a routine procedure. The resultant equations of motion , excluding the dynamics of the electronic control devices and the gear friction, are a set of second-order coupled nonlinear differential equations. Hence this method is suitable for the analysis of mechanical systems, except for the difficulty and the complexity of the calculation. On the other hand, the Newton-Euler formulation has the advantage of both speed and accuracy. The resultant dynamical equations, excluding the dynamics of the control and gear friction, are a set of forward and backward recursive equations. It seems to involve difficulties in the analysis, because the input/output relation is not clear from the resultant set of equations. But as the equations are usually simple and recursive in nature, they are convenient for calculation, and this formulation is often used for on-line calculations of dynamics. It must be noted that these two different formulations are equivalent in mathematical meaning. In the following, both types of formulations will be used: the lagrangian formulation for analysis, and the Newton-Euler formulation for real time control.

In general, using the Lagrangian formulation, the dynamic equation of a manipulator with n degrees of freedom, which is supposed to be a spatial link mechanism, is described in the matrix vector notation [Paul 72] as

$$t_c = J(\theta)\ddot{\theta} + F(\theta,\dot{\theta}) + G(\theta) \tag{1}$$

where

$J(\theta)$: n x n matrix of inertia terms,

$F(\theta,\dot{\theta})$: n dimensional vector of the Coriolis and, centrifugal terms,

$G(\theta)$: n dimensional vector of gravity terms,

t_c : n dimensional vector of the generalized joint forces.

The Newton-Euler method is based on a set of mathematical equations that describe the kinematic relation of the moving links of a manipulator with respect to the base coordinate system. An efficient recursive formulation has been used to develop a symbolic manipulator program to derive the equations of motion of an open kinematic chain ([Sreenath-Krishnaprasad 86], Eqs. (13)). To summarize that paper,

If the joint i is *revolute*, the torque needed to be applied at the motor of joint i in the i'th coordinate system is

$$\begin{aligned} q_i^T \cdot T_i^{motor} &= \sum_{j=i}^{N} q_i^T \cdot (\phi_j \cdot \dot{\omega}_j + m_j(\Gamma_j - \Gamma_i + \alpha_i) \times \ddot{\Gamma}_j) \\ &+ q_i^T \cdot \sum_{j=i}^{N} \omega_j \times \phi_j \cdot \omega_j - q_i^T \cdot \sum_{j=i}^{N} T_j^{ext} \\ &- q_i^T \cdot \sum_{j=i}^{N} (\Gamma_j - \Gamma_i + \alpha_i) \times (m_j g + F_j^{ext}) \end{aligned} \tag{2}$$

If the joint i is *prismatic*, the force needed to be applied at the motor of joint i in the i'th coordinate system is

$$q_i^T \cdot F_i^{motor} = \sum_{j=i}^{N} q_i^T \cdot m_j \ddot{\Gamma}_j - q_i^T \cdot \sum_{j=i}^{N} (m_j g + F_i^{ext}) \tag{3}$$

Here,

$$q_i = ACB_{i-1} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

denotes the axis of the degree of freedom with

ACB_i the 3×3 Rotation matrix to carry the i'th coordinate system to the inertial coordinate system.

The other parameters are:

α_i Denavit-Hartenberg parameter 'α' of the i'th link;

m_i Mass of link i;

ϕ_i Inertial tensor of link i;

ω_i Inertial angular velocity of link i;

$\dot{\omega}_i$ Inertial angular acceleration of link i;

Γ_i Vector from inertial origin O_o to center of mass (c.m.) of link i;

$\ddot{\Gamma}_i$ Inertial acceleration of link i c.m.;

T_i^{ext} Total external torque acting on link i in the i'th coordinate system;

F_i^{ext} Total external force acting on link i (through the c.m. of link i) in the inertial coordinate system.

In OOMANEX, the equations based on the Newton-Euler method are used since they can be symbolically generated by DYNAMAN, a software package written in MACSYMA [Sreenath-Krishnaprasad 86].

3.3 Control Laws for the Manipulator

Given the motion equations of a manipulator, the purpose of manipulator control is to achieve a prescribed motion of the manipulator along a desired trajectory by joint motors which supply appropriate drive torques. The motors should also be able to exert corrective compensation torque to the joint to adjust for any deviation of the manipulator from the planned trajectory.

To control the manipulator, one of the basic control schemes — the

partitioned control law, will be used, which is of the form as

$$t_c = at + b \tag{4}$$

with

$$a = J(\theta)$$

$$b = F(\theta,\dot{\theta}) + G(\theta)$$

$$t = \ddot{\theta}_d + K_v(\dot{\theta}_d - \dot{\theta}) + K_p(\theta_d - \theta)$$

where $J(\theta)$, $F(\theta,\dot{\theta})$, and $G(\theta)$ are as in equation (1); $\ddot{\theta}_d$, $\dot{\theta}_d$, and θ_d: are desired joint acceleration, velocity and position derived by trajectory planning; $\dot{\theta}$ and θ are the actual joint velocity and position measurements; K_v and K_p are velocity and position feedback gain matrices.

Rewriting Eq.(4) yields

$$\begin{aligned} t_c &= J(\theta)\ddot{\theta}_d + F(\theta,\dot{\theta}) + G(\theta) + J(\theta)K_v(\dot{\theta}_d - \dot{\theta}) + J(\theta)K_p(\theta_d - \theta) \\ &= t_1 + t_2 \end{aligned} \tag{5}$$

where

$$t_1 = J(\theta)\ddot{\theta}_d$$

$$t_2 = F(\theta,\ \dot{\theta}) + G(\theta) + J(\theta)K_v(\dot{\theta}_d - \dot{\theta}) + J(\theta)K_p(\theta_d - \theta)$$

In Eq.(5), t_1 is the feed-forward control part which generates the desired torque for each joint given desired joint acceleration $\ddot{\theta}_d$ under the condition that no modeling errors exist and system parameters are known. But this situation does not occur in practice because of modeling inaccuracies and parameter

variations as well as the errors due to backlash, gear friction etc, which cause deviations from the desired joint trajectory. Rate and position feedback is therefore used to compute the correction torque in t_2 to compensate for small deviations. The remaining part of t_2 is the nonlinear feedback part to generate torques in order to balance the Coriolis and gravity terms, which are usually treated as "known" disturbances. Thus, t_1 and t_2 form two parts of the required control torque, from which the name "partitioned control law" derives.

This control law is analysed below. Suppose that the input torque to the manipulator was calculated as follows:

$$\begin{aligned} t_c \;=\; J_a(\theta)\ddot{\theta}_d \;+\; F_a(\theta,\,\dot{\theta}) \;+\; G_a(\theta) \;+\; J_a K_v(\dot{\theta}_d \;-\; \dot{\theta}) \\ +\; J_a(\theta)K_p(\theta_d \;-\; \theta) \end{aligned} \tag{6}$$

where the subscript a denotes the term being calculated using incorrect parameters. By substituting Eq.(5) into (1), we have

$$\begin{aligned} & J_a(\theta)\left[(\ddot{\theta}_d \;-\; \ddot{\theta}) \;+\; K_v(\dot{\theta}_d \;-\; \dot{\theta}) \;+\; K_p(\theta_d \;-\; \theta)\right] \\ & \quad = \left[(J(\theta) \;-\; J_a(\theta)\right]\ddot{\theta} \;+\; \left[F(\theta),\,\dot{\theta}) \;-\; F_a(\theta,\,\dot{\theta})\right] \\ & \qquad +\; [G(\theta) \;+\; G_a(\theta)] \end{aligned} \tag{7}$$

The right hand side is the error due to modeling and can be treated as a disturbance, which is denoted as f.

Usually, as $J_a(\theta)$ contains the inertia of drive, such as motors and gear train etc, it can be treated as a nonsingular matrix. Hence Eq.(7) can written as

$$(\ddot{\theta}_d \;-\; \ddot{\theta}) \;+\; K_v(\dot{\theta}_d \;-\; \dot{\theta}) \;+\; K_p(\theta_d \;-\; \theta) \;=\; J_a^{-1}(\theta)f \;=\; f' \tag{8}$$

Further, let

$$\begin{aligned} e &= \theta_d - \theta, \\ \dot{e} &= \dot{\theta}_d - \dot{\theta}, \\ \ddot{e} &= \ddot{\theta}_d - \ddot{\theta}, \end{aligned} \tag{9}$$

and substitute Eq.(9) in Eq.(8) we have

$$\ddot{e} + K_v\dot{e} + K_p e = f' \tag{10}$$

It can be seen from Eq.(10) that if we choose K_v, K_p as diagonal matrices then the error dynamics of the manipulator (10) is decoupled and K_v, K_p can be chosen as appropriate for each joint. This approximate approach is based on Paul's experiments with the Stanford manipulator in 1972, where he found that the contribution by the Coriolis and centrifugal terms is relatively insignificant. This is especially true when the hand (end effector) approaches its goal position and orientation since the joint velocities are low during that time.

Hence, we can write the error dynamics for joint i as

$$\ddot{e}_i + k_{v_i}\dot{e}_i + k_{p_i}e = f' \tag{11}$$

Taking the Laplace transform of the homogeneous part of (11) and letting

$$k_{v_i} = 2\xi_i\omega_i,\ k_{p_i} = \omega_i^2 \tag{12}$$

yields

$$s^2 + 2\xi_i\omega_i s + \omega_i^2 = 0 \tag{13}$$

where

$\xi_i =$ the error system damping ratio of joint i,

$\omega_i =$ the error system undamped natural frequency of joint i.

Common practice is to select $\xi_i = 0.7$ and ω_i as high as possible to increase the response or tightness of the servo control loop. Here $\xi_i = 1$ is selected which is reasonable since the overshoot for a step input is then reduced from 7% ($\xi_i = 0.7$) to zero. Thus we can set

$$2\omega_i = k_{v_i}$$

$$\omega_i^2 = k_{p_i}$$

or

$$k_{p_i} = \frac{{k_{v_i}}^2}{2} \tag{14}$$

3.4 Optimization of control parameters

In §3.3, we determine the position and the velocity feedback matrices to be diagonal under the assumption that the coupling terms resulting from the Coriolis and centrifugal terms are relatively *small*. However, this approach is too crude to design a controller for a complicated system such as the manipulator system mentioned in previous sections. This is because that manipulator systems in the real world are highly coupled and the diagonal matrices K_v, K_p would not meet specifications. For example, recent research shows that approaches based on simply dropping the velocity term is not adequate, since it can be shown that the velocity product terms have the same significance relative to the acceleration dynamic terms for all speeds of movement [Sahar 85]. Therefore, we should permit the control matrices K_v, K_p to be in general form to improve the dynamic behavior of the manipulator system.

The task of choosing the controller parameters is too complicated to be done manually, especially for medium or large problems. Only with the help of

certain optimization software packages can we approach the complicated design problem. CONSOLE is one of such packages. When using it, one first formulates the design problem in a form that can be understood by CONSOLE. Then, with the help of CONSOLE, one could possibly find a set of design parameters satisfying all specifications.

In more detail, CONSOLE contains two distinct programs, CONVERT and SOLVE. A design problem is first described in the problem description file in a form that can be understood by CONVERT. The essential information needs to be included in the problem description file is the declaration of design parameters, assigning their initial values and the declaration of various specifications, such as objectives and constrains. CONVERT then could be invoked to read the problem description file, detect all syntax errors and translate into an equivalent program written in C programming language. Once the translation is successful, one can invoke SOLVE to integrate problem description, simulators and itself together in a way that its complexity is not more than linking programs. SOLVE then interacts with the user to perform optimization algorithm to find a set of design parameters satisfying all specifications. The algorithm in SOLVE is a *feasible direction method*, which was developed by Nye and Tits [Fan 87], and it proceeds the optimization with increasing satisfaction.

One of the reasons for using CONSOLE to solve for the optimal controller is that CONSOLE can accept simulation models written in FORTRAN, C or Pascal. This is very attractive since the manipulator simulation needs only to be "read" out in its natural form from the generator DYNAMAN instead of being formulated in some restrictive form. Before now, the optimal design problems of this kind have typically remained unsolvable because of the complexity of

the system itself and difficulty to formulate it to adapt to existing optimization software packages. CONSOLE makes it possible to approach the optimal design problem for the manipulator system quantitatively.

As is usual engineering practice, solving an optimization problem begins by setting a set of objectives and constraints. Here, the objective is to minimize the deviation of certain joint angles from the nominal or preplanned trajectory. A too large deviation is usually undesired since it might either cause collision with some object close to the preplanned trajectory or delays in the settling time. These deviations need to be minimized so that the joint angles do not go too "far" from the preplanned trajectory but are still varying "fast" enough. This can be stated as a usual tracking problem: Suppose that a joint angle is expected to track a step input function $u(t)$ as shown in Figure 8. One hopes that the joint angle θ_i will stay between the upper curve and the lower curve, where the upper curve represents the largest overshoot tolerated and the lower curve is a measure of the time response. For example, one usually sets the overshoot to be 5% of the height of the step input, the tolerated stability to be 2% of the height of the step input, and the settling time to be three times the sampling time.

If the control parameters K_v, K_p can be chosen such that the time response of the joint angle θ_i lies exactly between the upper and lower curves, the parameters chosen are thought to be optimized. But, usually the constraints of the upper and lower curves are too hard to satisfy due to the setting of the optimization problem and the number of constraints. Hence, one has to compromise his constraints and "soften" them so long as they do not violate constraints from the real world such as that they allow the joint angles to have overshoot but of not too large a magnitude to come a collision with some objects close to

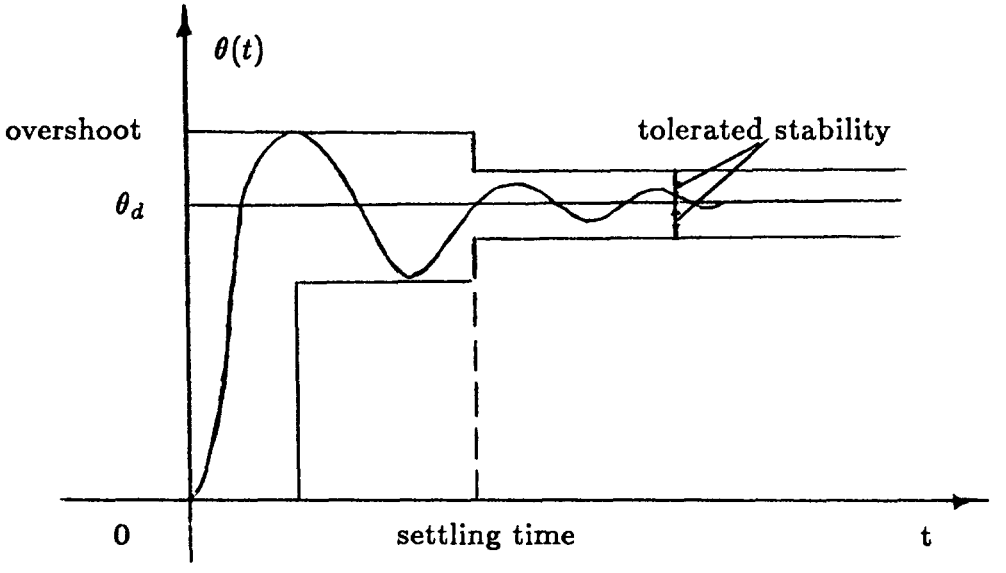


Figure 8. Step response of a joint angle

the preplanned trajectory. These "softened" constraints can be represented by two so-called *bad* curves, in which the upper curve allows larger overshoot while the lower one permits the system to reach its stable state shown by the shaded area between the two curves more slow than the the limit permitted by the good lower curve. This can be seen in Figure 9.

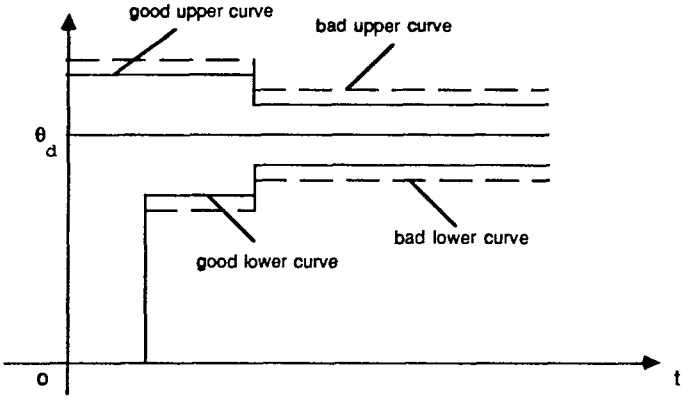


Figure 9. The good and bad constraint curves for a step response

In practice, the preplanned trajectory can only be tracked by a manipu-

lator in a point-by-point sense, since the manipulator system is a second order nonlinear system. Therefore, a trajectory, say, a straight line, is divided into finite segments and the manipulator is regulated to trace each segment (see Chapter 4.). For this reason, there is actually a sequence of step inputs for the manipulator to follow, and naturally a set of constraints in the form of good and bad curves. One such example is given in Figure 10.

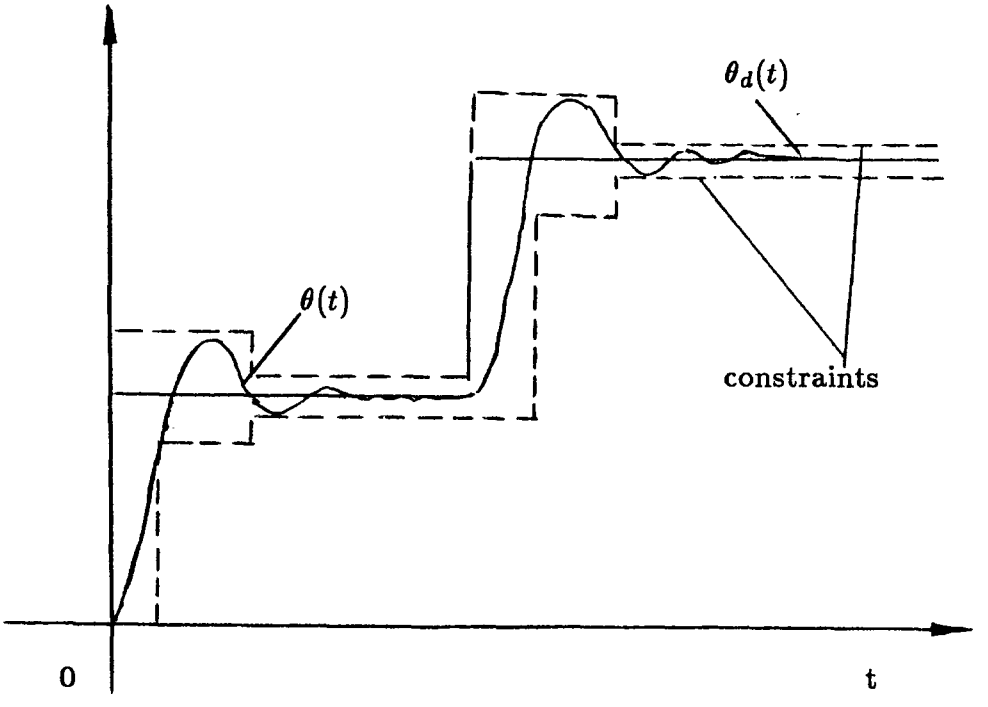


Figure 10. Multi-step input and the constraints

The objective for the optimization process is then to determine the best control parameters K_v, K_p so that the joint angles θ_i, i *from* 1 *to* n follow the preplanned trajectory point by point and their time responses lie in the areas between the corresponding upper and lower curves.

Once one defined all the objectives and constraints and written in the prob-

lem description file, one is ready to have CONSOLE solve for optimal feedback gain matrices K_v, K_p. As expected, the matrices K_v, K_p chosen by CONSOLE are general matrices, using which the time response of the joint angles will meet the objectives and satisfy the constraints one sets up in the problem description file.

CHAPTER
FOUR

TRAJECTORY PLANNING

4.1 Trajectory planning for a planar manipulator

A *trajectory* is defined as a time course or sequence along a path in space. The time course or sequence can be defined by a set of *configurations* of an object either in Cartesian space or joint space. The configuration of a rigid object defined in Cartesian space is defined by a 3-tuple (x, y, θ) where (x, y) is the position of a reference point P and the θ is the orientation of an internal coordinate system x', y' of the object with respect to the reference point p (Figure 11). The configuration defined in joint space is a sequence of vectors of joint variables. The set of all possible configurations is called the *configuration space* [Brady 82].

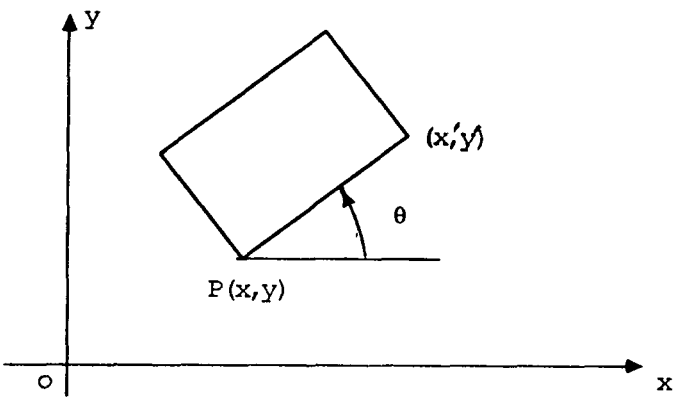


Figure 11. The configuration of an object

It is natural to describe a trajectory by the Cartesian space description of

configuration. But from the point of view of the control system and formulation of kinematics and dynamics, the most convenient description of a configuration is the joint space description. Generally, the transformation from joint space to Cartesian space is simple and efficient to compute. However, the inverse kinematics is often intractable except by numerical methods, and even where it is solvable, several joint vectors typically give rise to the same Cartesian space description of a configuration. Nevertheless, because it is easier to keep track of the behavior of a trajectory calculation algorithm using joint vector description of configurations, and because it largely finesses the expense of inverse kinematics computations, it is normally preferable to compute trajectories in joint space, even if the source and destination configurations are specified in terms of Cartesian space.

For a planar manipulator with N rigid links, the configuration of each link can be defined by a 3-tuple in Cartesian description. Assume that the base is fixed and the links are constrained at the joints. Then the 3-tuples defining the configuration of the manipulator links is uniquely determined by the joint angles $\theta_i,\ i = 1,2,\ldots,N$. For example, consider the two-link manipulator shown in Figure 12, where the joint positions are denoted by θ_1 and θ_2; the links have length L_1 and L_2 and both are assumed to be right circular cylinders of radius R. The 3-tuple for the first link is $(0,0,\theta_1)$; the 3-tuple for the second link is (x_1,y_1,θ_2) with

$$\begin{aligned} x_1 &= L_1 cos\theta_1 \\ y_1 &= L_2 sin\theta_2 \end{aligned} \tag{14}$$

Like most current industrial robots, the manipulators considered here are *positioning* devices, which can be "taught" by being led them through a se-

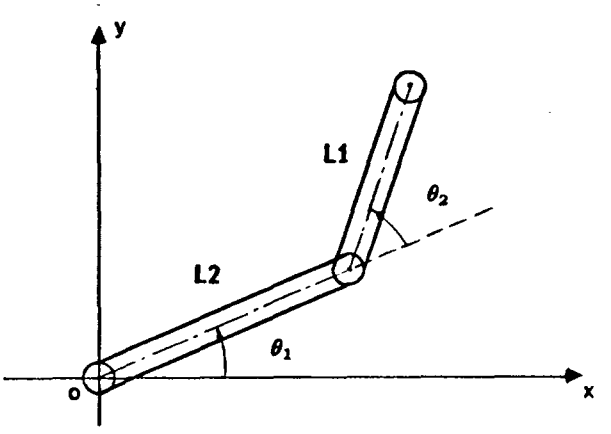


Figure 12. A two-link planar manipulator with revolute joints

quence of motions that can be executed repeatedly. The individual motions are specified by placing the manipulator in configurations corresponding to the beginning and end of the motion. The process of computing configurations corresponding to a sequence of motions is called *trajectory planning*. In other words, *trajectory planning* converts a *description* of the desired motion to a trajectory defining the time sequence of intermediate configurations of the arm between the beginning point and the destination point.

The output of trajectory planning is a sequence $\{\theta_i^j\}$, $i = 1, 2, \ldots, N$; $j = 1, 2, \ldots, n$ of configurations of arm. The configurations are shipped off in succession to the servo mechanisms controlling the joint motors that actually move the arm.

4.2 Planning straight line trajectories

Many applications naturally involve trajectories in which the end effector is required to follow a straight line path from an initial position to the destination. However, the Cartesian space straight line motion can not be achieved by simply using linear interpolation between two points under the *joint space descriptions*.

One way to plan a straight line path is first to divide the path by finite number of "knots" or *Cartesian midpoints*; then compute corresponding configuration for each knot.

But constraining the path of the end effector does not totally constrain the arm configuration since for fixed position of the end effector the arm may have more than one configuration. This can be seen from Figure 13, where the end effector of a two-link manipulator is positioning the position P which can be determined by two sets of joint angles: (θ_1, θ_2) and (θ_1', θ_2').

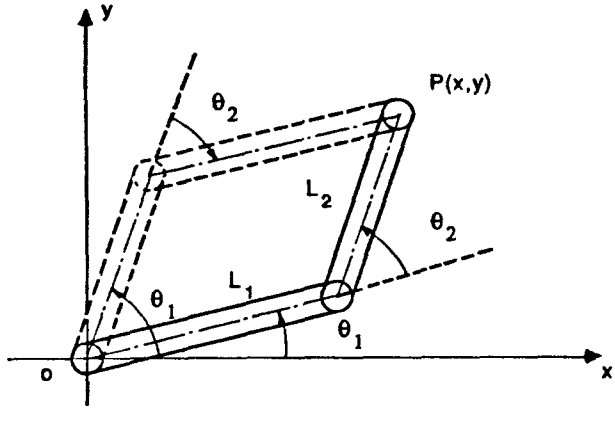


Figure 13. Configurations of a two-link manipulator

Besides, every manipulator has a *Cartesian work space* which contains all the points the manipulator can approach to. If some point along a path is outside of the work space, no configuration can be found for the corresponding point. Thus, when we talk about trajectory planning, we have to check if a desired path is entirely inside the work space. One simple example is the work space of a two-link planar manipulator shown in Figure 12. The position of its end effector

is

$$\begin{aligned} x &= L_1 cos\theta_1 + L_2 cos(\theta_1 + \theta_2) \\ y &= L_1 sin\theta_1 + L_2 sin(\theta_1 + \theta_2) \end{aligned} \tag{15}$$

After some calculations, (15) yields

$$x^2 + y^2 = L_1^2 + L_2^2 + 2L_1 L_2 cos\theta_2 \tag{16}$$

or

$$|L_1 - L_2| \leq \sqrt{x^2 + y^2} \geq |L_1 + L_2| \tag{17}$$

From (17), the work space for the two-link manipulator is obtained as shown in Figure 14.

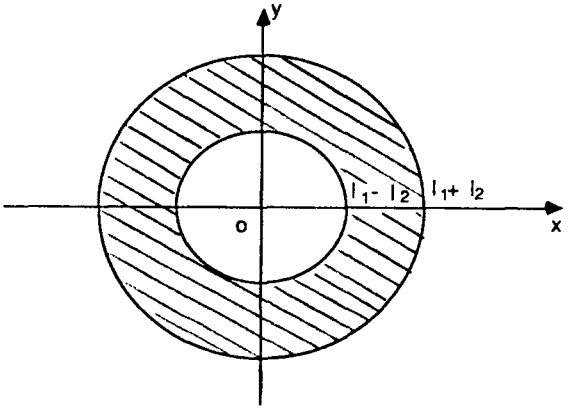


Figure 14. The work space for the two-link manipulator

In order to compute the individual configurations the *inverse kinamatic* problem for a manipulator has to be solved, which can be quite complicated. As a first step, we consider how to plan straight line trajectories for a two-link manipulator shown in Figure 15. Then, we extend some of the basic methods to plan for a three-link manipulator.

(i) Planning for a two-link manipulator

As mentioned above, for each point in the work space there are two corresponding configurations of the two-link manipulator. When the manipulator moves from one point to another, one of the two configurations must be chosen for the next point. Usually, it is undesirable that the manipulator executes a big "jump" when moving to the next point since it will need more energy. Based on this, the manipulator is always required to move in one of the two *gestures* depending on its initial gesture. These two gestures are called "stretch up" and "stretch down", as shown in Figure 15.

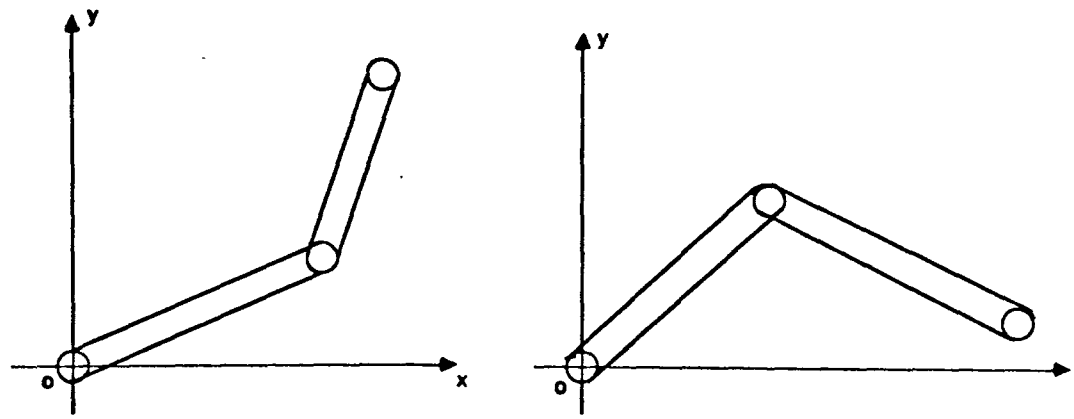


Figure 15.

(a) The "stretch up" gesture. (b) The "stretch down" gesture

Here, the planning for the manipulator in "stretch up" gesture is considered. For the "stretch up" gesture there are eight types of configurations, which are shown in Figure 16.

Draw one of the configurations in "stretch up" gesture as in Figure 17.

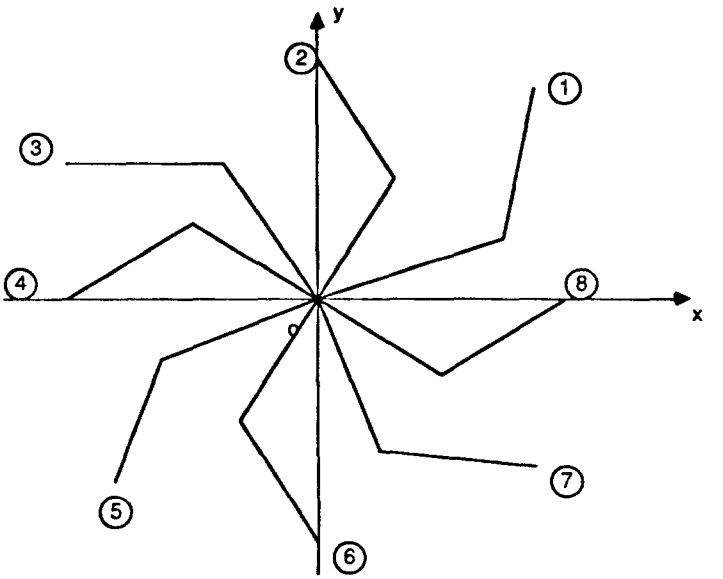


Figure 16. Eight possible types of configuration for the "stretch up" gesture

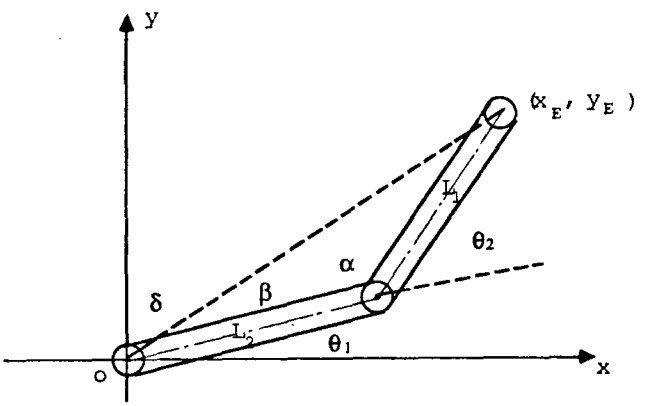


Figure 17. Calculation of joint angles θ_1, θ_2

Let (x_E, y_E) denote the position of the end effector, and let

$$\alpha = cos^{-1} \frac{L_1^2 + L_2^2 - x^2 - y^2}{2L_1 L_2},$$

$$\beta = cos^{-1} \frac{x^2 + y^2 + L_1 - L_2}{2\sqrt{x^2 + y^2} L_1}, \tag{18}$$

$$\delta = tg^{-1} \frac{|y|}{|x|}$$

Then, the joint angles for each type of configuration are:

(1) $x > 0,\ y > 0$

$$\begin{aligned} \theta_1 &= \delta - \beta, \\ \theta_2 &= \pi - \alpha \end{aligned} \tag{19}$$

(2) $x = 0,\ y > 0$

$$\begin{aligned} \theta_1 &= \frac{\pi}{2} - \beta, \\ \theta_2 &= \pi - \alpha \end{aligned} \tag{20}$$

(3) $x < 0,\ y > 0$

$$\begin{aligned} \theta_1 &= \pi - (\beta + \delta), \\ \theta_2 &= \pi - \alpha \end{aligned} \tag{21}$$

(4) $x < 0,\ y = 0$

$$\begin{aligned} \theta_1 &= \pi - \beta, \\ \theta_2 &= \pi - \alpha \end{aligned} \tag{22}$$

(5) $x < 0,\ y < 0$

$$\begin{aligned} \theta_1 &= \pi - (\delta - \beta), \\ \theta_2 &= \pi - \alpha \end{aligned} \tag{23}$$

(6) $x = 0,\ y < 0$

$$\begin{aligned} \theta_1 &= \frac{3}{2} \times \pi - \beta, \\ \theta_2 &= \pi - \alpha \end{aligned} \tag{24}$$

(7) $x > 0,\ y < 0$

$$\begin{aligned} \theta_1 &= -(\delta + \beta), \\ \theta_2 &= \pi - \alpha \end{aligned} \tag{25}$$

(8) $x > 0,\ y = 0$

$$\begin{aligned} \theta_1 &= -\beta, \\ \theta_2 &= \pi - \alpha \end{aligned} \tag{26}$$

With the Eqs. (19)-(26), the configuration of "stretch up" gesture corresponding to any point inside the work space can be calculated. The remaining problem is to obtain a sequence of points along a straight line path inside the work space. As an example, suppose that it is required to get the sequence of configurations corresponding to the knots along the straight line path L as shown in Figure 18. Assume that the number of knots is n. Then, for $i = 1, 2, \ldots, n$, the position of i'th point is

$$\begin{aligned} x_i &= x_{i-1} + \frac{(x_n - x_0)}{n}, \\ y_i &= y_{i-1} + \frac{(y_n - y_0)}{n} \end{aligned} \tag{27}$$

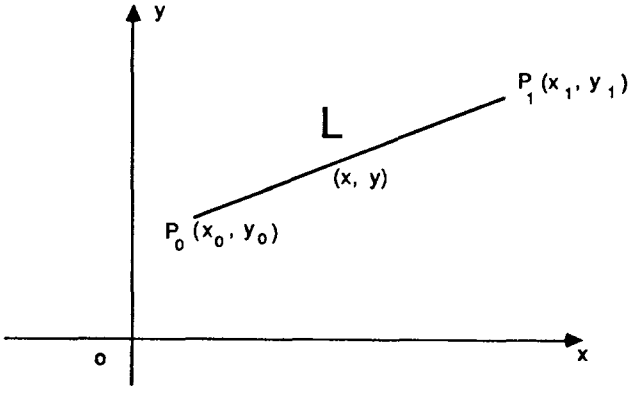


Figure 18. Planning the trajectory L

Similar calculation can be done for the configurations of "stretch down" gesture.

(ii) Planning for a three-link manipulator

A three-link planar manipulator has three degrees of freedom (d.o.f.) (see Figure 19). Hence, when planning a trajectory for the three-link manipulator, it is necessary to specify at each "knot" not only the position but also the orientation for the end effector.

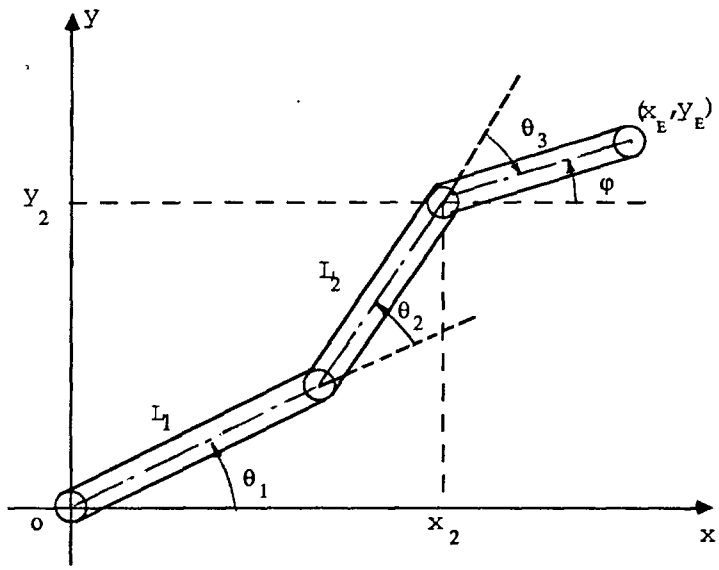


Figure 19. A three-link manipulator

As shown in Figure 19, the orientation of the end effector is denoted by ϕ. Denote the position of the end effector as (x_E, y_E) and the end position of the second link as (x_2, y_2). Then (x_2, y_2) can be calculated from (28):

$$\begin{aligned} x_2 &= x_E - L_3 \times cos\phi, \\ y_2 &= y_E - L_3 \times sin\phi \end{aligned} \tag{28}$$

Substituting x_2, y_2 for x, y in Eqs. (18)-(25), θ_1 and θ_2 can be obtained. θ_3 can be simply calculated from (29).

$$\theta_3 = \phi - (\theta_1 + \theta_2) \tag{29}$$

4.3 Task associated trajectories planning

Having shown in the previous section the methods to plan straight line trajectories for a manipulator with two or three links, now we can consider how to plan trajectories for a free flying robot with two dexterous manipulator arms [Posbergh 86]. This space robot can be used to replace an astronaut to accomplish certain tasks such as

* Mating an electrical connector;
* Replacement of end-effector;
* Unfasten a cover panel, replace an under laying failed instrument unit, close and fasten the cover panel.

To begin the basic planning, the trajectories are planned for the free flyer so that it can accomplish the first task in the list above, i.e., mating an electrical connector. First, different states can be defined for the robot during the process electrical connector as listed in Figure 20.

We utilize straight lines to connect the initial position to final position (destination) and let the robot travels along each straight line segment in one of the states listed in Figure 20. All those straight lines can be divided into two groups:

(1) the lines are to be followed by the center of body;

(2) the lines are to be followed by arms in their local coordinate system. This is illustrated by Figure 21.

As can be seen from Figure 21, the lines $\overline{AB}$, $\overline{BC}$, $\overline{CD}$, $\overline{DA}$ are to be followed by the center of body, while $\overline{EF}$, $\overline{GH}$ must be followed by *arm-2* and *arm-1*, respectively.

Object State Mode	*body*	*arm-1*	*arm-2*
initial state	initial position	folded	folded
move *body* towards connector	move towards the connector	move with *body*, fixed w.r.t. *body*	move with *body*, fixed w.r.t. *body*
move *arm-2* towards connector	no movement	no movement	move towards the connector
grasp connector	no movement	no movement	move two fingers towards connector
move *body* towards holder with connector	move towards the holder	move with *body*, fixed w.r.t. *body*	move with *body*, fixed w.r.t. *body*
move *arm-1* towards holder	no movement	move towards the holder	no movement
grasp holder	no movement	move two fingers to grasp holder	no movement
mate connector with holder	no movement	no movement	move with connector towards holder
move body with mated connector and holder towards the destination	move towards the destination	move with holder fixed w.r.t. *body*	move with holder fixed w.r.t. *body*
release connector and holder	no movement	move two fingers to release the holder	move two fingers to release the connector
return to initial state	move to initial position	folded	folded

Figure 20. States of a Sample Task: Mating connector

For lines in group (1), the required trajectories for the body are:

$$\begin{aligned} x_n &= x_{n-1} + p \times (x_{final} - x_{begin}) \\ y_n &= y_{n-1} + p \times (y_{final} - y_{begin}) \end{aligned} \tag{30}$$

where p is the number of points to divide the line from $(x_{begin},\ y_{begin})$ to $(x_{final},\ y + final)$, and $n = 1, 2, \ldots, p$.

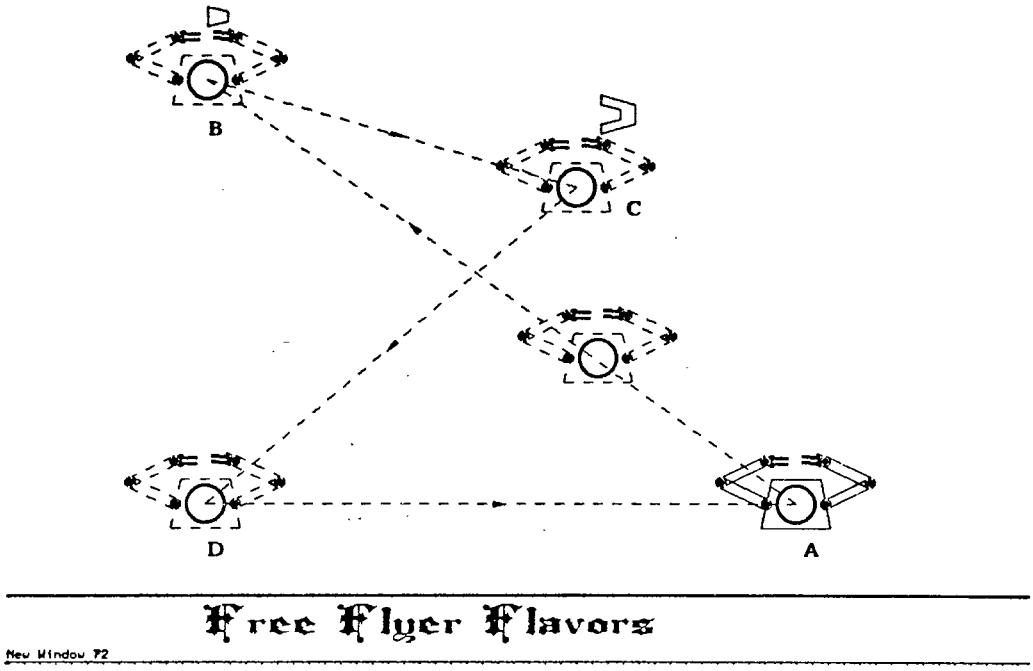


Figure 21. Trajectories for Space robot to mate connector

For those straight lines in group (2), the required joint angles for *arm-1* and *arm-2* can be obtained by using the trajectory planning method for the three-link manipulator mentioned in part (ii) of the previous section. Or more specifically, the straight line paths for *arm-1* can be calculated using the formulations for configuration of "stretch up" gesture; while for *arm-2* using those formulations for configuration of "stretch down" gesture.

Combining all these trajectories, the space robot can be built to accomplish the task, mating electrical connector. The simulation is illustrated in Figure 22.

As shown in Figure 22, the space robot moves from the initial position A towards the connector. After arriving at position B in the vicinity of the

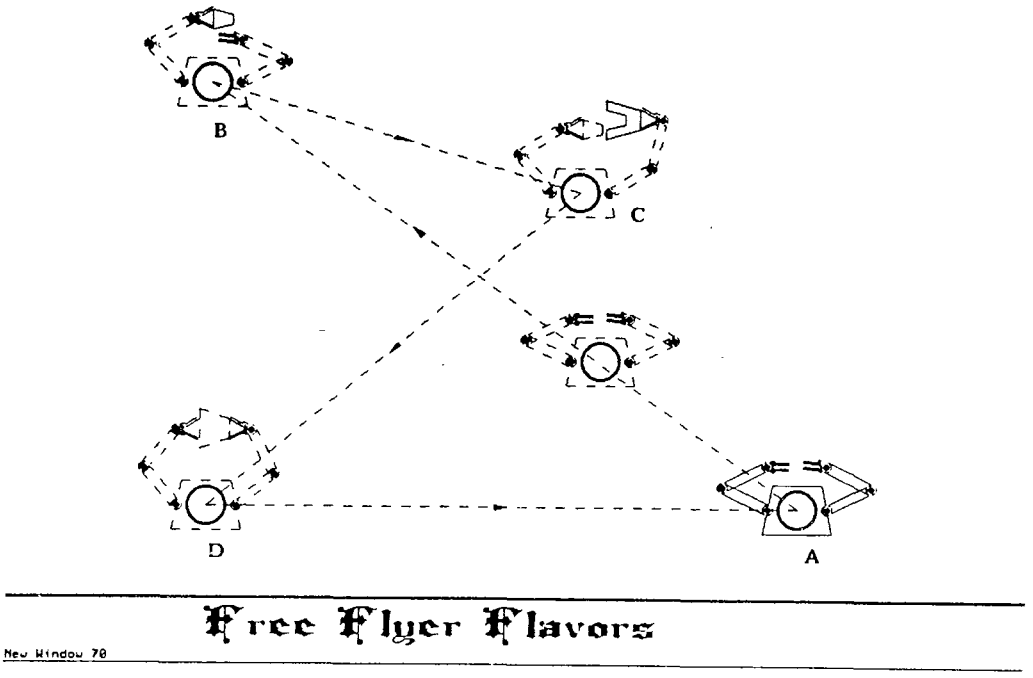


Figure 22. Consecutive Movements for Simulation

connector, the robot grasps the connector using its left arm. Then, the robot moves towards the holder with the connector. When the right arm is in its approaching range, the robot stops at C and stretches its right arm out to grasp the holder. After that, the two arms mate the connector with the holder. Finally, the robot moves towards the destination D with mated connector and holder; releases them and goes back to the initial position A.

CHAPTER
FIVE

DESIGN EXAMPLE

In this chapter, the design of a three-link manipulator system will be outlined by using OOMANEX.

Suppose the structure of the three-link manipulator is given as shown in Figure 19. The design goal is to choose the optimal controller parameters, i.e., the position and velocity feed-back gain matrices so that the dynamical behavior of the manipulator system satisfies certain given constraints and its end effector can travel along a straight line path inside its work space with "small" deviation from the preplanned trajectory.

The design procedure contains following steps:

Step 1: generating the dynamic equations for the three-link manipulator using Symbolic Processor.

To begin with the design, a set of structure data needs to be prepared. These data include the types of the joints; the lengths and masses of the links (either numerically or symbolically); the moment of the link's inertial in the link's local coordinate system; the force or torque applied at each link; and Denavit-Hartenberg parameters which is used to describe the manipulator kinematics (definition of Denavit-Hartenberg parameters and examples of setting up the Denavit-Hartenberg for a specified manipulator can be found in [Sreenath-Krishnaprasad 86]).

From Figure 19, the Denavit-Hartenberg parameters for the three-link planar manipulator (with three revolute joints) are list in Figure 23.

D. H. Parameters Link	θ	α	r	a
1	θ_1	0	0	L_1
2	θ_2	0	0	L_2
3	θ_3	0	0	L_3

Figure 23. Denavit-Hartenberg parameters for the three-link planar manipulator

When Symbolic Processor inquires, the user needs to give the data he prepared one by one according to the questions that Symbolic Processor shows on the screen. Then Symbolic Processor will "translate" the data that the user gave into its internal form and stores in a file for later use.

After obtaining sufficient information from the user, Symbolic Processor generates the dynamic equations for the three-link manipulator system and FORTRAN simulation program upon request.

Step 2: assigning numerical values for the structure variables given symbolically in Step 1 such as the lengths and masses of the links; and specifying the initial positions and velocities of the joint angles.

Data assignment is realized via Numeric Simulator by invoking several *pop—up menu*. When the pop-up menus are brought up one by one, the user can specify the values. The pop-up menus also set up *default* value for each variable needed to specified and let the user change them if he desires. For a novice, these default values can be used to establish a simple simulation process for a manipulator system and obtain hands-on experience about OOMANEX.

Step 3: setting up the objectives which are expected to be satisfied with the controller parameters chosen by Gain Optimizer and specifying the initial guess for the controller parameters.

Since there are three joint angles as the output in the manipulator control system, six functional objectives need to be set up, two for each joint angle. Thus, each joint angle has an upper and a lower limit, and each joint angle needs to be regulated by using the controller optimized by Gain Optimizer so that it lies between the upper and lower curves. For simplicity, a two-step input for each joint angle has been set as shown in Figure 10. where the six objectives are established such that joint angles have the same upper and the same lower curves.

Gain Optimizer can help the user to set up the objectives by listing several questions to allow the user to give a set of data to specify the upper and lower curves quantitatively. Following is an inquiring list which is shown when Gain Optimizer begins to solve for the optimal controller.

```
************************************************************
—— Please give ——
overshoot(% of the magnitude of the input) = 5
upper steady state(% of the magnitude of the input) = 2
rising state(% of the magnitude of the input) = 5
lower steady state(% of the magnitude of the input) = 2
rising time(% of the period for tracing one point) = 3
setting time(% of the period for tracing one point) = 10
************************************************************
```

For the three-link manipulator system to be designed, following matrices are

given as an initial guess:

The position feedback gain matrix is

$$K_p = \begin{pmatrix} 6.25 & 0.0 & 0.0 \\ 0.0 & 6.25 & 0.0 \\ 0.0 & 0.0 & 6.25 \end{pmatrix}$$

The velocity feedback gain matrix is

$$K_v = \begin{pmatrix} 5.0 & 0.0 & 0.0 \\ 0.0 & 5.0 & 0.0 \\ 0.0 & 0.0 & 5.0 \end{pmatrix}$$

After setting up necessary information about the controller to be optimized in a problem description file, Gain Optimizer solves for the controller parameters, presents them on the screen and stores them into a file for later use. Figure 24(a) and (b) show the step response of the first joint angle with the controller given by the user's initial guess. After 16 iterations of optimization, CONSOLE chose the controller containing the position feedback gain matrix

$$K_p^{16} = \begin{pmatrix} 6.434481108e+00 & -2.911914425e-01 & -9.808694992e-01 \\ -1.039298531e+00 & 5.839826945e+00 & -7.152507381e-01 \\ -1.189433573e+00 & -9.458351033e-02 & 6.243905150e+00 \end{pmatrix}$$

and the velocity feedback matrix

$$K_v^{16} = \begin{pmatrix} 6.431257946e+00 & 3.828377657e-01 & 1.383145870e+00 \\ 1.054184961e+00 & 6.093635447e+00 & 1.959748154e+00 \\ 1.092932719e+00 & 1.031120096e+00 & 7.298404457e+00 \end{pmatrix}$$

Using this optimized gain matrices K_p^{16} and K_v^{16}, the time responses of each joint angle is improved so that the constraints given above can be satisfied.

One such example, the time response of the first joint angle using the optimized controller, is illustrated in Figure 24(c) and (d).

Step 4: planning trajectory for the three-link manipulator to travel along a straight line path.

When the user *calls* Trajectory Planner from the OOMANEX's house, it shows a pop-up menu to let the user specify a straight line. If the whole line is inside the workspace of the three-link manipulator, Trajectory Planner shows on the screen the straight line to be traced by the manipulator; the corresponding joint angles of the manipulator which are required so that the end effector of the manipulator can travel along the specified straight line path. If some part of the line is outside of the workspace, error message will be given on the screen to inform the user.

One planning example is illustrated in Figure 25, where the end effector of the three-link manipulator is expected to travel along a straight line from (0.5,0) to (0.5,0.5) (some scaling has been set up so that the user can easily specify the line, where the range for both x and y is (-1,1)). The largest dot in the *Robot Pane* shows the actual path that the end effector travels. *Message Pane* displays the sequence of the angle vector $(\theta_1, \theta_2, \theta_3)$ which need to be achieved by the manipulator to ensure its end effector to be able to travel along the specified straight line.

Step 5: displaying the dynamical simulation.

Having chosen the optimal controller and set up the trajectory for the manipulator to track, the dynamical simulator obtained in Step 1 can now be connected into a closed control loop by using the *partitioned control law*. The dynamical

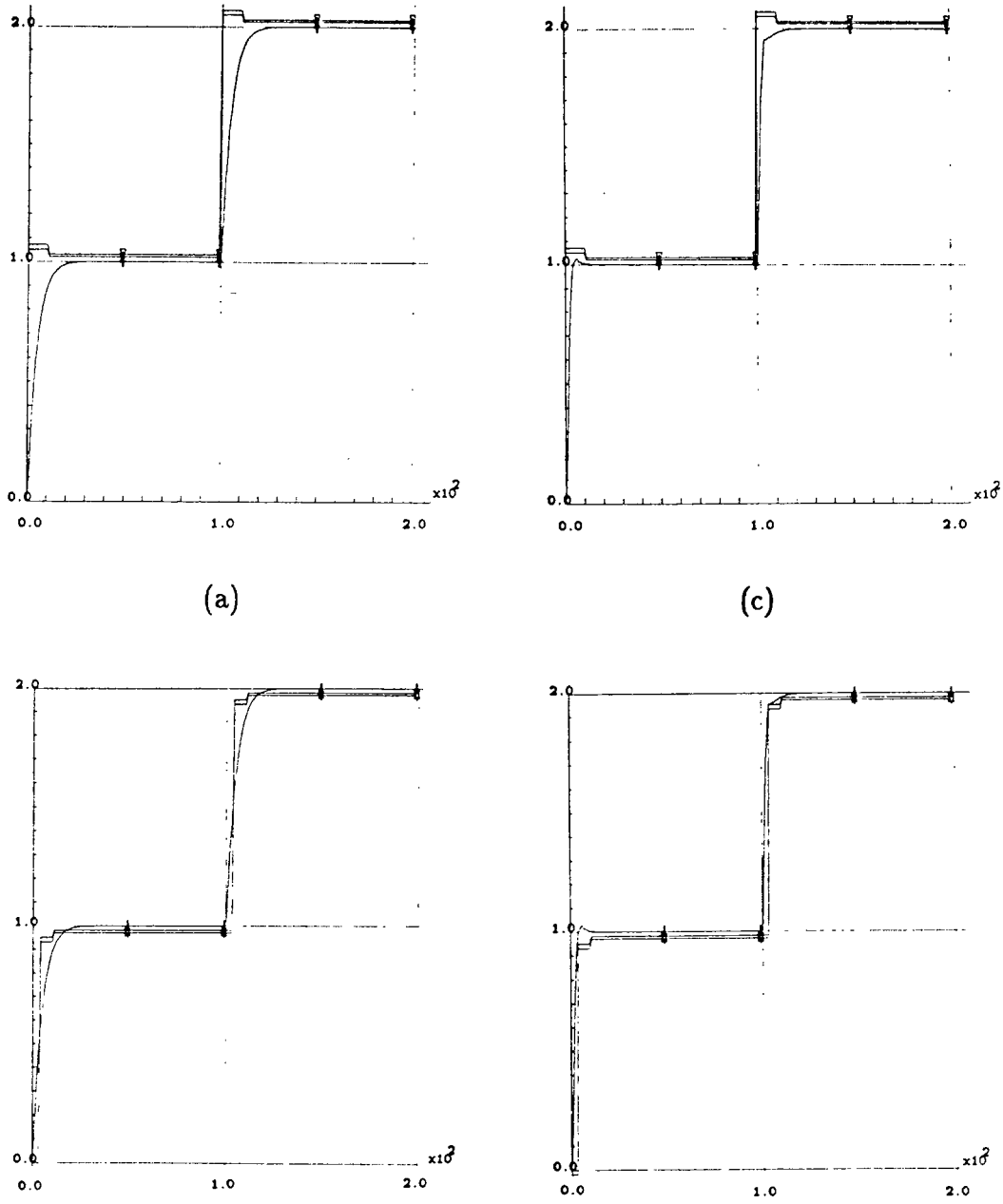


Figure 19. The step response of the first joint angle
(a) Initial x1-upper and Its Good/Bad Curves
(b) Initial x1-lower and Its Good/Bad Curves
(c) Optimized x1-upper and Its Good/Bad Curves
(d) Optimized x1-lower and Its Good/Bad Curves

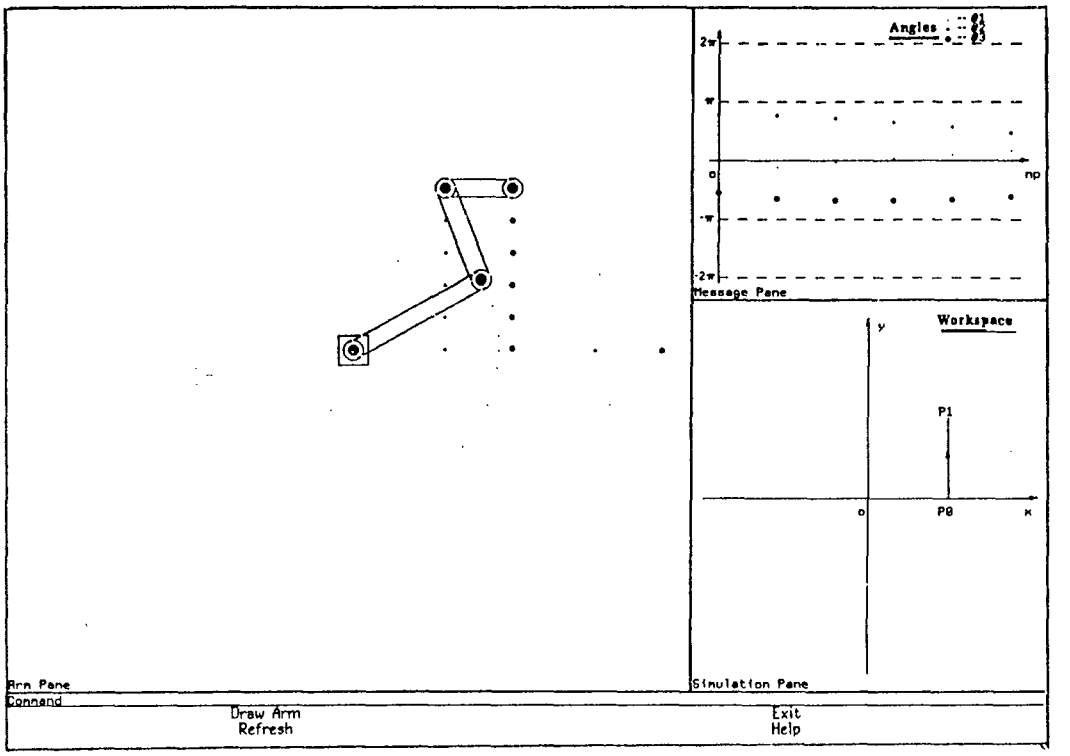


Figure 25. Planning trajectory for the three-link manipulator

behavior of the manipulator system is thus simulated numerically and graphically on the screen while the end effector of the manipulator moves along the straight line path specified by the user. Figure 26 shows one example where the manipulator travels along the preplanned trajectory, a straight line path.

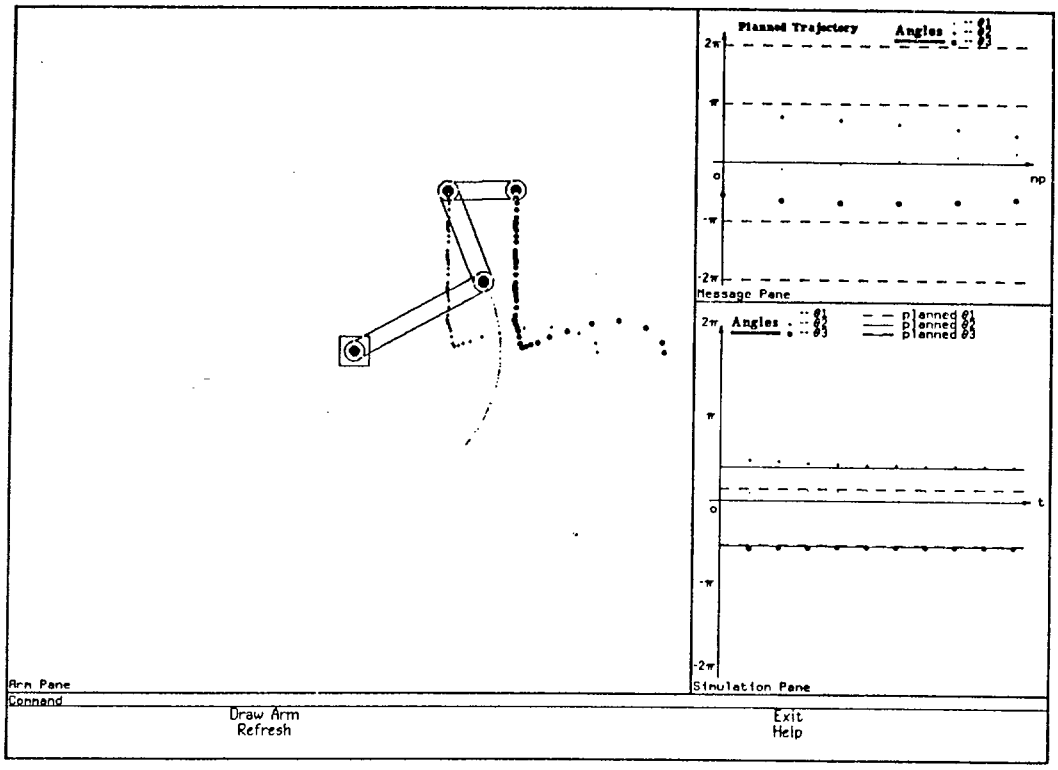


Figure 26. The dynamic simulation of the three-link manipulator

CHAPTER
SIX

CONCLUSION

With the help of OOMANEX, the user can easily create a robot manipulator system and initiate a corresponding simulation process via its graphical representation. Furthermore, OOMANEX can be easily extented to use other control schemes and include other constraints, say, force constraints. Although with the present implementation, OOMANEX only enables us to program the robot manipulator at a level of guiding the manipulator through a simple straight line path (sometimes called *programming at the joint level*) by 'manually' solving the inverse kinematical problem, extension to program at higher level, i.e. *object* or *objective level*, is possible. At the object level not only the robot is considered, but also the rest of the environment. At objective level only the task to be performed by the robot is prescribed and the required robot program is determined by the robot itself. In this case one can speak about *artificial intelligence*.

Bibliography

[Bejczy 74] Bejczy, A. K. "Robot Arm Dynamics and Control", *Tech. Memo 33-669*, Jet Propulsion Laboratory, Feb.1974.

[Brady 82] Brady, M., J. M. Hollerbach, T. L. Johnson, T. Lozano-Pérez, M. T. Mason eds "Robot Motion: Planning and Control", The MIT Press, 1982.

[Cox 86] Cox, B. J. *Object Oriented Programming - An Evolutionary Approach*, Addison-Wesley Publishing Company, 1986.

[Fan 87] Fan, M. K. H., L. Wang, J. Koninckx and A. L. Tits "CONSOLE: A CAD Tandem for Optimization-Based Design Interacting with Arbitrary Simulators" (in preparation).

[Furuta 84] Furuta, K. , K. Kosuge, O. Yamano, K. Nosaki, "Robust control of a robot manipulator with nonlinearity", *Robotica*, Vol.2, pp.75-81, 1984.

[Keahler 86] Keahler, T. and D. Patterson "A Small Taste of Smalltalk" *Byte*, August 1986, pp.145-159.

[Kawamura 86] Kawamura, K., G. Beale, J. Schaffer, B.-J. Hsieh, S. Padalker, J. Rodriguez-Moscoso, F. Vinz and K. Fernandez "Development of a Coupled Expert System for Spacecraft Attitude Control Problem", in: *Coupling Symbolic and Numerical Computing in Expert System* edited by J. S. Kowalik, Elsevier Science Publishers B. V. (North-Holland), 1986.

[Lee 82] Lee, C.S.G. "Robot Arm Kinematics, Dynamics and Control", *IEEE Computer*, December 1982.

[Luh 80(1)] Luh, J. Y. S., M. W. Walker and R. P. C. Paul, "On-Line Computational Scheme for Mechanical Manipulators", *Journal of Dynamic Systems, Measurement, and Control*, Vol.102, pp.69-76, June 1980.

[Luh 80(2)] Luh, J. Y. S., M. W. Walker and R. P. C. Paul, "Resolved-Acceleration Control of Mechanical Manipulators", *IEEE Trans. Auto. Contr.*, Vol. AC-25, No. 3, June 1980.

[Markiewicz 73] Markiewicz, B. R. "Analysis of the Computed Torque Drive Method and Comparision with Conventional Position Servo for a Computer-Controlled Manipulator", *Technical Memorandum 33-601*, Jet Propulsion Laboratory, Mar 1973.

[Paul 72] Paul, R. P., "Modeling, Trajectory Calculation and Servoing of a Computer Controlled Arm", *A.I. Memo 177*, Stanford Artificial Intelligence Laboratory, Stanford University, Sept. 1972.

[Posbergh 86] Posbergh, T. A. "Design Consideration for a Free Flying Space Robot", *Final Report NASA/USRA Summer Project 1986* (NASA/USRA Contract 01-4-33055), College Park, Maryland, 1986.

[Sahar 85] Sahar, G. and J. M. Hollerbach "Planning of Minimum-Time Trajectories for Robot Arms" in: *IEEE International Conference on Robotics and Automation*, 1985.

[Sreenath-Krishnaprasad 86] Sreenath, N. and P. S. Krishnaprasad, "DYNAMAN: A Tool for Manipulator Design and Analysis", IEEE Conf. on Robotics & Automation, San Fransisco, CA., April 7-11, 1986.

[Tompkins 86] Tompkins, J. W. "Using An Object-oriented Environment to Enable Expert Systems To Deal With Existing Problems", in: *Coupling Symbolic and Numerical Computing in Expert System* edited by J. S. Kowalik, Elsevier Science Publishers B. V. (North-Holland), 1986

[Vukobrato 82] Vukobratovic, M. and D. Stokic, *Control of Manipulation Robots — Theory and Applocation*, Spring-Verlag, 1982

[Vukobrato 85] Vukobratovic, M. and N. Kircanski, *Real-time Dynamics of Manipulation Robots*, Spring-Verlag, 1985

[Symbolics 85a] *Reference Guide to Symbolics – Lisp, Symbolics*^{tm}, Inc., Cambridge, MA (1985)

[Symbolics 85b] *Programming the User Interface, Symbolics*^{tm}, Inc., Cambridge, MA (1985)

the user can give to OOMANEX. Then the user can click on left button of the mouse to choose one of the items in the command list.

Each command the user gives invokes a new frame. On each frame, there is a *command* pane which contains several command *buttons*. Moving the cursor over each command shows a box around that button. Then click right button of the mouse to bring up a menu and click left button of the mouse to choose one of the item listed on the menu.

For on-line information, the user can use *Help* button on each frame. Clicking on left button on *Exit* button quits from the current frame.

Appendix A: OOMANEX User's Guide

1. Loading OOMANEX System

1.1. Login on Symbolics

In Lisp Listener, type *login user-name*. Example:

command: *login xin*

1.2. Load OOMANEX System

After logging in on Symbolics, one needs to load all binary code of OOMANEX into the current programming environment. In Lisp Listener, type

command: *load system oomanex*.

1.3. Invoke OOMANEX System

User can invoke OOMANEX system in three ways:

(a) Hit < *Select* > key, then type *B*;

(b) Hit < *Function* > key, then type *B*;

(c) Click twice on the right button of the mouse to bring up the *System Menu* of the Symbolics, then move the cursor over *OOMANEX System* of the menu and click on the left button of the mouse.

2. Using Menus and Mouse

After the user invokes the OOMANEX system, the OOMANEX's house appears on the screen. The windows and the door are *mouse sensitive*, i.e. when moving the cursor over these areas, a rectangular box will appear. Clicking on the right button of the mouse brings up a menu which is a list of commands that

Appendix B: Program Development for OOMANEX

OOMANEX divides its program into two parts on two machines: on a Symbolics 3600 and on a VAX 11/785 running 4.3BSD UNIX. The gain optimization module is on VAX, other modules are on Symbolics.

Part I: On Symbolics

1. Organization

OOMANEX has following system structure:

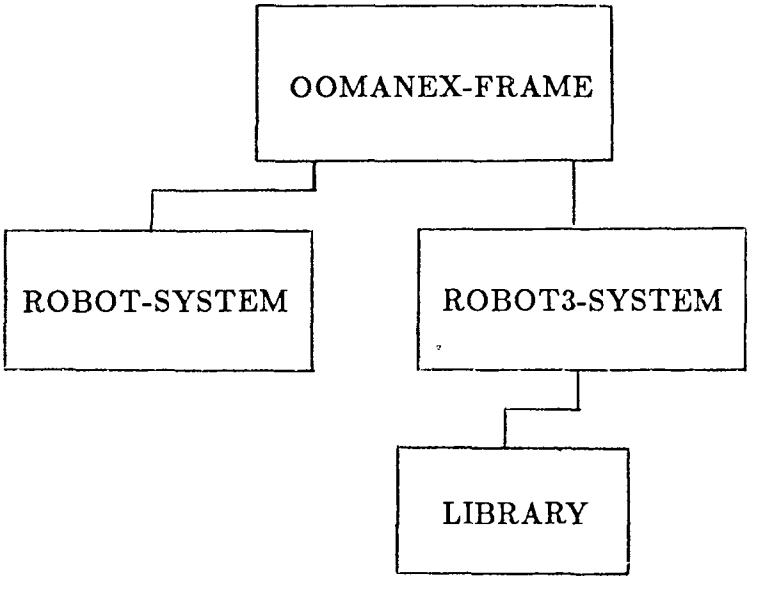


Figure 1. System structure of OOMANEX

In Figure 1, OOMANEX-FRAME is the user interface of OOMANEX, ROBOT-SYSTEM is a subsystem for simulation of a planar manipulator with its two links, ROBOT3-SYSTEM is a subsystem for simulation of a three-link planer manipulator, and LIBRARY contains some FORTRAN subroutines to be used in ROBOT3-SYSTEM.

OOMANEX-FRAME	ROBOT-SYSTEM	ROBOT3-SYSTEM
oomanex-frame.lisp	arm-calculation.lisp	arm3-ref.lisp
	arm-output.lisp	get-simulation.lisp
		plan-simulation.lisp
		arm3-simulation.fortran
		top-level.lisp

Table 1. The files in OOMANEX

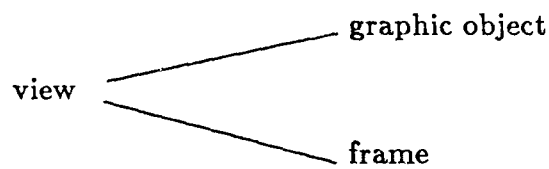
OOMANEX was developed on a Symbolics 3600, which is used as input, calculation and display device. Accordingly, functions and flavors in OOMANEX were divided into following three classes:

(1) *menu* to obtain information or commands from the user;

(2) *model* to accomplish all kinds of calculation;

(3) *view* to display information in alpha-numerics and pictures on screen.

Further, "view" class can be divided as:



Each subsystem in Figure 1 has functions or subroutines belonging to at least one class above. However, the calssification for all functions is not very rigorous sometime. For example, some function may do certain calculation, but at the same time display the results on screen. The designer tried to make this classification as clear as possible. Nevertheless, some mixtures may still remain.

All Zeta-Lisp functions and flavors are listed in Table 2, 3, and 4.

2. Sample Code

Following code is taken from ROBOT-SYSTEM.

2.1. Flavor and Method

In Zeta-Lisp, Flavors are created with **defflavor** *special form*; and methods are created with the **defmethod** *special form*. Here, *special form* works in a manner similar to usual functions or subroutines in a procedural language, but differs in the way each special form is defined, since a function usually follows a simple set of rules; each special form is idiosyncratic.

More precisely, a flavor is defined by a form such as

(**defflavor** *flavor-name* (*var*1 *var*2...) (*flav*1 *flav*2...)
*opt*1 *opt*2...)

where *flavor-name* is a symbol that serves to name the flavor; *var*1, *var*2, and so on, are the names of the instance variables containing the local state for this flavor; *flav*1, *flav*2, and so on, are the names of the component flavors out of which this flavor is built; *opt*1, *opt*2, and so on, are options for handling the instance variables.

A method is defined as

(**defmethod** *flavor-name method-type message*) *lambda-list*
*form*1 *form*2...)

where *flavor-name* is a symbol that is the name of the flavor that is to receive the method, *method-type* is a keyword symbol for the type of method; it is omitted when defining a primary method, which is the usual case, *message* is a keyword symbol that names the message to be handled, *lambda-list* describes the arguments and "aux variables" of the function, *form*1, *form*2, and so on, are the function body; the value of the last form is returned.

In ROBOT-SYSTEM, a flavor "link" was created as follows:

```
(defflavor link
        (length
        angle
        origin-x
        origin-y)
        ()
    :gettable-instance-variables
    :settable-instance-variables
    :initable-instance-variables)

(defmethod (link :end-x) (x)
        (floor (+ origin-x (* (cos x) length))))

(defmethod (link :end-y) (x)
        (floor (- origin-y (* (sin x) length))))

(defmethod (link :compute-link-points) (x)
        (let* ((diff-from-origin-x (floor (* (sin x) *large-circle-radius*)))
          ( diff-from-origin-y (floor (* (cos x) *large-circle-radius*)))
          (l1-1-x (+ origin-x diff-from-origin-x))
          (l1-1-y (+ origin-y diff-from-origin-y))
          (l1-2-x (+ (send self ':end-x x) diff-from-origin-x))
          (l1-2-y (+ (send self ':end-y x) diff-from-origin-y))
          (l2-1-x (- origin-x diff-from-origin-x))
          (l2-1-y (- origin-y diff-from-origin-y))
          (l2-2-x (- (send self ':end-x x) diff-from-origin-x))
          (l2-2-y (- (send self ':end-y x) diff-from-origin-y)))
    (values l1-1-x l1-1-y l1-2-x l1-2-y l2-1-x l2-1-y l2-2-x l2-2-y)))
```

The methods "end-x" and "end-y" calculate the x- and y-position of the link end, which are called by another method "compute-link-points" sending *self* message. Later, after making an instance of flavor "link", say, link-1, and requesting the x-position of its end, we can send the message "end-x" as

```
(send link-1 ':end-x θ - 1)
```

More complicated methods can be defined according to the application.

After creation for all the necessary flavors, a two-link arm can be constructed by instantiating suitable flavors. Here, the arm has two links. Denote the first link as *arm-1* and the second link *arm-2*. The links can be declared to be the instances of flavor 'link' as following:

```
(setq *arm-1* (make-instance 'link :length *arm-1-length*
                                   :origin-x *center-x*
                                   :origin-y *center-y*))

(setq *arm-2* (make-instance 'link :length *arm-2-length*))
```

2.2. Function

Functions are defined by using a *special form* defun. A defun form looks like:

(defun *name lambda − list*

body ...)

name is the function name. The *lambda − list* is a list of the names to give to the arguments of the function. *body* is the function body. The function named *name* returns the value of the last form in its function body.

One example is the "refresh" function as following:

```
;;;
;;; ;;; refresh function
;;;
(defun refresh-pane ()
   (let (a)
     (setq a
          (tv:multiple-choose
            "Refresh Pane" selection-item-list
            selection-keyword-alist))
```

```
(if (equal (cadar a) 'Yes) (send *simulation-pane* ':clear-window))
(if (equal (cadadr a) 'Yes) (send *message-pane* ':clear-window))
(if (equal (cadar (cddr a)) 'Yes) (send *r-window* ':clear-window))
(if (equal (cadar (cdddr a)) 'Yes) (send *robot-system-frame* ':refresh))))
```

2.3. Menu

The window system for the Lisp Machine contains a variety of facilities to allow the user to make choices interactively. These all work by displaying some arrangement of items in a window. By pointing to an item with the *mouse* and pressing a mouse button, the user selects the item.

In OOMANEX, following types of menus were used : pop-up, momentary, command, multiple choice, choose variable values, user options and mouse-sensitive items and areas.

An example of menu is described as following, which defines a multiple choice menu. This menu can be invoked by calling the "refresh" function above.

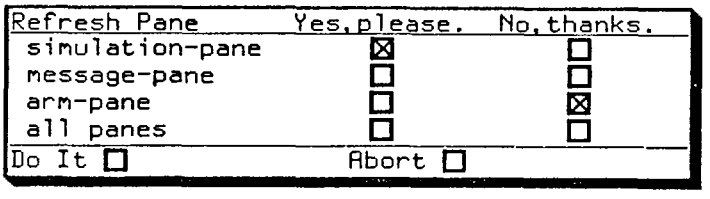


Figure 1. The Refresh Menu

```
(defvar refresh-choices )

(defvar selection-item-list)
(defvar selection-keyword-alist)

(setq refresh-choices '(Yes No)

(setq selection-item-list
    (list (list 1 " simulation-pane" refresh-choices)
        (list 2 " message-pane" refresh-choices)
```

```
                (list 3 " robot-pane" refresh-choices)
                (list 4 " rs-frame" refresh-choices)))

(setq selection-keyword-alist
        (list '(Yes "Yes,please. ")
              '(No "No,thanks. ")))
```

2.4. Frame

A *frame* is a window that is designed to contain other windows inside it. A direct inferior window of a frame is called a *pane*. Many activities consist of a frame and its direct and indirect inferior windows. The frame is the representative window of this kind of activity.

For example, a frame *robot-system-frame* was defined for ROBOT-SYSTEM as:

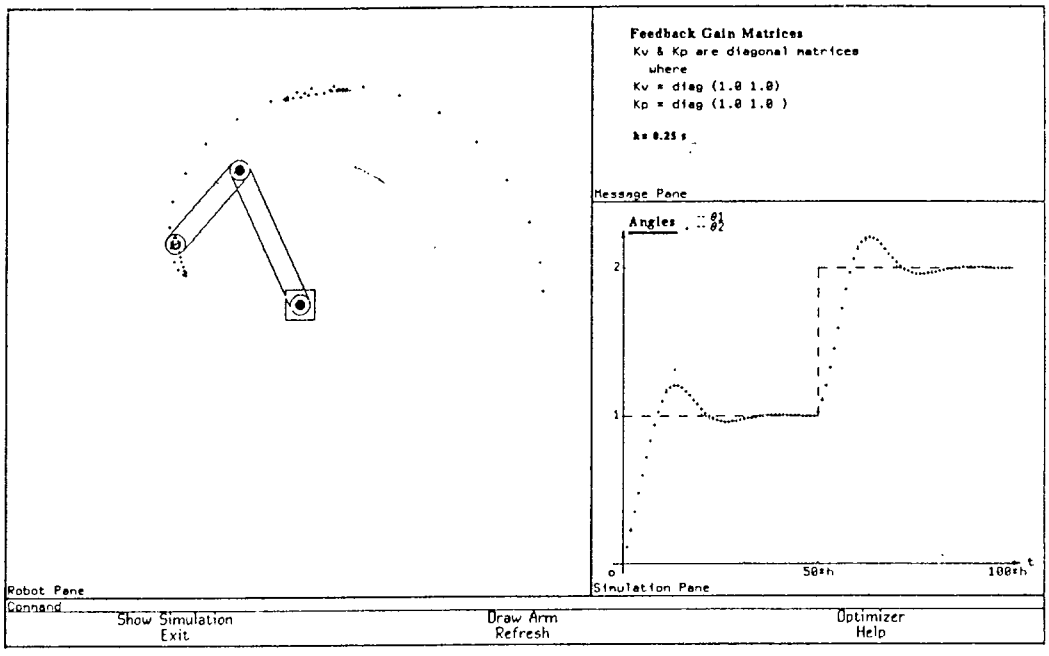


Figure 2. The Frame of ROBOT-SYSTEM

```
;;; Define the frame and its panes
(setq *robot-system-frame*
```

```
(tv:make-window 'tv:bordered-constraint-frame-with-shared-io-buffer
    ;; Select the graphics pane when it is exposed
    ':selected-pane 'graphics-pane
    ;; Specify the panes
    ':panes
     '((graphics-pane tv:window
                      :blinker-p nil
                      :label "Robot Pane")
      (simulation-pane tv:window-pane
                      :blinker-p nil
                      :label "Simulation Pane" )
      (message-pane tv:window-pane
                      :blinker-p nil
                      :label "Message Pane" )
      (interaction-pane tv:window-pane
                      :blinker-p t
                      :label "Interaction Pane")
      (menu-pane tv:command-menu-pane
                 :label "Command"
                 :item-list
                 (("Show Simulation" :value :show-simulation
                 :documentation "Show Simulation")
                 ("Draw Arm" :value :draw-arm
                 :documentation "Draw Arm")
                 ("Optimizer" :value :optimizer
                 :documentation "Call Optimizer")
                 ("Exit" :value :exit
                 :documentation "Exit this frame.")
                 ("Refresh" :value :refresh-p
                 :documentation "Refresh")
                 ("Help" :value :help-p
                 :documentation "Information"))))
     ':constraints
     '((main . ((top-strip menu-pane interaction-pane)
                ((top-strip :horizontal (600)
                   (graphics-pane column-strip)
```

```
        (graphics-pane 600 )
        (column-strip :vertical (465)
        (message-pane simulation-pane)
        ((message-pane 200)
        (simulation-pane 400))))))
    ((menu-pane 2 :lines ))
    ((interaction-pane :even)))))))
```

After defining the frame, the function $set-frame$ was created to invoke the frame.

```
(defun set-frame ()
  ;; Get access to the panes
  (setq *r-window*
        (send *robot-system-frame* ':get-pane 'graphics-pane))
  (setq *simulation-pane*
        (send *robot-system-frame* ':get-pane 'simulation-pane))
  (setq *vax-pane*
        (send *robot-system-frame* ':get-pane 'simulation-pane))
  (setq *interactive-pane*
        (send *robot-system-frame* ':get-pane 'interactive-pane))
  (setq *message-pane*
        (send *robot-system-frame* ':get-pane 'message-pane))
  (setq *menu-pane*
        (send *robot-system-frame* ':get-pane 'menu-pane))
  (send *robot-system-frame* :expose)
  (set-frame-info)
  ;; blip holds the list returned by :any-tyi
  (loop as blip = (send *r-window* ':any-tyi)
        as result-value =
          (cond ((and (listp blip) (eq (car blip) ':menu))
                 send (fourth blip) ':execute (second blip)))
                 (t nil)) ;just ignore keyboard input
        do
  ;; Check the value and draw the appropriate object
  (selectq result-value
```

```
(:draw-arm
  (set-up))
(:show-simulation
  (show-simu))
(:optimizer
  (call-vax))
(:exit
  (send *robot-system-frame* ':deactivate))
(:refresh-p
  (refresh-pane))
(:help-p
  (on-line-info *message-pane*)))))
```

2.5. System

When a program gets large, it is often good to split it up into several files. One reason is to keep the parts of the program organized, to make things easier to find. Another is that programs broken into small pieces are more convenient to edit and compile. It is particularly important to avoid the need to recompile all of a large program every time any small piece of it changes. On Symbolics, *system facility* provides a way to maintain and manage a large program of many files efficiently, similar to *make* used to maintain program facility under UNIS-VAX.

A *system* is a set of files and a set of rules and procedures that defines the relations among these files; together these files, rules, and procedures constitute a complete program. These files and relationships are defined by the system facility's **defsystem** special form. Another two files , *system* – *name*.system and *system* – *name*.translations, must be created to make the system site-independent. Both files should be put in Sys:site directory of a Symbolics on which the system is created.

Following example illustrates how the system ROBOT-SYSTEM was created:

```
;;; -*- Mode: LISP; Package:USER; Syntax :Zeta-Lisp; Base :10 -*-
;;; Created 8/01/87 by xin

;;; definition of the robot-system package
(defpackage robot-system (:nicknames RS))

;;; definition of the prime system
(defsystem robot
   (:package "robot-system")
   (:name "Robot System")
   (:pathname-default "syms:¿xin¿ROBOT¿arm2¿")
   (:bug-reports "robot" "Report problems with the Robot System.")
   (:patchable)
   (:not-in-disk-label)

   (:module info "arm-information")
   (:module macros "arm-output.lisp")
   (:module main "arm-calculation.lisp")

   ;; transformation should be performed on each module.
   (:compile-load info)
   (:compile-load macros)

   (:compile-load main))
```

The .system file contains:

```
(fs:make-logical-pathname-host "robot-system")
(si:set-system-source-file "robot"
                           "robot-system:robot-system;robot-sytem")
```

The .translations file looks like:

```
(fs:set-logical-pathname-host "robot-system"
:physical-host "syms"
```

```
:translations '(("robot-system;" "¿xin¿robot¿arm2¿"))
```

Part II: On VAX

The gain optimization part of OOMANEX on VAX consists of problem description files and simulation files. The problem description file is written in the syntax of CONSOLE. The simulation files are FORTRAN programs for simulation the dynamic behavior of a robot manipulator, which are generated by the Symbolic Processor (DYNAMAN).

An example is illustrated by the gain optimization program for a two-link planar manipulator as follows.

(1) The problem description file (part):

```
design_parameter kp11 init=5. max=12
design_parameter kp12 init=0. max=12
design_parameter kp21 init=0. max=12
design_parameter kp22 init=5. max=12

design_parameter kv11 init=5. max=10
design_parameter kv12 init=0. max=10
design_parameter kv21 init=0. max=10
design_parameter kv22 init=5. max=10

functional_objective "x1-upper"

    for u from 0 to 199 by 1

    minimize {
        import kp11 kp12 kp21 kp22 kv11 kv12 kv21 kv22
        double kv[2][2], kp[2][2];
        extern double x1[];
        int i;
```

```
        kp[0][0] = kp11;
        kp[0][1] = kp12;
        kp[1][0] = kp21;
        kp[1][1] = kp22;

        kv[0][0] = kv11;
        kv[0][1] = kv12;
        kv[1][0] = kv21;
        kv[1][1] = kv22;

        i = u + 0.5;
        j2(kv, kp);
        return x1[i];
        }
    good_curve = {
        int i;
        i = u+0.5;
        if (i ≥ 0 && i ≤ 10) return 1.05;
        if (i ≥ 11 && i ≤ 99) return 1.02;
        if (i ≥ 100 && i ≤ 110) return 2.05;
        if (i ≥ 111 && i ≤ 199) return 2.02;
        }
    bad_curve = {
        int i;
        i = u+0.5;
        if (i ≥ 0 && i ≤ 10) return 1.07;
        if (i ≥ 11 && i ≤ 99) return 1.03;
        if (i ≥ 100 && i ≤ 110) return 2.07;
        if (i ≥ 111 && i ≤ 199) return 2.03;
        }

functional_objective "x1-lower"

    for u from 0 to 199 by 1

    maximize {
```

```
    import kp11 kp12 kp21 kp22 kv11 kv12 kv21 kv22
    double kv[2][2], kp[2][2];
    extern double x1[];
    int i;

    kp[0][0] = kp11;
    kp[0][1] = kp12;
    kp[1][0] = kp21;
    kp[1][1] = kp22;

    kv[0][0] = kv11;
    kv[0][1] = kv12;
    kv[1][0] = kv21;
    kv[1][1] = kv22;

    i = u + 0.5;
    j2(kv, kp);
    return x1[i];
    }
good_curve = {
    int i;
    i = u+0.5;
    if (i ≥ 0 && i ≤ 3) return 0.00;
    if (i ≥ 3 && i ≤ 10) return 0.95;
    if (i ≥ 11 && i ≤ 99) return 0.98;
    if (i ≥ 100 && i ≤ 103) return 0.98;
    if (i ≥ 103 && i ≤ 110) return 1.95;
    if (i ≥ 111 && i ≤ 199) return 1.98;
    }
bad_curve = {
    int i;
    i = u+0.5;
    if (i ≥ 0 && i ≤ 3) return -.02;
    if (i ≥ 3 && i ≤ 10) return 0.93;
    if (i ≥ 11 && i ≤ 99) return 0.97;
    if (i ≥ 100 && i ≤ 103) return 0.97;
```

```
if (i ≥ 103 && i ≤ 110) return 1.93;
if (i ≥ 111 && i ≤ 199) return 1.97;
}
```

(2) The FORTRAN simulation program (part):

```
c     ------------------------------
c     J2
c     ------------------------------
      subroutine j2(kv,kp,x1,x2)

      PARAMETER(NLIN = 2)

      double precision x(nlin),dx(nlin)
      double precision xs(nlin),dxs(nlin) ,ddxs(nlin)
      double precision x1(200), x2(200)
      double precision tempx1(100),tempx2(100)
      double precision kv(1),kp(1)

      integer i,j, k
      common/pos/x,dx
      common/vel/xs,dxs ,ddxs

      open(7,file='init2')
c     Read the initial conditions from 'init2'
      do 1 i=1,nlin
         read(7,*)x(i)
         read(7,*)dx(i)
1     continue
      close(7)

      open(8,file='ref2')
      do 30 j=1,nlin
c     Read the references point j from 'ref2'
      do 10 i=1,nlin
         read(8,*)xs(i)
```

```
            read(8,*)dxs(i)
            read(8,*)ddxs(i)
10      continue

c       Call the simulator
        call j2simu(kv,kp,tempx1,tempx2)
        do 20 i = 1,100
            k = i + (j-1)*100
            x1(k) = tempx1(i)
            x2(k) = tempx2(i)
20      continue
30      continue

        close(8)
        return
        end

c       ------------------------------------------
c       TQ1 calculates the 1st part of control torque.
c       ------------------------------------------
        subroutine TQ1(x,dx,ddx,tm)
        PARAMETER(NLIN = 2)
        DOUBLE PRECISION L1, L2, M1, M2
        double precision x(nlin),dx(nlin),ddx(nlin)
        double precision tm(nlin)
        COMMON/PARAM/L1, L2, M1, M2
c
      TM(1) = (((2*DDX(2)+2*DDX(1))*L2**2+((-3*DX(2)**2-6*DX(1)*DX(2))*S
     1 IN(X(2))+(3*DDX(2)+6*DDX(1))*COS(X(2)))*L1*L2+6*DDX(1)*L1**2)*M
     2 2+2*DDX(1)*L1**2*M1)/6.0
                                                          TM(2)      =
((2*DDX(2)+2*DDX(1))*L2**2+(3*DX(1)**2*SIN(X(2))+3*DDX(1)*
     1 COS(X(2)))*L1*L2)*M2/6.0
c
        return
        end
```

	menu	model		view
			graphic object	frame
Function	symbolic			make-up-oomanex--window
	numerical			set-up-oomanex-window
	optimal			draw-house
	trajectory-3			quit
	graphics-2			set-up-lisp-listner
	graphics-3			back-to-lisp--listner
	help			oomanex
Flavor				oomanex-pane
				o-frame

Table 2. Functions and flavors in OOMANEX-FRAME

	menu	model	view	
			graphic	frame
			object	
Function	operate-r	trajectory	base	set-frame
	old-trec		turn-arms	set-frame-info
	new-trec		show	set-telnet-frame
	operate-s		set-up	call-vax
	data-s		show-simu	
	program-s		show-title	
			show-step-size	
			show-control-	
			-parameters	
			draw-simu-coord	
			on-line-info	
Flavor			link	graphics-frame
(method)			(compute-link-	
			-points	
			draw-link	
			draw-dashed-link	
			end-x	
			end-y)	

Table 3. Functions and flavors in ROBOT-SYSTEM

	menu	model	view	
			graphic object	frame
Function	get-init	trajectory	base3	set-r3-frame
	get-structure-data	find-theta-1	turn-arm3	
	get-init-data	cal	clean-arm3	
	get-matrix-type	store-theta	arm-home-position	
	get-diagonal--position-gain	traj-for-plan	draw-coord-for-simu	
	get-symmetric--position-gain	cal-for-plan	show-desired-traj	
	get-general--position-gain	plan-traj	on-line-info	
	get-diagonal--velocity-gain	r3	plan-traj-info	
	get-symmetric--velocity-gain	store-data	r3-simu-info	
	get-general--velocity-gain	run-simulation	draw-planned-traj	
	refresh-pane	wait	draw-coord-for-dyn-simu	
			set-up r3-simu	draw-dyn-data
			get-y-coord	
Flavor (method)			link3	r3-frame
			(draw-link3	
			end-x	
			end-y)	

Table 4. Functions and flavors in ROBOT3-SYSTEM

CURRICULUM VITAE

Name: Xin Chen

Permanent Address: c/o Prof. Zhongyi Chen
The Management Research Group
Xianlie-zhong Road, No. 99
Canton, Kwangtung
P. R. China

Degree and date to be conferred: M.S., August 1987.

Date of birth: May 30, 1960

Place of birth: Jiangxi, P.R. China

Secondary education: Middle School affiliated to Huazhong (Central China) University of Science and Technology, Wuhan, Hubei, P. R. China

Collegiate institutions attended:	Dates	Degree	Date of Degree
University of Maryland College Park, Maryland	09/85 - 08/87	M.S.	08/87
Technical University of Braunschweig Braunschweig, West Germany	10/83 - 05/85	-	05/85
Jiao-Tong University Shanghai, P. R. China	09/78 - 07/82	B.S.	07/82

Major: Electrical Engineering.