July 1988

A Distributed System for Robot Manipulator Control

Richard P. Paul
University of Pennsylvania

Hong Zhang
University of Pennsylvania

Minoru Hashimoto
University of Pennsylvania

Alberto Izaguirre
University of Pennsylvania

Jeffrey Trinkle
University of Pennsylvania

See next page for additional authors

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-49.

A Distributed System for Robot Manipulator Control

Abstract

This is the final report representing three years of work under the current grant. This work was directed to the development of a distributed computer architecture to function as a force and motion server to a robot system. In the course of this work we developed a compliant contact sensor to provide for transitions between position and force control; we have developed an end-effector capable of securing a stable grasp on an object and a theory of grasping; we have built a controller which minimizes control delays, and are currently achieving delays of the order of five milliseconds, with sample rates of 200 hertz; we have developed parallel kinematics algorithms for the controller; we have developed a consistent approach to the definition of motion both in joint coordinates and in Cartesian coordinates; we have developed a symbolic simplification software package to generate the dynamics equations of a manipulator such that the calculations may be split between background and foreground.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-49.

Author(s)

Richard P. Paul, Hong Zhang, Minoru Hashimoto, Alberto Izaguirre, Jeffrey Trinkle, Nathan Ulrich, Yangsheng Xu, and Yehong Zhang

A DISTRIBUTED SYSTEM FOR ROBOT MANIPULATOR CONTROL NSF GRANT ECS84-11879 FINAL REPORT

MS-CIS-88-49
GRASP LAB 149

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

July 1988

A Distributed System for Robot Manipulator Control NSF Grant ECS84-11879 Final Report

Richard P. Paul
Hong Zhang Minoru Hashimoto Alberto Izaguirre
Jeffrey Trinkle Nathan Ulrich Yangsheng Xu
Yehong Zhang

The University of Pennsylvania
Moore School
Philadelphia PA 19104

ABSTRACT

This is the final report representing three years of work under the current grant. This work was directed to the development of a distributed computer architecture to function as a force and motion server to a robot system. In the course of this work we developed a compliant contact sensor to provide for transitions between position and force control; we have developed an end-effector capable of securing a stable grasp on an object and a theory of grasping; we have built a controller which minimizes control delays, and are currently achieving delays of the order of five milliseconds, with sample rates of 200 hertz; we have developed parallel kinematics algorithms for the controller; we have developed a consistent approach to the definition of motion both in joint coordinates and in Cartesian coordinates; we have developed a symbolic simplification software package to generate the dynamics equations of a manipulator such that the calculations may be split between background and foreground.

Contents

1 PAST RESEARCH

Research during the prior years of this grant related to the problems of multi-sensor control of robots, sensor fusion, and grasp planning. A distributed computing architecture was proposed in which sensors and actuation controllers run on separate processors coupled together by a network and supervised by a coordinator. The coordinator used Bayesian techniques to cluster sensor observations and to provide a robust estimate of environment state.

Problems of grasp planning were considered together with the design of a new three fingered hand of medium complexity. A contact sensor was developed which was to provide contact detection information, compliance during contact, and relative end-effector displacement.

Actuation was handled by a special purpose, concurrent processor which provided for both force and motion control. The aim of this processor was to remove the computational limitations on manipulator performance. This current system involves delays of the order of 5 milliseconds between changes in Cartesian coordinates and a response at the manipulator actuator level. A number of software and hardware tools were developed in the course of this work. Algorithms were carefully studied in order to reduce the real time complexity of manipulator control.

Documentation relating to this work was as follows:

- the integration, coordination, and control of multi-sensor systems [1] [2] [3] [4] [5] [6];
- the planning of grasps [7] [8];
- the initial development of the distributed force and motion server [9] [10];
- research on general manipulation and dynamics [11] [12] [13] [14] [15].

2 CURRENT RESEARCH

During this final year of the Grant we have, in the absence of the availability of the high performance Hughes Systolic/Cellular Array Processor concentrated on developing the robotics applications for which it will be used. These applications require substantially more computing capability that can be obtained by any reasonable practical machine available. We have, however, been concerned with the array processor host and manipulator interface, itself a significant computational problem. This proposed host system has been used to demonstrate many of our ideas.

Problems of robot manipulation in unstructured environments may be viewed in terms of the manipulator's wrist: the interface between the end-effector and the manipulator. The end-effector contacts the environment, grasps objects, and with a grasped objects contacts the environment. The manipulator is used to position and to orient the wrist and to exert forces and moments on the wrist so as to apply a required force or moment on the environment.

In terms of the wrist we must be concerned with:

- the orienting and positioning of the end-effector along with its ability to make arbitrary translations and rotations from any given initial position and orientation.

- making contact and breaking contact with the environment, in which case, impulsive forces and discontinuities in force will occur; the manipulator must be shielded from these potentially destructive forces.

- the grasping of objects, the establishment of stable grasps at the appropriate position and orientation on the object so that necessary manipulative surfaces of features of the object are exposed in the correct orientation to perform any required actions.

- the ability of the end-effector, or the end-effector holding an object, to exert forces or moments on the environment while maintaining stability in unconstrained directions.

- the computational algorithms necessary to control the manipulator joints to provide the necessary task motions and to provide the neces-

sary task forces, this involves a dynamic model of the manipulator so that we may separate internal forces from external forces.

2.1 Four Joint Wrist Design

The wrist is required to provide an orienting mechanism, free of kinematic singularities to orient the end-effector relative to the terminal link of the manipulator. In this work we consider the manipulator to be a three-dimensional positioning mechanism with a well defined terminal link orientation, a function of position. The wrist is then to provide a three-degree-of-freedom orienting mechanism free of singularities. It has been shown in prior work that this requires a minimum of four revolute axes. It is also desirable that the wrist does not introduce any appreciable translation as it provides for rotational change. If these joints are to be of limited rotational ability, and continuous rotation to be ruled out, then we might require that the wrist be able to rotate some arbitrary finite angle, say $\pm 90°$ from any given initial orientation. We have been working on developing the design by Fisher which uses an additional joint to provide motion to control the rates of the three primary orienting joints [16].

2.2 Compliant Wrist Design

We have also been working on a six-degree-of-freedom passive compliance located at wrist and instrumented for displacement and orientation [17, 18] see Appendix A.1. With this compliance we are able to provide a high bandwidth low pass filter between the environment and the manipulator. It is used to absorb the impulsive forces on contact until they may be dissipated by the manipulator in a controlled manner. The displacement sensor has been programmed to increase the stiffness of the passive compliance by driving the manipulator in directions opposite to the detected displacement in order to provide for stable position control. The displacement sensor has also been programmed to decrease the stiffness of the passive compliance so as to provide for force control. In this case the manipulator is programmed to move so as to maintain a constant deflection of the passive compliance thus providing a constant force or moment on the environment. We will look eventually to the combination of the passive into the four revolute wrist to make a non-singular passive compliant wrist.

2.3 Grasping — Theory

The wrist will interface to the end-effector. We are concerned here with an end-effector capable of securing a stable grasp on an object, not on the dextrous manipulation of the object. We rely on the manipulator to do that. We are thus primarily interested in enveloping grasps in which an object is held in form closure not in the less stable force closure which relies on friction between the finger tips and the object for the holding force. In our case friction is a problem as we would like objects to slide along the fingers into a form closure grasp against a palm surface. The palm of the end-effector is instrumented for tactile array information to help determine the grasp on the object. We thus consider the end-effector to be a soft surface at the end of the wrist which may be used to exert forces and moments on objects and the environment and to provide a tactile image of the contact. Fingers are provided to hold an object against the palm so that the manipulator may pull as well as push on objects.

Given an arbitrary object it is difficult, if not impossible, to determiningg how if might be picked-up. If we restrict the class of objects to polyogonial cylinders we are able to determine grasping strategies and to prove that a given object will rest in a final position against the palm if the determined strategy is followed [19, 20, 21]. See Appendix A.2. A large class of objects may be approximated by polyogonial cylinders and grasping strategies developed. For other objects specific grasping surfaces must be specified in order for a stable grasp to be planned.

2.4 Grasping — Practice

We have also developed a medium-complexity end-effector which will enable us to provide these grasping and force exerting strategies [22]. See Appendix A.3. The end-effector consists of a palm around which three fingers are positioned. On finger is fixed in place, the other two fingers move together, symmetrically around the palm to provide for a number of grasps. The fingers may be rigid or passively curl around the object. The end-effector is strong and is suitable for holding wrenches or in using a hammer; it does not provide for any dextrous motion.

2.5 Dynamics

The control of the manipulator is itself a computational problem. A straightforward implementation of manipulator kinematics and dynamics results in matrix algorithmns, which while simple, involve enormous amounts of arithmetic computation. Historically this problem has been solved by generating the symbolic equations that would be evaluated by the numeric algorithm. Taking advantage of the simple form of many manipulator's kinematics and dynamics, which result in many terms being either zero of one, the generated, closed form symbolic equations, may be evaluated at a far reduced computational cost. We have been in the forefront of this approach and have developed not only kinematic equations but have also studied dynamics [15, 23, 24]. The approach to dynamics is to separate those computations which must be performed at the control rate and those which may be computed at a much reduced background computational rate dependent only on manipulator configuration state. The control rate computation is only six multiplies and additions per joint. Background computations compute inertias, coupling inertias, and velocity and gravity dependent forces. See Appendix A.4.

2.6 Robot Force and Motion Server

Even with these symbolic techniques it is necessary many of the low-level computations in parallel. See Appendix A.5. To this end we have developed a multi-processor computer controller based on Intel 8086 and 8087 processors. The controller employs one processor per joint and involves no pipelining. The joint processors look at the desired Cartesian set-point and determine, in parallel, their own joint coordinate [25].

2.7 Parallel Kinematics

This parallel approach involved also developing a parallel approach to manipulator kinematics [26, 27]. Here all joints obtain their individual joint coordinates based on the Cartesian set-point and on previous values of the other joint coordinates. Good convergence and tracking is obtained using this method. See Appendix A.6

3 DOCUMENTATION

[1] Richard P. Paul, Hugh F. Durrant-Whyte, and Max Mintz. A robust, distributed sensor and actuation robot control system. In Oliver Faugeras and Georges Giralt, editors, *Robotics Research: The Third International Symposium*, pages 93–100, MIT Press, Cambridge, Massachusetts, 1986.

[2] Hugh F. Durrant-Whyte and R. P. Paul. Integration of distributed sensor information: an application to a robot system coordinator. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, page 415, November 1985.

[3] Hugh F. Durrant-Whyte. Consistent integration and propogation of disparate sensor observations. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1464–1469, April 1986.

[4] Hugh F. Durrant-Whyte. Consistent integration and propogation of disparate sensor observations. To appear, International Journal of Robotics Research, Fall 1986.

[5] Hugh F. Durrant-Whyte. Concerning uncertain geometry in robotics. In *International Workshop on Geometric Reasoning*, June 1986.

[6] Hugh F. Durrant-Whyte, Ruzena Bajcsy, and Richard Paul. Using a blackboard architecture to integrate disparate sensor observations. In *DARPA Workshop on Blackboard Systems for Robot Perception and Control*, June 1986.

[7] Jeffrey C. Trinkle, Jacob M. Abel, and R. P. Paul. *Enveloping, Frictionless, Planar Grasping.* Technical Report MS-CIS-86-57, University of Pennsylvania, CIS Dept., Moore School, Philadelphia, PA 19104, July 1986.

[8] Jeffrey C. Trinkle, Jacob M. Abel, and R. P. Paul. Enveloping, frictionless, planar grasping. In *Proceedings of the 1987 IEEE International Conference on Robotics and Automation*, 1987.

[9] R. P. Paul and Hong Zhang. Design and implementation of a robot force/motion server. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1878–1883, 1986.

[10] Hong Zhang and R. P. Paul. Hybrid control of robot manipulators. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 602–607, March 1985.

[11] R. P. Paul and Hong Zhang. Robot motion trajectory specification and generation. In Hideo Hanafusa and Hirochika Inoue, editors, *Robotics Research: The Second International Symposium*, pages 373–380, MIT Press, Cambridge, Massachusetts, 1985.

[12] R. P. Paul and Hong Zhang. Computationally efficient kinematics for manipulators with spherical wrists based on the homogeneous transformation representation. *The International Journal of Robotics Research*, 5(2), 1986. Special Issue on Kinematics.

[13] Alberto Izaguirre and R. P. Paul. Computation of the inertia and gravitational coefficients of the dynamic equations of the robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1024–1032, March 1985.

[14] Alberto Izaguirre and Richard P. Paul. Automatic generation of the dynamics equations of the robot manipulators using a LISP program. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 220–226, April 1986.

[15] Alberto Izaguirre, Minoru Hashimoto, Richard P. Paul, and Vincent Hayward. Identification of the parameters of the dynamic equations of robot manipulators. In *IEEE International Whorkshop on Robotics: Trends, Technology and Applications*, Madrid, 1987.

[16] Gregory Long and Richard P. Paul. Avoiding orientation degeneracies with a spherical four-joint wrist. 1988. Work in progress.

[17] Yangsheng Xu and Richard. P. Paul. On position compensation and force control stability of a robot with a compliant wrist. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1173–1178, April 1988.

[18] Richard P. Paul, Yangsheng Xu, and Xiaoping Yun. Terminal link force and position control of a robot manipulator. September 1988. To appear in the Proceedings of CISM Conference on Theory and Practics of Robots Manipulator.

[19] Jeffery C. Trinkle. *The Mechanics and Planning of Enveloping Grasps.* Technical Report MS-CIS-87-46, University of Pennsylvania, CIS Dept., Moore School, Philadelphia, PA 19104, June 1987.

[20] Jeffery C. Trinkle and Richard P. Paul. An investigation of frictionless, enveloping grasps. *The International Journal of Robotics Research*, 7(3):33–51, June 1988.

[21] Jeffery C. Trinkle and Richard P. Paul. Planning for dextrous manipulation with sliding contacts. *The International Journal of Robotics Research*, 1988. submitted for publication.

[22] Nathan Ulrich, Richard Paul, and Ruzena Bajcsy. A medium-complexity compliant end effector. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 434–436, April 1988.

[23] Alberto Izaguirre, Minoru Hashimoto, Richard P. Paul, and Vincent Hayward. A new computational structure for real time dynamics. In *Identification of parameters in dynamics*, S.I.C.E. Conference, Hiroshima, JAPAN, July 1987.

[24] Alberto Izaguirre, Minoru Hashimoto, Richard P. Paul, and Vincent Hayward. A new computational structure for real-time dynamics. *The International Journal of Robotics Research*, 1988. accepted for publication.

[25] Zhang Hong. *RFMS Software Reference Manual.* Technical Report MS-CIS-88-01, University of Pennsylvania, CIS Dept., Moore School, Philadelphia, PA 19104, January 1988.

[26] Hong Zhang and Richard P. Paul. Non-kinematic errors in robot manipulators. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1138–1139, April 1988.

[27] Hong Zhang and Richard P. Paul. A parallel solution to robot inverse kinematics. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1140–1145, April 1988.

A APPENDICES

A.1 Terminal Link Force and Position Control of a Robot Manipulator

TERMINAL LINK FORCE AND POSITION CONTROL OF A ROBOT MANIPULATOR

Richard P. Paul

Yangsheng Xu

Xiaoping Yun

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

ABSTRACT

The rigidity of robot manipulators, while an assest in the control of end-effector position, is a problem when force is to be controlled or when a controlled contact with a rigid environment is to be achieved. The introduction of flexibility into the manipulator structure may simplify force and contact control but considerably complicate the control of position. We propose a semi-flexible terminal link for a rigid manipulator which, while providing the necessary flexibility for force and contact control, provides a structure which may still be controlled in position. We choose an intermediate stiffness for the terminal link so that position feedback in the manipulator can either increase the effective stiffness of the end-effector, when under position control, or can decrease the stiffness, when under force control. This paper describes such a terminal link -- a compliant wrist device and analyzes its stability and performance under both position and force control.

1. INTRODUCTION

In many applications of robots the manipulator's end-effectors work with objects which are in contact with the environment. The manipulator continually moves between constrained and unconstrained modes, constrained when the object is in contact with the environment and unconstrained when an object is being moved in free space. When the manipulator is constrained, force is controlled, when the manipulator is in free space and unconstrained, position is controlled. Between these two modes is a transition, the manipulator moving in free space comes into contact with a rigid environment and the manipulator in constrained by the environment breaks contact and becomes unconstrained [13]. Changes between modes can involve discontinuities in state variables such as velocity and force. In order to accommodate these discontinuities passive compliance may be inserted between the manipulator and its end-effector, in the same manner as springs and shock absorbers are used in automobiles.

In the constrained mode force is to be controlled. If only some degrees of freedom are constrained then those degrees of freedom are appropriate for force control while the unconstrained degrees of freedom are appropriate for position control. Force control may be provided in either joint coordinates [14] or in Cartesian coordinates [15] and the control may take place in either coordinate system.

The simplest, and most direct method of force control involves colocated force sensors at the manipulator's joints. This method while only providing approximate control of end-effector force is simple and fast [14]. Stiff wrist force sensors may be used to control force by feedback over the entire

This material is based on work supported by the National Science Foundation under Grant No. DMC-8512838. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

The system shows an actuator controlling the motion of a link and thereby controlling the motion of the end effector through a compliant wrist sensor device which is attached between the end effector and link. The feedback control loop is used to make the end effector position reach a command position against the influence of an external force which could be gravity load, unbalancing harmonic force, or random excitation.

We will assume the following: the link drive train is rigid compared with the flexibility of the compliant wrist sensor, the contribution from the viscous damping and static friction of the actuator is negligible, the rotational inertia of the actuator and the link is J , the proportional feedback gain and rate gain of PD controller are K_p and K_v respectively, the load and end effector mass is m , the stiffness and damping of the wrist sensor are K and C respectively.

A system block diagram of the single joint manipulator is shown in Fig.2. The wrist sensor records the difference of the motions between link and end effector. These signals are the input to the compensating controller H , together with the input command motion X_c (corresponding to the angular displacement of the link) which drives the system controller and thereby the actuator.

It is our aim to determine the form of the compensating controller H so that the deflection of the end effector due to external forces applied to the compliant wrist sensor can be compensated. In other words, the response of the end effector becomes independent of the force and compliance of the system.

We define G_c the transfer function of the actuator, PD controller, and rigid link. H is the transfer function of position compensator in the feedback loop. G_1 is the transfer function of end effector motion/command position, and G_2 is the transfer function of end effector motion/applied external force. An equivalent block diagram is shown in Fig.3. From Fig.3, we set $F = 0$ and $X_c = 0$ respectively, to obtain:

$$\frac{X}{X_c} = \frac{G_1 G_c}{1 - HG_c + HG_c G_1} \tag{1}$$

$$\frac{X}{F} = \frac{(1 - H\ G_c)\ G_2}{1 - H\ G_c + G_1\ G_c\ H} \tag{2}$$

where

$$G_1 = \frac{C\ s + K}{m\ s^2 + C\ s + K} \tag{3}$$

$$G_2 = \frac{1}{m\ s^2 + C\ s + K} \tag{4}$$

$$G_c = \frac{K_p}{J\ s^2 + K_v\ s + K_p} \tag{5}$$

Therefore, the characteristic equation of the closed loop system is

$$1 - H\ G_c + G_1\ G_c\ H = 0 \tag{6}$$

or,

$$(ms^2 + Cs + K)(Js^2 + K_v s + K_p) - HK_p ms^2 = 0 \tag{7}$$

The steady state characteristics of the closed loop system can be determined by setting s to zero.

$$\lim_{s \to 0} \frac{X}{X_c} = \frac{G_c(s) G_1(s)}{1 - H(s) G_c(s) + H(s) G_c(s) G_1(s)}$$

$$= \frac{KK_p}{KK_p - H(0) K_p K + H(0) KK_p} = 1 \tag{8}$$

$$\lim_{s \to 0} \frac{X}{F} = \frac{[1 - H(s) G_c(s)] G_2(s)}{1 - H(s) G_c(s) + H(s) G_c(s) G_1(s)}$$

$$= \frac{K_p - H(0) K_p}{KK_p - H(0) K_p K + H(0) K_p K}$$

$$= \frac{1 - H(0)}{K} \tag{9}$$

$$\lim_{s\to 0}\frac{X}{F}=\frac{K_p-K_1K_p}{K_pK}=\frac{1-K_1}{K} \tag{15}$$

In order to compensate for all deflection, K_1 should be chosen as unity. The stability conditions are

$$K_2<\frac{K_v}{K_p} \tag{16}$$

$$K_1<1+\frac{CK_v}{K_pm} \tag{17}$$

$$K_1<1+\frac{CK_v}{mK_p}-\frac{K(K_v-K_pK_2)}{KK_v+K_pC} \tag{18}$$

Compared (18) with (12), one may see the stability condition with PD compensator is better than that with proportional compensator. The critical damping for a stable system is fairly small. Therefore, for the small damping sensor case, the P compensator is not a good choice, while the PD compensator is better. Simulation was performed for various parameters. For the case in which the inertia of actuator and link is considered, the simulation shows that more damping is necessary. From (16), the rate gain of the compensator is restricted by the joint PD controller parameters. In the other words, the active damping cannot be set arbitrarily high. Here again passive damping is critical.

2.3. Lead-lag Network Compensator

Compensating also can be achieved using a simple lead-lag network in the feedback loop. The lead-lag network can be represented in the form $H(s)=K_{p_1}(1+K_{v_1}s)/(1+K_Is)$. If the inertia is neglected in the system, the transfer function is

$$\frac{X}{F}=\frac{(1+K_Is)(K_p+K_vs)-K_pK_{p_1}(1+K_{v_1})}{(1+K_Is)(ms^2+Cs+K)(K_p+K_vs)-ms^2K_pK_{p_1}(1+K_{v_1}s)} \tag{19}$$

$$\frac{X}{X_c}=\frac{(Cs+K)(1+K_Is)K_p}{(1+K_Is)(ms^2+Cs+K)(K_p+K_vs)-ms^2K_pK_{p_1}(1+K_{v_1}s)} \tag{20}$$

The characteristic equation is

$$(mK_IK_v)s^4+(mK_IK_p+mK_v+K_ICK_v-mK_pK_{v_1}K_{p_1})s^3+(CK_IK_p+K_vC+K_vK_IK)s^2$$
$$+(K_IKK_p+CK_p+K_vK)s+KK_p=0 \tag{21}$$

The steady-state compensation requires that K_{p_1} be chosen as unity. The damping ratio needed for a stable system is weaker than that needed in the P compensator, but stronger than that in PD compensator.

2.4. Compensator as an Inverse of G_c

From the equations (1) and (2), an interesting fact is that if the compensator transfer function is the inverse of the transfer function of the actuator, the PD controller, and the link, $H(s)=1/G_c(s)$, the ratio X/F will be made zero at all the time, which means all deflections in the end-effector will be compensated no matter how much and what kind of the external force is. At this condition, the ratio of X/X_c will be equal to G_c and tend to unity in the steady state. The system will become independent of the external force and passive compliance of the robot system.

If the robots move slowly or the robot is light-weight and the inertia of the link and actuator is neglected, there is no the acceleration term in the transfer function G_c and H is the exact proportional and derivative control, which is easy to realize. In the case that the acceleration term has to be considered, the inertia must be specified. Therefore, we also investigated the sensitivity of the system performance to the inertia estimated error. The result showed that the effect of inertia error is not significant. For both cases, the simulation has been performed.

The steady-state error can be expressed as

$$e_{ss} = \lim_{t\to\infty} e(t) = \lim_{s\to 0} \frac{sP_d(s)}{1+G(s)} \tag{27}$$

The step, ramp, and paraboloid are simple mathematical expressions for the input force, namely, $p_d(t)$ is defined as $U(t)$, $tU(t)$, $t^2U(t)/2$, respectively, where the notation $U(t)$ means the unit step force for $t>0$. The system error at the steady-state is following:

$$e_{ss} = \frac{1}{1+K_p}, \qquad as \qquad P_d(s) = \frac{1}{s}$$

$$e_{ss} = \infty, \qquad as \qquad P_d(s) = \frac{1}{s^2}$$

$$e_{ss} = \infty, \qquad as \qquad P_d(s) = \frac{1}{s^3}$$

The above results are obvious because the open-loop is a zero order system. Therefore, the closed-loop system has a force error at the steady-state under the step command force if the P controller is used.

Suppose the wrist device and force controller parameters are $m = 2\,kg$, $K_w = 4000\,N/m$, $C_r = 200\,N/m/s$, $K_p = 30$, the simulation of the step force response is performed for the different wrist dampers as shown in Fig.8. In Fig.8, the dashed line at force level 1 is the desired contact force. At $K_p = 30$, there is 3.2% force error at the steady state.

As the PD controller is represented as $H(s) = K_p(1+K_v s)$, the open-loop transfer function becomes

$$G(s) = \frac{K_p K_w (1+K_v s)}{ms^2+(C_r+C_w)s+K_w} \tag{28}$$

The closed-loop system transfer function is in form of

$$\frac{P_c(s)}{P_d(s)} = \frac{K_p K_w (1+K_v s)}{ms^2+(C_r+C_w+K_p K_w K_v)s+K_w(1+K_p)} \tag{29}$$

From (28), since the open-loop is still a zero order system, the steady-state performance will be identical with that in the P controller.

A simulation was performed for the step force input for various parameters and it is shown that the system response is much improved because the rate gain in the PD controller has a contribution to the system damping. Therefore, a proper value of the rate gain is beneficial to the improvement of the system behavior. For the P and PD force controllers, we have the following summaries.

1) From (25) and (29), the system is stable and independent of the wrist compliance and the controller. The relative stability of the system depends upon the stiffness of the system. The higher the proportional gain, the more compliant the wrist can be made.
2) There is a force error at the steady-state. The error is inversely proportional to the P gain of controller at a step force input, and becomes infinite at a ramp and paraboloid force input.
3) The rate gain in the PD controller provides the damping in the system. Therefore, in a slight damping system, an increasing of the rate gain can achieve the same system response as that with a large damping.

For the PI controller represented as $H(s) = K_p(1+\frac{K_I}{s})$, the open-loop transfer function is

$$G(s) = \frac{K_p K_w (1+\frac{K_I}{s})}{ms^2+(C_r+C_w)s+K_w} \tag{30}$$

From (30), we know that the open-loop becomes a first order system in this case, and the steady-state performance can be obtained as follows.

$$e_{ss} = 0, \qquad as \qquad P_d(s) = \frac{1}{s}$$

$$\frac{P_c(s)}{P_d(s)} = \frac{K_p K_w (m_e s^2+C_e s+K_e)}{[ms^2+(C_r+C_w)s+K_w][m_e s^2+(C_e+C_w)s+(K_w+K_e)]-(C_w s+K_w)^2+K_p K_w (m_e s^2+C_e s+K_e)} \quad (37)$$

Generally, the controller can be expressed as $H(s)$ and the open-loop system transfer function is

$$G(s) = \frac{H(s)K_w (m_e s^2+C_e s+K_e)}{[ms^2+(C_r+C_w)s+K_w][m_e s^2+(C_e+C_w)s+(K_w+K_e)]-(C_w s+K_w)^2} \quad (38)$$

From (38), the characteristic equation of the closed-loop system is

$$f_4 s^4+f_3 s^3+f_2 s^2+f_1 s+f_0 = 0 \quad (39)$$

where

$$f_4 = mm_e$$

$$f_3 = m_e(C_r+C_w)+m(C_e+C_w)$$

$$f_2 = m(K_w+K_e)+mK_w(1+K_p)+C_r(C_e+C_w)+C_w C_e$$

$$f_1 = K_e(C_w+C_r)+K_w(C_e+C_r)+C_e K_w K_p$$

$$f_0 = K_w K_e(1+K_p)$$

Since all coefficients of the characteristical equation are positive, the stability condition is

$$f_1 f_2 f_3 > f_1^2 f_4+f_3^2 f_0 \quad (40)$$

We can investigate the system performance for some particular cases. For the elastic environment and wrist, there is no passive damping C_e and C_w. Also, we suppose the mass $m = m_e = 1$, and the proportional gain is larger than unity, taking K_p=10 for instance, as the steady-state error can be made the smallest possible. In this case, the stability condition is

$$C_r^2(12K_w+K_e)(K_e+K_w > (K_e+K_w)^2 C_r^2+11C_r^2 K_w K_e$$

or

$$12K_w^2 > K_w^2$$

which is always satisfied. Further, Eq.(39) in this case can be interpreted as

$$\frac{1}{K_e}s^4+\frac{C_r}{K_e}s^3+[\frac{(2+K_w)}{K_e}+1]s^2+C_r(1+\frac{K_w}{K_e})s+K_w(1+K_p)=0 \quad (41)$$

From (41), if K_e is infinite, the characteristics of the system is identical with that in the rigid case. However, if K_e is not infinite, the system performance is improved because the effective damping ratio is increased.

In the case that the end-effector mass is small compared with the effective mass of the manipulator, for example m=10, m_e=1, the environment is elastic C_e=0, and the actuator viscous damping is as same as the wrist damping, i.e, $C_w = C_r$, the system stability condition is

$$11K_w^2-20K_e^2-1000K_w K_e+C_r^2(2K_2+k_w) > 0 \quad (42)$$

It is clear that the damping C_r or C_w plays an important part in the system stability.

Summary

1) In most cases, the system including the environmental compliance can be maintained stable. The effective damping of the closed-loop system in this case is increased and the high order dynamics are introduced.

2) The passive damping in the wrist is always beneficial, especially for the case that there is no damping and the environment system has infinite stiffness.

3) In the case that the the system damping is small, the alternative way to stablize the system is by utilizing a PD controller in the force feedback loop, because the rate gain of the controller can provide

because of the active sensing and compensation control in the feedback loop. The compliance and compliance ratio in our device is programmable in each direction and depends upon the task operation and the positioning compensation capability of the system.

5. SENSING MECHANISM KINEMATICS

The sensor has to be able to measure six DOF motions of the upper plate relative to the bottom one. If one considers the device mechanism as robot, this task is just the direct kinematics of the robot manipulator. Namely, the joint space motion is measured and the Cartesian space motion is identified. We at first, intented to use a parallel mechanism as in the paper [10], and a LVDT as a displacement sensor. However, the direct kinematics is difficult for a parallel mechanism, while inverse kinematics is easy. On the contrary, for a series mechanism the direct kinematics is much easier than the inverse one [16]. After judging all the possible structure, we finally choses the series mechanism as the sensing structure with six potentiometers are used as displacement sensors.

The mechanism kinematics skeleton is as Fig.10 with coordinate frame assigned to the links.

$$T_w = Trans(-l_7,l_3,l_1,)\ Rot(z,\theta_1)\ Trans(-l_2,0,0)\ Rot(x,\theta_2)\ Trans(0,-l_3,0)\ Rot(x,\theta_3)\ Trans(l_4,-l_5,0)$$
$$Rot(z,\theta_4)\ Trans(0,0,l_6)\ Rot(y,\theta_5)\ Trans(0,l_5,0)\ Rot(z,\theta_6)\ Trans(l_7,0,l_8) \quad (43)$$

Using the notation in [7], the A transformation matrix for the device are as follows.

$$A_1 = Trans(-l_7,l_3l_1))\ Rot(z,\theta_1) = \begin{bmatrix} C_1 & -S_1 & 0 & -l_7 \\ S_1 & C_1 & 0 & l_3 \\ 0 & 0 & 1 & l_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (44)$$

$$A_2 = Trans(-l_2,0,0)\ Rot(x,\theta_2) = \begin{bmatrix} 1 & 0 & 0 & -l_2 \\ 0 & C_2 & -S_2 & 0 \\ 0 & S_2 & C_2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (45)$$

$$A_3 = Trans(0,-l_3,0)\ Rot(x,\theta_3) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & C_3 & -S_3 & -l_3 \\ 0 & S_3 & C_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (46)$$

$$A_4 = Trans(l_4,-l_5,0)\ Rot(z,\theta_4) = \begin{bmatrix} C_4 & -S_4 & 0 & l_4 \\ S_4 & C_4 & 0 & -l_5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (47)$$

$$A_5 = Trans(0,0,l_6)\ Rot(y,\theta_5) = \begin{bmatrix} C_5 & 0 & S_5 & 0 \\ 0 & 1 & 0 & 0 \\ -S_5 & 0 & C_5 & l_6 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (48)$$

$$A_6 = Trans(0,l_5,0)\ Rot(z,\theta_6) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & C_6 & -S_6 & l_5 \\ 0 & S_6 & C_6 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad (49)$$

$$U_{414} = C_4 U_{514} + l_4$$

$$U_{424} = S_4 U_{514} - l_5$$

Since

$$A_2 A_3 = \begin{bmatrix} 1 & 0 & 0 & -l_2 \\ 0 & C_{23} & -S_{23} & -l_3 C_2 \\ 0 & S_{23} & C_{23} & -l_3 S_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{55}$$

$$U_2 = A_{23} U_4 = \begin{bmatrix} U_{411} & U_{412} & U_{413} & U_{414} + l_2 \\ C_{23} U_{421} + S_{23} S_5 & C_{23} U_{422} + -S_{23} U_{432} & C_{23} U_{423} - S_{23} U_{433} & C_{23} U_{424} - S_{23} U_{434} + l_3 C_2 \\ S_{23} U_{421} - C_{23} S_5 & S_{23} U_{422} + C_{23} U_{432} & S_{23} U_{423} + C_{23} U_{433} & S_{23} U_{424} + C_{23} U_{434} - l_3 S_2 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{56}$$

let

$$U_{221} = C_{23} U_{421} + S_{23} S_5$$

$$U_{231} = S_{23} U_{421} - C_{23} S_5$$

$$U_{222} = C_{23} U_{422} + -S_{23} U_{432}$$

$$U_{232} = S_{23} U_{422} + C_{23} U_{432}$$

$$U_{223} = C_{23} U_{423} - S_{23} U_{433}$$

$$U_{233} = S_{23} U_{423} + C_{23} U_{433}$$

$$U_{224} = C_{23} U_{424} - S_{23} U_{434} + l_3 C_2$$

$$U_{234} = S_{23} U_{424} + C_{23} U_{434} - l_3 S_2$$

$$U_1 = A_1 U_2 = \begin{bmatrix} C_1 U_{411} - S_1 U_{221} & C_1 U_{412} - S_1 U_{222} & C_1 U_{413} - S_1 U_{223} & C_1 (U_{414} - l_2) - S_1 U_{224} - l_7 \\ S_1 U_{411} + C_1 U_{221} & S_1 U_{412} + C_1 U_{222} & S_1 U_{413} + C_1 U_{223} & S_1 (U_{414} - l_2) + C_1 U_{224} + l_3 \\ U_{231} & U_{232} & U_{233} & U_{234} + l_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{57}$$

The determinant of the Jacobian matrix was calculated to investigate singularities. The results shows that there is no singularity around the home position where the device works.

6. CONTROL STRATEGIES AND PRELIMINARY EXPERIMENT

The experiment of the compliant wrist was performed on the PUMA 560. Before the experiment, six potentiometers were adjusted in a proper range and an A/D board was designed. The control was executed by the RCCL system. The preliminary experiment has shown that the compliant wrist works well in both passive compliance and active sensing, and that stable compensation in position control is possible.

We tested the following two control strategies:

(1) **Position compensation in the free space**

We suppose the transformation between the base and robot wrist coordinate is T_6, that between two plates of the compliant wrist is T_w, and that between the base and upper plate of the compliant device is B which is assumed as the task coordinate transformation. The kinematics relation at the initial state is

$$T_6 T_w = B \tag{58}$$

REFERENCES

[1] S. D. Eppinger and W. P. Seering, "On dynamic models of the robot force control", *Proceedings of the IEEE International Conference on Robotics and Automation*, P.29-34, 1986

[2] S. D. Eppinger and W. P. Seering, "Understanding bandwidth limitations in robot force control" *Proceedings of the IEEE International Conference on Robotics and Automation*, P.904-909, 1987

[3] R. K. Roberts, R. P. Paul, and B. M. Hillberry, "The effect of wrist force sensor stiffness on the control of robot manipulators", *Proceedings of the IEEE International Conference on Robotics and Automation*, P.269-274, 1985

[4] R. P. Paul, "Problems and research issues associated with the hybrid control of force and displacement", *Proceedings of the IEEE International Conference on Robotics and Automation*, P.1966-1971, 1987

[5] D. E. Whitney, "Historical perspective and state of the art in robot force control", *Proceedings of the IEEE International Conference on Robotics and Automation*, P.262-268, 1985

[6] D. E. Whitney and J. M. Rourke, "Mechanical behavior and design equations for elastomer shear pad remote center compliance", *ASME Journal of Dynamic System, Measurement, and Control*, Vol. 108, P.223-232, 1986

[7] R. P. Paul, *"Robot Manipulators: Mathematics, Programming and Control"*, Cambridge, MIT press, 1981

[8] K. Takuse, H. Inoue, K. Sato, and S. Hagiuara, "The design of the articulated manipulator with torque control ability", *Fourth International Symposium on Industrial Robots*, Nov. 1974

[9] H. Asada and K. Ogawa, "On the dynamic analysis of a manipulator and its end effector interacting with the environment", *Proceedings of the IEEE International Conference on Robotics and Automation*, P.751-756, 1987

[10] H. Inoue, Y. Tsusaka, and T. Fukuizumi, "Parallel manipulator", *Proceedings of Third International Symposium of Robotics Research*, P.321-327, 1986

[11] D. S. Seltzer, "Compliant robot wrist sensing for precision assembly", *Robotics: Theory and Application*, P.161-168, 1986

[12] H. Kazerooni, and J. Guo, "Direct-drive, active compliant end-effector" *Proceedings of the IEEE International Conference on Robotics and Automation*, P.758-766, 1987

[13] R. C. Goertz, "Manipulators used for handling radioactive materials" *Human Factors in Technology*, edited by E. M. Bennett, McGraw Hill, 1963

[14] R. P. Paul, and H. Zhang, "Design and implementation of a robot force /motion server", *Proceedings of the IEEE International Conference on Robotics and Automation*, P.1878-1883, 1986

[15] O. Khatib, and J. Burdick, "Manipulators motion and force control", *Proceedings of the IEEE International Conference on Robotics and Automation*, 1986

[16] Kenneth J. Waldron and Kenneth H. Hunt, "Serial-prallel dualities in actively coordinated mechanisms", *Fourth International Symposium on Robotics Research*, Santa Cruz, Sept. 1987

[17] Chae H. An, and John M. Hollerbach, "Dynamic stability issues in the force control of manipulator", *Proceedings of the IEEE International Conference on Robotics and Automation*, P.890-896, 1987

[18] De Schutter J., "Compliant robot motion control methods for regid manipulators based on a generic scheme", *Proceedings of the IEEE International Conference on Robotics and Automation*, P.1060-1065, 1987

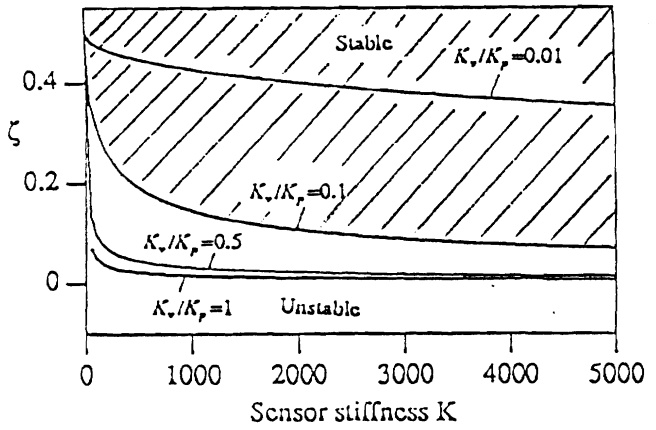


Fig.4 Stability condition for sensor stiffness and sensor damping ratio with P compensator

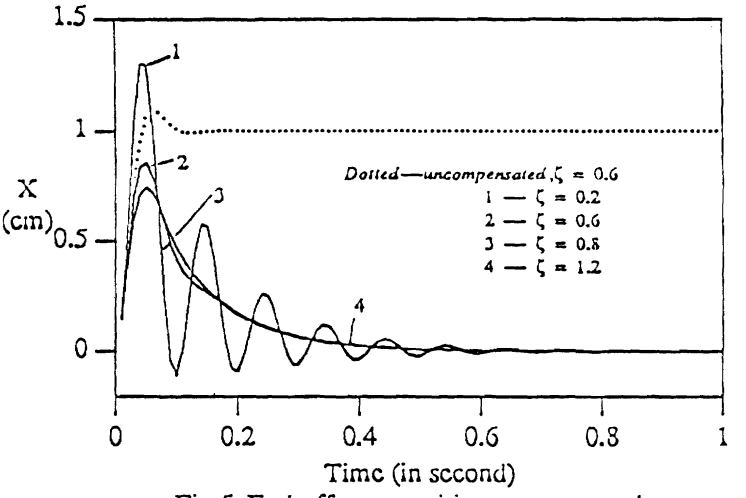


Fig.5 End-effector position response under 4kg step force with P compensator (The inertia is not included)

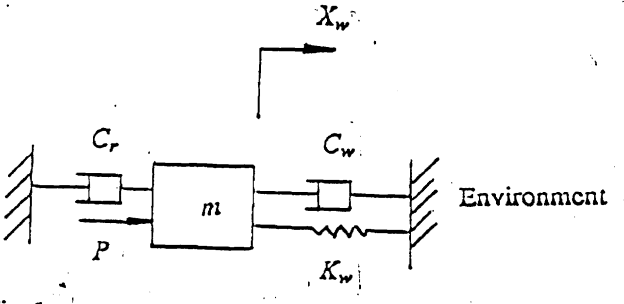


Fig.6 The rigid environment system

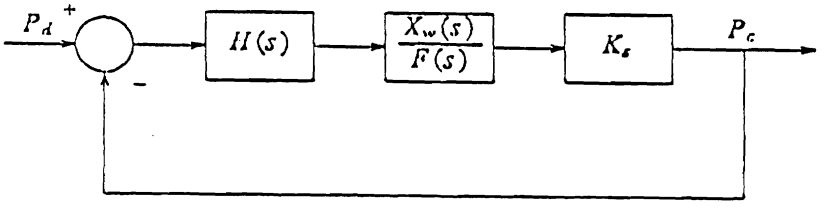
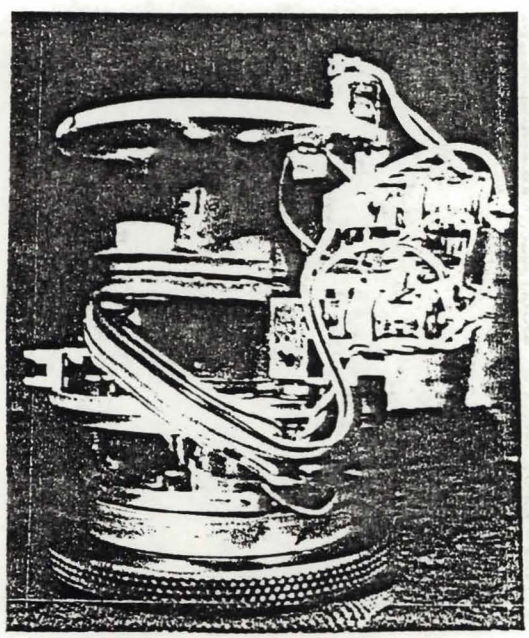


Fig.7 The closed-loop control system block diagram



A.2 Planning for Dextrous Manipulation with Sliding Contacts

Draft submitted to the International Journal of Robitics Research, June 3, 1988.

PLANNING FOR DEXTROUS MANIPULATION WITH SLIDING CONTACTS

J. C. Trinkle
Department of Systems and Industrial Engineering
University of Arizona
Tucson, AZ 85721

R. P. Paul
Department of Computer and Information Science
GRASP Laboratory
University of Pennsylvania
Philadelphia, PA 19104

ABSTRACT

We study the problem of gaining a secure and enveloping grasp of a two dimensional object by exploiting sliding at the contacts between the object and the hand. This is done in two steps: first, choosing an initial grasp with which the object can be manipulated away from the support and second, continuously altering the grasp so that the desired final configuration is achieved. The plans generated by our technique may be executed with only position control.

The main contributions of this paper are: the derivation of *liftability regions* of a planar object for use in manipulation planning; the use of the *lifting phase plane* in manipulation planning; and the derivation of the quasi-static *object motion problem* which provides a basis for general, three-dimensional manipulation planning.

1. INTRODUCTION

One desirable application of robotic technology is automatic assembly using articulated mechanical hands and flexible fixturing systems. Assuming that the parts of the product to be assembled are within reach of the robot and the sequence of assembly operations is known, the following fundamental problems must be addressed: 1) part acquisition, by which is meant the selection and achievement of a useful grasp; 2) fixture set-up, which is closely related to grasp selection, but requires the synthesis of an accessible partial fixture [Asada 1984] as an intermediate step; and 3) parts mating which requires path planning [Brooks 1983] and compliant motion control [Mason 1979, Mason 1985, and Whitney 1982].

This paper concentrates on part acquisition via *liftability regions*. These regions provide a sound means to select a suitable initial grasp of an object resting on a support and a geometric method for planning subsequent manipulations. Contact forces are not controlled, they are rendered insignificant by the geometry of the grasp. Therefore, manipulation can proceed under position control. Force control is unnecessary!

Techniques for choosing a desired grasp for an articulated mechanical hand have been developed based on a quasi-static analysis coupled with optimization methods [Jameson 1985], independent regions of stable contact [Nguyen 1986, Paul 1972], expected task forces and fine motion requirements [Kobayashi 1984, Li 1987], the forces required to cause one or more contacts to slip [Cutkosky 1985, Holzmann 1985], minimizing the contact forces arising due to external forces [Trinkle 1985] and the potential energy in compliant fingers [Hanafusa 1982]. However, none of these techniques have included a means to achieve the grasp, nor have they dealt with the fact that most objects are initially at rest on a supporting surface (Wolter [1984] and Laugier [1983] consider the support, but only for the case of a parallel-jawed gripper). Others have considered dexterous manipulation of the object, but begin their analysis from the point of an achieved grasp. For example, Okada [1982] controlled a hand to turn a nut onto a bolt, Kobayashi's [1984] experimental hand drew simple figures with a pencil, Fearing [1987] demonstrated "baton twirling" using the Stanford/JPL hand, and Kerr [1984] developed the general differential equations for dexterous manipulation. All of these studies were done assuming that only rolling contacts exist. Enforcing this assumption requires that manipulation be carried out under force and position control and unduly limits the manipulation which can be performed. Mason [1985] and later Brost [1985]

and Peshkin [1987] studied the motion of objects sliding on a horizontal plane. They were able to devise manipulation strategies which were guaranteed to achieve a desired result despite uncertainty in the contact forces and the object's velocity.

In the following analysis, we consider the quasi-static motion of a manipulated object. Even though uncertainty exists in the precise descriptions of the contact forces, the resultant force and the object's geometry are always known. These facts allow exact computation of the object's instantaneous velocity given the instantaneous velocity of the palm and fingers.

2. LIFTABILITY OF RIGID BODIES

One reason to grasp an object is to gain complete control over its position and orientation. Thus we propose that an object is grasped by a robot if the object contacts only the robot's hand. If other bodies such as the support were allowed to contact the object, then those bodies would usurp a portion of the control over the object's motion. Therefore, the first goal in grasping is to manipulate the object in such a way as to cause it to lose all contact with the support. For this to be possible, the object must be *liftable* (The notion of liftability is a generalization of tippability which was discussed elsewhere [Trinkle 1988]).

> **Definition:** An object is *liftable* if and only if there exist finger contact positions on its surface for which increasing the contact forces applied by the fingers causes at least one of the supporting contacts to break.

2.1. Liftability Regions for Frictionless Objects

Liftability regions define the qualitative motion of a squeezed object (*i.e.* rotate left, rotate right, translate, or jam) based on the geometry of the grasp. To determine these regions, we need a contact model which accurately represents the kinematic constraints and the appropriate limiting cases of the contact force distributions which identify the qualitative motion. The exact force distribution of a contact region is irrelevant. A model which satisfies these requirements is to consider all contacts as a set of one or more point contacts. A contact of small area is considered to be a single

point. One with a large planar area is approximated by the set of points defining the vertices of the convex hull of the contact. Curved contact areas can be approximated by several polygons. During the following development, objects are assumed to be two-dimensional. However, any three-dimensional object which can be approximated as a generalized cylinder can be analyzed by our method by considering a suitable cross section of the cylinder.

Consider the two-point initial grasp of the frictionless, rigid, planar object depicted in Figure 1. The forces acting on the object are the finger contact forces ($\mathbf{f}_1$ and $\mathbf{f}_2$), the supporting contact forces ($\mathbf{f}_3$ and $\mathbf{f}_4$), and the weight of the object ($m\,\mathbf{g}$).

Under quasi-static conditions, the object must always satisfy the equilibrium relationships which may be written as

$$\mathbf{W}\,\mathbf{c} = -\mathbf{g}_{ext} \tag{1}$$

$$\mathbf{c} \geqslant 0 \tag{2}$$

[Salisbury 1982] where $\mathbf{W}$ is the wrench matrix, $\mathbf{c}$ is the vector of wrench intensities, $\mathbf{g}_{ext}$ is the external wrench (*i.e.* force and moment) [Ohwovoriole 1980] acting on the object and the vector inequality (2) applies element by element. If we choose the external wrench $\mathbf{g}_{ext}$ to be that caused by gravity, the solution to equation (1) is given by

$$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ c_{30} \\ c_{40} \end{bmatrix} + c_2 \begin{bmatrix} n_{21} \\ 1 \\ n_{23} \\ n_{24} \end{bmatrix} \tag{3}$$

where

$$c_{30} = \frac{mgt_g}{-t_3} > 0\,, \qquad c_{40} = \frac{mg\,(t_3 + t_g)}{t_3} > 0\,, \tag{4}$$

$$n_{21} = -\frac{\cos(\psi_2)}{\cos(\psi_1)}\,, \qquad n_{23} = -\frac{t_2}{t_3}\,, \tag{5}$$

$$n_{24} = \frac{t_2}{t_3} + \frac{\sin(\psi_1 - \psi_2)}{\cos(\psi_1)}\,, \tag{6}$$

ψ_i is the angle of the i^{th} inward contact normal measured counter-clockwise with respect to horizontal, and t_i is the moment arm of the i^{th} contact force taken with

respect to the summing point q_{14} which is the point of intersection of the lines of action of f_1 and f_4. Note that the second term on the right hand side of equation (3) is known as the internal grasping force [Salisbury 1982], because increasing c_2 increases the contact forces without changing the total force applied to the object.

To lift the object by squeezing, either c_3 or c_4 must be reduced to zero by increasing c_1 and c_2. The first row of equation (3) implies that for c_1 to increase with c_2, the following inequalities must hold

$$\cos\psi_1 < 0 \tag{7}$$

$$\cos\psi_2 > 0 \tag{8}$$

where without loss of generality, the first contact has arbitrarily been chosen to be on the right hand side of the object. If inequalities (7) and (8) are not simultaneously satisfied, then squeezing will result in an unstable grasp; the object will slide out of the grasp to the left. For c_3 or c_4 to be driven to zero, at least one of n_{23} and n_{24} must be negative. Equating the third and fourth rows of equation (3) to zero gives the values of c_2 required to break the third and fourth contacts, respectively

$$c_{23} = \frac{mgt_g}{-t_2} \tag{9}$$

$$c_{24} = \frac{-mg\,(t_3 + t_g)\cos(\psi_1)}{t_2\cos(\psi_1) + t_3\sin(\psi_1 - \psi_2)}\,. \tag{10}$$

Since c_2 is increased gradually after achieving the initial grasp, the contact which will break is the one corresponding to the smaller nonnegative value of c_2 (negative values of c_2 violate inequality (2)). Thus equations (9) and (10) in conjunction with inequalities (7) and (8) can be used to predict the motion caused by squeezing for every possible grasping configuration.

The magnitudes c_{23} and c_{24} depend on the grasp parameters ψ_1, ψ_2 and t_2. If we fix the position of the first finger's contact, then ψ_1 is constant and ψ_2 and t_2 vary. Considering all possible contact points and angles (at vertices) for the second finger, the perimeter P may be partitioned into five mutually exclusive *liftability regions* S, J, $B3$, $B4$ and T which satisfy the following relationship

$$S \cup J \cup B3 \cup B4 \cup T = P\,. \tag{11}$$

These regions correspond to possible contact points for the second finger for which

squeezing causes the object to: jam the fingers, resulting in the object's being pressed against the support; slide along the support; tip breaking the third contact; tip breaking the fourth contact; and translate (or rotate) so as to cause both the third and fourth contacts to break simultaneously. Figure 2 shows the liftability regions using a coded curve off-set from the perimeter of the object. The codes corresponding to S, J, $B3$, $B4$ and T are: dashed curve, no curve, solid bold curve, thin solid curve, and double-bold solid curve. In Figure 2 the translation region is a set of distinct points, so no double-bold curve segments are visible.

2.2. Liftability Regions of Frictionless Polygons

A polygon can be used to approximate any two-dimensional object with arbitrary precision. Therefore we discuss the liftability regions of polygons in detail and then show how the results can be applied to curved objects.

The sliding region S is the portion of the perimeter for which the inward normal of the second contact has either no horizontal component or has a horizontal component with the same sense as that of $\mathbf{f}_1$. In Figure 3, S is comprised of edges 0, 1, 2 and 3, vertices 1, 2 and 3, and a portion of vertex 4.[1] If the second finger contacts the polygon in S, squeezing will cause sliding to the left.

The regions J, $B3$, $B4$ and T are partitions of the remaining perimeter denoted by S'. Consider the k^{th} edge of the polygon in Figure 3. Points $\mathbf{p}$ lying on the line l_k containing the edge can be written in parametric form as

$$(1 - s)\,\mathbf{v}_k + s\,\mathbf{v}_{k+1} = \mathbf{p} \tag{12}$$

where $\mathbf{v}_i$ represents the i^{th} vertex of the polygon and the k^{th} edge is defined by $s \in [0,1]$. The line l_{ppd} is the unique line which is perpendicular to l_k and contains the point $\mathbf{q}_{14}$. The intersection of l_{ppd} with the k^{th} edge defines the contact point where t_2, the moment arm of the second contact force, is zero. The variables s and t_2 are linearly related by

$$t_2 = s - a \tag{13}$$

[1] What is meant by a portion of a vertex will be made clear later.

where a is the value of s at the intersection of l_k and l_{ppd}. Since t_2 varies linearly along the edge, c_{23} and c_{24} describe hyperbolas along l_k as shown in Figure 4. Note that the vertical asymptote of c_{24} is located at a positive value of t_2. This is the case defined by the following inequality

$$\sin(\psi_1 - \psi_2) < 0. \tag{14}$$

When inequality (14) is satisfied, a *jamming region* J lies between the vertical asymptotes of the two hyperbolas and the *jamming window* JW is the closed line segment $[q_{13}, q_{14}]$. The *breaking regions* $B3$ *and* $B4$ lie to the right and left of the jamming region. Since the values of c_{23} and c_{24} are not equal at any point on the edge, the *translation region* T is empty, implying that translational lifting is impossible if the second finger contacts that edge. Note that the physical significance of inequality (14) is that the resultant of the finger contact forces is in the direction of the gravity force. Therefore, to avoid jamming and to cause tipping, one must push down on the edge at a suitable point.

If the sense of inequality (14) is reversed,

$$\sin(\psi_1 - \psi_2) > 0 \tag{15}$$

then the functions c_{23} and c_{24} overlap, eliminating the jamming region (see Figure 5). The regions $B3$ and $B4$ meet at the cross-over point t_{2c}

$$t_{2c} = \frac{t_g \sin(\psi_1 - \psi_2)}{\cos(\psi_1)}. \tag{16}$$

For the edge in question, t_{2c} is the only point which is an element of T. The contact normal from the point t_{2c} passes through the point q_{1g}, called the *translation window* TW. If inequality (15) is satisfied, then the resultant of the finger's contact forces oppose gravity. Therefore, as the hand squeezes more and more tightly, the object must rise because its weight is overcome.

The second contact point need not occur on an edge of the polygon. It may occur on the k^{th} vertex, in which case the contact angle ψ_2 is free to vary between the inward normals of edges k and $k-1$ (see Figure 6), so that t_2 varies according to

$$t_2 = |\rho_{14}| \sin(\alpha_0 - \psi_2). \tag{17}$$

where the vector ρ_{14} is the position of the second contact point with respect to q_{14} and

α_0 is the angle of ρ_{14} measured counter-clockwise with respect to horizontal. Substituting equation (17) into equations (9) and (10) allows one to determine the liftability regions of a vertex. Figure 7 shows the edge of the second finger against the vertex in the jamming region. Tilting the finger clockwise or counter-clockwise eventually places the contact in region $B\,4$ or $B\,3$ respectively. Thus for a vertex, the liftability regions are defined as partitions of the range of possible contact angles.

2.3. Translational Lift-Off

The first goal during dexterous manipulation is to break all contact with the support. Therefore, it makes most sense to use the translation region in planning the initial grasp. With only two finger contacts, the translation region is a set of distinct points and is impossible to contact (practically speaking).[2] However, a three-point initial grasp generates a translation region with finite length, making its use practical.

One way to achieve a third finger contact (see f_5 in Figure 8) is by laying a finger against an edge of the polygon. Equilibrium equation (1) becomes

$$\mathbf{W}\,\mathbf{c} = \mathbf{g}_{ext} \tag{1}$$

$$\begin{bmatrix} \hat{a}_1 & \hat{a}_2 & \hat{a}_3 & \hat{a}_4 & \hat{a}_5 \\ t_1 & t_2 & t_3 & t_4 & t_5 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} = -mg \begin{bmatrix} \hat{a}_g \\ t_g \end{bmatrix} \tag{18}$$

where $\hat{a}_i$ is the i^{th} unit contact normal and $\hat{a}_g$ is the direction in which gravity acts. The particular solution of equation (18) in which we are interested is the one for which the third and fourth contacts break and the first, second and fifth contacts are maintained. These conditions can be stated as

$$c_3 = 0 \quad c_4 = 0 \qquad c_1 > 0 \quad c_2 > 0 \quad c_5 > 0\,. \tag{19}$$

Removing the third and fourth columns from $\mathbf{W}$ and noting that $\psi_1 = \psi_5$ and $\hat{a}_3 = \hat{a}_4 = -\,\hat{a}_g$, equation (18) can be solved for the type of initial grasp shown in

[2] The points can be vertices of the polygon, but precise contact angles are required for translation. Positioning errors make it impossible to achieve the exact contact angles.

Figure 8. Substituting the result into inequality (19) yields

$$\sin(\psi_1 - \psi_2) > 0 \tag{20}$$

$$t_5 > 0 \tag{21}$$

$$\frac{t_g \sin(\psi_1 - \psi_2)}{\cos(\psi_1)} + \frac{t_5 \cos(\psi_2)}{\cos(\psi_1)} < t_2 < \frac{t_g \sin(\psi_1 - \psi_2)}{\cos(\psi_1)} < 0 \,. \tag{22}$$

Inequalities (15) and (21) are necessary conditions for translation. We observe that inequality (21) can always be satisfied by suitably numbering the contact points. However, inequality (15) can only be satisfied by contacting the polygon on certain edges or portions of vertices. Inequality (22) defines the translation region T in which squeezing with the second finger causes the object to translate along the first finger breaking both support contacts. This region consists of all points external to the sliding region S whose normals satisfy inequality (15) and pass through the translation window. The addition of the fifth contact has caused the translation window to grow, from the point $\mathbf{q}_{1g}$ to the open line segment $(\mathbf{q}_{1g}, \mathbf{q}_{5g})$. Figure 9 illustrates the translation window TW and the translation region (and $B\,3$ and $B\,4$) of one edge for a specific placement of the first finger. For a vertex, the translation region is determined by substituting equation (17) into inequality (22) (see Figure 10).

2.4. Graphical Construction of Liftability Regions

A graphical method to determine the liftability regions of any planar curve with or without vertices for two- and three-point initial grasps has been developed based on the above analysis. It is best to illustrate the method with the following example.

2.4.1. Two-Point Initial Grasps

First, the perimeter of the object is partitioned into the complementary regions S and S' as shown in Figure 11. The region S is the set of points p for which all local contact normals have a nonpositive component in the x-direction. The region S' is the set of points whose normals have positive x-components.

$$S = \{\, \mathbf{p}:\ cos\,(\psi) \leqslant 0 \,\} \tag{23}$$

$$S' = \{\, \mathbf{p}:\ cos\,(\psi) > 0 \,\} \tag{24}$$

where the apostrophe denotes the set compliment operation.

Second, the first finger's contact is chosen to satisfy inequality (7), *i.e.* the first contact point is in the interior of S. Therefore, to satisfy equilibrium relationships (1) and (2), the second finger's contact must be in S'. Next we divide S' into regions of possible translation PT and possible jamming PJ based on inequalities (14) and (15)

$$PT = \{ \mathbf{p}: \sin(\psi_1 - \psi_2) > 0 \text{ and } \mathbf{p} \in S' \} \tag{25}$$

$$PJ = \{ \mathbf{p}: \sin(\psi_1 - \psi_2) \leqslant 0 \text{ and } \mathbf{p} \in S' \}. \tag{26}$$

The partitions are shown in Figure 12.

Third, we define the points $\mathbf{q}_{13}$, $\mathbf{q}_{14}$ and $\mathbf{q}_{1g}$. They are at the intersections of third, fourth, and gravity forces, respectively, with the line of action of the first contact force (see Figure 13).

Fourth, the region PT is broken into $B3_T$, $B4_T$ and T. Points in PT whose contact normals pass through the translation window $\mathbf{q}_{1g}$ belong to T. Points whose normals produce positive or negative moments with respect to the translation window belong to $B4_T$ or $B3_T$, respectively (see Figure 13 again).

$$B3_T = \{ \mathbf{p}: \rho_{1g} \times \hat{a}_2 < 0 \text{ and } \mathbf{p} \in PT \} \tag{27}$$

$$B4_T = \{ \mathbf{p}: \rho_{1g} \times \hat{a}_2 > 0 \text{ and } \mathbf{p} \in PT \} \tag{28}$$

$$T = \{ \mathbf{p}: \rho_{1g} \times \hat{a}_2 = 0 \text{ and } \mathbf{p} \in PT \} \tag{29}$$

where recall ρ_{ij} is the position of the second contact point relative to $\mathbf{q}_{ij}$ and $\hat{a}_2$ is the normal unit vector of the second contact.

Fifth, PJ is divided into $B3_J$, $B4_J$, and J. Points in PJ whose contact normals intersect the jamming window $[\mathbf{q}_{13}, \mathbf{q}_{14}]$ are elements of J. The points whose normals do not intersect the jamming window and generate a positive or negative moment with respect to $\mathbf{q}_{1g}$ belong to region $B4_J$ or $B3_J$, respectively (see Figure 14).

$$B3_J = \{ \mathbf{p}: \rho_{14} \times \hat{a}_2 < 0 \text{ and } \mathbf{p} \in PJ \} \tag{30}$$

$$B4_J = \{ \mathbf{p}: \rho_{13} \times \hat{a}_2 > 0 \text{ and } \mathbf{p} \in PJ \} \tag{31}$$

$$J = \{ \mathbf{p}: \rho_{13} \times \hat{a}_2 \leqslant 0 \text{ and } \rho_{14} \times \hat{a}_2 \geqslant 0 \text{ and } \mathbf{p} \in PJ \} \tag{32}$$

Finally, the liftability regions J and T are complete. However, the regions $B3$ and $B4$ must be formed by the unions of the individual $B3$'s and $B4$'s found in steps 4

and 5.

$$B3 = B3_T \cup B3_J \tag{33}$$

$$B4 = B4_T \cup B4_J \ . \tag{34}$$

Figure 15 shows all of the liftability regions. Note that by construction, the liftability regions are mutually exclusive and contain every point on the perimeter P, *i.e.*

$$J \cup B3 \cup B4 \cup T = S' \tag{35}$$

$$S' \cup S = P \tag{36}$$

$$Q \cap R = \emptyset; \qquad Q \in \{ T, B3, B4, J, S' \} \tag{37}$$

$$R \in \{ T, B3, B4, J, S' \} \tag{38}$$

where $\emptyset$ represents the empty set.

2.4.2. Three-Point Initial Grasp

The liftability regions for a three-point grasp can be formed by combining the liftability regions corresponding to the two possible two-point grasps. Let S_i, J_i, $B3_i$, $B4_i$ and T_i denote the liftability regions when considering only the i^{th} contact; $i \in \{1,5\}$. Denote by S, J, $B3$, $B4$ and T the liftability regions for the three-point grasp. In the Appendix we show that the following relationships hold:

$$S = S_1 = S_5 \tag{39}$$

$$J = J_1 \cup J_5 \cup J_T \tag{40}$$

$$B3 = B3_1 \cap B3_5 \tag{41}$$

$$B4 = B4_1 \cap B4_5 \tag{42}$$

$$T = (B3_1 \cap B4_5 \cap J_T') \cup (T_1 \cap B4_5) \cup (T_5 \cap B4_1) \tag{43}$$

Equations (39)-(43) imply that the regions J and T grow at the expense of $B3$ and $B4$. Thus we see that including an extra contact point allows a grasp to be achieved in the translation region, but makes it more difficult to tip the object.

The additional jamming region J_T and the new translation region T can be found graphically by using the new translation window (q_{1g}, q_{5g}) (see Figure 16). There are two cases: q_{15} on the right of the translation window and q_{15} on the left. For q_{15} on

the right, the translation region consists of those points, elements of S', whose contact normals pass through the translation window and (q_{5g}, q_{15}). The region J_T contains all points in S' whose normals pass through the translation window and $[q_{14}, q_{15}]$. For q_{15} on the left, the translation region consists of the points in S', whose contact normals pass through the translation window and (q_{1g}, q_{15}). The region J_T contains all points in S' whose normals pass through the translation window and $[q_{15}, q_{53}]$. These facts can be used to find the most important liftability region, T, without computing all the liftability regions for both two-point grasps.

Figure 17 shows the positions of contacts 1 and 5 for several polygons. The perimeter of each object is grown and coded to illustrate the liftability regions. Placing the second finger tip against the object where the offset perimeter is dashed indicates that squeezing the fingers will cause jamming J. Placing the second finger against the object beside the thin solid line, the bold solid line or the double-bold solid line indicate that squeezing will cause the right support contact to break $B\,4$, the left support contact to break $B\,3$, or both support contacts to break T, respectively. Squeezing with the second finger touching the object in the uncoded portion of the perimeter causes the object to slide to the left, S. The first row of the Figure shows the liftability regions for two-point initial grasps and therefore no translation regions are visible. The second row shows the regions for three-point initial grasps. Note that translation regions have appeared, but that the jamming regions have grown.

2.5. Liftability Regions with Friction

When friction is present, the method for computing the liftability regions becomes more complicated. However, a subset of the translation region can be determined much like before. The translation window is the portion of the line of action of the gravitational force lying between the friction cones of the forces $\mathbf{f}_1$ and $\mathbf{f}_5$ as illustrated in Figure 18. The translation region consists of the points on the object's perimeter for which the cone of $\mathbf{f}_2$ is completely within the translation window. Under these conditions all finger contacts will be maintained during squeezing. As in the frictionless case, we can guarantee that the object will slide up finger 1 if the following sufficient conditions are met. First,

$$\sin(\phi_{15} - \phi_2) > 0 \tag{44}$$

where $\phi_{15} = \max\{\phi_1, \phi_5\}$, ϕ_1 and ϕ_5 are the angles of the counter-clockwise most edges of cones 1 and 5, and ϕ_2 is the angle of the clockwise most edge of cone 2. Second, finger 2 contacts the object in the translation region.

Figure 19 shows a force diagram for the grasp depicted in Figure 18. The lower cone $\mathbf{f}_2$ corresponds to the possible forces acting at the second contact point. The upper cone represents all possible linear combinations of the forces generated at the other two contact points. The point E represents a particular combination of contact forces which result in the object's equilibrium. If E is on the interior of the quadrilateral $ABCD$, then the object remains fixed relative to the hand. As the internal grasp force is increased, the magnitude of $\mathbf{f}_2$ increases. Eventually E reaches the boundary AB at which point the object begins sliding up finger 1. Alternatively the internal grasp force could be reduced until the object slides down. Because in general, sliding on finger 1 results in sliding on finger 2, it is expected that the trajectory of E will terminate at points B and D. If termination occurs on AB or CD then the second finger's motion would be required to comply with that of the object. For example, if E lies on the interior of the line segment AB, then contacts 1 and 5 are sliding, because the contact forces $\mathbf{f}_1$ and $\mathbf{f}_5$ lie on the edge of their friction cone. However, contact force $\mathbf{f}_2$ lies within its cone, which implies that the second contact point on the object and finger must have identical velocities.

If inequality (44) is not satisfied, edges AB and BC become infinite making it impossible to cause the object to slide up finger 1 by squeezing. An example of this situation occurs when the contacts are on parallel edges of an object. It should be noted, however, that edges AD and DC are always finite for stable grasps, which implies that an object will always slide out of the hand if the internal grasp force is reduced enough.

One remaining concern is that the boundaries AB and CD of the quadrilateral are conservative estimates. The cone $-(\mathbf{f}_1 \cup \mathbf{f}_5)$ allows for all possible combinations of the first and fifth contact forces. Since effects of deformation will determine the nature of load sharing between contacts 1 and 5, realistic boundaries AB and CD will be on the interior of the quadrilateral, so that the predicted internal grasp force to cause sliding will be greater than the actual value.

The motion of the object for all other grasp configurations and finger motions can be predicted by solving the object motion problem. Trinkle [1988] found the motion of a frictionless object as the solution of a linear program. It was determined that such an object will move so as to minimize its rate of gain of potential energy while adhering to the velocity constraints imposed by the fingers.

2.6. The Object Motion Problem with Coulomb Friction

The object motion problem can be extended to include Coulomb friction using Peshkin's minimum power principle [Peshkin 1988]. Roughly speaking the "...minimum power principle states that a system chooses at every instant the lowest energy of 'easiest' motion in conformity with the constraints." This principle applies only to quasi-static systems subject to forces of constraint (*i.e.* normal forces), Coulomb friction forces, and forces independent of velocity. For this principle the power is defined as

$$P_{zc} = - \sum_i \mathbf{f}_{zci} \cdot \mathbf{v}_i \tag{45}$$

where $\mathbf{v}_i$ is the velocity of the i^{th} point of application of external forces and $\mathbf{f}_{zci}$ is the sum of the external forces, excluding constraint forces, applied to the i^{th} point. Included in P_{zc} are the friction and gravitational forces. The normal forces at the contacts are omitted. Thus P_{zc} is only a fraction of the object's power.

The wrench $\mathbf{w}_i$, applied to the object through the i^{th} point contact with friction can be written as the product of the i^{th} contact's unit wrench matrix $\mathbf{W}_i$ and the wrench intensity vector $\mathbf{c}_i$

$$\mathbf{w}_i = \mathbf{W}_i \, \mathbf{c}_i \qquad i = 1, \ldots, n_c \tag{46}$$

where n_c is the number of contact points,

$$\mathbf{W}_i = \begin{bmatrix} \hat{n}_i & \hat{o}_i & \hat{a}_i \\ \mathbf{r}_i \times \hat{n}_i & \mathbf{r}_i \times \hat{o}_i & \mathbf{r}_i \times \hat{a}_i \end{bmatrix}, \qquad \mathbf{c}_i = \begin{bmatrix} c_{in} \\ c_{io} \\ c_{ia} \end{bmatrix}, \tag{47}$$

$\mathbf{r}_i$ is the position of the i^{th} contact point, $\hat{a}_i$ is the contact's unit normal directed inward with respect to the object, $\hat{n}_i$ and $\hat{o}_i$ are orthogonal unit vectors defining the contact tangent plane, and the elements of $\mathbf{c}_i$ are the magnitudes of the i^{th} contact

force in the $\hat{n}_i$, $\hat{o}_i$ and $\hat{a}_i$ directions. Including all of the contacts, the equilibrium equation (1) can be written as

$$\begin{bmatrix} \mathbf{W}_n & \mathbf{W}_o & \mathbf{W}_a \end{bmatrix} \begin{bmatrix} \mathbf{c}_n \\ \mathbf{c}_o \\ \mathbf{c}_a \end{bmatrix} = - \mathbf{g}_{ext} \tag{48}$$

where

$$\mathbf{W}_p = \begin{bmatrix} \hat{p}_1 & \hat{p}_2 & \cdots & \hat{p}_{n_c} \\ \mathbf{r}_1 \times \hat{p}_1 & \mathbf{r}_2 \times \hat{p}_2 & \cdots & \mathbf{r}_{n_c} \times \hat{p}_{n_c} \end{bmatrix}; \qquad p \in \{n, o, a\}, \tag{49}$$

$$\mathbf{c}_p = \begin{bmatrix} c_{1p} \\ c_{2p} \\ \cdot \\ \cdot \\ \cdot \\ c_{n_c p} \end{bmatrix}; \qquad p \in \{n, o, a\}. \tag{50}$$

This partitioning of the wrench matrix allows us to write the sum of the friction wrenches as $\mathbf{W}_n \ \mathbf{c}_n + \mathbf{W}_o \ \mathbf{c}_o$ and the sum of the contact normal wrenches as $\mathbf{W}_a \ \mathbf{c}_a$. We now form the equation for the power as follows

$$P_{zc} = -\dot{\mathbf{q}}_{ob}{}^{\mathrm{T}} \{ \mathbf{g}_{ext} + \begin{bmatrix} \mathbf{W}_n & \mathbf{W}_o \end{bmatrix} \begin{bmatrix} \mathbf{c}_n \\ \mathbf{c}_o \end{bmatrix} \} \tag{51}$$

where $\dot{\mathbf{q}}_{ob}$ represents the object's linear and angular velocities and the superscript $^{\mathrm{T}}$ denotes matrix transposition. Given the joint velocities of the hand and arm, $\dot{\theta}$, the velocity of the object may be found by minimizing P_{zc} subject to the rigid body velocity constraints and the Coulomb friction constraints. The velocity constraints disallow interference between rigid bodies and may be written as

$$\mathbf{W}_a{}^{\mathrm{T}} \dot{\mathbf{q}}_{ob} \geqslant \mathbf{L}_a \ \dot{\theta} \tag{52}$$

where $\mathbf{W}_a{}^{\mathrm{T}} \dot{\mathbf{q}}_{ob}$ and $\mathbf{L}_a \ \dot{\theta}$ are the vectors of the normal velocity components of the contact points on the object and the hand respectively and $\mathbf{L}_a$ is related to the grasp Jacobian. The friction constraint for the i^{th} contact force as derived by Jameson [1985] is given by inequalities (53) and (54)

$$\mathbf{f}_i{}^{\mathrm{T}} \mathbf{D}_i \ \mathbf{f}_i \geqslant 0; \qquad i = 1, \ldots, n_c \tag{53}$$

$$\mathbf{f}_i^{\mathrm{T}}\,\hat{a}_i \geqslant 0\,; \qquad i = 1, \ldots, n_c \tag{54}$$

where

$$D_i = \frac{1}{1+\mu^2}\,I - \hat{a}_i\,\hat{a}_i^{\mathrm{T}}\,, \qquad \mathbf{f}_i = [\hat{n}_i\ \ \hat{o}_i\ \ \hat{a}_i]\,\mathbf{c}_i\,, \tag{55}$$

where μ is the coefficient of friction, and I is the identity matrix. Inequalities (53) and (54) may be rewritten in terms of wrench intensities as

$$\mathbf{c}_i^{\mathrm{T}}\,\Phi_i\,\mathbf{c}_i \geqslant 0\,; \qquad i = 1, \ldots, n_c \tag{56}$$

$$c_{ia} \geqslant 0\,; \qquad i = 1, \ldots, n_c \tag{57}$$

where

$$\Phi_i = \begin{bmatrix} \hat{n}_i^{\mathrm{T}} \\ \hat{o}_i^{\mathrm{T}} \\ \hat{a}_i^{\mathrm{T}} \end{bmatrix} \mathbf{D}_i \begin{bmatrix} \hat{n}_i & \hat{o}_i & \hat{a}_i \end{bmatrix}. \tag{58}$$

Thus far we know that we must minimize P_{zc} subject to inequalities (52), (56) and (57). One might think that the equilibrium equation (48) should also be used to constrain the solution. However, by formulating and examining the dual optimization problem, one finds that equilibrium equation (48) is implicitly satisfied and that inequality (57) is redundant.

The primal problem is defined as

$$\text{Minimize} \qquad P_{zc} = -\dot{\mathbf{q}}_{ob}^{\mathrm{T}}\left\{\mathbf{g}_{ext} + \begin{bmatrix}\mathbf{W}_n & \mathbf{W}_o\end{bmatrix}\begin{bmatrix}\mathbf{c}_n \\ \mathbf{c}_o\end{bmatrix}\right\} \tag{51}$$

$$\text{Subject to:} \qquad \mathbf{W}_a^{\mathrm{T}}\,\dot{\mathbf{q}}_{ob} \geqslant \mathbf{L}_a\,\dot{\theta} \tag{52}$$

$$\mathbf{c}_i^{\mathrm{T}}\,\Phi_i\,\mathbf{c}_i \geqslant 0\,; \qquad i = 1, \ldots, n_c \tag{56}$$

To derive the dual problem, the objective function is augmented by attaching the constraints with Lagrange multipliers. Applying the Kuhn-Tucker optimality conditions [Beveridge 1970] gives

$$\text{Maximize} \qquad Z = \dot{\theta}^{\,T}\,\mathbf{L}_a\,\lambda \tag{59}$$

$$\text{Subject to:} \qquad \mathbf{g}_{ext} + \begin{bmatrix}\mathbf{W}_n & \mathbf{W}_o\end{bmatrix}\begin{bmatrix}\mathbf{c}_n \\ \mathbf{c}_o\end{bmatrix} + \mathbf{W}_a\,\lambda = 0 \tag{60}$$

$$\mathbf{W}_n^{\mathrm{T}}\,\dot{\mathbf{q}}_{ob} + 2\sum_i \eta_i\,\Phi_{in}\,c_{in} = 0 \tag{61}$$

$$\mathbf{W}_o{}^{\mathrm{T}}\,\dot{\mathbf{q}}_{ob} + 2\sum_i \eta_i\ \Phi_{io}\ c_{io} = 0 \tag{62}$$

$$2\sum_i \eta_i\ \Phi_{ia}\ c_{ia} = 0 \tag{63}$$

$$\lambda \geqslant 0 \tag{64}$$

$$\eta \geqslant 0 \tag{65}$$

where η and λ are vectors of Lagrange multipliers associated with inequalities (56) and (52) respectively and $\Phi_{:p}$ is the partition of Φ corresponding to the component of the i^{th} contact force in the direction of the unit vector $p \in \{\hat{n}, \hat{o}, \hat{a}\}$. Since the vector of Lagrange multipliers, λ, associated with the velocity constraints is equivalent to the vector of normal wrench intensities, $\mathbf{c}_a$, constraint (60) is the equilibrium equation and constraint (64) is equivalent to inequality (57).

At the optimal solution, both the primal constraints and the dual constraints are satisfied, therefore the primal problem defined by the nonlinear program, (51), (52) and (56), need not include the equilibrium equation. Another interesting point is that for all feasible solutions, the primal and dual objective functions satisfy the following relationship

$$\dot{\theta}^{\mathrm{T}}\,\mathbf{L}_a{}^{\mathrm{T}}\,\mathbf{c}_a \leqslant -\dot{\mathbf{q}}_{ob}{}^{\mathrm{T}}\,\mathbf{g}_{ext} - \dot{\mathbf{q}}_{ob}{}^{\mathrm{T}}\begin{bmatrix}\mathbf{W}_n & \mathbf{W}_o\end{bmatrix}\begin{bmatrix}\mathbf{c}_n \\ \mathbf{c}_o\end{bmatrix} \tag{66}$$

with equality holding only at the optimal solution. The term on the left hand side of the inequality is the power applied to the object by the forces of constraint. The terms on the right are the rate of gain of potential energy and the power dissipation through Coulomb friction. Thus, at the optimal solution, expression (66) has the following physical interpretation. The motion of the fingers in the direction of the contact normals supplies power to the object. Some of that power is lost to friction. What remains goes into lifting the object. Consequently every suboptimal solution must defy conservation of energy.

The primal problem (51), (52) and (56) is complete for rolling contacts, but not for sliding contacts. In the case of sliding, the contact force must be anti-parallel to the relative velocity at the contact and lie on the boundary of the cone. This constraint was concisely expressed by Jameson [1985] as

$$\mathbf{f}_i \cdot (\mathbf{v}_{ti} \times \hat{a}_i) = 0; \qquad i = 1, \ldots, n_c \tag{67}$$

$$\mathbf{f}_i \cdot \mathbf{v}_{ti} \leqslant 0 ; \qquad i = 1, \ldots, n_c \tag{68}$$

where $\mathbf{v}_{ti}$ is the relative contact velocity. Noting that

$$\mathbf{v}_{ti} = \mathbf{W}_i^{\mathrm{T}} \dot{\mathbf{q}}_{ob} - \mathbf{L}_i \dot{\theta} ; \qquad i = 1, \ldots, n_c , \tag{69}$$

where $\mathbf{L}_i$ is the transmitted Jacobian of the i^{th} contact [Trinkle 1987], relations (67) and (68) may be written in terms of the wrench intensities and the object and arm velocities as

$$\mathbf{c}_i^{\mathrm{T}} \mathbf{P}_i \dot{\mathbf{q}}_{ob} - \mathbf{c}_i^{\mathrm{T}} \mathbf{Q}_i \dot{\theta}_i = 0 ; \qquad i = 1, \ldots, n_c \tag{70}$$

$$\mathbf{c}_i^{\mathrm{T}} \mathbf{R}_i \dot{\mathbf{q}}_{ob} - \mathbf{c}_i^{\mathrm{T}} \mathbf{S}_i \dot{\theta}_i \leqslant 0 ; \qquad i = 1, \ldots, n_c \tag{71}$$

where

$$\mathbf{P}_i = \mathbf{W}_i^{\mathrm{T}} \mathbf{B}_i \mathbf{W}_i , \qquad \mathbf{Q}_i = \mathbf{W}_i^{\mathrm{T}} \mathbf{B}_i \mathbf{L}_i , \tag{72}$$

$$\mathbf{R}_i = \mathbf{W}_i^{\mathrm{T}} \mathbf{C}_i \mathbf{W}_i , \qquad \mathbf{S}_i = \mathbf{W}_i^{\mathrm{T}} \mathbf{C}_i \mathbf{L}_i , \tag{73}$$

$$\mathbf{B}_i = [\, \mathbf{I} \;\; \mathbf{0} \,], \qquad \mathbf{C}_i = [\, \mathbf{A}_i \;\; \mathbf{0} \,], \tag{74}$$

$$\mathbf{A}_i = \begin{bmatrix} 0 & -a_{iz} & a_{iy} \\ a_{iz} & 0 & -a_{iz} \\ -a_{iy} & a_{iz} & 0 \end{bmatrix} , \tag{75}$$

$\mathbf{I}$ is the identity matrix and $\mathbf{0}$ is a matrix of zeros.

The complete object motion problem is now given by

$$\text{Minimize} \qquad P_{zc} = -\dot{\mathbf{q}}_{ob}^{\mathrm{T}} \{ \mathbf{g}_{ext} + \begin{bmatrix} \mathbf{W}_n & \mathbf{W}_o \end{bmatrix} \begin{bmatrix} \mathbf{c}_n \\ \mathbf{c}_o \end{bmatrix} \} \tag{51}$$

$$\text{Subject to:} \qquad \mathbf{W}_a^{\mathrm{T}} \dot{\mathbf{q}}_{ob} \geqslant \mathbf{L}_a \dot{\theta} \tag{52}$$

$$\mathbf{c}_i^{\mathrm{T}} \Phi_i \mathbf{c}_i \geqslant 0 ; \qquad i \in \Omega \qquad \textit{rolling} \tag{56}$$

$$\mathbf{c}_i^{\mathrm{T}} \Phi_i \mathbf{c}_i = 0 ; \qquad i \in \Psi \qquad \textit{sliding} \tag{76}$$

$$\mathbf{c}_i^{\mathrm{T}} \mathbf{P}_i \dot{\mathbf{q}}_{ob} - \mathbf{c}_i^{\mathrm{T}} \mathbf{Q}_i \dot{\theta}_i = 0 ; \qquad i \in \Psi \qquad \textit{sliding} \tag{70}$$

$$\mathbf{c}_i^{\mathrm{T}} \mathbf{R}_i \dot{\mathbf{q}}_{ob} - \mathbf{c}_i^{\mathrm{T}} \mathbf{S}_i \dot{\theta}_i \leqslant 0 ; \qquad i \in \Psi \qquad \textit{sliding} \tag{71}$$

where Ω and Ψ represent the set of contact points assumed to be sliding and rolling, respectively. This nonlinear program is called the velocity formulation of the object motion problem. Constraints (56), (70), (71) and (76) define the Coulomb friction

model without which friction forces could create rather than dissipate power resulting in an unbounded objective function.

To determine $\dot{q}_{ob}$, the object motion problem must be solved for all possible permutations of sliding, rolling, and separating at each contact point while enforcing the appropriate constraints. If there are n_p permutations, then there will be $n_f \leqslant n_p$ feasible solutions. The motion corresponding to the feasible solution of least power is the one which the object will execute. If no feasible solution exists, then the motion of the hand is kinematically inadmissible. If the minimum power solution is unbounded, then the finger motions cause the grasp configuration to become unstable.

Clearly the general object motion problem depends on not just the grasp configuration, but the velocities of the contacts. Therefore it is not possible to define liftability regions and use them in planning. Each grasp and desired vector of joint velocities must be considered separately.

3. MANIPULATION PLANNING

The ultimate goal of our analysis is to provide a framework in which intelligent dexterous manipulation can be planned for articulated mechanical hands. Intelligent dexterous manipulation can be considered to be the continuous evolution of a stable grasp from an undesirable configuration to one appropriate to the performance of a given task. The simplest task is a pick-and-place operation which can be easily performed with a parallel-jawed gripper if friction is significant. However, it is useless in the frictionless case. To hold an object without friction requires that the hand envelop the object much as one would grip a wet piece of ice. Therefore under slippery conditions an articulated hand is necessary.

Figure 20 shows the simplest two-dimensional articulated hand performing an enveloping grasp (also called a form closure grasp [Lakshminarayana 1978]) of a frictionless object. For the remainder of this paper, objects are considered to be convex polygons and the hand is assumed to be the one pictured in Figure 20.

An enveloping grasp must satisfy the equilibrium relationships

$$\mathbf{W}_a \, \mathbf{c}_a = -\mathbf{g}_{ext} \tag{1}$$

$$\mathbf{c}_a \geqslant 0 \tag{2}$$

for any external wrench acting on the object (recall that the subscript a identifies the normal components of the contact forces). Equivalently, the nonnegative column span of $\mathbf{W}_a$ must be equal to the space of possible external wrenches (for the two-dimensional problem, $g_{ext} \in R^3$ where R^3 represents Euclidean three-space). Another way to think of envelopment is that if the joints are locked, then the object cannot move. That is, the object is completely restrained by the form or surface of the hand. This constraint condition is expressed by substituting 0 for θ in inequality (52),

$$\mathbf{W}_a^{\mathrm{T}} \dot{\mathbf{q}}_{ob} \geqslant 0 \tag{77}$$

and requiring that only the trivial solution exist, *i.e.* $\dot{\mathbf{q}}_{ob} \equiv 0$.

A second type of stable grasp is called a force closure grasp. It still satisfies relationships (1) and (2), but not for all possible external wrenches. For force closure, the nonnegative column span of $\mathbf{W}_a$ defines a convex cone C^+ [Goldman 1956] which is a subset of the space of possible external wrenches. If the negative of the external wrench lies within C^+, then the grasp exhibits force closure. Therefore stability depends on the external wrench or force, hence the name force closure. Figure 21 shows a force closure grasp. If gravity were acting up the page instead of down, the object would fall toward the palm.

Assuming our goal is to perform safe pick-and-place operations, each object must be manipulated away from its support surface and into an enveloping grasp. To achieve this goal, planning is broken into two phases: the *pre-lift-off* phase and the *lifting phase.*

3.1. Pre-Lift-off Phase

The objective of the pre-lift-off phase is to find a realizable initial grasp which guarantees that the object can be manipulated away from the support. The simplest way to achieve this objective is to choose a grasp in the translation region. Squeezing then causes the object to translate upward, breaking all contact with the support (see Figure 22). All possible initial grasps of this type can be found using the following procedure.

1. Designate one finger to lie along an edge of the polygon.
2. Compute the translation region T.

3. Solve for the joint angles to contact the object in T with the other finger.

4. Check for geometric interference.

5. If T is empty or step 3 has no solution or interference is detected, reject the grasp; otherwise accept the grasp as feasible.

6. Return to step 1 until all combinations of finger and edge have been considered.

When choosing an initial grasp, preference should be given to those for which the second contact point is near the center of a large translation region, because those grasps will be least sensitive to position errors. For example, consider the intended initial grasp shown in Figure 22. Position errors could give rise to any or all of the following scenarios:

1. Error in vertical position of finger 1, δy: q_{1g} moves up or down.

2. Error in the angle of finger 2, $\delta\psi_2$: the normal of contact 2 is altered.

3. Error in the angle of finger 1, $\delta\psi_1$: contact 1 or contact 5 is not achieved.

Errors of types 1 and 2 do not deleteriously affect the nature of lift-off of the object as long as the vertical error δy and the angular error $\delta\psi_2$ adhere to the following inequality

$$\delta y - \rho_{cg} \frac{\delta\psi_2}{\cos\psi_2} < l_{2g} \tag{78}$$

where ρ_{cg} is the distance from the center of gravity of the object to the second contact point and l_{2g} is the distance between the points q_{1g} and q_{2g}. Inequality (78) is valid provided that $\delta\psi_2$ is small.[3] Violation of inequality (78) implies that the second contact normal $\hat{a}_2$ passes below the translation window placing the contact in region $B4$. Thus the object will tip maintaining the third contact, defeating our goal. The third type of error causes the translation window to shrink to a point, either q_{1g} or q_{5g}. Therefore $\hat{a}_2$ passes either above or below the translation window, respectively. In the former case, the angle of finger 1 is less than commanded. Since the translation window becomes the point q_{1g}, $\hat{a}_2$ passes above it, so the grasp is in region $B3$. Upon squeezing, the object will rotate clockwise, aligning its edge with finger 1. This alignment

[3] A similar expression applies if the contact is on an edge of the object rather than a vertex.

opens the translation window and changes the nature of the grasp back to what was originally intended. Continued squeezing causes the object to translate up finger 1. In the latter case the angle of finger 1 is too large causing the translation window to become the point q_{5_g} and the second contact to be in region $B4$. Again squeezing causes an aligning rotation of the polygon followed by translation up the finger as planned. Figure 23 illustrates and initial grasp exhibiting the third type of error. As squeezing commences, the polygon's right-most edge aligns with the edge of the finger. Continued squeezing causes the object to translate up the finger.

3.2. Lifting Phase

The lifting phase begins when the object no longer contacts the support. For this to occur, the object must be in a force closure grasp in the hand. No contact may remain on the support.

The goal of the lifting phase is to manipulate the object into an enveloping grasp. In doing so, the object may either be stable through force closure or unstable. Instability is undesirable, because it results in the object's falling. Even though an unstable object will eventually come to rest in a stable configuration, the final configuration cannot be predicted by our quasi-static technique. Therefore, it is imperative that an enveloping grasp be gained without ever losing force closure.

Assume that the initial grasp has been chosen in a translation region. As lifting begins, the object contacts the hand at two points on one finger and at one point on the other. Since force closure requires three contacts, all of these contacts must be maintained until a fourth contact is achieved. If the object is enveloped, *i.e.*, the grasp has form closure, then the grasp is complete and the lifting phase ends. If not, one of the contacts must break as manipulation continues. Thus it is apparent that during the lifting phase, the object must translate relative to one of the fingers (assuming flat fingers) until the object contacts the palm. Once the palm has been contacted, translation is possible only if one finger loses contact with the object. In an enveloping grasp, both fingers and the palm must contact the object. Therefore, we prefer to manipulate the object maintaining contact with both fingers. However, we analyze one planning strategy which allows contact to be lost with one finger as the object slides on the other.

A force closure grasp is one for which the negative of the gravitational wrench acting on the object is within the convex cone C^+ defined by equations (1) and (2). Figure 24 shows a convex cone and an external wrench for a typical force closure grasp. Examination of the equilibrium relationships (1) and (2) reveals how to manipulate a force closure grasp. Denote by $\mathbf{y}_i$, the i^{th} column of $\mathbf{W}_a$

$$\mathbf{y}_i = \begin{bmatrix} \cos\psi_i \\ \sin\psi_i \\ t_i \end{bmatrix} \tag{79}$$

where recall ψ_i is the angle of the i^{th} contact normal and t_i is the moment of that contact normal measured with respect to the summing point $\mathbf{q}$. Choosing $\mathbf{q}$ to be the center of gravity of the object, the gravity moment is zero during manipulation. Thus the convex cone can be projected onto the *Lifting Phase Plane* (LPP) formed by the $\cos\psi_i$ and t_i axes. In this plane, the cone becomes the LPP triangle, the gravity force maps to the origin, and two contacts on a flat link map to points on a vertical line separated by the distance that separates the contacts on the link (see Figure 25). The necessary and sufficient conditions for a grasp to have force closure are that the LPP triangle enclose the origin and the *sine* of the difference of the contact angles on the two fingers be greater than zero

$$\sin(\psi_1 - \psi_2) > 0. \tag{15}$$

We desire to squeeze the object until it contacts the palm. However, while squeezing we must make sure that the LPP triangle always contains the origin and inequality (15) and is never violated. If the initial grasp is in the translation region, then initially both of the conditions are satisfied. As the fingers are squeezed together, the quantity, $\psi_1 - \psi_2$, may only decrease (if the singly-contacted finger contacts a vertex of the object) or remain constant (if the singly-contacted finger contacts an edge of the object). Because the palm eventually prevents squeezing from continuing, the quantity is bounded from below by zero. At the start of manipulation, the angular difference between the contact normals, $\psi_1 - \psi_2$, is in the interval bounded by zero and pi. During squeezing, the difference reduces, but remains in the interval. Because the *sine* function is positive in that interval, the second condition is guaranteed to be satisfied throughout the entire manipulation.

The condition that the LPP triangle contain the origin at all times must be checked by considering the trajectories of the triangle's vertices. Their positions are affected by three variables, the two joint angles, θ_1 and θ_2, and the angle of the palm θ_p (see Figure 22). Consider the hypothetical trajectory shown in Figure 26. Because the object will translate up finger 1, finger 2, called the *pusher,* is rotated counter clockwise while finger 1 is held stationary. As the pusher rotates, vertices $\mathbf{y}_1$ and $\mathbf{y}_5$ of the LPP triangle remain fixed while vertex $\mathbf{y}_2$ follows a path qualitatively like the one shown beginning at point A. At the point C, $\mathbf{y}_2$ jumps to D. The discontinuity is caused by the edge of the second finger contacting the k^{th} vertex $\mathbf{v}_k$ of the object. At the instant the discontinuity occurs, there are four contacts, but only three can be maintained as the fingers continue to squeeze. Since D is within the valid region for the second vertex, the new contact remains and the previous contact on that finger breaks. At the point E, the trajectory jumps outside of the valid region to the point F. If the new contact were to remain (as it did at D), the interior of the LPP triangle would exclude the origin and the object would become unstable. However, the trajectory jumped into the region labeled $B\,5$. This means that the fifth contact point (which is on finger 1) will break. The new LPP triangle has vertices labeled $\mathbf{y}_1$, $\mathbf{y}_2$ and $\mathbf{y}_6$ (see Figure 27). Since the new vertices contain the origin, the grasp is still table, but now the object will slide up the second finger rather than the first. Continuing squeezing, the first finger now acts as the pusher, rotating clockwise and the second finger is held fixed. This strategy of using one finger as a pusher and holding the other finger fixed is called the *pusher.*

Figure 27 shows the trajectory of vertex $\mathbf{y}_1$ crossing the boundary of the new valid region into the region $B\,2$. When this happens, the second contact breaks, leaving only the first contact on finger 1 and the sixth contact on finger 2. The grasp loses force closure becoming unstable, so the object falls. However, if an enveloping grasp is achieved before the object becomes unstable, then the pusher strategy can be used successfully.

An interesting property of the LPP trajectory is that if a vertex moves out of the valid region in a continuous manner (as at G), the object becomes unstable, because a contact point is lost without gaining a new one. However, if the vertex jumps outside of the valid region (as at F), the object remains stable and the finger on which the object translates switches. Any motion of the trajectory within the valid region represents stable translation of the object without switching pushers.

If the pusher strategy fails (see Figure 28), one could try the *roll* strategy during which the finger angles are fixed as the palm is rotated. If the hand can be rotated far enough without losing force closure, the object will slide down the finger until it touches the palm. Afterward, the fingers may be closed around the object creating an enveloping grasp. Figure 29 shows the trajectories of the vertices of the LPP triangle corresponding to a clockwise rotation of the hand shown in Figure 21. As the hand rotates, the object does not move relative to it and therefore the moment arms, t_i; $i \in \{1, 2, 5\}$, of the contacts do not change. The result is that the corners of the LPP triangle can move only horizontally and since the normals of contacts 1 and 5 have the same direction, $\mathbf{y}_1$ and $\mathbf{y}_5$ move at a common rate. At B the right finger becomes horizontal. After slightly more rotation, the second contact breaks and the object slides towards the palm. Closing the fingers around the object achieves the enveloping grasp.

We would like to know what conditions guarantee the success of the roll strategy. A condition of necessity is that t_1 and t_5 have opposite signs. If they have the same sign, the object could not be stable when sliding down the finger, because the gravity force would not pass between the two supporting contacts. Given that necessity is met, a sufficient condition is that t_2 equal zero. The validity of this condition can be argued for as follows. Since the grasp satisfies inequality (15), $\mathbf{y}_2$ is always on the left side of the lifting phase plane. As the hand rotates clockwise, $\mathbf{y}_1$ and $\mathbf{y}_5$ move toward the left. Until the right finger becomes horizontal, $\mathbf{y}_1$ and $\mathbf{y}_5$ are on the right side of the LPP. Therefore the LPP triangle always contains the origin. As the right finger passes through horizontal, $\mathbf{y}_1$ and $\mathbf{y}_5$ cross the t_i axis causing the second contact to break as the object slides towards the palm on finger 1.

The pusher and roll strategies can be combined as illustrated in Figure 30. If possible, the pusher strategy should be used to cause vertex $\mathbf{y}_2$ to move to the $\cos\psi_i$ axis (see point B in Figure 26). After this, the roll strategy can be used to safely complete the grasp provided that t_1 and t_5 have opposite signs.

If friction is present, the same strategies are valid, however inequality (44)

$$\sin(\phi_{15} - \phi_2) > 0 \tag{44}$$

must be satisfied rather than inequality (15) and each edge, $\mathbf{y}_i$ of the convex cone must be replaced with $\tilde{\mathbf{y}}_i$, the appropriate edge of the friction cone for each contact

$$\tilde{\mathbf{y}}_i = \begin{bmatrix} \cos(\psi_i + \alpha) \\ \sin(\psi_i + \alpha) \end{bmatrix}; \quad i \in \{1,5\} \qquad \tilde{\mathbf{y}}_2 = \begin{bmatrix} \cos(\psi_2 \pm \alpha) \\ \sin(\psi_2 \pm \alpha) \end{bmatrix} \tag{80}$$

where $\tilde{t}_i$ is the moment arm of the i^{th} contact force and α is the friction angle. The sign of α in the expression for $\tilde{y}_2$ is dependent on the relative velocity of the second contact point. Since it would be useful to control the sign of α, it would be preferable that the second finger to have more than one link.

4. CONCLUSION

Manipulation with articulated hands is usually carried out under force control to prevent slipping at the contacts. Dissallowing slipping unnecessarily limits the dexterous capability of a hand and cannot be done in the absence of friction. We have addressed the problem of achieving an enveloping grasp in the plane based on sliding contacts. Our solution was based on the frictionless case, but extended where appropriate to include Coulomb friction. Planning was broken into two phases: the *pre-lift-off phase* and the *lifting phase*. The goal of the pre-lift-off phase was to manipulate the object so as to cause it to lose contact with its support. This led us to define and analyze the *liftability* of planar objects. Given the contact configuration of one finger, the liftability regions of the object could be determined and used to plan the placement of the other finger to complete the initial grasp. It was determined that initial grasps in the *translation region* of the object should be used, because they achieve the goal of the pre-lift-off phase most easily and are insensitive to position errors. For the lifting phase, planning was done geometrically in the *lifting phase plane* providing a simple method to monitor grasp stability and to predict which contacts were gained and lost as the grasp evolved. Manipulation trajectories generated by our planning technique can be executed under position control. Force control is unnecessary even when friction is included.

Even though our analysis in Section 3 was two-dimensional, the planning methods can be applied to three-dimensional objects which can be modeled as generalized cylinders by planning manipulation using the appropriate cross sections of the cylinders. In the event that such modeling is inappropriate, the *object motion problem* we have formulated can be used incrementally to plan manipulation trajectories in three dimensions.

5. ACKNOWLEDGEMENTS

The authors would like to extend their thanks to Dr. J.M. Abel and Dr. M.C. Peshkin for their suggestions.

This research was performed at the University of Pennsylvania and the University of Wollongong, and was supported in part by the following grants: IBM 6-28270, ARO DAA6-29--84-k-0061, AfOSR 82-NM-299, NSF ECS 8411879, NSF MCS-8219196-CER, NSF MCS 82-07294, AVRO DAAB07-84-K-F077, and NIH 1-RO1-HL-29985-01. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the granting agencies.

6. REFERENCES

Asada, H. and By, A., 1984 (Aug.), "Kinematic Analysis and Design for Automatic Workpart Fixturing in Flexible Assembly," *Proc. Second International Symposium of Robotics Research,* Kyoto, pp. 50-57.

Beveridge, G.S.G. and Schechter, R.S., 1970, *Optimization: Theory and Practice,* New York, McGraw-Hill.

Brooks, R.A., 1983, "Planning Collision-Free Motions for Pick-and-Place Operations, *International Journal of Robotics Research,* 2(4):19-44.

Brost, R.C., 1985 (July), "Planning Robot Grasping Motions in the Presence of Uncertainty," *CMU-RI-85-12* Department of Computer Science, Carnegie-Mellon University, Pittsburgh.

Cutkosky, M.R., 1985, *Robotic Grasping and Fine Manipulation,* Boston, Kluwer Academic Publishers.

Fearing, R.S., 1987 (March/April), "Some Experiments with Tactile Sensing During Grasping," *Proc. IEEE International Conference on Robotics and Automation,* Raleigh, NC, pp. 1637-1643.

Goldman, A. and Tucker, A., 1956 "Polyhedral Convex Cones," in *Linear Inequalities and Related Systems,* ed. H. Kuhn and H. Tucker, pp. 19-40, Princeton University Press, Princeton.

Hanafusa, H. and Asada, H., 1982, "Stable Prehension by a Robot Hand with Elastic Fingers," in *Robot Motion,* ed. Brady et al., Cambridge, MIT Press, pp. 337-359.

Holzmann, W. and McCarthy, J.M., 1985 (March), "Computing the Friction Forces Associated with a Three-Fingered Grip," *Proc. IEEE International Conference on Robotics and Automation,* St. Louis, pp. 594-600.

Jameson, J.W., 1985 (June), "Analytic Techniques for Automated Grasp," *Ph.D. dissertation,* Department of Mechanical Engineering, Stanford University, Stanford.

Kerr, J.R., 1984 (Dec.), "An Analysis of Multi-Fingered Hands," *Ph.D. dissertation,* Department of Mechanical Engineering, Stanford University, Stanford.

Kobayashi, H., 1984 (Aug.), "On the Articulated Hands," *Proc. Second International Symposium on Robotics Research,* Kyoto, pp. 128-135.

Lakshminarayana, K, 1978, "The Mechanics of Form Closure," *ASME Report no. 78-DET-32.*

Laugier, C. and Pertin, J., 1983 (Jan.), "Automatic Grasping: A Case Study in Accessibility Analysis," *research report no. 342.,* Toulouse.

Li, Z. and Sastry, S., 1987 (March/April), "Task Oriented Optimal Grasping by Multifingered Robot Hands," *Proc. IEEE International Conference on Robotics and Automation,* Raleigh, NC, pp. 389-394.

Mason, M.T., 1979 (April), "Compliance and Force Control for Computer Controlled Manipulators," *MS thesis,* MIT AI Lab, Cambridge.

Mason, M.T. and Salisbury, J.K, 1985, *Robot Hands and the Mechanics of Manipulation,* Cambridge, MIT Press.

Nguyen, V-D., 1986 (May), "The Synthesis of Stable Force-Closure Grasps," *MS thesis,* MIT AI Lab, Cambridge.

Ohwovoriole, M.S., 1980 (April), "An Extension of Screw Theory and Its Application to the Automation of Industrial Assemblies," *Ph.D dissertation,* Department of Mechanical Engineering, Stanford University, Stanford.

Okada, T., 1982 (May/June), "Computer Control of Multijointed Finger System for Precise Object Handling," *IEEE Transactions on Systems, Man, and Cybernetics,* SMC-12(3):289-298.

Peshkin, M.A. and Sanderson, A.C., 1988 (April), "Minimization of Energy in Quasistatic Manipulation," *proc. IEEE International Conference on Robotics and Automation,* Philadelphia.

Peshkin, M.A. and Sanderson, A.C., 1987 (April), "Planning Robotic Manipulation Strategies for Sliding Objects," *proc. IEEE International Conference on Robotics and Automation*, Raleigh.

Paul, R.P. 1972, "Modeling, Trajectory Calculation and Servoing of a Computer Controlled Arm," *Stanford Artificial Intelligence Laboratory, AIM 177,* Stanford University, Stanford.

Salisbury, J.K., 1982 (July), "Kinematic and Force Analysis of Articulated Hands," *Ph.D. dissertation,* Department of Mechanical Engineering, Stanford University, Stanford.

Trinkle, J.C., 1985 (November), "Frictionless Grasping," *MS-CIS-85-46, GRASP Lab 52,* Department of Computer and Information Science, University of Pennsylvania, Philadelphia.

Trinkle, J.C., 1987 (June), "The Mechanics and Planning of Enveloping Grasps," *Ph.D. dissertation in Systems Engineering, MS-CIS-87-46, GRASP Lab 108,* Department of Computer and Information Science, University of Pennsylvania, Philadelphia.

Trinkle, J.C., 1988, "An Investigation of Frictionless, Enveloping Grasping in the Plane," *International Journal of Robotics Research,* in press.

Whitney, D.E., 1982 (March), "Quasi-Static Assembly of Compliantly Supported Rigid Parts," *Journal of Dynamic Systems, Measurement, and Control,* vol. 104, pp. 65-77.

Wolter, J.D., Volz, R.A., and Woo, A.C., 1984 (Feb.), "Automatic Generation of Gripping Positions," *technical report,* University of Michigan, Ann Arbor.

7. Appendix

Here we show how the liftability regions of two two-point initial grasps can be combined to form the liftability regions of one three-point initial grasp.

7.1. The Sliding Region, S

The sliding region S for an object is independent of the number and positions of finger contacts; it depends only on the geometry of the object (see inequality (23)). Therefore we immediately write the following equation

$$S = S_1 = S_5 . \tag{39}$$

7.2. The region S'

The other relevant liftability regions are
J, $B3$, $B4$, T, J_1, $B3_1$, $B4_1$, T_1, J_5, $B3_5$, $B4_5$, and T_5, where the non subscripted regions are due to the three-point initial grasp and the subscript i ; $i \in \{1, 5\}$ implies a two-point grasp using contacts 2 and i. The liftability regions of the two two-point initial grasps must satisfy inequality (35)

$$J_1 \cup B3_1 \cup B4_1 \cup T_1 = S' \tag{A1}$$

$$J_5 \cup B3_5 \cup B4_5 \cup T_5 = S'. \tag{A2}$$

We begin our derivation of equations (40)-(43) with the following true statement.

$$S' \cap S' = S' . \tag{A3}$$

Substituting equations (A1) and (A2) into (A3) and expanding gives

$$\begin{aligned}
&(J_1 \cap J_5) \cup (J_1 \cap B3_5) \cup (J_1 \cap B4_5) \cup (J_1 \cap T_5) \\
&\cup (B3_1 \cap J_5) \cup (B3_1 \cap B3_5) \cup (B3_1 \cap B4_5) \cup (B3_1 \cap T_5) \\
&\cup (B4_1 \cap J_5) \cup (B4_1 \cap B3_5) \cup (B4_1 \cap B4_5) \cup (B4_1 \cap T_5) \\
&\cup (T_1 \cap J_5) \cup (T_1 \cap B3_5) \cup (T_1 \cap B4_5) \cup (T_1 \cap T_5) = S' .
\end{aligned} \tag{A4}$$

As a consequence of equations (37) and (38), the 16 sets formed by intersection are mutually exclusive.

Any set which is formed by intersection with J_1 or J_5 belongs to J. This statement is motivated physically by the fact that for a two-point initial grasp in the jamming region the object can only be further constrained by adding another contact point. Noting that the top row of, and the left hand most column of the left hand side of equation (A4) are equivalent to J_1 and J_5 respectively, we write:

$$J \supset J_1 \bigcup J_5 . \tag{A5}$$

Equation (A5) accounts for 7 of the sets in equation (A4).

The nature of lift-off for the remaining 9 sets can be deduced by considering Figure A1 using the following facts:

1. A stable grasp must have 3 contact points during manipulation (more contacts cause static indeterminacy and interference; fewer lead to grasp instability).
2. The positive or negative cones of a stable grasp must "see each other" [Nguyen 1986] where the positive force cone of $\mathbf{f}_4$ and $\mathbf{f}_5$ is labeled in Figure A1 as C_{54}.
3. The positive or negative force cone is the set of points defined respectively by a positive or negative linear combination of the forces using their intersection as the cone's apex.
4. A pair of cones see each other or are mutually visible if each cone contains the other's apex.

Consider the set $B\,3_1 \bigcap B\,3_5$. Referring to Figure A1, a point in S' can only belong to both $B\,3_1$ and $B\,3_5$ in two ways: the contact normal $\hat{a}_2$ passes upward through both open half lines (q_{5g}, ∞_5^+)[4] and (q_{1g}, ∞_1^+) (as shown in Figure A1) or downward through both half lines (q_{14}, ∞_1^-) and (q_{54}, ∞_5^-). Let one force cone be C_{2g} as shown.[5] To determine the nature of lift-off we must find a cone within sight of C_{2g} which can see C_{2g}. Only cones C_{14} and C_{54} satisfy this requirement. Since neither

---

[4] By "... upward through the half-line (q_{5g}, ∞_5^+) ...," we imply the satisfaction of $\sin(\psi_5 - \psi_2) > 0$. Similarly "downward" implies satisfaction with the inequality reversed. To define upward and downward with respect to half lines along the line of the first contact force, substitute ψ_1 for ψ_5.

[5] To make two cones requires four forces. They are the three contact forces and the gravity force.

cone is constructed using the third contact force, that contact must break, *i.e.* the grasp must be in $B\,3$. Both cones include the fourth contact force, so that contact is maintained. However, to maintain exactly 3 contacts during manipulation, either the first or fifth contact must also break. The one which will break is determined by considering the motions which will be made by the fingers. For example, if finger 1 remains fixed as finger 2 squeezes, then the first contact will break while the fifth contact is maintained. In this case the instantaneous center of rotation of the object is either the point $\mathbf{q}_{14}$ or $\mathbf{q}_{54}$. If we assume that the fifth contact breaks implies that the center of rotation is $\mathbf{q}_{14}$. Rotation counter-clockwise about $\mathbf{q}_{14}$ causes interference at the third contact; clockwise rotation causes interference at the fifth contact. Thus the assumption that the fifth contact breaks is inconsistent with the instantaneous kinematics of the grasp. Therefore, the first contact (and the third contact) must break while the fifth contact (and the second and fourth contacts) is maintained. This conclusion is validated by the fact that instantaneous clockwise rotation about $\mathbf{q}_{54}$ does not cause interference.

The result of the above arguments is that we may write:

$$B\,3 \supset B\,3_1 \bigcap B\,3_5\,. \tag{A6}$$

By a similar argument one can show that

$$B\,4 \supset B\,4_1 \bigcap B\,4_5 \tag{A7}$$

$$T \supset B\,3_1 \bigcap T_5 \tag{A8}$$

$$T \supset B\,4_5 \bigcap T_1\,. \tag{A9}$$

Also, because in S' the contact normals $\hat{a}_2$ must have a horizontal component to the right, we note that it is impossible for any contact point to be in both sets $B\,4_1$ and $B\,3_5$

$$B\,4_1 \bigcap B\,3_5 = \emptyset \tag{A10}$$

where $\emptyset$ represents the null set. For a contact point to be elements of both sets would require that the contact normal pass upward through $(\mathbf{q}_{1g}, \infty_1^-)$ or downward through $(\mathbf{q}_{13}, \infty_1^+)$ and upward through $(\mathbf{q}_{5g}, \infty_5^+)$ or downward through $(\mathbf{q}_{54}, \infty_5^-)$ which is impossible. Similar arguments result in the following three equations

$$B\,3_5 \bigcap T_1 = \emptyset \tag{A11}$$

$$B\,4_1 \bigcap T_5 = \emptyset \tag{A12}$$

$$T_1 \bigcap T_5 = \emptyset \,. \tag{A13}$$

The only set not accounted for is $B\,3_1 \bigcap B\,4_5$. All contact normals belonging to this set must pass through the translation window $(q_{1g}\,,\, q_{5g}\,)$. Consider the cone C_{2g} shown in Figure A2. If $\hat{a}_2$ passes through $(q_{5g}\,,\, q_{15})$, then C_{15} and C_{2g} see each other, *i.e.* the grasp is in the translation region. If $\hat{a}_2$ passes through $(q_{14},\, q_{15}]$[6], then the only pairs of mutually visible cones are $C_{2g}\,,\, C_{14}$ and $-C_{2g}\,,\, -C_{53}$. The corresponding instantaneous centers of rotation are the points q_{14} and q_{53}, respectively. The first pair of cones implies that the third contact must break which requires counter-clockwise rotation about q_{14}. This rotation causes interference at the fifth contact. The second pair of cones implies that the fourth contact must break which requires clockwise rotation about q_{53}. This rotation causes interference at the first contact. Therefore we conclude that motion is impossible. Thus points in S' whose normals pass through the translation window TW and through $(q_{14},\, q_{15}]$ belong to J_T which is a subset of J. For the case of q_{15} on the left of the translation region, the procedure for defining T and J_T are identical except the segments $(q_{5g}\,,\, q_{15})$ and $(q_{14},\, q_{15}]$ must be replaced by $(q_{15},\, q_{1g}\,)$ and $[q_{15},\, q_{53})$, respectively. The set $(B\,3_1 \bigcap B\,4_5)$ can now be written as the union of two mutually exclusive sets

$$B\,3_1 \bigcap B\,4_5 = J_T \bigcup (J_T{}' \bigcap B\,3_5 \bigcap B\,4_1). \tag{A14}$$

where the second term on the right hand side represents the portion of the set $(B\,3_1 \bigcap B\,4_5)$ which is in the translation region. Also note that for a grasp in the translation region with finger 1 fixed, q_{15} is the instantaneous center of rotation and must lie to the left or right of the line of action of f_3 or f_4, respectively. If q_{15} lies between f_3 and f_4, rotation counter-clockwise or clockwise causes interference at the third and fourth contacts, respectively; a contradiction that both contacts three and four break simultaneously.

The nature of lift-off for all 16 sets of S' have been determined and are now be combined to yield the liftability regions for the three-point initial grasp

$$B\,3 = B\,3_1 \bigcap B\,3_5 \tag{40}$$

---

[6] Note that contact normals which pass through TW and $(q_{1g}\,,\, q_{14}]$ have already been assign to the jamming region by relationship (A5).

$$B4 = B4_1 \cap B4_5 \tag{41}$$

$$J = J_1 \cup J_5 \cup J_T \tag{42}$$

$$T = (B3_1 \cap B4_5 \cap J_T{}') \cup (B3_1 \cap T_5) \cup (B4_5 \cap T_1) . \tag{43}$$

q.e.d.

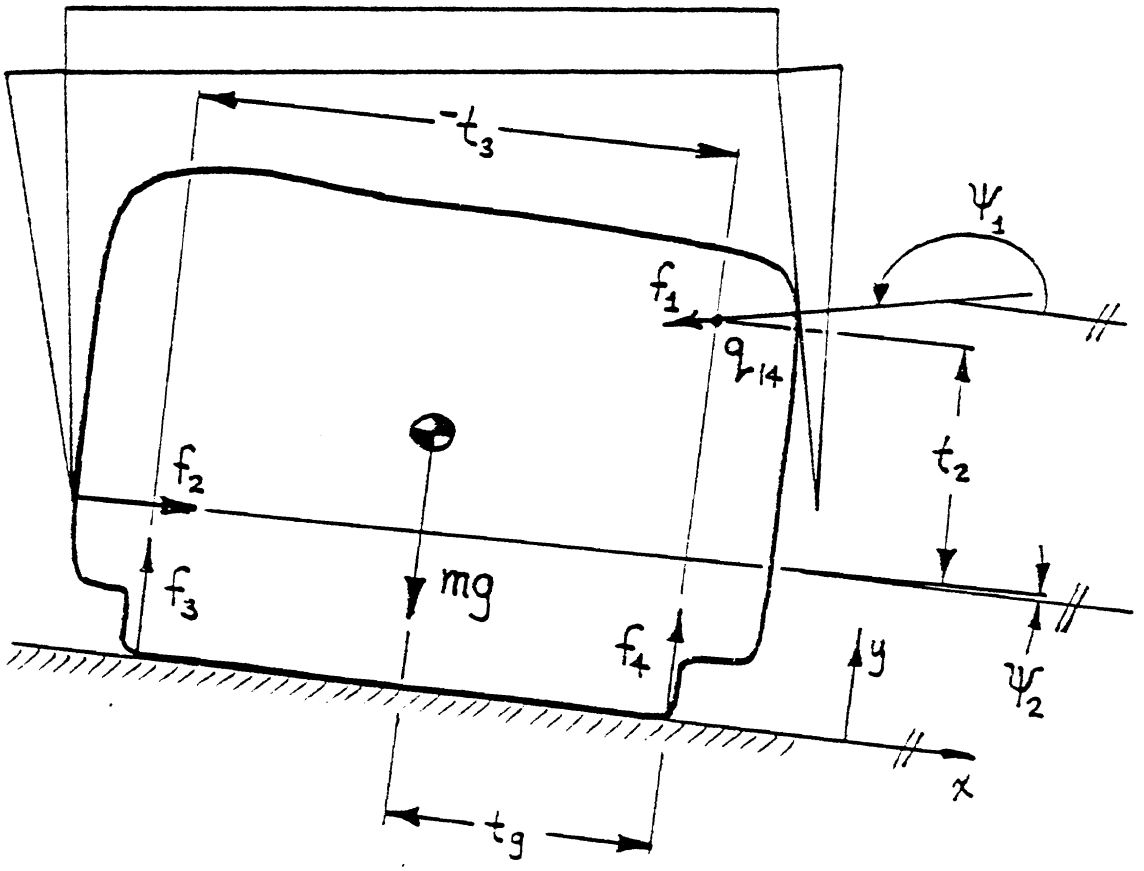


Figure 1: Two-Point Initial Grasp

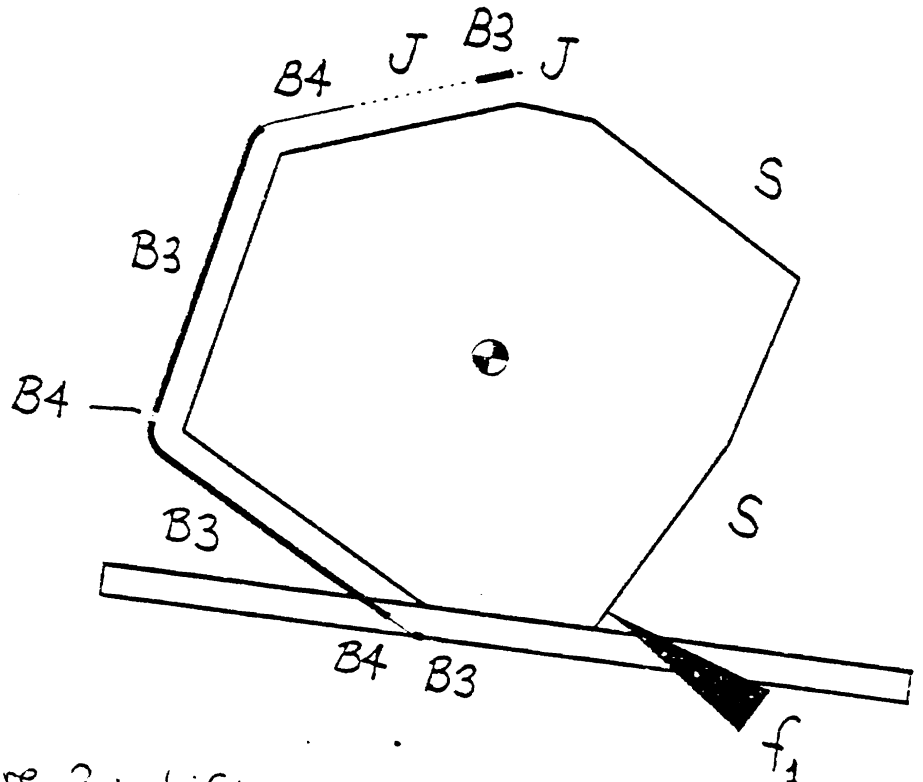


Figure 2: Liftability Regions

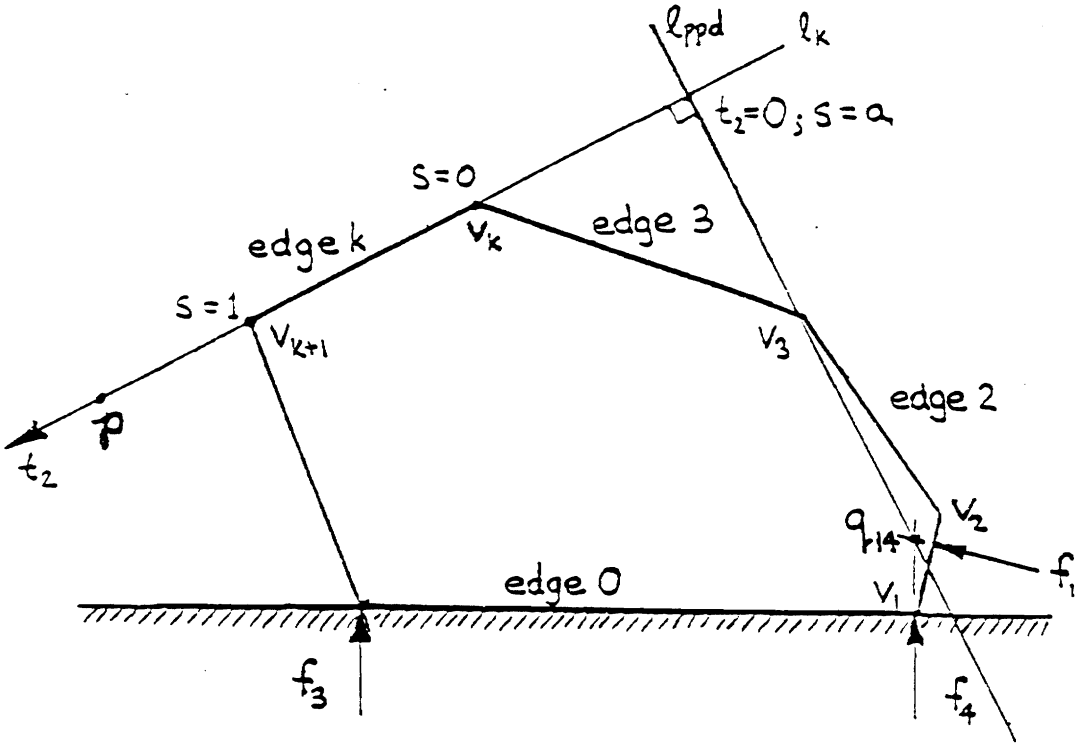
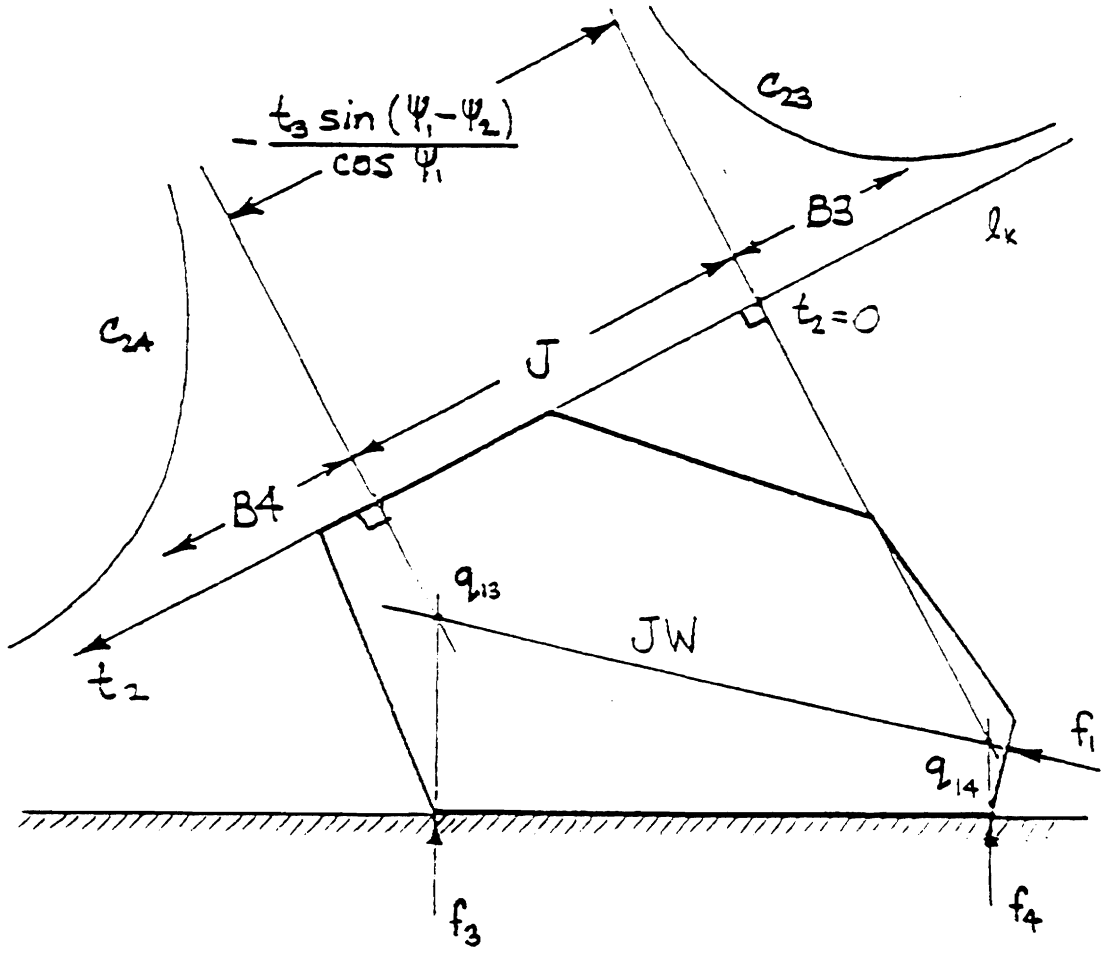


Figure 3: Quantities For Edge Liftability Regions



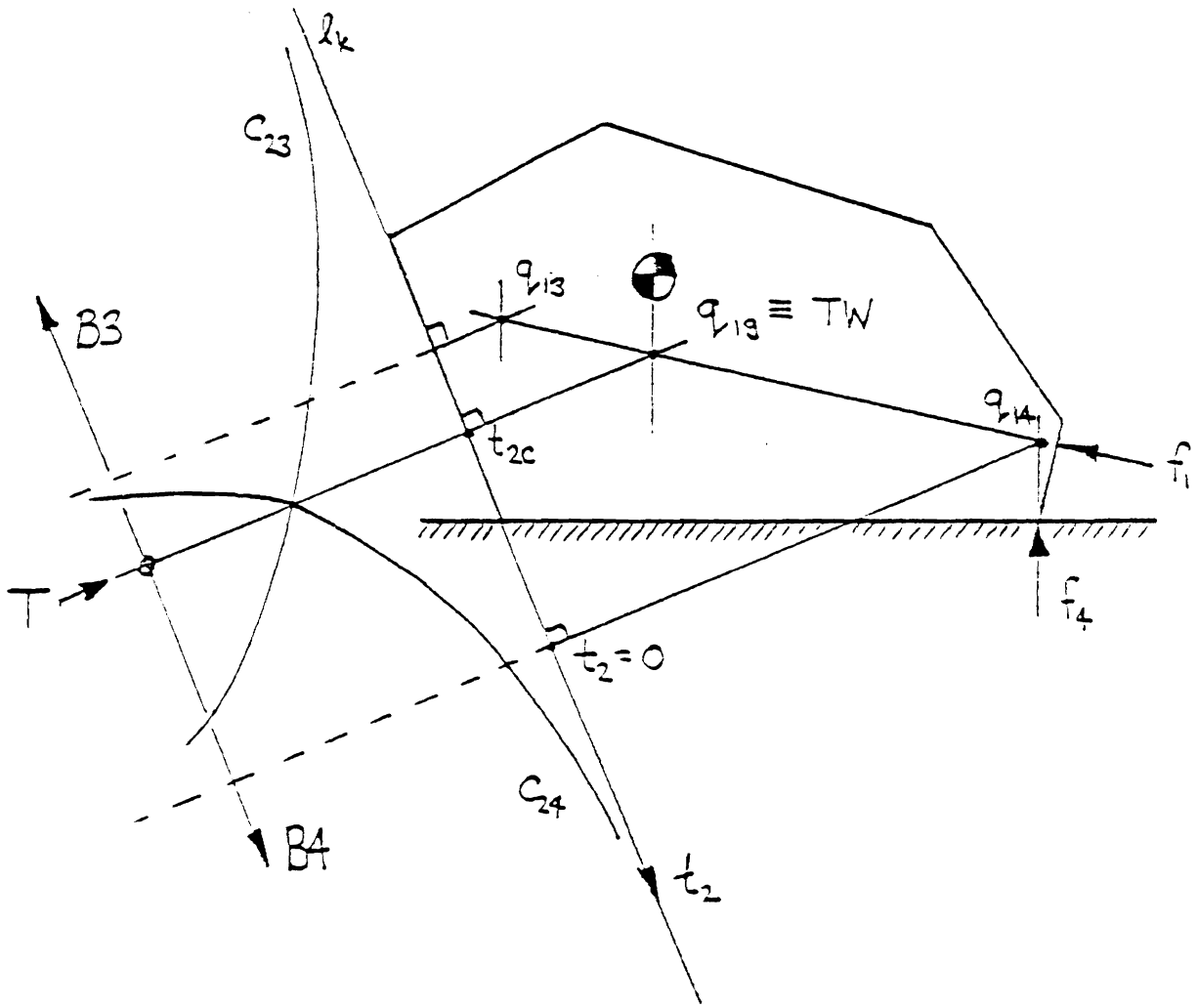


Figure 5: Formation of Edge Regions B3, B4, and T

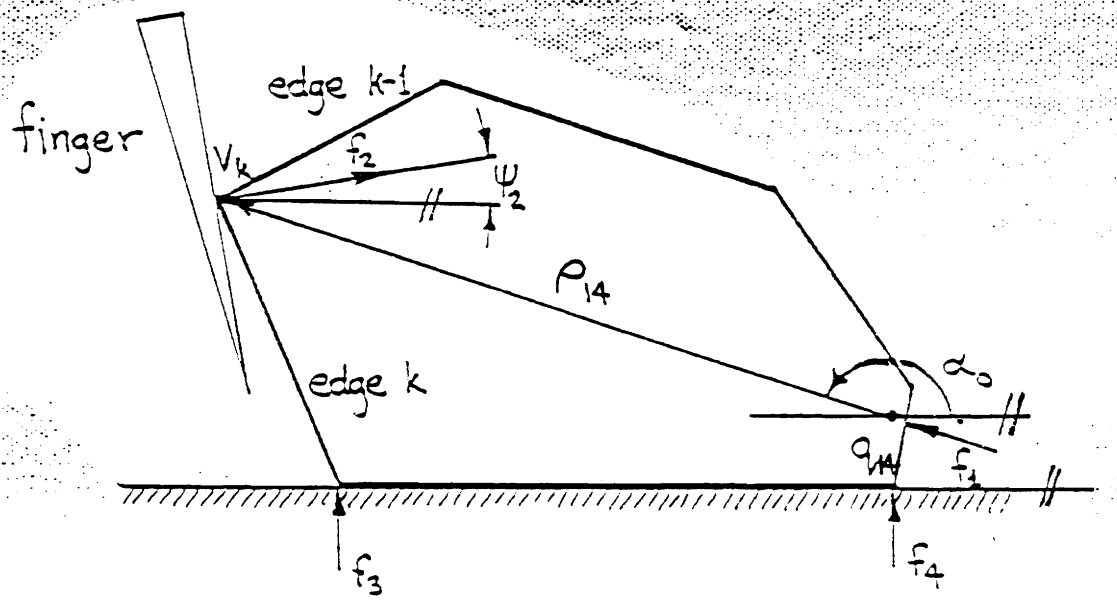


Figure 6: Quantities For Vertex Liftability Regions

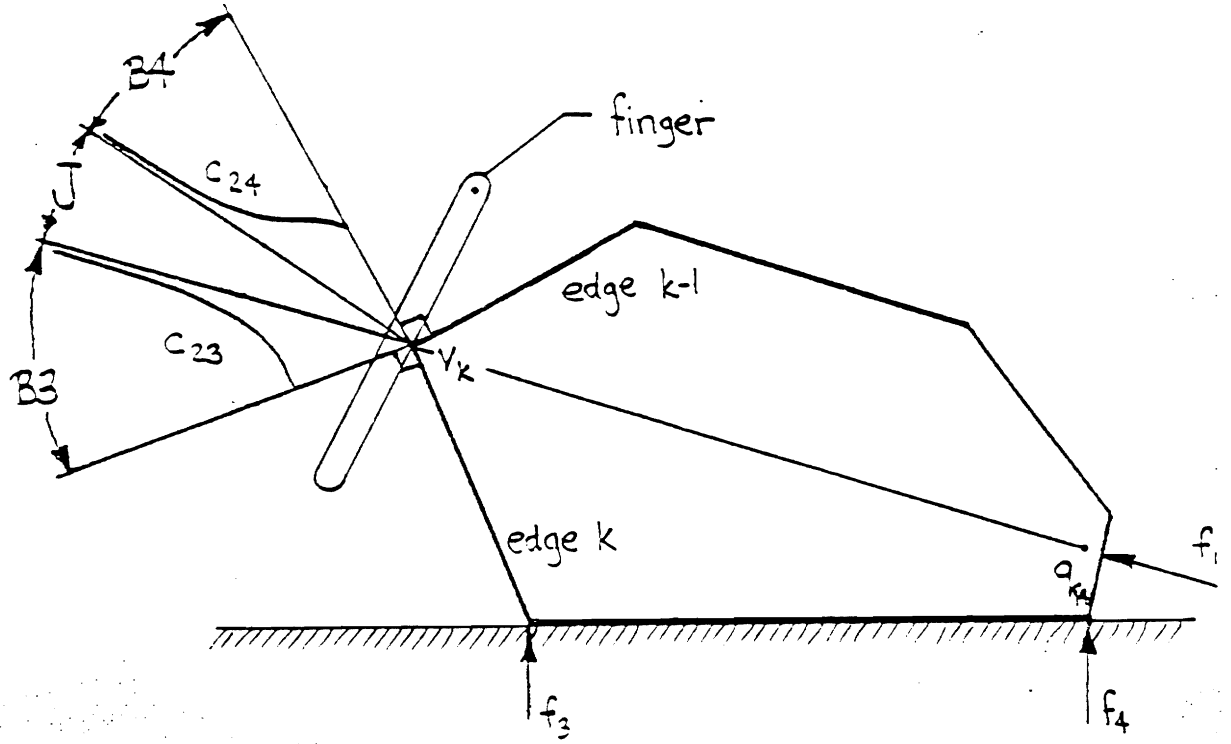


Figure 7: Vertex Liftability Regions

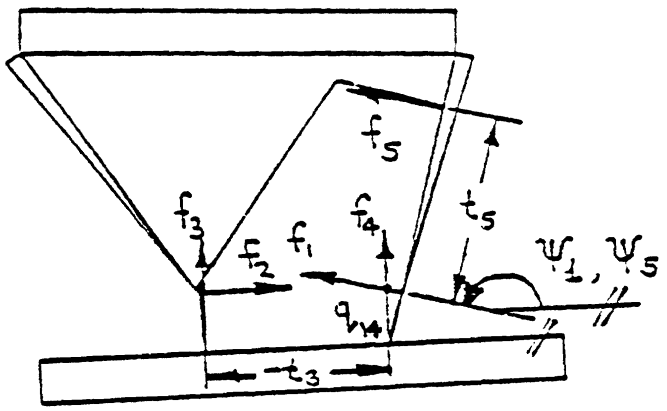


Figure 8: Three-Point Initial Grasp

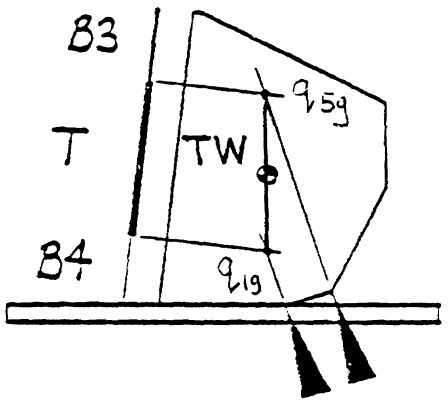


Figure 9: Region, T, for an Edge

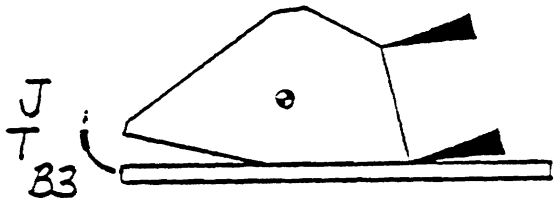


Figure 10: Region, T, of a Vertex

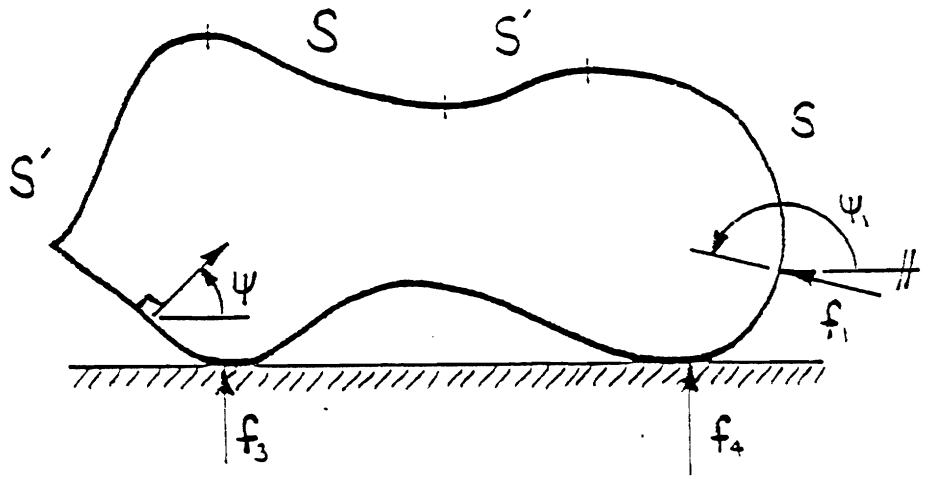


Figure 11: Regions S and S'

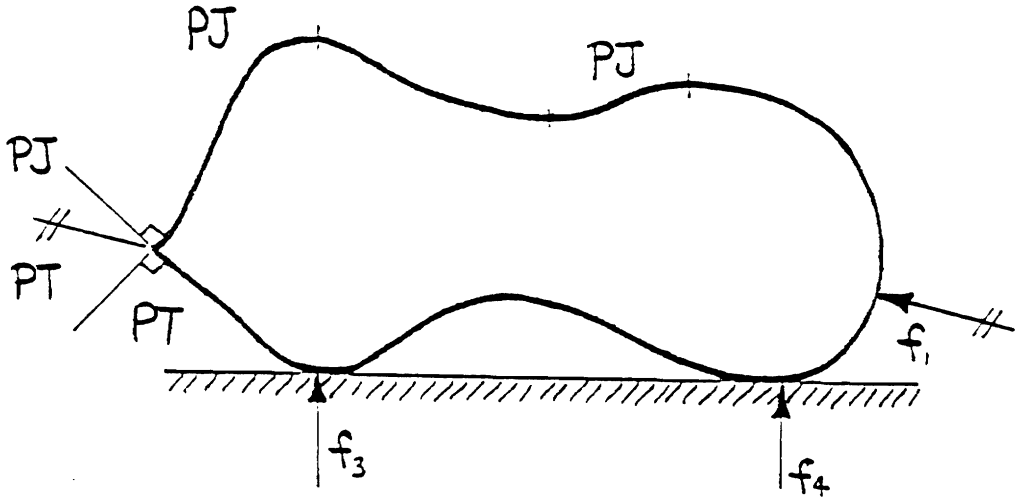


Figure 12: Regions PT and PJ

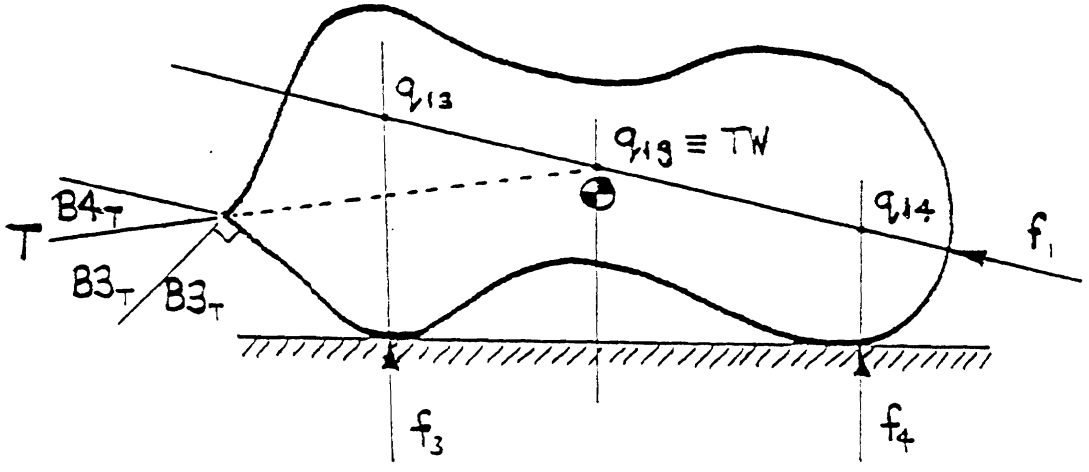


Figure 13: Regions $B3_T$, $B4_T$ and T

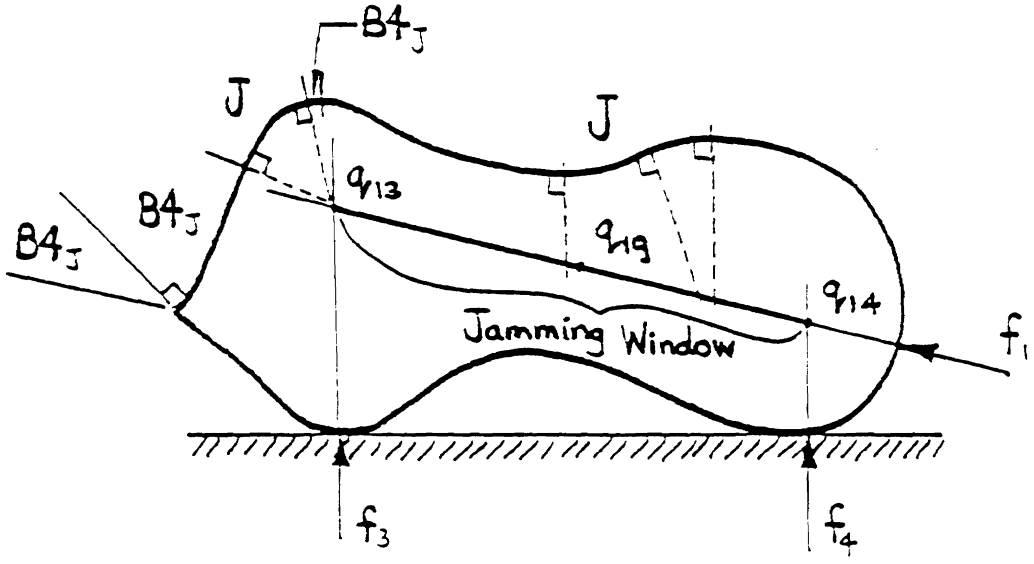


Figure 14: Regions $B3_J$, $B4_J$, and J

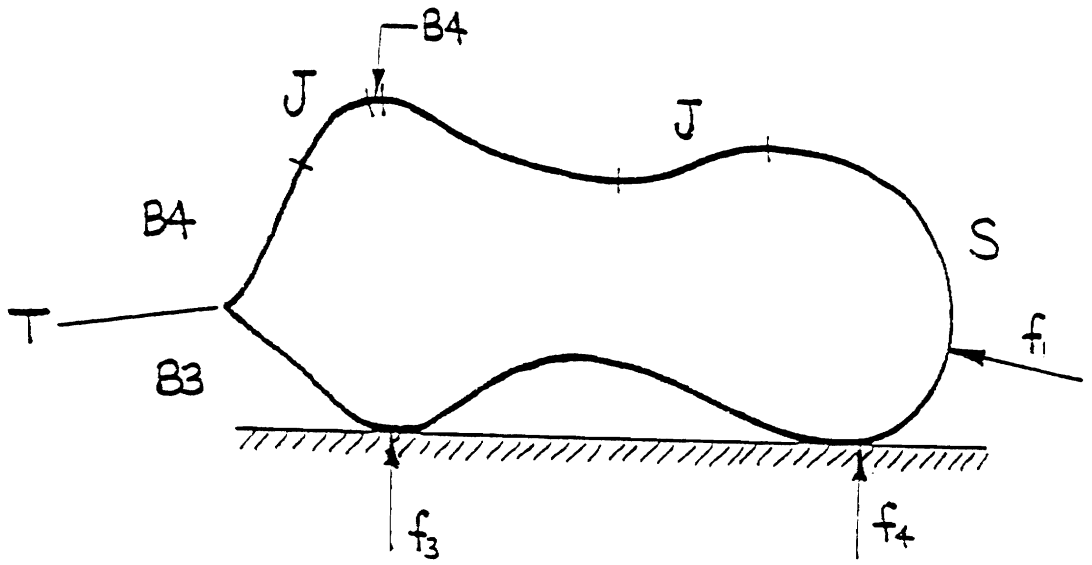


Figure 15: Liftability Regions for a Two-Point Initial Grasp

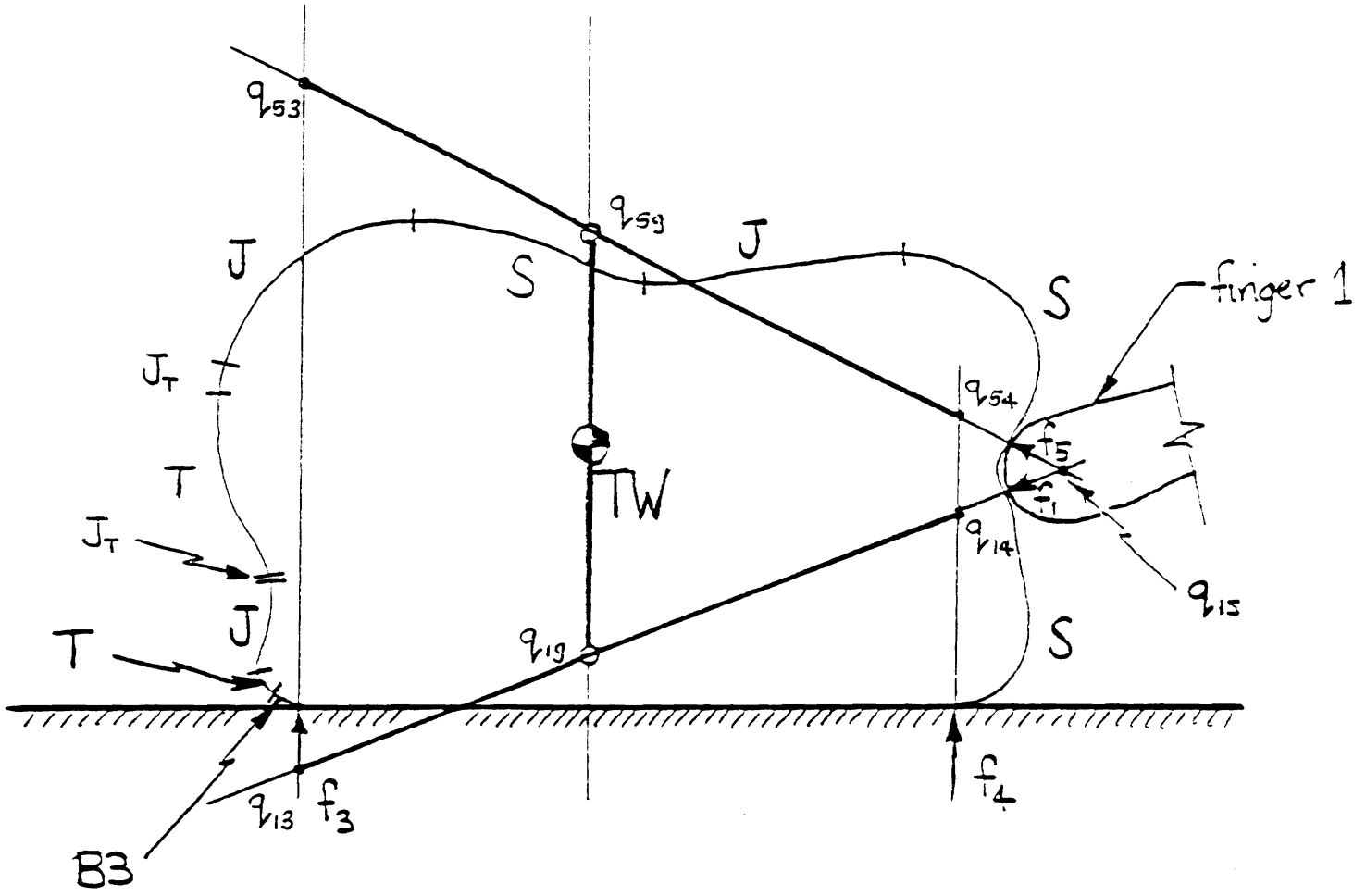


Figure 16: Liftability Regions for a Three-Point Initial Grasp.

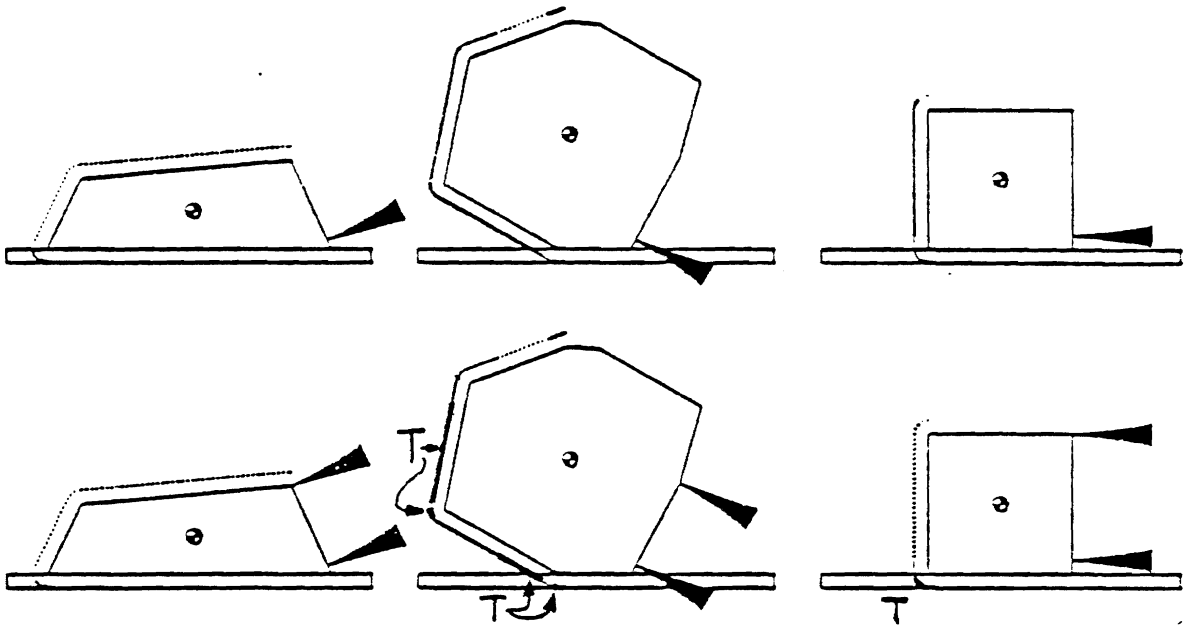


Figure 17: Liftability Regions for Two-Point and Three-Point Initial Grasps of Several Polygons. Note that the object on the left does not develop a translation region.

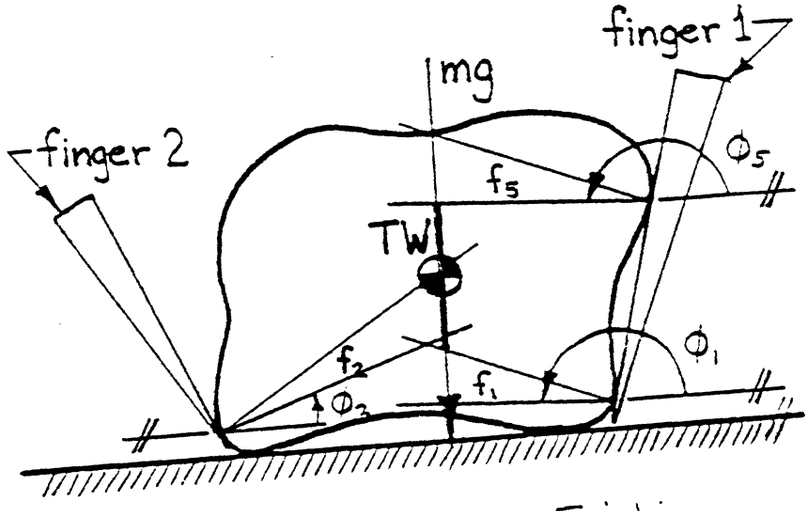


Figure 18: Grasp in T with Friction

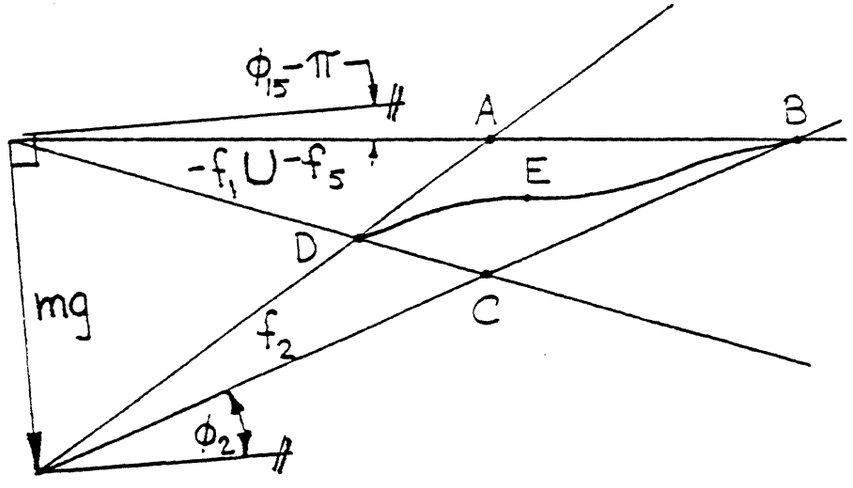


Figure 19: Grasp Force Diagram

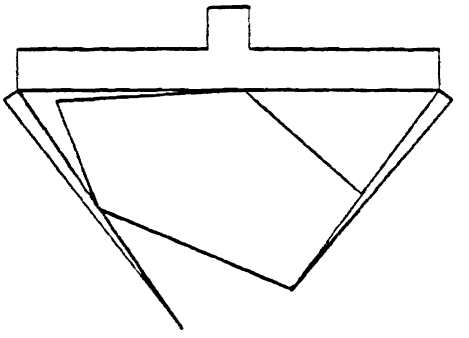


Figure 20: a form closure grasp of the object. If the fingers are locked the object cannot move at all.

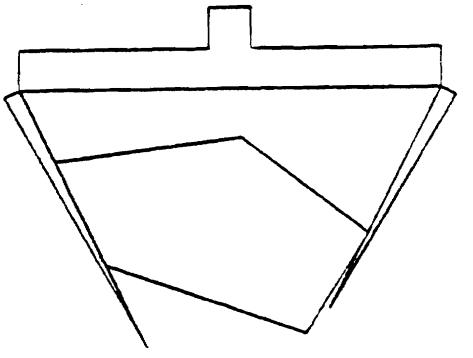


Figure 21 :a typical force closure grasp.

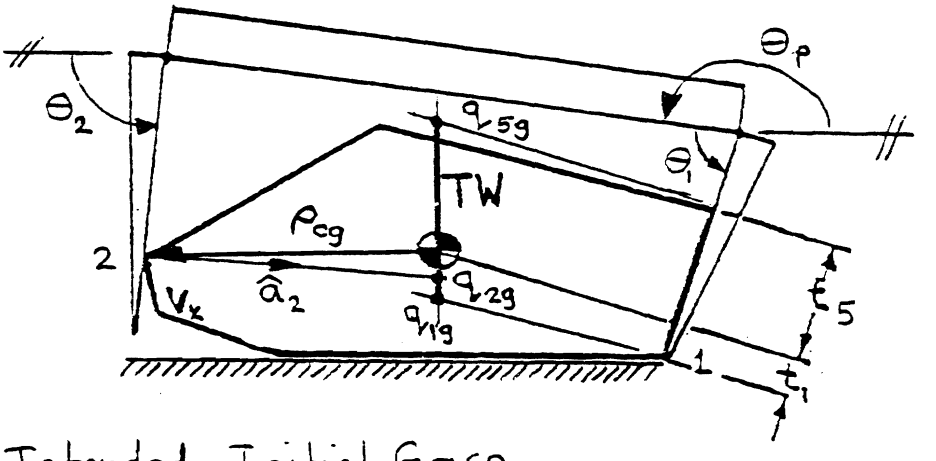


Figure 22: Intended Initial Grasp

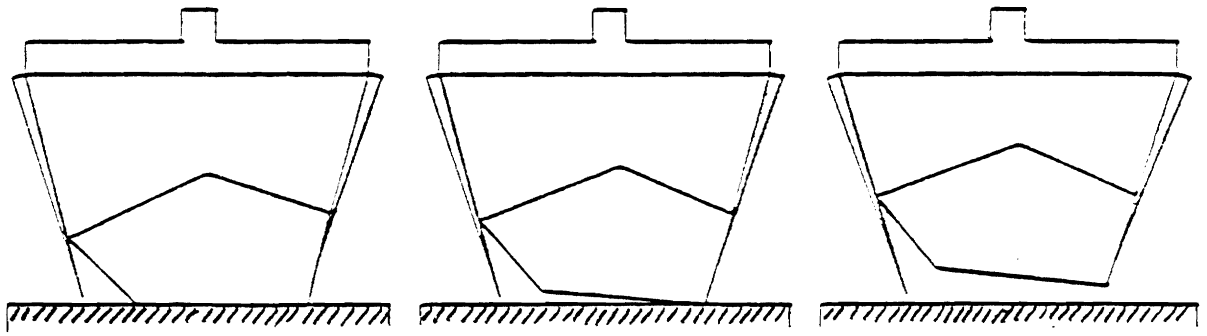


Figure 23: Grasp with Self-Correcting Type 3 Error

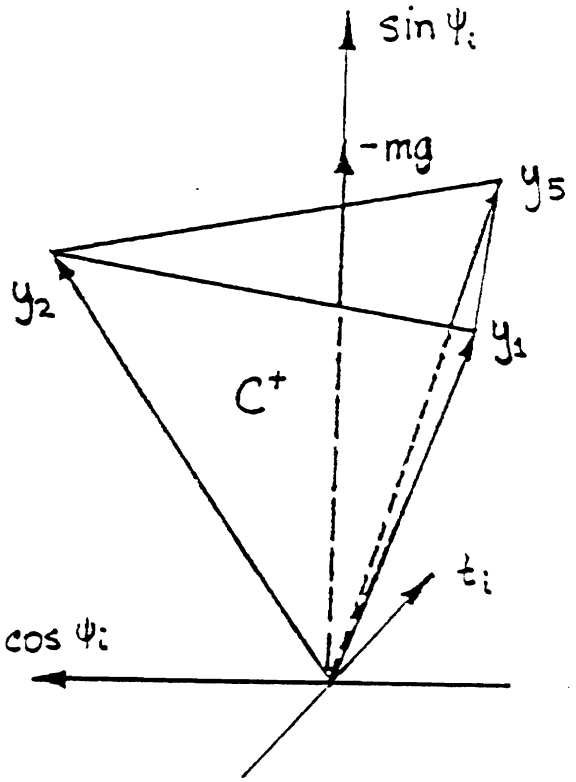


Figure 24: Convex Cone and External Force

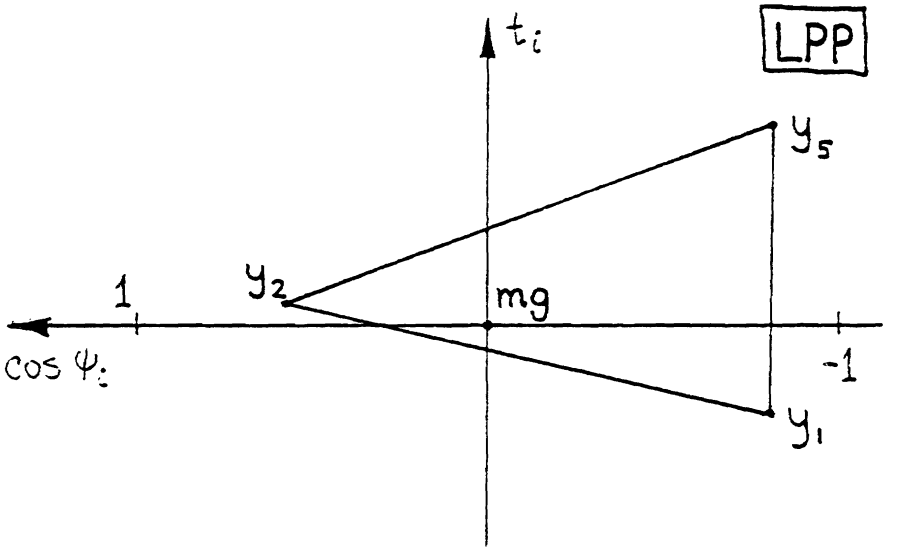


Figure 25: Convex Cone C^+ Mapped onto the Lifting Phase Plane.

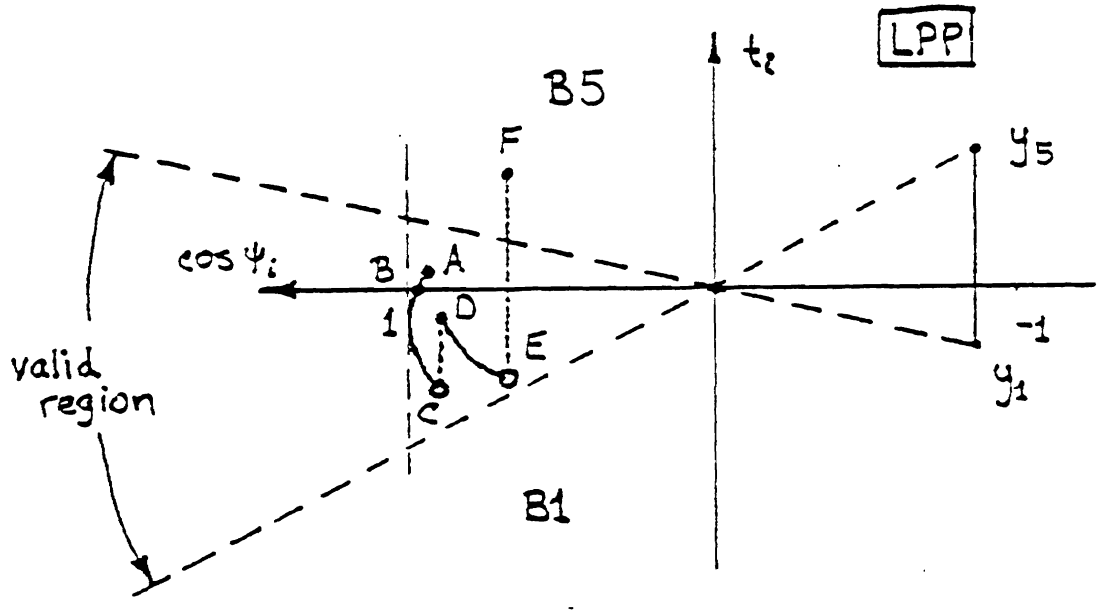


Figure 26 : Trajectory of y_2.

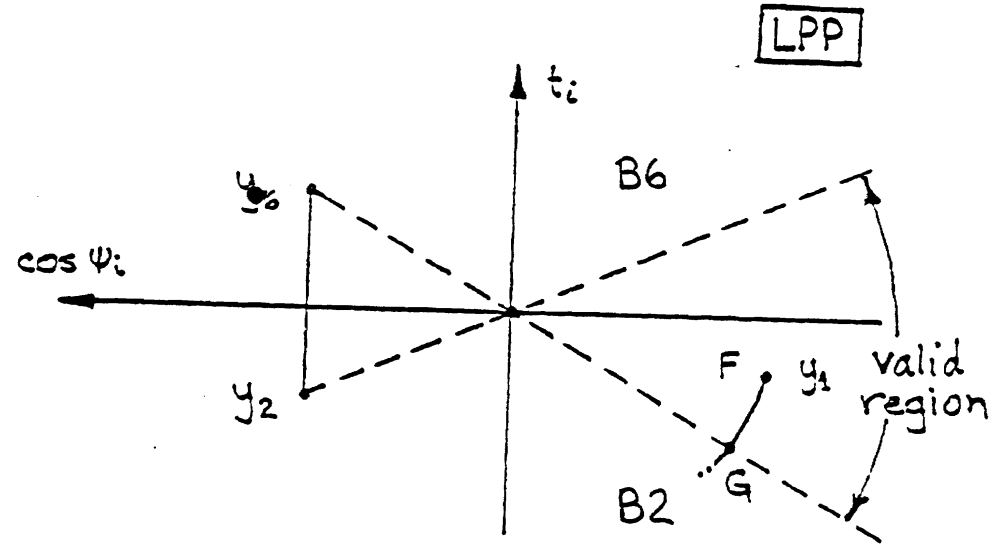


Figure 27 : Trajectory after Switching Pushers

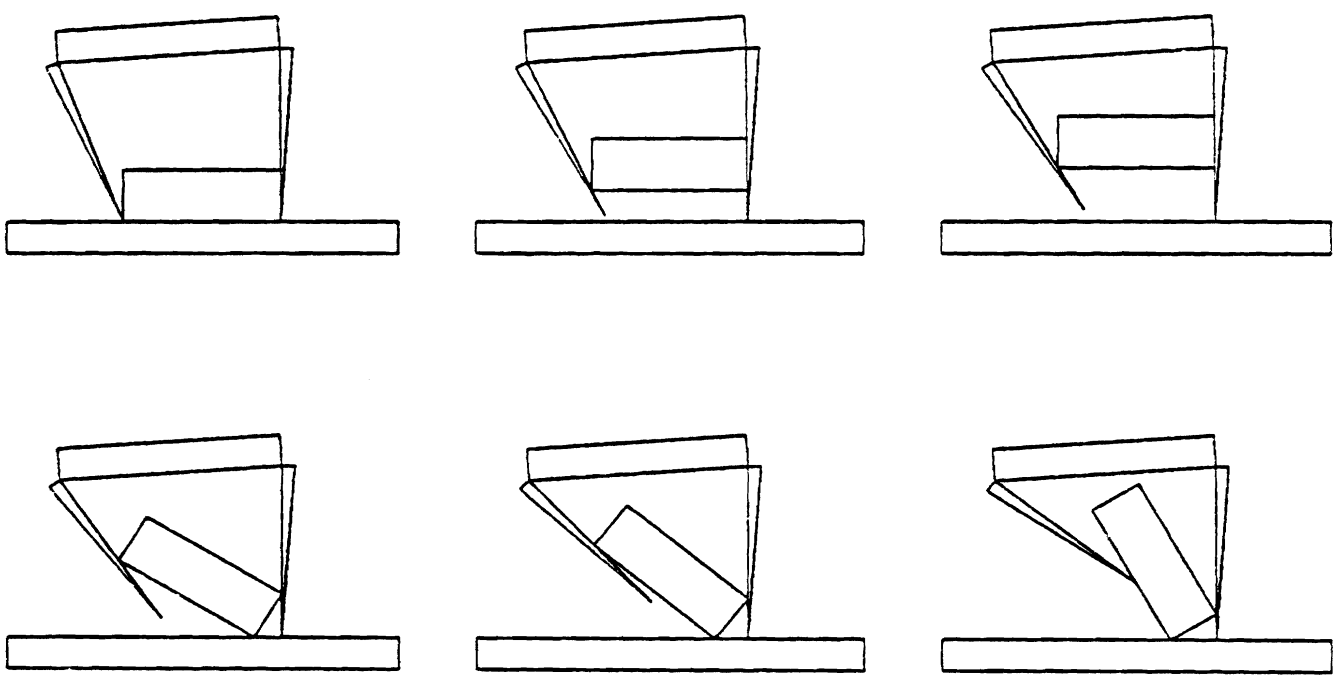


Figure 28: Failed Pushing Strategy

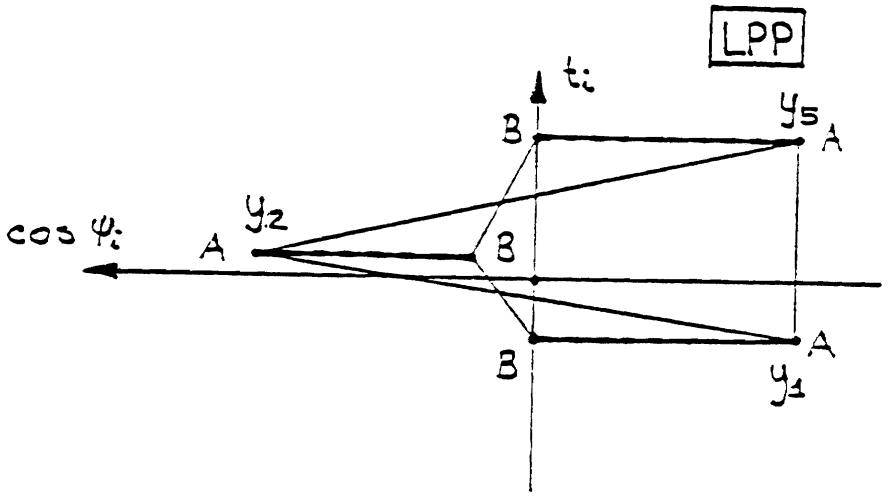


Figure 29: Hand Rolling Strategy

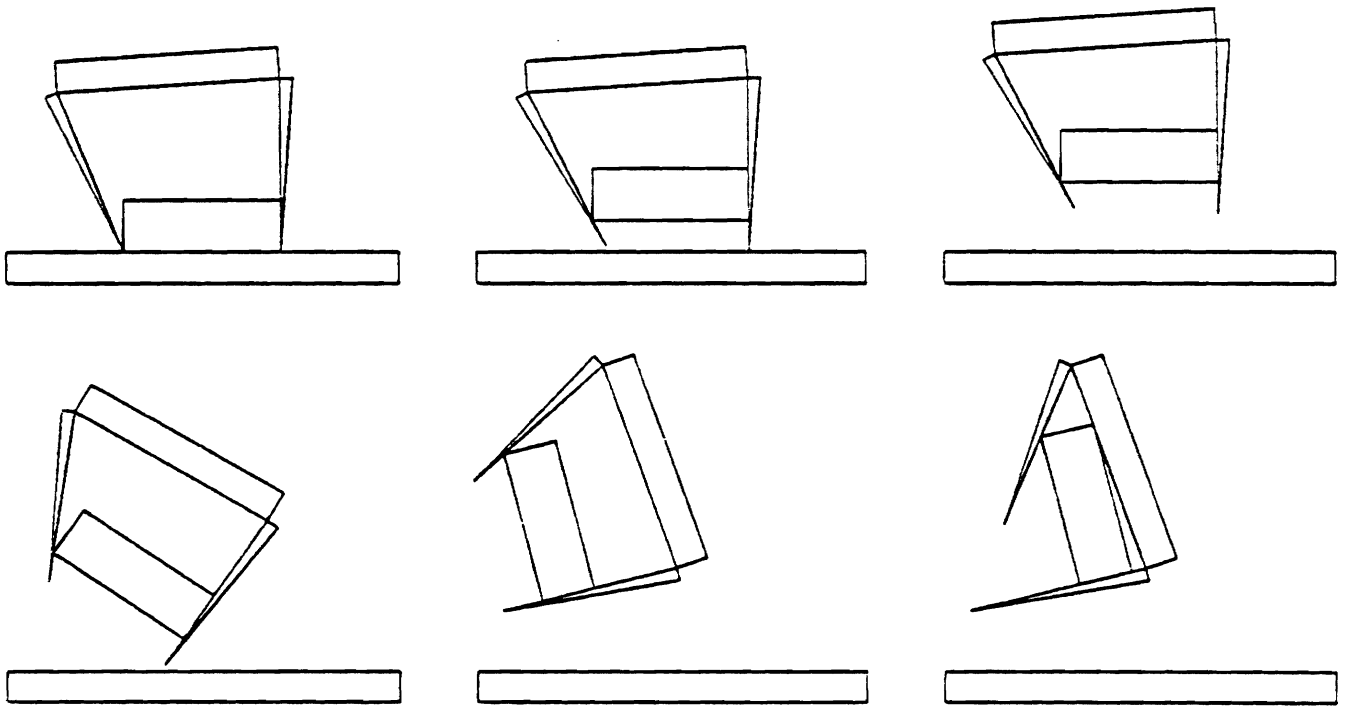


Figure 30: Successful Push and Roll Strategy

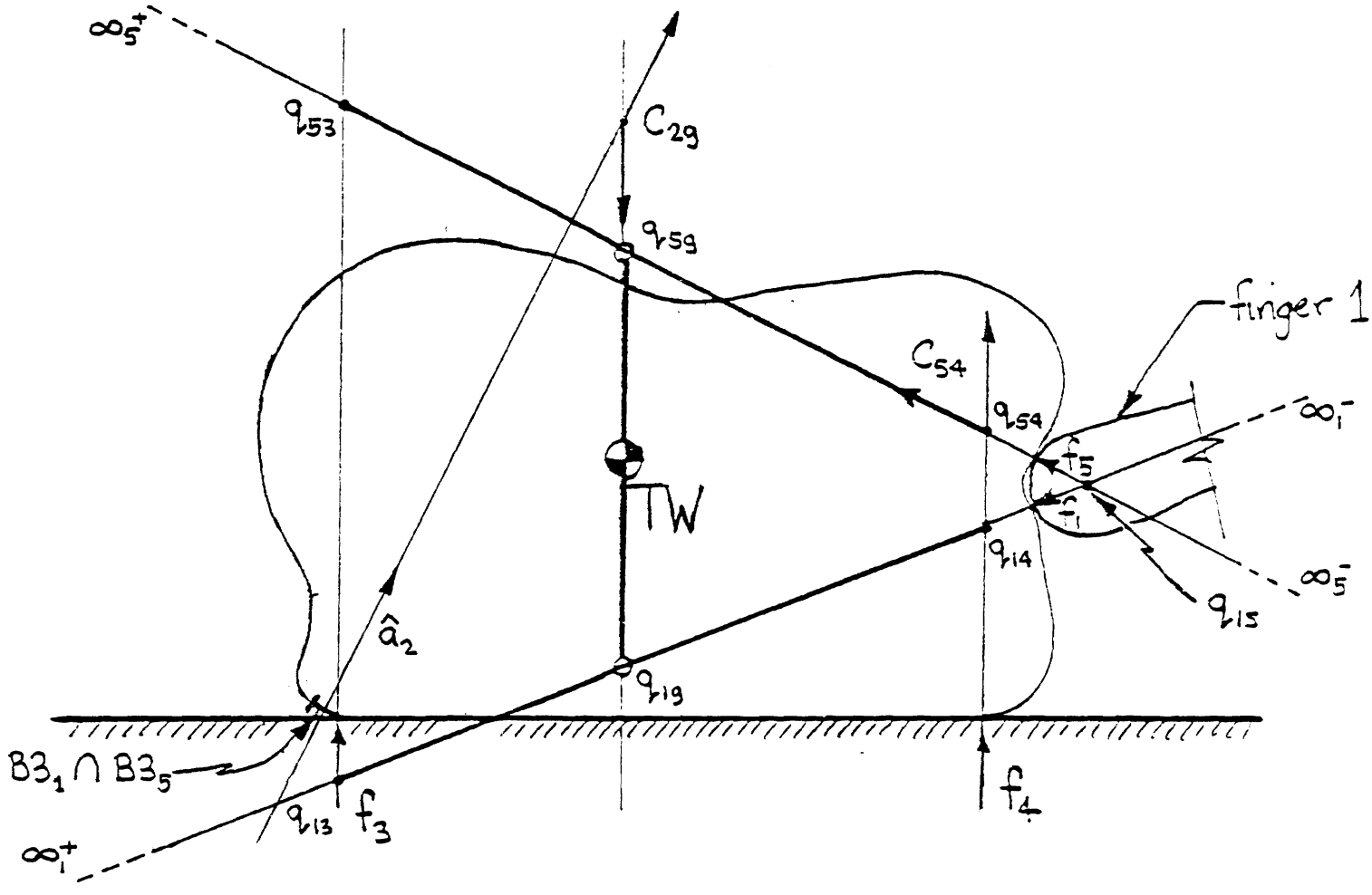


Figure A1: Formation of the Region $B3_1 \cap B3_5$.

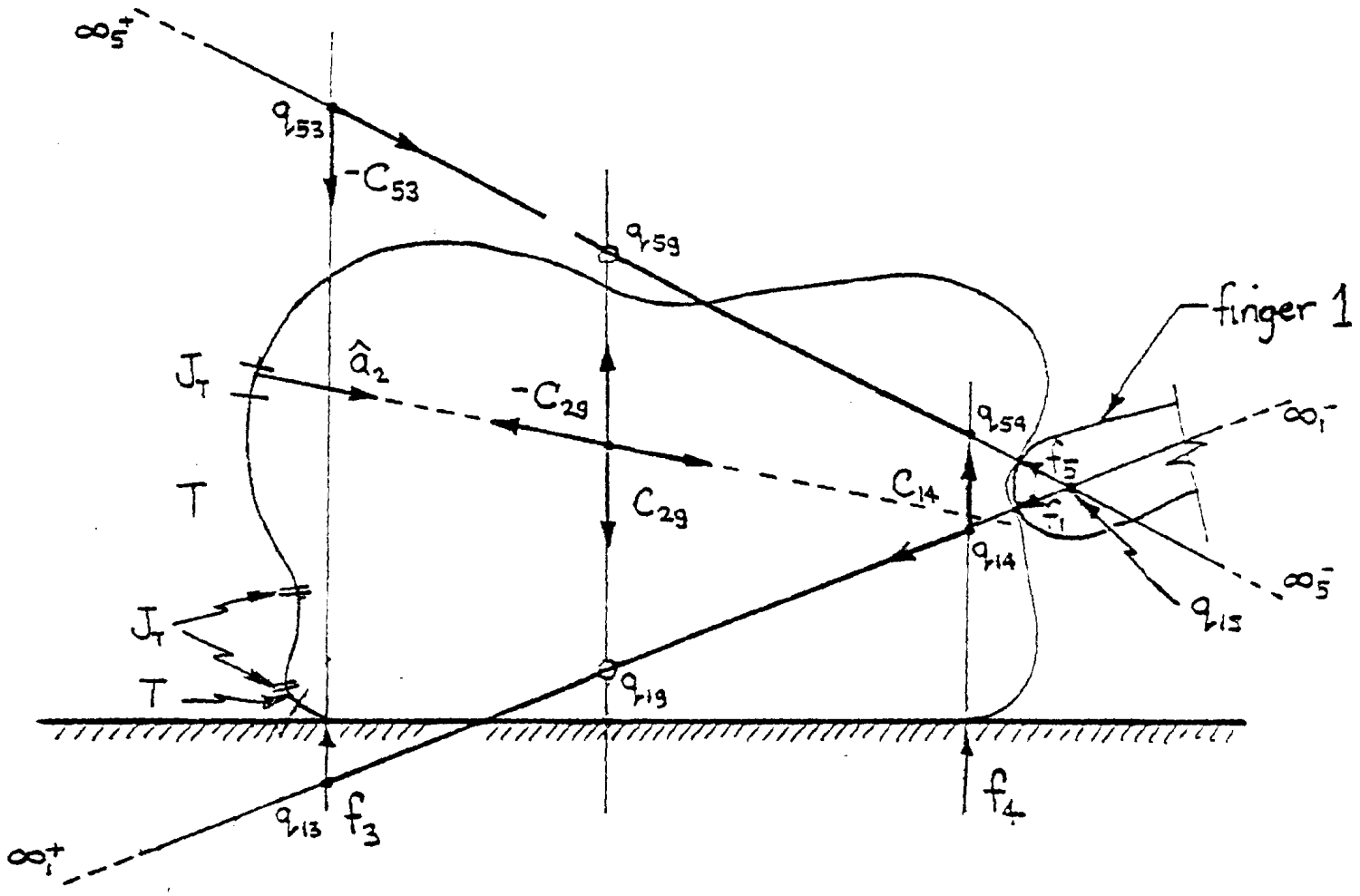


Figure A2: Partitioning of the Region $B3_1 \cap B4_5$ into Regions J_T and T.

A.3 A Medium-Complexity Compliant End Effector

A Medium-Complexity Compliant End Effector*

Nathan Ulrich Richard P. Paul Ruzena Bajcsy

Department of Mechanical Engineering
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

Abstract

Recent interest in end effector design has not yet resulted in a versatile yet simple mechanism appropriate for a wide range of manipulation tasks. The design of a novel end effector under development at the University of Pennsylvania is explained in detail in this paper. The rationale supporting this mechanism is explored, its geometry is described, experimental results from the first prototype are shown, and some ideas for future work are presented.

Introduction

In recent years there has been a great deal of attention focused on the design of end effectors. Progress in grasping research, active sensing, assembly, and

*Supported by NSF grants MEA-8119884, DCR-8410771, CER/DCR-8219196, INT-8514199, DMC-8517315, and DARPA/ONR grant N0014-85-K-0807. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the supporting agencies.

prototype construction has created a need for a versatile, robust, and economical mechanical hand that can be used for experimentation. Although many designs have been proposed and several prototypes built, a comprehensive effort which combines the desire for performance with the reality of application has yet to be undertaken. As a result, no single device is in common use.

Most previous end effector designs fall into two categories: complex "hands" or simple grippers. Notable in the first class are the Utah/MIT Dextrous Hand [1] and the Salisbury hand [2]. They incorporate a large number of degrees of freedom (degrees of freedom) into a complex multi-fingered hand design which imitates the human hand in speed, dexterity, and versatility. The resulting performance is impressive, but the increased complexity precludes simple planning procedures. The simple grippers do not have this problem—they are generally one or two degrees of freedom and are powered by means of remote pneumatic or self-contained electric actuators. They pay for this simplicity by being limited in application, usually specialized for one type of task.

We feel that what is needed is a *medium-complexity* end effector: a device that combines the simplicity characteristic of the simple grippers with some of the versatility of the complex hands.

Design Philosophy

The design of any tool requires a precise definition of its intended use. It is important to not only decide what tasks a robotic end effector needs to be able to perform, but to also determine the limits of its performance. Previous hand designs have used the human hand as a so-called "existence proof" of the appropriateness of such a geometry. Since our hands are capable of many varied

tasks, any mechanical end effector which duplicated the human hand would also be capable of these tasks. But this is not sufficient reason for an anthropomorphic geometry. The design of an end effector should be pursued in the same way as any other design; establish the criteria for its performance and synthesize a mechanism which satisfies these goals. For our specific research environment, the end effector is required to machine and assemble parts, handle many different sizes and shapes of objects, and perform exploratory and sensing tasks—it does not need to be able to perform tasks outside of this environment. While the human hand seems to be ideal for performing the wide range of tasks required of a person—from playing basketball to changing diapers to driving nails—it is not necessarily the perfect tool for the specific areas in which robotic research is now concentrated. Witness the number of tools to assist the human hand found in a machine shop. It should be possible to design an end effector that is more suited than the human hand for such an environment.

Design Criteria

The Medium-complexity Compliant End Effector (McCEE) is designed primarily for three research areas: active sensing, assembly (and disassembly), and grasping.[1] Although these subjects encompass a wide range of criteria, we feel that they overlap sufficiently for the use of one basic end effector design.

Grasping research requires a versatile mechanism that allows application of theoretical methods to experimental situations. The state of the art at this point demands a more flexible tool than the simple grippers commonly used,

[1]Research in the application of this design to prosthetics is continuing, but is beyond the scope of this paper.

but it is extremely important that the complexity of the end effector be limited. Since theoretical principles cannot support a complex (e.g. 9 or more degrees of freedom) model of grasping in three dimensions, we feel that a medium-complexity device is most appropriate at this time. The simplicity of planning, movement, and control associated with fewer degrees of freedom is an important consideration—such a tool would be more accessible to the researcher. However, it is important to note that 9 degrees of freedom is the minimum necessary to allow arbitrary positioning of three fingertips in space. For this reason, our design will concentrate on *enveloping* grasps; those that rely on the palmar surfaces of the inside of the fingers and the palm to constrain an object, as opposed to fingertip manipulation utilizing friction and fingertip contacts[3]. An extension of the two degree of freedom grippers is necessary, but in interest of utility, we would like to limit our end effector design to three or four degrees of freedom.

Although recent advances in vision and other passive sensing techniques have resulted in increased reliability and information gathering ability, it has been shown that the use of active sensing is necessary to adequately define the shape and orientation of an object[4][5][6]. In addition, psychological research has defined a number of "exploratory procedures" that can be used to collect such characteristics of an object such as texture, hardness, thermal conductivity, and shape[7]. Such sensing will allow us to classify an object or verify a hypothesis; an exact description is essential to allow us to perform manipulation in an assembly operation or to support grasping experimentation. Therefore, the end effector will need to serve as a platform for a number of specialized sensors necessary for this work. It is necessary that a sensor package be incorporated in

the design of the end effector, but that the end effector be sufficiently versatile to accomodate changes in sensor type and application. The primary sensors—those integral to the design—provide position, tactile, force, and moment information on contact surfaces. But the design must also consider easy mounting and dismounting of other more exotic sensors (thermal and electrical conductivity, proximity, specialized textural, etc.).

Assembly of parts and objects is an important area of robotics research because of its relevance to industrial applications. However, assembly tasks performed by robots today are limited to rigid, structured operations which usually require complex jigs and parts-feeding devices. Any appreciable uncertainty in such an operation cannot be accomodated. This is essentially automation and *not* robotics. At a certain level of production capacity, such automation becomes cost effective. However, below this critical level, human workers are necessary to supplement any generic automatic devices in use. A true robotic assembly operation would combine grasping and sensing with computational sophistication, and would be able to tolerate much larger errors in positioning and description. Necessary to such an operation, however, are one or more versatile end effectors that are suited for both a wide range of grasps and a variety of sensors. Such a device should be able to handle both parts and tools, as well as possessing the sensor sophistication to recognize and differentiate objects. But even with these capabilities, an assembly operation still requires a model and procedure to follow. Previous research has used human-based techniques to synthesize assembly algorithms. However, the strengths and weaknesses of a robotic system are inherently very different from those found in humans. By taking an object apart, finding seams, joints, and fasteners, such a system could determine the

best way for a robot to reassemble the object. The ability to perform effectively in such a disassembly operation is an important criterion for our end effector design.

A number of criteria for the design of an end effector that could perform the operations suggested above are related to convenience and utility. The mechanism would ideally be self-contained; discrete from the manipulator and able to be mounted and dismounted quickly and easily to facilitate adjustment and repair. A compact, sleek design integrating all cabling, sensing, and actuation is important, but since it will be a research tool, the mechanical design should be accessible, allowing changes in structure and operation without radical reconstruction or redesign. The use of the end effector to learn about objects necessitates it use as a platform for many types of sensors. All of these sensors do not initially need to be built-in, but the design must be able to accomodate their use. The end effector should, ideally, satisfy the research imperatives described previously while attaining these objectives as well.

Supporting Research

Many researchers have attempted to classify the grasps required by a robotic end effector. Schlesinger defined six prehension types used by humans in his work[8], and Cutkosky and Wright further defined the grasps used by a machinist at work[9]. Although other, different, classifications have been used (see [10] for a complete grasp taxonomy), we find these two sets of descriptive labels most appropriate for our applications. The grasps required by assembly, disassembly, prototype construction, and grasping research are contained within these types, represented graphically in Figures 1 and 2.

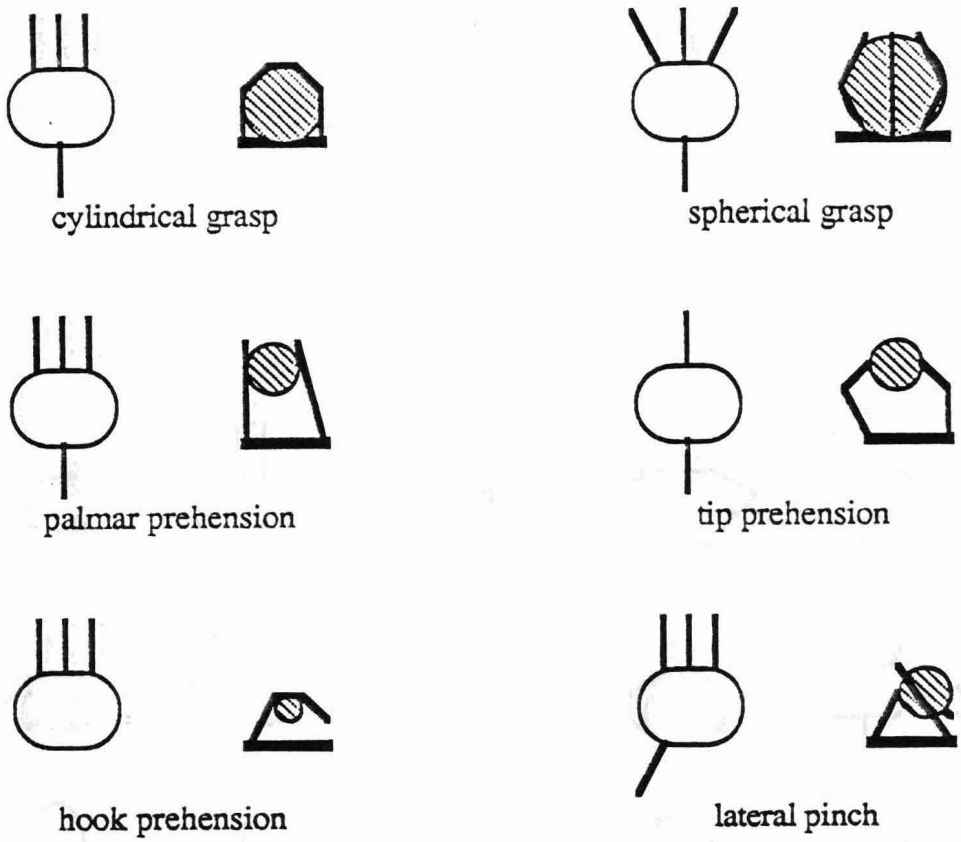


Figure 1: Schlesinger's prehension types

While the actual apprehension of an object with a robotic end effector can be modeled using the above classifications, the use of the device as a tool for active sensing requires expansion of these models. Although a great deal of haptic (kinesthetic plus tactile) information can be gained by simply holding an object, the exploratory procedures described by Klatzky *et al* require other sensory methods. Figure 3, adapted from [7], shows the properties that we need to obtain by active sensing and the necessary actions of the end effector to determine these properties. In order to perform these movements with an end effector, we need several abilities. First, we need to be able to use the end effector with one finger extended as a probe. This will allow us to perform the exploratory procedures to test for texture, hardness, temperature, and will allow us to determine the shape of the object by means of the procedures suggested by Allen [5] and Stansfield [6]; i.e. determine surfaces, cavities, holes and

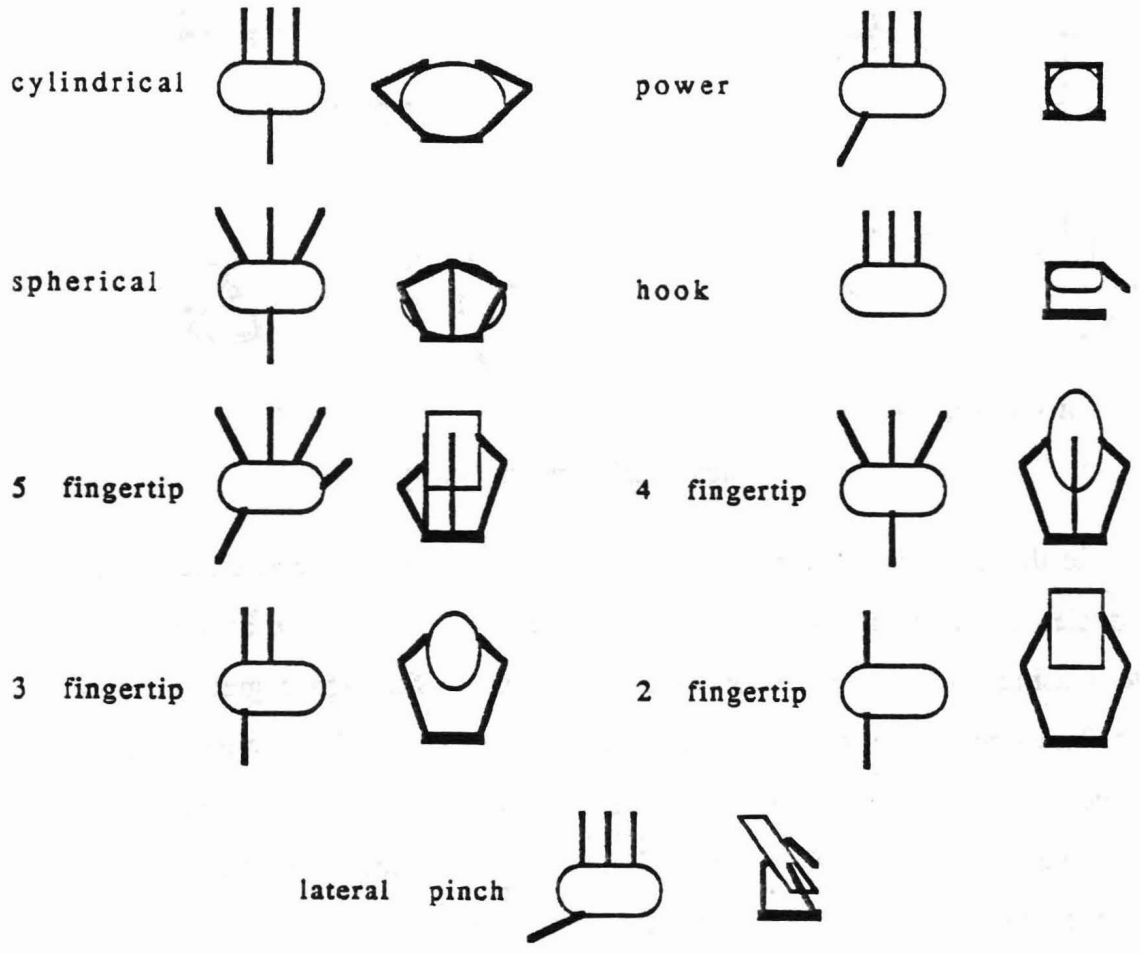


Figure 2: Cutkosky and Wright's manufacturing grips

	Properties	Hand Movements
Surface Properties	Texture	Lateral Motion
	Hardness	Pressure
	Temperature	Static Contact
	Weight	Unsupported Holding
Structural Properties	(Weight)	(Unsupported Holding)
	Global Shape	Enclosure, Contour Following
	Exact Shape	Contour Following
	Volume	Enclosure

Figure 3: Classification of properties and exploratory procedures

contours. In order to accomplish these tasks, this finger would need tactile sensing capability, force and position sensing, and also specialized temperature sensors.

The end effector must also be able to enclose an object within its grasp and lift it free of support. This will allow us to determine the weight, shape, and volume of the object. Such a function requires similar properties as those required by other aspects of our goals, but also requires precise sensing of the object within the grasp. A determination of an object's properties by means of the exploratory procedures described above is essential to an accurate classification of the object; such a classification is necessary for success in the assembly, disassembly, and prototype construction workplaces described previously. It follows, then, that in order for an end effector to be useful in these task-oriented environments, it must also be a efficient tool for active sensing.

Mechanical Configuration

The shape of the end effector design was determined by the need to achieve wide versatility with as few degrees of freedom as possible. We found that in order

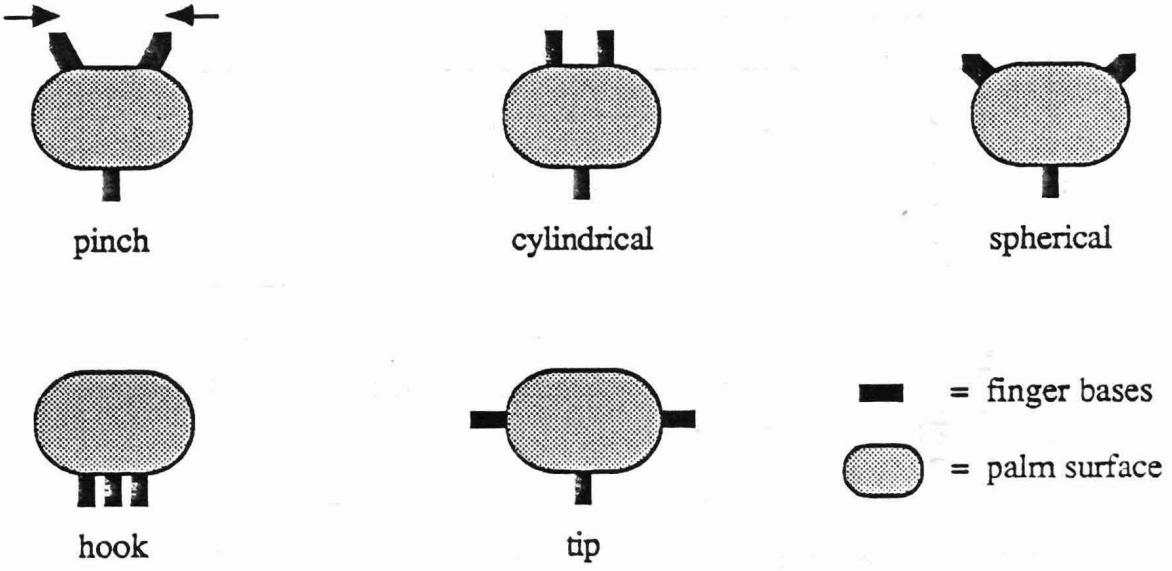


Figure 4: The five grasping modes of McCEE

to obtain the grasping and sensing configurations necessary for our research, we needed an end effector with at least four degrees of freedom. The actual mechanical geometry is separated into two parts: the shape of the palm and its relationship to the fingers, and the finger design.

The palm/finger relationship consists of a one degree of freedom movement of the fingers around the palm. Skinner proposed a similar movement of the fingers, but his design did not incorporate the palm into the grasping arrangement[11]. We wish the palm to be an important tool in the manipulation of objects. Not only can the palm be used as a base against which to hold objects, as a tool to perform pushing operations on objects, but also (with tactile sensors) as a information-gathering instrument which will allow "footprints" of objects to be obtained. By separating the centers of rotation of the fingers, we obtain a number of grasping configurations. Figure 4 shows these different modes. One finger (which, although not precise biologically, we call the thumb) has its base fixed with respect to the palm, while the other two move synchronously around two different axes. The resulting scheme allows a very wide range of grasping

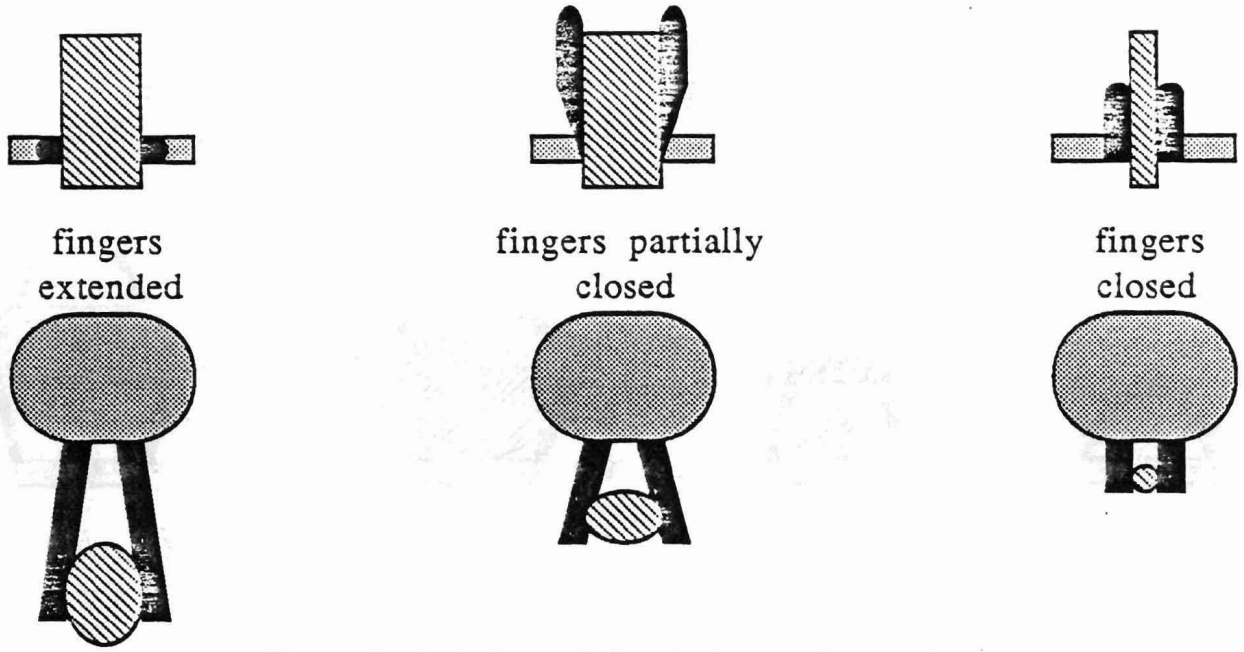


Figure 5: Variations of the pinch grasping mode

types and, in addition, yields a pinching grasp between the two fingers similar to that used by amputees who use a split hook. Another advantage to this configuration is that the palmar surfaces of the fingers are always facing directly inwards—simplifying the sensing of an object within a grasp—in contrast to the human hand, where the lateral movement of the fingers does not allow this. The five grasping modes are described below with their parallels in Schlesinger's and Cutkosky and Wright's work defined as well:

The **pinch** grip occurs when the two movable fingers are brought together on the opposite side of the palm from the thumb. The inside of these two fingers are lined with rubber, which allows for friction grasping of small objects. This is primarily a precision grasp, used for picking up small, delicate objects. It is similar to the lateral pinch grasp described by both Schlesinger and Cutkosky and Wright. In addition, some operations which are usually performed by Schlesinger's tip prehension and Cutkosky and Wright's two-finger precision grasp can be achieved in this configuration. The flexibility of this grasp is enhanced by the ability to change its nature by changing the angle of the fingers. In Figure 5, this technique is illustrated. This grasp is very similar

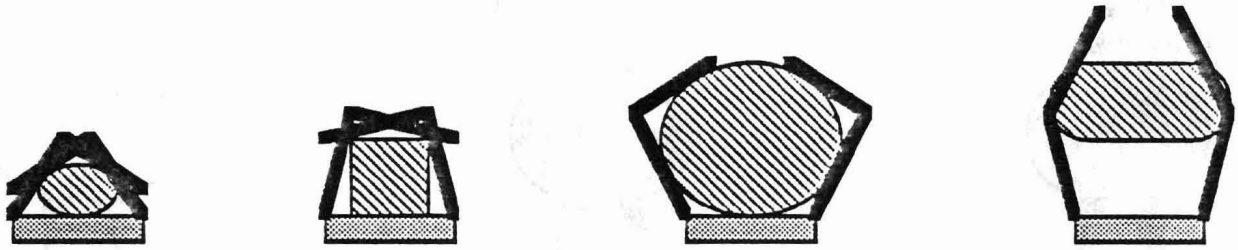


Figure 6: Variations in the cylindrical grasping mode

to the precision grasp used by amputees who have been fitted with a split hook prosthesis. In this case, a cylindrical groove between the halves of the hook allow for stable grasping of a pencil or similar small cylindrical objects. Such an implementation in the robotic end effector could prove useful.

The **cylindrical** grasp, when the two fingers are opposite the thumb, is analogous to Schlesinger's cylindrical grasp and Cutkosky and Wright's cylindrical power and precision grips. This mode allows for the apprehension of a wide range of shapes and sizes, from small cylindrical objects to larger rectangular box-shaped objects (see Figure 6). In addition, this mode allows a version of the lateral pinch grasp, when an object is held between the three fingertips. The attractiveness of this grasp lies in its strength. Since the palmar surfaces of all three fingers are holding the object against the palm, objects are held very securely.

The **spherical** grasp, with the three fingers roughly 120 degrees apart, is similar to Schlesinger's spherical grasp and Cutkosky and Wright's spherical power and 3-finger, 4-finger, and 5-finger precision grasps. In a power grasp, the palmar surfaces of the fingers are used to hold a spherical object against the palm, while in a precision grip, the three fingertips form a three-sided fingertip

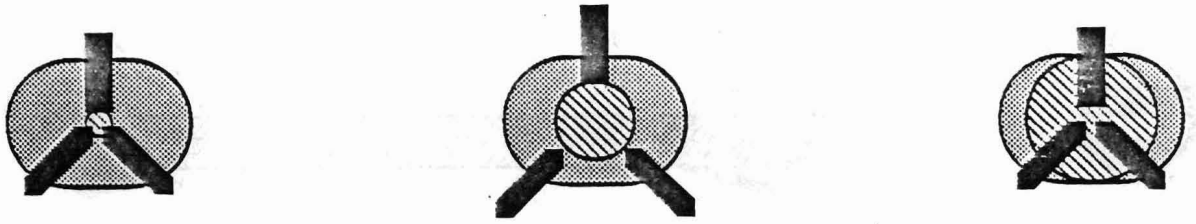


Figure 7: Variations of the spherical grasp

grasp which is similar to the chuck on a drill. In Figure 7, the application of this grasp to various objects is shown.

When the two fingers are rotated until they are opposite each other, they can be used in a **tip** grasping mode. This is exactly the tip prehension described by Schlesinger and the 2-finger precision grip described by Cutkosky and Wright. Although this grasp relies primarily on friction for stability, it can be useful in apprehending objects that are ackwardly placed or for manipulating objects securely held in some manner. The pinch grasp provides a more stable grasp of most small objects.

The **hook** mode of grasping uses all three fingers located together on one side of the palm. This allows for two types of grasping: a passive grip on a handle or similar structure where the fingers act as a hook, or an active grasp where all three fingers hold a large object against the palm. This is a grasp that could be used to lift one side of a large flat object (in cooperation with another hand) where the size of the object precludes an enveloping grasp. Figure 8 shows these uses.

Although these modes provide wide versatility in grasping, an equally flexible finger design is necessary in order to fulfill our design objectives. A finger of fixed shape pivoting around the edge of the palm would provide only limited capability. Although it could hold many objects, such a finger could only perfectly

Figure 8: Variations of the hook grasp

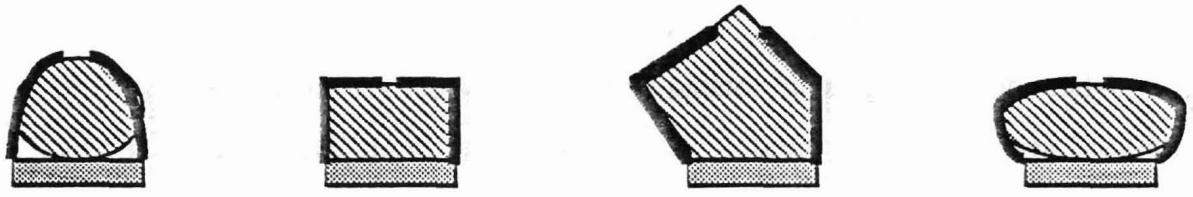


Figure 9: Variations in finger shape with changes in object shape

grasp a small number of objects with optimum contact points corresponding to its fixed shape. In Figure 9, we show how ideal finger shape varies with object geometry. We would like to have a finger which could change its geometry in response to the shape of the object. A multi-jointed finger such as those found on the Utah/MIT DH [1] and in the Salisbury hand [2] can comply to the object shape by integration of sensor feedback and position control. However, these fingers have 3 or 4 degrees of freedom. We need a finger which can achieve this same function without the control and actuation complexity associated with these added degrees of freedom.

The author originally proposed such a finger design in the Compliant Articulated Mechanical Manipulator (CAMM) [12], which incorporated a four-joint finger with two degrees of freedom. We have modified the design to yield a two-jointed one degree-of-freedom compliant finger design. The single degree

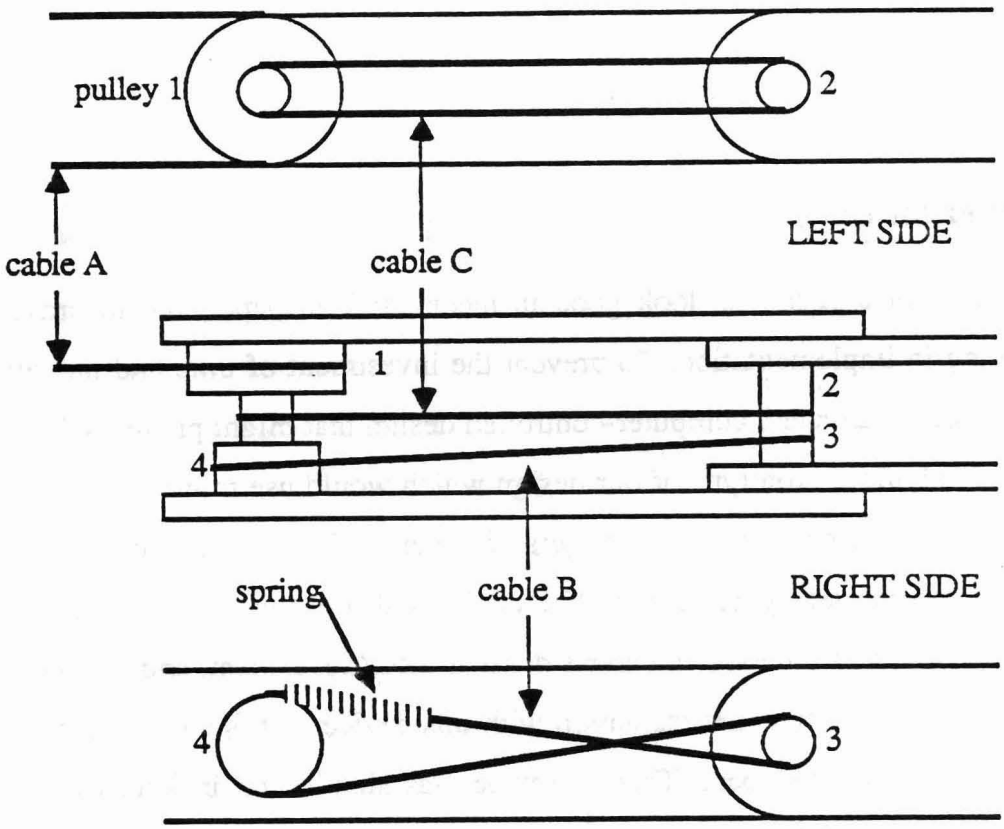


Figure 10: Schematic representation of actuation linkages

of freedom satisfies our need for simplicity, yet allows flexibility in object apprehension. Figure 10 shows a schematic of the linkages involved. This finger will *passively* shape itself to an object without the use of control computation or sensor feedback. The finger incorporates a spring in its linkage to provide compliance in one direction; this allows the second joint of the finger to continue to rotate once the first joint contacts an object. However, no matter how much the joints rotate independently, the finger will not comply in opening; that is, it will always maintain pressure on the object dependent only on the torque produced by the actuator. The compliance is implemented in the linkage contained on the right side of the finger, while at the same time the drive linkage on the left side of the finger actuates the finger and transfers gripping force. For a more detailed description of this finger and its kinematics, see [13].

Experimentation

It is common for a design to look good in theory and on paper, but to prove disappointing in implementation. To prevent the investment of time and money into a electrically-actuated, computer-controlled design that might prove useless, we decided to build a prototype of our design which would use movement of an experimenter's fingers to actuate the fingers of the end effector. This device was in essence a manual teleoperated end effector. This allowed us to test our ideas very quickly, utilizing the experimenter's brain as a control system, and his body as the actuator. It was in experimentation with this device that the actual design presented here was developed. This prototype was simple and inexpensive to build and allowed quick modification. In combination with prototypes of the finger design, we were able to finalize the design with little effort.

In the process of our experimentation, we found the device very useful; that all of the grasps necessary for enveloping grasps and tool handling were possible, and that the actions necessary for assembly and disassembly could be achieved. However, the device does have limitations. As anticipated, the design is more suited to enveloping grasps and handling large tools. Associated with the low number of degrees of freedom is a loss of dexterity in small parts manipulation. Although such objects can be grasped securely, movement of the objects within the grasp requires interaction with a table surface or another hand. We do not find this a serious fault for our work, since the use of two hands for assembly tasks is probably necessary anyway.

Conclusion

We have presented the basis of a medium-complexity compliant end effector design. The end result of our identification of a gap in end effector development has led to a four degree of freedom flexible end effector design that is especially suited for work in active sensing, assembly and disassembly, and grasping. We have attempted to support the rationale for this design on fundamental good engineering practice as well as on previous research. There are obviously many details of the design which have not been described here, but an electrically-actuated self-contained end effector for use on the end of a robotic manipulator is under construction. Use of this device will allow expansion of present research topics and allow for experimentation in new areas related to robotic manipulation.

References

[1] Jacobsen, S.C., E.K. Iversen, D.F. Knutti, R.T. Johnson, and K.B. Biggers, "Design of the Utah/MIT Dextrous Hand," *Proceedings of the IEEE Conference on Robotics and Automation*, San Francisco, April 1986.

[2] Salisbury, J.K., "Kinematic and Force Analysis of Articulated Hands," *Ph.D. Thesis, Stanford University,* July 1982.

[3] Trinkle, Jeffrey C., "The Mechanics and Planning of Enveloping Grasps," *Ph.D. dissertation*, University of Pennsylvania, June 1987.

[4] Bajcsy, R., "What can we learn from one finger experiments?" in M. Brady and R. Paul, editors, *The First International Symposium on Robotics Research*, MIT Press, Cambridge, 1984.

[5] Allen, P.K., "Object Recognition Using Vision and Touch," *Ph.D. dissertation*, University of Pennsylvania, September 1985.

[6] Stansfield, S.A., "Primitives, Features, and Exploratory Procedures: Building a Robot Tactile Perception System," *Proceedings of the IEEE International Conference on Robotics and Automation*, San Francisco, 1986.

[7] Klatzky, R.L., Ruzena Bajcsy, and Susan J. Lederman, "Object Exploration in One and Two Fingered Robots," *Proceedings of the IEEE Conference on Robotics and Automation*, 1987.

[8] Schlesinger, G., *Der Mechanische Aufbau der kunstlichen Glieder*, Part II of *Ersatzglieder und Arbeitshilfen*, Springer Verlag, Berlin, 1919.

[9] Cutkosky, Mark R. and Paul K. Wright, "Modeling Manufacturing Grips and Correlations with the Design of Robotic Hands," *Proceedings of the IEEE Conference on Robotics and Automation*, San Francisco, 1986.

[10] Iberall, Thea, "The Nature of Human Prehension: Three Dextrous Hands in One," *Proceedings of the IEEE Conference on Robotics and Automation*, 1987.

[11] Skinner, F., "Design of a multiple prehension manipulator system," *ASME publication 74-DET-25*, October 1974, and *Mechanical Engineering*, September 1975, pp. 30-37.

[12] Ulrich, Nathan, "The Compliant Articulated Mechanical Manipulator," University of Pennsylvania, GraspLab Memo, April 1985.

[13] Ulrich, Nathan, "A Two-Jointed One Degree-of-Freedom Compliant Finger," University of Pennsylvania, September 1987.

A.4 A New Computational Structure for Real-Time Dynamics

A NEW COMPUTATIONAL STRUCTURE FOR REAL TIME DYNAMICS

Alberto Izaguirre, Minoru Hashimoto, Richard Paul, Vincent Hayward

ABSTRACT

In this paper we present a new structure for the computation of robot dynamics in real time. The basic characteristic of this structure is the division of the computation into a high priority synchronous task and a low priority background task. The background task computes the inertial and gravitational coefficients as well as the forces due to the velocities of the joints. Each control sample period, the high priority synchronous task computes the product of the inertial coefficients by the accelerations of the joints as well as performing the addition of the torques due to the velocities and gravitational forces. Kircanski (Kircanski86) has shown that the band-width of the variation of joint angles and their velocities is an order of magnitude less than the variation of the joint accelerations. This result agrees with the experiments that we have carried out on a PUMA260 robot.

Two main strategies have been adopted to reduce the computational burden of the dynamic equations. The first involves the selection of efficient algorithms for the computation of the equations. The second is the reduction in the number of dynamic parameters by identifying linear dependencies among parameters, as well as by making a significance analysis on the contribution of the parameters to the torques.

We chose an iterative procedure for the computation of the inertial and gravitational coefficients (Izaguirre86, Renaud85, Featherston84), and a recursive iteration for the computation of the velocity torques (Khalil86). In our experiments using the PUMA260 we obtained a set of 52 linearly independent parameters from an initial set of 78 parameters. The identification of the parameters revealed only 23 parameters to be significant.

These reductions permit the calculation of the inertias and gravitational coefficients, for the PUMA260 without load, with 98 multiplications and 70 additions; and the calcultion of the velocity torques with 140 multiplications and 110 additions. In the case of an arbitrary load at the end effector, the calculation of the inertias and gravitational coefficients require 190 multiplications and 150 additions and the velocity torques 200 multiplications and 170 additions. Velocity torques, inertial coefficients, and gravitational coefficients can be computed in the background in 20 milliseconds using an Intel 8087 microprocessor. The synchronous task requires only 6 multiplications and 6 additions per joint.

1 INTRODUCTION

The inverse dynamics equations of a robot expresses the necessary generalized forces or torques, to be applied to the different joints of the manipulator as functions of the required position, velocity

and acceleration of the joints. The computed torque control scheme is based on calculating the generalized forces from a model of the manipulator. These signals can be added as feed-forward terms in a conventional feed-back control loop, in order to linearize and decouple the system. The equations may be expressed in the following form

$$F_i = (D_{ii} + I_{ai}) * \ddot{q}_i + \sum_{j=1;j\neq i}^{n} D_{ij} * \ddot{q}_j + \sum_{i,j,k} D_{i,jk} * \dot{q}_j * \dot{q}_k + D_i + F_{di} * \dot{q}_i + F_{si} * sign(\dot{q}_i) \quad (1)$$

Where :

D_{ii} is the effective inertia at the joint i
D_{ij} is the coupling inertia between joints i and j
$D_{i,jk}$ are the Coriolis and centrifugal coefficients at the joint i
D_i is the gravitational force at the joint i
I_{ai} is actuator inertia
F_{di} is the viscous friction
F_{si} is the Coulomb friction
F_i is the generalized force at the joint i
$\dot{q}_i$ is the velocity of the joint i
$\ddot{q}_i$ is the acceleration of the joint i

This control scheme requires to compute the dynamic equations in real time, that is, within the sample period of the controller. For this purpose, a great deal of research has been done to the reduction of the complexity of these equations. However, the problems of the automatic generation of the equations, and of the identification of the constant parameters in the equations, functions of the moments of the links, frictions, dampings and inertias on the motors, are related problems in a practical fashion.

In this paper, we are address these three problems making appropiate references to previous works; explain our contributions, and show the results obtained from our experiments with the PUMA260 robot.

2 OVERVIEW OF THE INVERSE DYNAMICS COMPUTATION

In this section we review the work that have been done in the past years from the point of view of complexity, implementation and identification of the parameters of the inverse dynamic equations of robot manipulators.

2.1 COMPLEXITY IN THE CALCULATION OF INVERSE DYNAMICS

There are two main approaches to derive the required equations: the Lagrange formalism and the Newton-Euler formalism. The lagrangian approach, developped by Uicker (Uicker68), has been used by several researchers, including Khan (Khan71), R. Paul (Paul72) and Bejcsy (Bejcsy74). The pricipal disadvantage of this formulation is the complexity in the order $O(n^4)$, due to redundancies in the calculation. A simplification using a forward recursion on the velocities and accelerations of the joints and a backward recursion on the generalized forces was introduced by Hollerbach (Hollerbach80,82). His approach simplified the computation substantially reducing the complexity to a linear function of the number of joints, i.e. $O(n)$. However, his method cannot compute the dynamic coefficients that depend only on the joint angles. It thus encounters the same problem as does the Newton-Euler computation. Megahed (Megahed84) calculated the dynamic equations based on the Lagrange equations and the dynamic coefficients. The complexity acheived is in the order of $O(n^3)$, requiring approximatively 1000 multiplications and 700 additions for a general manipulator of 6 degrees of freedom.

Recently, Featherston (Featherston84), following a spatial notation, and Renaud (Renaud83,85), following tensorial notation, have reduced the computation of the dynamic coefficients, using the Lagrangian method and the notion of the "compound link". We used the theorem of "conservation of the momentum" (Izaguirre86), acheiving equivalent results in the calculation of the inertial and gravitational coefficients. Basically, the computation consists of the calculation of the moments of the "compound link i" (i.e. the link formed by the links i through the last link) in function of the moments of the "compound link $i+1$". This recursion leads to a big saving on the calculations, as well as a systematic way of calculating the dynamic coefficients. The complexity using this approach is $O(n^3)$ if the entire dynamic model is computed, and $O(n^2)$ if only the inertial and gravitational coefficients are calculated.

The second approach, i.e. the Newton-Euler method, consists of the calculation of the generalized by using Newton's law to calculate forces, and Euler's law to calculate torques. One of the first methods of calculation of the generalized forces was developed by Likins (Likins71). Luh, Walker and Paul (Luh80) developed an algorithm using a forward recursion for the velocities and accelerations of the joints, and a backward recursion for the calculation of the generalized forces. The complexity of the algorithm is $O(n)$ and it requires only 800 multiplications and 600 additions for a general six degrees of freedom manipulator. Khalil, (Khalil86), based on Luh's work, reduced the computation by regrouping common terms. The dynamic equations can be calculated by his algorithm in 540 multiplications and 480 additions in the case of a general six degrees of freedom manipulator.

2.2 STRUCTURES OF COMPUTATION OF THE INVERSE DYNAMIC EQUATIONS

Different approaches have been developed for the computation of dynamic equations in real time. Luh, Walker and Paul (Luh80), based on the Newton-Euler method, computed the dynamic equations for the Stanford manipulator in 4.5 milliseconds using floating point assembly language in a PDP-11/45. Raibert (Raibert77) used look-up tables to compute the dynamic equations. 460 multiplications and 260 additions were required to calculate the equations of a general 6 degrees of freedom manipulator, reducing the complexity of the Newton-Euler method by a factor of 2. Luh and Lin (Luh82) described a procedure for scheduling subtasks of a group of 6 microprocessors, one per joint, in order to compute the Newton-Euler equations. Their estimation indicates that 320 multiplications and 280 multiplications are required to compute the dynamics of a 6 degrees of freedom manipulator.

Recently, Orin (Orin85) has introduced a structure of control by dividing it into ten different tasks. He arrives at the conclusion that the longest task is the one that computes the inverse dynamics. Lathrop (Lathrop85) has studied a parallel computation of the inverse dynamics. He has proposed a pipelined architecture reducing the Newton-Euler computation by two orders of magnitude. The latency to compute the dynamics of a 6 degrees of freedom manipulator would be of the order of 15 multiplications and 43 additions. He improves the Newton-Euler parallel implementation, reducing the complexity to a logarithmic expression on the number of joints. Only 11 multiplications and 28 additions are required to compute the manipulator dynamics. Also, a systolic pipeline implementation is possible reducing the latency to only 4 floating-points operations.

The disadvantage of this method lies in the difficulty of implementation, because custom designed VLSI devices are required, increasing the cost of the product. We will introduce later a solution based on microprocessors, which has the advantages of a very easy implementation, maintaining at the same time the required speed of computation and the accuracy in the calculation of the equations.

2.3 IDENTIFICATION OF THE ROBOT'S PARAMETERS

The constant parameters of the dynamic equations depend on the masses, centers of gravity and inertias of the links, as well as on the inertias and frictions of the motor. Ferreira (Ferreira84) realized that the expression of the torques could be expressed as a linear function of these parameters. He also realized that many parameters were linearly dependent, and that it was necessary to eliminate these dependences. However, he didn't give any algorithm for this purpose. In his experiments, he identified the parameters using the torque measured in the first joint of the robot TH8, using a Kalman filter.

Hollerbach (Hollerbach85) identified the parameters of a direct driven arm by using least squares method to fit the measured torques in a given trajectory. He measured the joint position and torque on the trajectory, estimating the velocity and acceleration. However he did not mention the problem

of the elimination of the linearly dependent parameters, that may introduce erroneous estimation of the parameters. However, still the fitting seems good. Khosla and Kanade (Koshla86) presented an algorithm to eliminate the linear dependencies in the dynamic model. However this paper was not ready on time for publication. Olsen (Olsen86) identified the constant parameters in simulation, but special cases were considered to identify different parameters. Finally, Khatib (Khatib86) identified the parameters of a PUMA 560 dismounting the robot and measuring directly the parameters. He was able to do a significance analysis to reduce some of the parameters.

In the next section we will present an identification method based on fitting of the measured torques over different trajectories, removing the linear dependencies and at the same time performing a significance analysis, that reduces the computation considerably.

3 OUR APPROACH

In this section, we present a new approach, based on microprocessors, to compute the inverse dynamics in real time. The scheme is based on the division of the computation into a high priority synchronous task and a background task (Izaguirre85,86). The background task updates the inertial and gravitational coefficients as well as the generalized forces due to the velocities of the joints. The synchronous task computes the final generalized forces by multiplying the inertial coefficients by the acceleration of the joints, adding at the same time the gravitational and velocities forces. This computational scheme agrees with the experiments that have been done by Kircanski (Kircanski86). He calculated trajectories for the PUMA560 robot, analyzing the bandwith of the position, velocity and acceleration of the joints respectively. He concluded that the position and velocitie's spectra are similar, and that the spectra of the acceleration is about 5 times larger. In our experiment we estimated the velocities and accelerations from the measurements of the joint angles. If we examine at Figure 1 through 5 in Appendix A, we can realize that the variations of position and velocity in the joints are much smaller than the variations in acceleration.

We require to calculate the inertial coefficients as well as the forces due to the velocities as fast as possible. To do so, we chose an efficient recursive algorithm to compute the inertial and gravitational coefficients (Izaguirre86) and an efficient recursive algorithm to compute the velocity forces (Khalil86). The identification of the parameters permits a further reduction by first eliminating the linear dependencies and by dropping the parameters that are not significant.

In the next sections we will explain in detail the computation of the dynamic coefficients, as well as the identification, explaining the results obtained for the PUMA260 robot.

4 CALCULATION OF THE DYNAMIC COEFFICIENTS

In this section we first develop the equations of the inertial and gravity loading coefficients for a differential mass dm located in link j. The equations are elaborated from the theorem of conservation of the momentum (Izaguirre85,86) leading to a very easy and understandable procedure. The calculation for the entire link will be obtained by integrating these formulas over the mass of the link.

4.1 CALCULATION OF THE TERMS D_{ij}

The coefficient D_{ij} corresponds to the generalized force in the joint j due to the acceleration of the joint i. We will consider only the terms for which $i < j$ and then show that $D_{ij} = D_{ji}$. We consider the four possible combinations for the joints i and j (revolute revolute, prismatic revolute, revolute prismatic, and prismatic prismatic). In all the cases we assume the acceleration $\ddot{q}_j$ to be different from zero, and that all the other accelerations and velocities are zero.

In the revolute-revolute case, the term D_{ij} corresponds to the torque in the joint i due to the acceleration $\ddot{q}_j$ only. This torque can be calculated by using the derivative of the angular momentum around the axis i with respect to time. We recall that the angular momentum of a point of mass dm with respect a coordinate system is calculated by the cross product $\mathbf{r} \times \mathbf{v}dm$, where $\mathbf{r}$ is the position and $\mathbf{v}$ is the velocity of the differential of mass dm in this coordinate system. The variation of the angular momentum around $\mathbf{z_i}$ is : $dm\ \mathbf{r_i} \times (\ \mathbf{z_j} \times \mathbf{r_j}) \cdot \mathbf{z_i}\ \ddot{q}_i \Delta t$. The torque is calculated by differentiating the angular momentum with respect to time:

$$\Gamma_i = dm\ \mathbf{r_i} \times (\mathbf{z_j} \times \mathbf{r_j}) \cdot \mathbf{z_i}\ \ddot{q}_i \tag{2}$$

The term D_{ij} is then

$$D_{ij} = dm\ \mathbf{r_i} \times (\mathbf{z_j} \times \mathbf{r_j})\ \cdot \mathbf{z_i} \tag{3}$$

or as $\boxed{D_{ij} = dm\ (\mathbf{z_i} \times \mathbf{r_i}) \cdot (\mathbf{z_j} \times \mathbf{r_j})}$.

This formula reveals the symmetry between D_{ij} and D_{ji}.

In the prismatic-revolute case, the coefficient D_{ij} corresponds to the force in the direction of the joint i due to the acceleration $\ddot{q}_j$. It can be calculated by differentiating the linear momentum in the direction of the axis i with respect to time. The linear momentum in the direction $\mathbf{z_i}$ is: $dm(z_j \times \mathbf{r_j}) \cdot \mathbf{z_i} \ddot{q}_i * \Delta t$, and the force is given by the formula:

$$F_i = dm\ (\mathbf{z_j} \times \mathbf{r_j}) \cdot \mathbf{z_i} \ddot{q}_j. \tag{4}$$

The coefficient D_{ij} is then

$$\boxed{D_{ij} = dm\ (\mathbf{z_j} \times \mathbf{r_j}) \cdot \mathbf{z_i}}. \tag{5}$$

In the revolute-prismatic case, the coefficient D_{ij} corresponds to the torque in the axis i due to the acceleration $\ddot{q}_i$. This torque is calculated by calculating the derivative of the angular momentum with respect to time, leading to the following equation:

$$\Gamma_i = dm\ (\mathbf{r_i} \times \mathbf{z_j}) \cdot \mathbf{z_i} \ddot{q}_j. \tag{6}$$

The terms D_{ij} can be expressed as

$$D_{ij} = dm\ (\mathbf{r_i} \times \mathbf{z_j}) \cdot \mathbf{z_i} \tag{7}$$

or as $\boxed{D_{ij} = dm\ (\mathbf{z_i} \times \mathbf{r_i}) \cdot \mathbf{z_j}}$.

In the prismatic-prismatic case it is very simple to see that the generalized force is

$$F_i = dm\ \ddot{q}_j\ \mathbf{z_j} \cdot \mathbf{z_i} \tag{8}$$

and D_{ij} is equal to

$$\boxed{D_{ij} = dm\ \mathbf{z_j} \cdot \mathbf{z_i}}. \tag{9}$$

4.2 CALCULATION OF D_i

In order to compute the gravity coefficients D_i, we consider both the revolute and prismatic cases. In the first case, the torque exerted by the gravity on the revolute joint is equal to

$$\Gamma_i = dm\ (\mathbf{r_i} \times \mathbf{g}) \cdot \mathbf{z_i}, \tag{10}$$

where :

$\mathbf{g}$ is the acceleration due to the gravity.

To compensate for the gravity load a torque must be exerted in the opposite direction. Thus the coefficient D_i is equal to

$$D_i = -dm\ (\mathbf{r_i} \times \mathbf{g}) \cdot \mathbf{z_i} \tag{11}$$

or as $\boxed{D_i = -dm(\mathbf{z_i} \times \mathbf{r_i}) \cdot \mathbf{g}}$.

Once again, in the prismatic case, the force is equal to $m\ \mathbf{g} \cdot \mathbf{z_i}$. To compensate for this force we must exert an opposing force. The coefficient D_i is then equal to

$$\boxed{D_i = -dm\ \mathbf{g} \cdot \mathbf{z_i}.} \tag{12}$$

4.3 INTEGRATION OF THE EQUATIONS

In this section we will integrate the equations derived earlier over one link. This link corresponds to the "compound link" j, i.e. the link formed by the links j trough the last link n. In fact, the calculation of the coefficients D_{ij} and D_j depend only on the acceleration of the joint j; the rest of the links don't move relatively to each other. An easy recursive calculation of the moments of the "compound link" j as a function of the "compound link" $j+1$ leads to a big reduction in the calculation of the inertial and gravitational coefficients. This recursion will be explained later in detail.

The most difficult term to integrate is the parameter D_{ij} for the revolute-revolute case. The expression of this term for a point mass situated in the "compound link" j, $(j > i)$, is :

$$\begin{aligned} D_{ij} &= dm\ (\mathbf{z_i} \times \mathbf{r_i}) \cdot (\mathbf{z_j} \times \mathbf{r_j}) \\ &= dm(\mathbf{z_i} \times (\mathbf{p_i} + \mathbf{l_j})) \cdot (\mathbf{z_j} \times \mathbf{l_j}), \end{aligned} \tag{13}$$

where:

- $\mathbf{p_i}$ is the vector between the origins of frames i and j,
- $\mathbf{l_j}$ is the vector between the origin of frame j and the elementary mass dm.

The expansion of this expression leads to the following formula:

$$\begin{aligned} D_{ij} &= dm[(\mathbf{z_i} \times \mathbf{p_i}) \cdot (\mathbf{z_j} \times \mathbf{l_j}) \\ &\quad +(\mathbf{z_i} \times \mathbf{l_j}) \cdot (\mathbf{z_j} \times \mathbf{l_j})]. \end{aligned} \tag{14}$$

The integration of the first term is obtained by the sustitution of the point mass by the center of gravity of the "compound link". The last term can be integrated using tensor notation. The term $(\mathbf{z_i} \times \mathbf{l_j})$ is expressed in tensor notation as $-\widehat{\mathbf{l}}_\mathbf{j}\ \mathbf{z_i}$ or as $\mathbf{z_i}\widehat{\mathbf{l}}_\mathbf{j}$, leading to the following expression :

$$dm(\mathbf{z_i} \times \mathbf{l_j}) \cdot (\mathbf{z_j} \times \mathbf{l_j}) = -dm\ \mathbf{z_i}\ \widehat{\mathbf{l}}_\mathbf{j}\ \widehat{\mathbf{l}}_\mathbf{j}\ \mathbf{z_j} \tag{15}$$

$$\begin{aligned} -\widehat{\mathbf{l}}_\mathbf{j}\widehat{\mathbf{l}}_\mathbf{j} &= -\begin{pmatrix} 0 & -l_{jz} & l_{jy} \\ l_{jz} & 0 & -l_{jx} \\ -l_{jy} & l_{jx} & 0 \end{pmatrix} \begin{pmatrix} 0 & -l_{jz} & l_{jy} \\ l_{jz} & 0 & -l_{jx} \\ -ljy & l_{jx} & 0 \end{pmatrix} \\ &= \begin{pmatrix} {l_{jz}}^2 + {l_{jy}}^2 & -l_{jx}l_{jy} & -l_{jx}l_{jz} \\ -l_{jx}l_{jy} & {l_{jx}}^2 + {l_{jz}}^2 & -l_{jy}l_{jz} \\ -l_{jx}l_{jz} & -l_{jy}l_{jz} & {l_{jx}}^2 + {l_{jz}}^2 \end{pmatrix} \end{aligned}$$

This term integrated over the link corresponds to the inertial matrix of the "compound link" j in the frame j. The final expression is :

$$\begin{aligned} D_{ij} &= (\mathbf{z_i} \times \mathbf{p_i}) \cdot (\mathbf{z_j} \times \mathbf{M_j} * \mathbf{D_j}) + \mathbf{z_i}\widehat{\mathbf{I}}_\mathbf{j}\mathbf{z_j} \\ &= (\mathbf{z_i} \times \mathbf{p_i}) \cdot (\mathbf{z_j} \times \mathbf{L_j}) + \mathbf{z_i}\widehat{\mathbf{I}}_\mathbf{j}\mathbf{z_j}, \end{aligned} \tag{16}$$

where:

- M_j is the mass of the compound link j,
- D_j is the center of gravity of the "compound link" j on the origin of the frame j,
- $\mathbf{L_j}$ is D_j multiplied by M_j, i.e. the first moment of the "compound link" j in the frame j.
- $\widehat{\mathbf{I}}_\mathbf{j}$ is the inertia matrix of the "compound link" j in the origin of the frame j.

All the other terms D_{ij} for the other cases and the parameters D_i are simple, with the integration resulting the substitution of the coordinates of the center of gravity of the link for those of the point mass.

4.4 CALCULATION OF THE COEFFICIENTS USING HOMOGENOUS TRANSFORMATIONS

In this section we explain how to compute the inertial and gravitational coefficients once the topology of the robot is defined by homogenous transformations. These homogenous transformations describe the relationships between the i^{th} and j^{th} coordinate frames. We will name these transformations the

$T_{i,j}$ matrix, with the convention that the first frame or base is the frame number 0, and that the last link corresponds to the frame number n.

The part of the homogenous transformation corresponding to the rotation is the matrix R_{ij}, and the part corresponding to the translation is the vector p_{ij}.

The matrix R_{ij} can be decomposed in three vectors n_{ij} o_{ij} and a_{ij} :

$$T_{ij} = \begin{pmatrix} n_{ijx} & o_{ijx} & a_{ijx} & p_{ijx} \\ n_{ijy} & o_{ijy} & a_{ijy} & p_{ijy} \\ n_{ijz} & o_{ijz} & a_{ijz} & p_{ijz} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{17}$$

4.4.1 EXPRESSIONS FOR THE D_{ij} COEFFICIENTS

In this section we will derive the expressions of the dynamic coefficients using homogenous transformations.

For the revolute-revolute case, the equation 16 corresponds to the formula:

$$\begin{aligned} D_{i,j} &= (0\ |0\ |1)\ R_{i,j}\ I_j\ (0\ |0\ |1)^T + \\ &+ (-p_{i,j,y}\ |p_{i,j,x}\ |0)\ R_{i,j}\ (-L_{j,y}\ |L_{j,x}\ |0)^T, \end{aligned} \tag{18}$$

where:

- $i \in [0, n-1]$ and $j \in [0, n-1]$, n being the number of degrees of freedom.

- $(L_{j,x},\ |L_{j,y},\ |L_{j,z})$ are the components of the first moments of the "compound link" j expressed in the j frame.

- I_j is the inertia matrix of the "compound link" j expressed on the j frame.

Equation 18 leads to the following:

$$\begin{aligned} D_{ij} = \quad & (n_{i,j,z}\ |o_{i,j,z}\ |a_{i,j,z}) \begin{pmatrix} I_{1,3,j} \\ I_{2,3,j} \\ I_{3,3,j} \end{pmatrix} \\ & +(A\ B) * (-L_{j,y}\ |L_{j,x})^T \end{aligned} \tag{19}$$

with

$$A = (-p_{i,j,y} * n_{i,j,x} + p_{i,j,x} * n_{i,j,y})$$
$$B = (-p_{i,j,y} * o_{i,j,x} + p_{i,j,x} * o_{i,j,y})$$

In the revolute-prismatic case, the formula 5 leads to :

$$D_{ij} = (n_{i,j,z} \mid o_{i,j,z})(-L_{j,y} \mid L_{j,x})^T \tag{20}$$

Similarly, in the prismatic-revolute case, equation 7 leads to :

$$\begin{aligned} D_{ij} &= M_j \, (-p_{i,j,y} \mid p_{i,j,x}) \begin{pmatrix} a_{i,j,x} \\ a_{i,j,y} \end{pmatrix} \\ &\quad -(n_{i,j,z} \mid o_{i,j,z})(-L_{j,y} \mid L_{j,x})^T. \end{aligned} \tag{21}$$

And finally for the prismatic-prismatic case, Equation 9 may be expressed as

$$D_{ij} = M_j \ * \ a_{i,j,z}. \tag{22}$$

4.4.2 EXPRESSION OF THE D_i COEEFICIENTS

We differentiate between the revolute and prismatic cases as before. In the first case Equation 11 can be expressed as

$$D_i = - \, (g_x \mid g_y \mid g_z) \, R_{0,i} \, (-L_{i,y} \mid L_{i,x} \mid 0)^T, \tag{23}$$

where :

- $(g_x \mid g_y \mid g_z)$ are the components of the acceleration of gravity in the base frame.

If the gravity is parallel to the z axis of the base frame, Equation 23 takes the following simplified form:

$$D_i = -g \, (n_{0,i,z} \mid o_{0,i,z}) \, (-L_{i,y} \mid L_{i,x})^T \tag{24}$$

In the second case, Equation 12 leads to the following expression:

$$D_i = -M_j \, (g_x \ g_y \ g_z) \, (a_{0,i,x} \mid a_{0,i,y} \mid a_{0,i,z})^T. \tag{25}$$

Once again if the gravity is in the direction of the z axis of the base frame Equation 24 may be expressed as:

$$D_i = -M_j \; g \; * \; |a_{0,i,z}. \tag{26}$$

4.5 RECURSION OF THE TERMS

In the previous sections we derived the expression of the inertial and gravitational terms for the "compound link" j. In this section we will work out the recursion for the calculation of the moments of the "compound link" j in function of those of the "compound link" $j+1$.

4.5.1 RECURSION FOR THE MASSES

Obviously $M_j = M_{j+1} + m_{j+1}$, where m_j is the mass of the link j and M_j is the mass of the "compound link" j.

4.5.2 RECURSION FOR THE FIRST MOMENTS

The first moment of the "compound link" j corresponds to the product of the mass M_j by the center of gravity of the "compound link" j. It can be expressed recursively by using the following expression:

$$\mathbf{L}_j(j) = R_{j,j+1} \; (\mathcal{L}_j(j+1) \; + \; \mathbf{L}_j(j+1) + \mathbf{p}_{j,j+1}(j+1) * M_j) \tag{27}$$

where:

- $\mathbf{L_j}(j)$ is first moment of the "compound link" j expressed in the frame j.

- $\mathcal{L}_j(j+1)$ is the first moment of the link j expressed in the frame $j+1$. The fact that we expressed it in the frame $j+1$ is due to the selection of the Denavit-Hartemberg parameters. The moments of a link are constant with respect to the next frame rather than with respect to the j^{th} frame. This is why modified Denavit-Hartemberg parameters have been sometimes used (Khalil86) (Craig86), in which case the moments of a link are constant with respect to the relative frame. However, computationally both approaches lead to similar results, the differences are in the notation.

- $\mathbf{p}_{j,j+1}(j+1)$ corresponds to the vector between the origins of frames j and $j+1$. It is also constant with respect to $j+1$ if we choose the Denavit-Hartemberg parameters, and is constant with respect to j if we choose the modified Denavit-Hartemberg parameters.

4.5.3 RECURSION FOR THE INERTIA

The inertia of the "compound link" j can be expressed as a function of the "compound link" $j+1$ by transforming the inertia of the "compound link" $j+1$ from the origin of the frame $j+1$ to its center of gravity and from the center of gravity to the origin of the frame j. Also we have to add the inertia of the link j in a similar procedure. This can be expressed as it follows :

$$\begin{aligned} I_j(j+1) = I_{j+1}(j+1) + M_{j+1} * (\widehat{D}_{j+1}(j+1)\ \widehat{D}_{j+1}(j+1)) \\ -M_{j+1}(j+1) * ((\widehat{D}_{j+1}(j+1) + \widehat{p}_{j,j+1}(j+1)) * (\widehat{D}_{j+1}(j+1) + \widehat{p}_{j,j+1}(j+1))) \\ + \mathcal{I}_j(j+1) + m_j * (\widehat{d}_j(j+1)\widehat{d}_j(j+1)) \\ - m_j * ((\widehat{d}_j(j+1) + \widehat{p}_{j,j+1}(j+1))(\widehat{d}_j(j+1) + \widehat{p}_{j,j+1}(j+1))) \end{aligned} \tag{28}$$

where :

- $I_j(j)$ is the inertia matrix of the "compound link" j (expressed in the j^{th} frame),

- $\mathcal{I}_j(j+1)$ is the constant inertia of the link j in the frame $j+1$,

- $\widehat{p}_{j,j+1}$ is the tensor correponding to the vector $p_{j-1,j}$,

- $\widehat{D}_{j+1}(j+1)$ is the tensor corresponding to the center of gravity of the compound link $j+1$ in the frame $j+1$,

- $\widehat{d}_j(j+1)$ is the tensor corresponding to the center of gravity of the link j in the frame $j+1$.

This leads to the following equation:

$$\begin{aligned} I_{j(j)} = R_{j,j+1}(\mathcal{I}_j(j+1) + I_{j+1}(j+1) + M_j * \widehat{p}_{j,j+1}(j+1)\widehat{p}_{j,j+1}(j+1)) \\ R_{j,j+1}{}^T - \widehat{p}_{j,j+1}(j)\ \widehat{L}_j(j)\ -\ \widehat{L}_j(j)\ \widehat{p}_{j,j+1}(j) \end{aligned} \tag{29}$$

where the term $p_{j,j+1}(j+1)$ is a constant if defined in the frame $j+1$.

These terms can be easily calculated using homogenous transformation as explained before for the cases of the inertial and gravitational coefficients.

5 CALCULATION OF THE VELOCITY TERMS

There are two main methods to calculate the velocity terms. The first requires the calculation of the velocity coefficients, Coriolis and Centrifugal terms, and multiplication of these coefficients by the

velocities of the joint. The Coriolis and Centrifugal terms, can be efficiently calculated by using the Christoffel symbols over the inertial terms (Renaud85). The second method comprises the calculation of the velocity torques by using the Newton-Euler method. A simplification of this method using intermediate variables has been proposed by Khalil (Khalil86). Using this algorithm, in order to calculate the velocity torque, we have to initialize the velocities and accelerations to zero since we don't consider gravitational effects.

The advantage of using this last method is that the inertial and gravitational coeeficients, and the velocity toques can be computed independently, unlike the Christoffel symbols which depend on the inertial coefficient. Also, the complexity of the method is linear, rather than $O(n^3)$ for the method using the Christoffel symbols.

The forward recursion for the velocities and accelerations considering only the influence due to the velocities of the joints is the following:

$$w_i(i) = {R^T}_{i-1,i} * (w_{i-1}(i-1) + (1-\sigma_i)(0\ |0\ |\dot{q}_i)^T \quad (30)$$

$$\dot{w}_i(i) = {R^T}_{i-1,i} * (\dot{w}_{i-1}(i-1) + (1-\sigma_i) * w_{i-1}(i-1) \times (0\ |0\ |\dot{q}_i)^T)) \quad (31)$$

$$\dot{v}_i(i) = {R^T}_{i-1,i} * \dot{v}_{i-1}(i-1) + 2 * \sigma_i * w_i(i) \times {R^T}_{i-1,i}(0\ |0\ |\dot{q}_i)^T + U_i * p_{i-1,i}(i) \quad (32)$$

where : $w_i(i)$ is the angular velovity of the frame i expressed in the frame i, $\dot{w}_i(i)$ and $\dot{v}_i(i)$ are the angular and linear accelerations of the frame i expressed on the frame i, and U_i is the matrix $U_i = \dot{\widehat{w}}_i(i) + \widehat{w_i(i)}\widehat{w_i(i)}$.

The backward recursion can be expressed by using the formula :

$$\mathcal{F}_i(i+1) = m_i \dot{v}_{i+1}(i+1) + U_{i+1} * \mathcal{L}_i(i) \quad (33)$$

$$\mathcal{N}_i(i+1) = \mathcal{I}_i(i+1)\dot{w}_{i+1}(i+1) + w_{i+1}(i+1) \times \mathcal{I}_i(i+1)w_{i+1}(i+1) \quad (34)$$

$$f_i(i) = R_{i,i+1}(f_{i+1}(i+1) + \mathcal{F}_i(i+1) \quad (35)$$

$$n_i(i) = R_{i,i+1}(\mathcal{N}_i(i+1) + n_{i+1}(i+1) \quad (36)$$

$$+\mathcal{L}_i(i+1) \times \dot{v}_{i+1}(i+1)) + p_{i,i+1}(i) \times f_i(i) \quad (37)$$

where $\mathcal{F}_i(i+1)$ is the force due to the motion of link i, $\mathcal{N}_i(i+1)$ is the torque due to the mtion of the link i, $f_i(i)$ is the total force in the link i expressed with respect to the frame i, and $n_i(i)$ is the total torque in the link i expressed with respect to the frame i.

Khalil chose some intermediate variables that further simplified the computation, and the reader is referred to (Khalil86) for more details.

6 IDENTIFICATION OF THE DYNAMIC COEFFICIENTS

The torques can be expressed as linear functions of the masses of the, the first moments (the masses of the links multiplied by the center of gravity of the links), and the second moments (the inertia matrix of the links). The easiest way of showing this linear relationship is by considering the Newton-Euler's method. The forces and torques due to the velocities and accelerations of these links, are calculated by linear functions of the moments of the links, i.e. multiplications of these moments by functions that depend on the velocities and accelerations of the links, as show above. In the backward recursion, these forces and torques transform from one link to the previous one in a linear form, and the influence of the new link is taken into account by the addition of the forces and torques due to previous link to the force and torque due to the actual link. Finally because the resulting force or torque in the joint is a projection of the forces and torques in the joint, we show that the dynamic equations can be expressed as a linear function of the link moments.

This result can be shown by using the results obtained in the calculation of the inertial and gravitational coefficients. First, it is easy to verify that the compound link moments are a linear function of the moments of the constituent links. Second, the coefficient D_{ij} and D_i are linear functions of the moments of the compound link. Third, as the D_{ijk} terms are calculated by the Christoffel symbols that are additions of partial derivatives of the D_{ij} terms with respect to the joint angle, we prove that the terms D_{ijk} are linear function of the link moments. Finally as the computation of the forces or torques is made by multiplications of these coefficients by the velocities and accelerations of the links, we prove that the forces/torques are linear functions of the link moments.

The problem is however not trivial as many of these terms are mutually linearly. It is important to calculate these dependencies in order to seek a reduction in the number of parameters to identify as well as to arrive to a unique estimation. In the following paragraphs we explain the method used to get rid off the dependencies as well as the experiments that we performed in the PUMA 260.

6.1 ALGORITHM TO IDENTIFY THE LINEAR DEPENDENCIES

Unless there are some analytical ways of calculating the linear dependencies between the moments of the links, it is not easy to identify them. Therefore, we have programmed a numerical procedure to find these dependencies. The algorithm is based in the calculation of the rank of a matrix.

The dynamic equations can be written as :

$$\Gamma_i = D(q, \dot{q}, \ddot{q}) * (par_1 \ par_2 ... par_m)^T \tag{38}$$

where

Γ_i is the generalized force of the joint i,

par_1, ... par_m are the moments of the links, the Coulomb and viscous frictions and the inertias of the motors,

$D(q, \dot{q}, \ddot{q})$ is a function of the position, velocities and accelerations of the joints, and multiplies the parameters to identify.

For the PUMA 260, there are 10 moments for each link, 6 static frictions, 6 dampings and 6 motors inertias, making a total of 78 parameters. The function D is computed numerically using the Newton-Euler method, each time taking into account the torque due to each unit parameter, i.e. the torque due to a parameter with value equal to 1. For the motor inertias, Coulomb frictions and viscous frictions, the D parameters correspond to the acceleration, sign of the velocity and velocity of the joint respectively.

Many measurements lead to :

$$\Gamma = (D_1 D_2 ... D_m) * (par_1 par_2 ... par_m)^T \tag{39}$$

If we consider n different values of the position, velocity and acceleration of the joints, this system has a dimension equal to (6 n, m) for the matrix D. If we consider 100 points, a 6 joint manipulator has a D matrix of dimension (600, 78). The system is overdetermined, and the linear dependencies will correspond to the dependencies between the columns of the matrix D. Suppose now that the columns D_1 ... D_i are linearly independent. We add the column D_{i+1} and compute the new rank. If the rank is equal to $i + 1$ then the new submatrix D_1 ... D_{i+1} is linearly independent. If not, D_{i+1} is linearly dependent of the previous columns, and we can obtain

$$\alpha_1 * D_1 + ... + \alpha_i * D_i + \alpha_{i+1} * D_{i+1} = 0 \tag{40}$$

with α_{i+1} different from zero by construction. We have next :

$$D_{i+1} = (-\alpha_1/\alpha_{i+1}) * D_1 - ... - (\alpha_i/\alpha_{i+1}) * D_i \tag{41}$$

The new equation will be :

$$\Gamma = (D_1 ... D_m) * ((par_1 - par_{i+1} * \alpha_1/\alpha_{i+1}) ... par_m)^T \tag{42}$$

We remove the dependencies by droping the corresponding parameters, and modifying the old parameters in this last equation. The algorithm to calculate the dependencies considers a D matrix with 100 random points, and succesively computes the rank of the submatrices D_1 ... D_i dropping one parameter each time that we get a new dependency. We implemented the algorithm by computing the rank using the singular value decomposition method. Each time we obtain the smallest singular value of the order of $10 \times e^{-10}$ we consider the matrix singular, due to the numerical errors.

Using the IMSL library in UNIX and "C" language, we found the dependencies for the PUMA 260 in 15 minutes of VAX CPU time. From a starting set of 78 independent parameters we got 52 linearly independent parameters. We chose the 6 motor inertias as the six first parameters because they are constant numbers. Two parameters were dependent with the actuator inertias of links 1 and 2. The static frictions and damping are independent of the other parameters. This means that from a total of 60 moments we get 34 linearly independent moments.

The identification carried out on these new independent parameters, we ran the PUMA260 robot over 10 different trajectories.

7 EXPERIMENTS

The experiments were done by running a PUMA 260 over 6 predetermined trajectories and 4 randomly generated trajectories. These trajectories were polynomials that fitted points inside the range of each link of the robot. We preserved the continuity of the trajectory, imposing condition zero velocities at the beginning and at the end of the trajectory. The experiment was performed at McGill University by Vincent Hayward, using RCCL under UNIX environment. The curves were time scale updated to obtain maximum torque responses and to enhance the noise to signal ratio. The sampling period was 28 msec.

The collected data from the measured torque and measured position has been used to calculate the velocities and accelerations of the joints and we substituted these values into the D model of the PUMA 260. The velocitites and accelerations were calculated using the formulas $v_i = (pos_{i+1} - pos_{i-1})/56msec$, and $a_i = (v_{i+1} - v_{i-1})/56msec$, to filter the accelerations and velocities values (see appendix A). We also dropped the first 15 and last 15 samples to eliminate the effects of the transients.

We used a weighted least squares procedure since the output torque due to the first three links is between 10 to 50 times bigger than the output torque due to the last three links. This weighting is possible because the influence of the last three links over the three first is not significant. We calculated the average, standard deviation, maximum and minimum values of the parameters. Table 1 shows the results for the fitting of all 53 parameters.

parameter	representation	average	standard deviation	maximum	minimum
1 *	ia1	0.098889	0.050155	0.155265	0.07256
2 *	ia2	0.1436638	0.086779	0.243456	-0.0067
3 *	ia3	0.048211	0.054446	0.136380	-0.031769
4 *	ia4	0.003262	0.012358	0.032090	-0.010854
5 *	ia5	0.012713	0.018743	0.041647	-0.030187
6 *	ia6	0.002808	0.002849	0.009251	-0.002870
7 *	m6	2.721895	1.119838	4.311959	0.402778
8	x6	-0.0012975	0.002621	0.002830	-0.005907
9	y6	-0.000464	0.002	0.002262	-0.002939
10 *	z6	0.012975	0.009833	0.023582	-0.004141
11	a6	0.00340	0.006396	0.010470	-0.00916
12	b6	-0.001549	0.005895	0.008865	-0.0107
13	c6	0.000513	0.001691	0.003652	-0.00208
14	d6	-0.000107	0.0.001391	0.002072	-0.0030
15	e6	0.000655	0.001307	0.003036	-0.00168
16	f6	-0.000123	0.000831	0.001565	-0.0013
17	x5	0.005288	0.012684	0.035485	-0.00479
18	y5	0.0067	0.014106	0.039692	-0.01085
19	a5	-0.000094	0.008077	0.012286	-0.0172
20	c5	-0.003878	0.016054	0.019385	-0.0296
21	d5	0.000206	0.004036	0.007888	-0.00736
22	e5	0.000139	0.002689	0.005776	-0.00354
23	f5	0.002137	0.005822	0.016140	-0.00345
24	x4	0.004657	0.018356	0.046432	-0.01414
25 *	y4	-0.344338	0.229325	0.117723	-0.6942
26 *	a4	0.086604	0.092222	0.196475	-0.11892
27 *	c4	0.078255	0.081800	0.176402	-0.11721
28	d4	0.006251	0.011871	0.032354	-0.01660
29	e4	-0.000380	0.004918	0.008801	-0.0053
30	f4	0.000146	0.001974	0.003355	-0.00283

Table 1: Statistics for the obtained parameters

parameter	representation	average	standard deviation	maximum	minimum
31	x3	0.015860	0.077495	0.242066	-0.05096
32	y3	0.068128	0.182303	0.254483	-0.33270
33	a3	-0.055563	0.09324	0.037403	-0.3020
34	d3	0.004356	0.020517	0.031289	-0.04308
35	e3	0.004208	0.027816	0.050051	-0.03657
36	f3	0.005254	0.021493	0.031724	-0.03638
37	x2	0.014250	0.235012	0.512454	-0.32158
38	y2	-0.007559	0.040522	0.075899	-0.0645
39	d2	-0.017301	0.049358	0.066779	-0.0837
40	e2	-0.015461	0.026013	0.025235	-0.0641
41 *	fs1	0.690330	0.074645	0.776301	0.565639
42 *	fs2	1.379400	0.229495	1.640865	0.929609
43 *	fs3	0.600034	0.127999	0.831090	0.382789
44 *	fs4	0.221244	0.038097	0.288517	0.162633
45 *	fs5	0.039534	0.037993	0.099793	-0.03222
46 *	fs6	0.097322	0.027456	0.135677	0.062352
47 *	ds1	0.640149	0.142105	0.957472	0.496185
48 *	ds2	1.054428	0.292792	1.639596	0.646664
49 *	ds3	0.419111	0.177024	0.653683	0.065445
50 *	ds4	0.087686	0.002107	0.045901	0.146658
51 *	ds5	0.099686	0.070966	0.281025	-0.00686
52 *	ds6	0.033299	0.012595	0.053150	0.019065

Table 1: Statistics for the obtained parameters

In this table, the first 6 parameters correspond to the actuator inertias. The last 12 parameters correspond to Coulomb frictions (i.e. fsi) and viscous frictions (i.e. dsi). The rest of the parameters corresponds to moments of the links of the robot. The first column in Table 1 corresponds to the parameter number in the identification. The parameters considered as significant are marked by a "*" . The second column contains the representation of each parameter, the third through 6^{th} columns contain the average value, standard deviation, maximum and minimum value respectively of the parameter over a set of 10 different trajectories.

The significant parameters were calculated by looking for the maximum contribution to the torque, for each parameter. The parameters whose contribution for all trajectories were less than 1 percent of the maximum measured torque were considered as non-significant. Only 23 parameters were found significant. A new identification was made using only these 23 parameters getting a better distribution, that is a smaller standard deviation. This result can be attributed to the condition of the

D matrix. In the first case the condition number, the quotient between the smallest singular value and the biggest singular value, is 0.033/828.8. If we fit only 23 parameters, we get a condition number of 0.35/146.0, which is 60 times better. The fitting of the parameters is shown in Table2.

parameter	representation	average	standard deviation	maximum	minimum
1 *	ia1	0.091631	0.015729	0.115515	0.069620
2 *	ia2	0.136312	0.037337	0.205561	0.084086
3 *	ia3	0.030843	0.022379	0.046165	-0.032295
4 *	ia4	0.001781	0.005213	0.013764	-0.005222
5 *	ia5	0.006759	0.012284	0.028195	-0.015815
6 *	ia6	0.001262	0.002065	0.003792	-0.004269
7 *	m6	2.768114	0.139741	2.971715	2.395409
8 *	z6	0.014041	0.006980	0.022369	0.002290
9 *	y4	-0.382190	0.025796	-0.315692	-0.423311
10 *	a4	0.079781	0.022904	0.118200	0.036028
11 *	c4	0.077761	0.021450	0.118039	0.033405
12 *	fs1	0.787033	0.048752	0.861973	0.703532
13 *	fs2	1.389280	0.221132	1.711836	1.080681
14 *	fs3	0.650706	0.045556	0.751500	0.593681
15 *	fs4	0.256854	0.031520	0.301434	0.199465
16 *	fs5	0.036607	0.041190	0.097840	-0.03550
17 *	fs6	0.106594	0.020605	0.140304	0.077563
18 *	ds1	0.575662	0.057868	0.685223	0.465081
19 *	ds2	0.944670	0.199482	1.362454	0.558198
20 *	ds3	0.417502	0.103817	0.589464	0.292471
21 *	ds4	0.066791	0.033318	0.113687	0.019180
22 *	ds5	0.101721	0.030324	0.155355	0.044809
23 *	ds6	0.030363	0.011697	0.056091	0.016647

Table 2: Statistics for the significant parameters

7.1 IDENTIFICATION ERRORS

We displayed the measured and fitting torques, using the 23 parameters (i.e. the average) that resulted from the fitting of the curves. The results for the first three joints are almost perfect, as very slight differencies are found (see Appendix B). For the last three joints there are larger discrepancies, due to the facts that the measured torques are small and that the last three links are mechanically coupled. This coupling is not yet taken into account in the model.

If we compare the results of the frictions measured for the same robot by J. Lloyd (Lloyd84) (see Table 3), we can see that they almost agree completely.

parameter number	representation	value found in J. Lloyd	value found by our method
1 *	fs1	0.760	0.7870
2 *	fs2	1.620	1.3892
3 *	fs3	0.850	0.6507
4 *	fs4	0.175	0.2568
5 *	fs5	0.178	0.0366
6 *	fs6	0.140	0.1065
7 *	ds1	0.71	0.5756
8 *	ds2	0.55	0.9446
9 *	ds3	0.50	0.4175
10 *	ds4	0.05	0.066
11 *	ds5	0.030	0.101
12 *	ds6	0.065	0.030

Table 3: Comparison of the results obtained for the frictions

We also compared the dynamic coefficients that J. Lloyd measured to calculate the gravitational coefficients. He found three parameters $cl5 = -0.192N - m$, $cl3 = -1.762N - m$ and $cl2 = 5.509N - m$. Developping our formulas (see Appendix A) we found that the coefficients have the following values : $cl5 = -0.13743N - m$, $cl3 = -1.768644N - m$ and $cl2 = 5.51792$ N-m.

Using these parameters we automatically generated the program that calculates the inertias and gravitational coefficients as well as the program that calculates the velocity torques (see Appendix A). These programs permit the calculation of the inertias and gravitational coefficients, for the PUMA260 without load, using 98 multiplications and 70 additions; and the calcultion of the velocity torques using 140 multiplications and 110 additions. In the case of an arbitrary load at the end effector, the calculation of the inertias and gravitational coefficients require 190 multiplications and 150 additions and the velocity torques 200 multiplications and 170 additions.

7.2 ERRORS IN TRAJECTORY FOLLOWING

To test the validity of our dynamic model we have computed the necessary torques to follow a predetermined trajectory. We have thus computed the trajectory by controlling the robot PUMA 260 in open-loop, using the computed torque, and obtaining the error with the real trajectory. We have experimented with several trajectories all of them giving very good results, at low and high speed, as well as we can totally compensate for the gravitational torque, freeing the robot.

In the experiments for the joints 1, 2 and 3 independently as well as for the 3 joints together, we realized that the errors are smaller at high speed than at low speed due to the effect that it is very difficult to modelize the static error when the robot is at rest. In the case that the motion is fast, (about 180 degrees/second), the error in a trajectory of 90 degrees is smaller than 10 percent (Figure 13).

We also realized that the gravitational force builds up errors as the robot is moving, being the errors of big magnitude in the case of slow motion. To correct this problem, we have compensated for the gravitational torque in function of the joint's position of the robot. We can see in the figure 15 of the appendix, that the errores are reduced considerably.

Finally in figure 14 and 16 we can realize that the model works well when three different joints move together, proving that the coupling inertias as well as the centrifugal and Coriolis terms are compensated.

8 CONCLUSION

The method developed in this paper allows for the real time computation of the dynamic equation of robot manipulators. This has been achieved by dividing the computation into background and synchronous task. Additionally, in the background computation we could divide the inertial gravitional part from the velocity torques, computing them in parallel. The reduction of parameters from 78 to 52 independent parameters, and the reduction from 52 to 23 significant parameters make this even more efficient. Effectively the computational of the inertial and gravitational terms takes 100 multiplications and 70 additions, and the computation of the velocity torques takes a similar computation effort. In the case that the manipulator has an arbitrary load at the end effector, the computation for the inertial and gravitational terms is 200 multiplications and 150 additions approximatively, the same number of operations is needed for the velocity torques.

We have also experimented measuring the errors by controlling the robot in open, fitting the robot controller with the calculated torques to follow predetermined trajectories. The results are very good for slow and fast motion, proving the validity of the model.

Several points could be taken into account in order to improve the results. First, we considered the friction constant, which is not true in a real manipulator. This might explain most of the discrepancies of our identification results. However, the range of variation is small, making the approximation reasonable, obtaining at the same time a very good fitting.

As a conclusion we feel that the identification procedure explained in this paper is very robust and applicable to real time control, with very small errors between the measured and calculated torque. Also, the computational structure should be adaptable to new robot manipulator design, as the research is focussed in obtaining manipulators having very small coupled inertias.

References

(Bejcsy74) A. K. Bejczy
Robot Arm Dynamic and Control
NASA Tech. Memo. J.P.L. 15 Fev (1974) 33-699.

(Craig86) Craig, J.
Introduction to robotics mechanics and control.
Addison-Wesley 1986, California.

(Featherstone84)R. Featherstone
Robot Dynamics algorithms
Ph D. Thesis, University of Edinburg 1984

(Ferreira84)E. Ferreira
Contribution a l'identification de parametres et a la commande dynamique adaptative de robots manipulateurs
Doct. Engenieur Thesis, Toulouse France, 1984

(Hollerbach80) J. Hollerbach
A recursive Lagrangian formulation and a comparative study of dynamics formulations.
IEEE Trans. on System Man and Cyber. vol SMC-10, n 11, pp 730-736

(Hollerbach82)J. Hollerbach
Dynamics
Robot Motion, Planning and Control, MIT Press 1982, chapter2

(Hollerbach85)Chae H. An, Christopher G. Akteson, J. Hollerbach
Estimation of inertial parameters of rigid body links of manipulators.
Proc. of the 24th Conf. on Dec. and Control, Fort Lauredale, Dec 1985, pp 990-1002.

(Izaguirre 85) A. Izaguirre, R. P. Paul
Computation of the inertia and gravitational coefficients of the dynamic equations for a robot manipulator with a load
IEEE Conference on Robotics and automation March 1985

(Izaguirre 86) A. Izaguirre, R. P. Paul
Automatic generation of the dynamic equations of the robot manipulators using a LISP program
IEEE Conference on Robotics and automation March 1985

(Khan71)M.E. Khan, B. Roth
The near minimum time control of open loop articulated kinematic chains.

Trans. of ASME. Journal of Dyn. Systems Eng. and Control, pp 164-172

(Khalil 86) W. Khalil, J.K. Fleifinger, M. Gautier
Reducing the Computational burden of the dynamic model robots
IEEE Conference on Robotics and automation March 1986

(Khatib86) B. Armstrong, O. Khatib, J. Burdick
The explicit dynamic model and inertial parameters of the PUMA560 Arm
Proc. Conf. IEEE Robotics and Automation, San-Francisco, pp 510-518

(Kircanski 86)N. Kircanski, M. Kircanski, M. Vukobratovic, O. Timcenko
An approach to development of real time robot models
IFToMM Symp. ROMANCY, KRAKOW 1986.

(Khosla86)P. Khosla
An algorithm to determine the identifiable parameters in dynamic robot models
Proc. Conf. IEEE Robotics and Automation, San-Francisco, (not-in-time)

(Lathrop85)R. Lathrop
Parallelism in manipulator dynamics
Proc. Conf. IEEE Robotics and Automation, Saint-Louis Missouri, pp 772-778

(Likins71)P. Likins
Passive and Semi-active attitude stabilizations-flexible spacecraft
ARGARD-LS pp 45-71, October 1971

(Lloyd 86) J. E. Lloyd
Implementation of a robot programming environment
Master Thesis, Mc Gill ,University 1986, Dept. of Elect. Eng.

(Luh80) J. Luh, M. Walker, R. Paul
On-line computational scheme for mechanical manipulators IEEE TRans. Automatic Control 25, 3 1980, pp 468-474

(Luh82) J. Luh, C. Lin
Schedulling of parallel computation for a computer-controlled mechanical manipulator
IEEE Trans. on System, Man and Cybernetics. vol SMC-12, n-2 1982,pp 214-234

(Megahed84)S.M. Megahed
Contribution à la modelisation geometrique et dynamique des robots manipulateurs à structure de chaine cinematique simple ou complexe
Thèse d' état, Université Paul Sabatier, Toulouse (1984).

(Orin85) D. Orin, H. Chao, K. Olson, W. Schrader
Pipeline/parallel algorithms for the Jacobian and inverse dynamics computations
Proc. Conf. IEEE Robotics and Automation, Saint-Louis Missouri, pp785-789

(Oslen86) H. Oslen, G. Bekey
Identification of Robot Dynamics
Proc. Conf. IEEE Robotics and Automation, San-Francisco, pp 1004-1010

(Paul 82)R. Paul
Modelling, trajectory calculation and servoing of a computer controlled arm
AIM 77, Nov 1972, Stanford University

(Paul 81)R. Paul
Robot manipulators : Mathematics, programming, and control
M.I.T. Press, Cambridge, Massachussetts, and London, England, (1981).

(Raibert77)M. Raibert
Analytical equatons vs. look-up table for manipulation : a unifying concept
Proc. IEEE Conf. on Decision and Control, New Orleans, LA. Dec 1977

(Renaud83)M. Renaud
An efficient iterative analytical procedure for obtaining a robot manipulator dynamic model
First International Symposium of robotic research, Brextton Woods, U.S.A., August (1983).

(Renaud 85) M. Renaud
An efficient iterative Analytical procedure for obtaining a robot manipulator model
Robotics Research 1984, pp. 749-764

(Uicker68)J. Uicker
Dynamic behaviour of spatial linkages
Trans. of ASME n 68, Mech 5, pp1-15.

A Automatic generator of dynamic equations using the LISP machine

To use the automatic generator in the LISP machine, one has to load the packages "LAG" and "VEL" respectively. The actual packages are implemented on a Symbolic Lisp machine using version 6.0. The command to load the packages are "(make-system 'inertia)" and "(make-system 'velocity)" respectively.

The package "LAG" permits the creation of a "C" program that calculates the inertia and gravitational coefficients of the dynamic equations. To do this, we have to type the LISP command "(LAG:principa <input-file> <output-file>)", where <input-file> corresponds to the input file containing the specifications of the robot (number of links, Denavith-Hartemberg parameters, masses, first-moments, inertias of the links,...) and <output-file> corresponds to the output file containing a "C" program.

The package "VEL" permits the creation of a "C" program that computes the torques due to the joint velocities. It includes the contribution due to the frictions. To do this, one has to type the LISP command : "(VEL:principa <input-file> <output-file>)", where <input-file> corresponds to the file containing the specifications of the robot, as explained before, and <output-file> contains the "C" program to calculate the velocity torques.

The source code for the package "LAG" and "VEL" are in the directories "upenn:usr:[alberto.lagrange]" and "upenn:usr:[alberto.velocity]" respectively.

A.1 Examples of input files for the PUMA260 robot

In the following lines we show the input file for the generation of the "C" program that calculates the inertial and gravitational coefficients of the dynamic equations, for the PUMA 260 without any load at the end effector. The input file is the following :

```
number_links 6
mass 0 0 0 0 0 m6
sigma 0 0 0 0 0 0
alpha 90 0 -90 90 -90 0
dpar   0 0 d3 d4 0 0
apar   0 a2 0 0 0 0
adyna 0 0 0 ad4 0 0
bdyna 0 0 0 0 0 0
cdyna 0 0 0 cd4 0 0
ddyna 0 0 0 0 0 0
edyna 0 0 0 0 0 0
fdyna 0 0 0 0 0 0
```

```
xgrav 0 0 0 0 0 0
ygrav 0 0 0 y4 0 0
zgrav 0 0 0 0 0 z6
ia ia1 ai2 ia3 ia4 ia5 ia6
option moment
option_update moment
variable m6 2.768114
variable z6 0.01401
variable y4 -0.382190
variable ad4 0.079781
variable cd4 0.077761
variable ia1 0.091631
variable ia2 0.136312
variable ia3 0.030843
variable ia4 0.001781
variable ia5 0.006759
variable ia6 0.001262
variablea a2 0.20320
variablea d3 0.12624
variablea d4 0.20320
stop
```

The file is almost self-explanatory. The first line contains the number of links of the manipulator. The second line contains the masses of the links. The third line contains the types of the joints, i.e. revolute or prismatic. In the case of the PUMA 260 all joints are revolute, so the values of the sigma parameters are zero. If the joints are prismatic the value of sigma is 1.

The next three lines correspond to the α, d and a, Denavit-Hartemberg parameters following the notation of R. Paul (Paul81). The next six lines correspond to the values of the inertia parameters of the links expressed on the corresponding link frame. The parameters $adyna$, $bdyna$ and $cdyna$ correspond to the diagonal terms of the inertia matrices in the x, y and z directions respectively. The parameters $ddyna$, $edyna$ and $fdyna$ correspond to the inertias in the $x * y$, $y * z$ and $x * z$ directions. The inertia matrix can thus be expressed in the following expression:

$$\begin{pmatrix} adyna & ddyna & fdyna \\ ddyna & bdyna & edyna \\ fdyna & edyna & cdyna \end{pmatrix}$$

The next three lines correspond to the center of gravities of the links or the first moments, i.e. the center of gravities multiplied by the masses of the link, depending on the value of option. In our case option is set to the value "moment" indicating that the values correspond to the first moment.

The next line correspond to "proper inertia" of the motors expressed on the link frame.
In the next two lines, if option is set to the value "update" the values of the inertia terms correspond to the inertia of the link in the origin of the link frame, and the values of the parameters $xgrav$, $ygrav$ and $zgrav$ correspond to the first moment of the link in the origin of the link frame. On the other hand, if the value of option is different from "update" the inertia parameters correspond to the inertia of the link expressed in a frame parallel to the link frame, placed on the center of gravity of the link. The parameters $xgrav$, $ygrav$ and $zgrav$, in this last case, correspond to the coordinates of the center of gravity of the link expressed on the link frame. The variable "option_update" is similar to "option" but it corresponds to the parameters that are variable in the last frame.

Finally, the rest of the file contain lines with the numerical values of the physical parameters. The last line contains always the word "stop" indicating the end of file.

The next file contains the information necessary to generate the inertial and gravitational coefficients of the dynamic equations for the case of the PUMA260 with an arbitrary load at the effector. The input file is the following:

```
number_links 6
mass 0 0 0 0 0 m6
sigma 0 0 0 0 0 0
alpha 90 0 -90 90 -90 0
dpar   0 0 d3 d4 0 0
apar   0 a2 0 0 0 0
adyna 0 0 0 ad4 0 ad6
bdyna 0 0 0 0 0 bd6
cdyna 0 0 0 cd4 0 cd6
ddyna 0 0 0 0 0 dd6
edyna 0 0 0 0 0 ed6
fdyna 0 0 0 0 0 fd6
xgrav 0 0 0 0 0 x6
ygrav 0 0 0 y4 0 y6
zgrav 0 0 0 0 0 z6
ia ia1 ai2 ia3 ia4 ia5 ia6
option moment
option_update moment
variablediff m6 2.768114
variablediff z6 0.01401
variablediff x6 0.0
variablediff y6 0.0
variablediff ad6 0.0
variablediff bd6 0.0
variablediff cd6 0.0
variablediff dd6 0.0
```

```
variablediff ed6 0.0
variablediff fd6 0.0
variable y4 -0.382190
variable ad4 0.079781
variable cd4 0.077761
variable ia1 0.091631
variable ia2 0.136312
variable ia3 0.030843
variable ia4 0.001781
variable ia5 0.006759
variable ia6 0.001252
variablea a2 0.20320
variablea d3 0.12624
variablea d4 0.20320
stop
```

The difference between this file and the previous one is in the parameters corresponding to the last link, i.e. the 6^{th} link. As this link may vary its parameters, i.e. masses, inertias and first moments may change due to the addition of the extra load, the numerical values of the extra link contain the word "variablediff" instead of the word "variable". The resulting program contains a procedure that permits to update the values of these parameters when new values of the extra load are identified.

The file for the generation of the velocity torques for the PUMA 260 without load contains the following information:

```
number_links 6
mass 0 0 0 0 0 m6
sigma 0 0 0 0 0 0
alpha 90 0 -90 90 -90 0
dpar   0 0 d3 d4 0 0
apar   0 a2 0 0 0 0
adyna 0 0 0 ad4 0 0
bdyna 0 0 0 0 0 0
cdyna 0 0 0 cd4 0 0
ddyna 0 0 0 0 0 0
edyna 0 0 0 0 0 0
fdyna 0 0 0 0 0 0
xgrav 0 0 0 0 0 0
ygrav 0 0 0 y4 0 0
zgrav 0 0 0 0 0 z6
ia ia1 ai2 ia3 ia4 ia5 ia6
friction rs1 rs2 rs3 rs4 rs5 rs6
```

```
damping rd1 rd2 rd3 rd4 rd5 rd6
option moment
option_update moment
variable m6 2.768114
variable z6 0.01401
variable y4 -0.382190
variable ad4 0.079781
variable cd4 0.077761
variable ia1 0.091631
variable ia2 0.136312
variable ia3 0.030843
variable ia4 0.001781
variable ia5 0.006759
variable ia6 0.001262
variable rs1 0.787033
variable rs2 1.389280
variable rs3 0.650706
variable rs4 0.256854
variable rs5 0.036607
variable rs6 0.106594
variable rd1 0.575662
variable rd2 0.944670
variable rd3 0.417502
variable rd4 0.066791
variable rd5 0.101721
variable rd6 0.030363
variablea a2 0.20320
variablea d3 0.12624
variablea d4 0.20320
stop
```

The only difference with the first file is in the lines that start with the words "friction" and "damping" respectively. The first line contains the information corresponding to the static friction of the motors expressed on the link frame. The second corresponds to the damping, i.e. viscosous friction, of the motor expressed on the link frame. The corresponding lines with numerical values are added to the file. It is to note, that this file may also be used to generate the "C" program to calculate the inertia and gravitational coefficients.

To generate the "C" program that calculates the velocity torques for the PUMA 260 with an arbitrary load, we used the following file:

```
number_links 6
```

```
mass 0 0 0 0 0 m6
sigma 0 0 0 0 0 0
alpha 90 0 -90 90 -90 0
dpar   0 0 d3 d4 0 0
apar   0 a2 0 0 0 0
adyna 0 0 0 ad4 0 ad6
bdyna 0 0 0 0 0 bd6
cdyna 0 0 0 cd4 0 cd6
ddyna 0 0 0 0 0 dd6
edyna 0 0 0 0 0 ed6
fdyna 0 0 0 0 0 fd6
xgrav 0 0 0 0 0 x6
ygrav 0 0 0 y4 0 y6
zgrav 0 0 0 0 0 z6
ia ia1 ai2 ia3 ia4 ia5 ia6
friction rs1 rs2 rs3 rs4 rs5 rs6
damping rd1 rd2 rd3 rd4 rd5 rd6
option moment
option_update moment
variablediff m6 2.768114
variablediff z6 0.01401
variablediff x6 0.0
variablediff y6 0.0
variablediff ad6 0.0
variablediff bd6 0.0
variablediff cd6 0.0
variablediff dd6 0.0
variablediff ed6 0.0
variablediff fd6 0.0
variable y4 -0.382190
variable ad4 0.079781
variable cd4 0.077761
variable ia1 0.091631
variable ia2 0.136312
variable ia3 0.030843
variable ia4 0.001781
variable ia5 0.006759
variable ia6 0.001262
variable rs1 0.787033
variable rs2 1.389280
variable rs3 0.650706
variable rs4 0.256854
variable rs5 0.036607
```

```
variable rs6 0.106594
variable rd1 0.575662
variable rd2 0.944670
variable rd3 0.417502
variable rd4 0.066791
variable rd5 0.101721
variable rd6 0.030363
variablea a2 0.20320
variablea d3 0.12624
variablea d4 0.20320
stop
```

The only differences with the above file are the changing parameters corresponding to the last link frame. These files are on the directory "upenn:usr:[alberto.gene]".

A.2 Ouput files

To generate the "C" program to calculate the inertias and gravitational coefficients for the case of the PUMA 260 without load, we used the LISP machine command "(LAG:principa "[alberto.gene]input260.lisp" [alberto.lagrange]output260.lisp")". The listing of the file "upenn:usr[alberto.lagrange]output260.lisp" is the following :

```
#define M6  2.768114
#define Z6  0.01401
#define Y4  -0.38219
#define AD4  0.079781
#define CD4  0.077761
#define IA1  0.091631
#define IA2  0.136312
#define IA3  0.030843
#define IA4  0.001781
#define IA5  0.006759
#define IA6  0.001262
#define MC6 M6
#define MC5 MC6
#define MC4 MC5
#define MC3 MC4
#define MC2 MC3
#define MC1 MC2
#define A2 0.2032
#define D3 0.12624
```

```
#define D4 0.2032
#define KP21 A2
#define KP32 (- D3)
#define KP42 D4
#define FP42 KP42 * MC4
#define FP32 KP32 * MC3
#define FP21 KP21 * MC2
#define NP411 MC4 * (KP42*KP42)
#define NP433 MC4 * (KP42*KP42)
#define NP311 MC3 * (KP32*KP32)
#define NP333 MC3 * (KP32*KP32)
#define NP222 MC2 * (KP21*KP21)
#define NP233 MC2 * (KP21*KP21)


#include <math.h>
dyn_robot(Q,DIJ,DI)
float Q[7],DIJ[7][7],DI[7];
{

float GRAV=9.81;
float S1,S2,S3,S4,S5,S6;
float C1,C2,C3,C4,C5,C6;
float C23,S23;
float C4C4,S4S4,S4C4,C3C3,S3S3,S3C3,C2C2,S2C2;
float S2S2,C1C1,S1C1,S1S1;
float T1214,T1224,T1411,T1414,T1424,T1421;
float BS21,BS23,BS22,BA31,BS33,BS31,BS32,BA42;
float BS43,BS42,BS41,BA52,BS52,BS51,BS63;
float JS222,JS223,JS233,JA322,JA333,JS311,JS312,JS322;
float JS313,JS323,JS333,JA411,JA433,JS411,JS412,JS413;
float JS422,JS423,JA511,JA533;
float PS031,PS041,PS042,PS131,PS141,PS142,PS241;
S1 = sin(Q[1]);
C1 = cos(Q[1]);
S2 = sin(Q[2]);
C2 = cos(Q[2]);
S3 = sin(Q[3]);
C3 = cos(Q[3]);
S4 = sin(Q[4]);
C4 = cos(Q[4]);
S5 = sin(Q[5]);
C5 = cos(Q[5]);
```

```
S6 = sin(Q[6]);
C6 = cos(Q[6]);
C23 = cos(Q[2]+Q[3]);
S23 = sin(Q[2]+Q[3]);
C4C4 = C4 * C4;
S4S4 = S4 * S4;
S4C4 = S4 * C4;
C3C3 = C3 * C3;
S3S3 = S3 * S3;
S3C3 = S3 * C3;
C2C2 = C2 * C2;
S2C2 = S2 * C2;
S2S2 = S2 * S2;
C1C1 = C1 * C1;
S1C1 = S1 * C1;
S1S1 = S1 * S1;

T1214 = (C2 * A2);
T1224 = (S2 * A2);
T1411 = ((C23 * C4));
T1414 = (- (S23 * D4) + T1214);
T1424 = ((C23 * D4) + T1224);
T1421 = ((S23 * C4));

BS63 = (Z6);
BS51 = (- (S5 * BS63));
BS52 = ((C5 * BS63));
BA52 = ((Y4 + BS52) + FP42);
BS41 = ((C4 * BS51));
BS42 = ((S4 * BS51));
BS43 = (BA52);
BA42 = (BS42 + FP32);
BS32 = ((S3 * BS41) + (C3 * BS43));
BS31 = ((C3 * BS41) - (S3 * BS43));
BS33 = (- BA42);
BA31 = (BS31 + FP21);
BS22 = ((S2 * BA31) + (C2 * BS32));
BS23 = (BS33);
BS21 = ((C2 * BA31) - (S2 * BS32));

JA533 = (CD4 - NP433);
JA511 = (AD4 - NP411);
JS423 = (0 - (D4 * BS42));
```

```
JS422 = ((S4S4 * JA511) + (C4C4 * JA533) + (2.0 * (D4 * BS43)));
JS413 = (0 - (D4 * BS41));
JS412 = ((S4C4 * (JA511 - JA533)));
JS411 = ((C4C4 * JA511) + (S4S4 * JA533) + (2.0 * (D4 * BS43)));
JA433 = (- NP333);
JA411 = (JS411 - NP311);
JS333 = (JS422);
JS323 = (- (S3 * JS412) - (D3 * BS32));
JS313 = (- (C3 * JS412) - (D3 * BS31));
JS322 = ((S3S3 * JA411) + (C3C3 * JA433) + (2.0 * (D3 * BS33)));
JS312 = ((S3C3 * (JA411 - JA433)));
JS311 = ((C3C3 * JA411) + (S3S3 * JA433) + (2.0 * (D3 * BS33)));
JA333 = (JS333 - NP233);
JA322 = (JS322 - NP222);
JS233 = (JA333 + (2.0 * ((T1224 * BS22) + (T1214 * BS21))));
JS223 = ((S2 * JS313) + (C2 * JS323) - (T1224 * BS23));
JS222 = ((S2S2 * JS311) + ((2 * S2C2) * JS312) + (C2C2 * JA322) + (2.0
* (T1214 * BS21)));

PS031 = ((C23 * D3));
PS041 = ((T1411 * D3) + (S4 * T1414));
PS042 = (- (S23 * D3));
PS131 = (- (C23 * T1224) + (S23 * T1214));
PS141 = (- (T1411 * T1424) + (T1421 * T1414));
PS142 = ((S23 * T1424) + (C23 * T1414));
PS241 = (- (C4 * D4));

DIJ[1][1] = ((JS222) + IA1);
DIJ[1][2] = (JS223);
DIJ[1][3] = ((S2 * JS313) + (C2 * JS323));
DIJ[1][4] = ((S23 * JS413) - (BS42 * PS031) + (BS41 * T1214));
DIJ[1][5] = (- (BS52 * PS041) + (BS51 * PS042));
DIJ[1][6] = (0);
DIJ[2][2] = ((JS233) + IA2);
DIJ[2][3] = (JS422 + (BS31 * A2));
DIJ[2][4] = (- JS423 - (BS42 * PS131));
DIJ[2][5] = (- (BS52 * PS141) + (BS51 * PS142));
DIJ[2][6] = (0);
DIJ[3][3] = ((JS422) + IA3);
DIJ[3][4] = (- JS423);
DIJ[3][5] = (- (BS52 * PS241));
DIJ[3][6] = (0);
DIJ[4][4] = IA4;
```

```
DIJ[4][5] = (0);
DIJ[4][6] = (0);
DIJ[5][5] = IA5;
DIJ[5][6] = (0);
DIJ[6][6] = IA6;
DI[1] = (0);
DI[2] = (GRAV * (BS21));
DI[3] = (GRAV * (- (BS32 * S2) + (BS31 * C2)));
DI[4] = (GRAV * (- (BS42 * S23)));
DI[5] = (GRAV * (- (BS52 * T1421) + (BS51 * C23)));
DI[6] = 0;
}
```

The routine "dyn_robot(Q,DIJ,DI)" calculates the inertia coefficients in the two dimensional array DIJ and the gravitational coefficients in the one dimensional array DI. The values in DIJ are expressed in $N - m/rad^2$, and those in DI are $N - m$. The one dimensional array Q contains the angles in rads, of the six joints.

To generate the "C" program that calculates the inertias and gravitational coefficients for the case of the PUMA 260 with arbitrary load, we used the LISP machine command "(LAG:principa "[alberto.gene]input260all.lisp" [alberto.lagrange]output260all.lisp")". The listing of the file "upenn:usr[alberto.lagrange]output260all.lisp" is the following :

```
#define Y4  -0.38219
#define AD4  0.079781
#define CD4  0.077761
#define IA1  0.091631
#define IA2  0.136312
#define IA3  0.030843
#define IA4  0.001781
#define IA5  0.006759
#define IA6  0.001262
#define A2 0.2032
#define D3 0.12624
#define D4 0.2032
#define KP21 A2
#define KP32 (- D3)
#define KP42 D4

struct var_const{
float M6;
float Z6;
```

```
float X6;
float Y6;
float AD6;
float BD6;
float CD6;
float DD6;
float ED6;
float FD6;
float MC6;
float MC5;
float MC4;
float MC3;
float MC2;
float MC1;
float FP42;
float FP32;
float FP21;
float NP411;
float NP433;
float NP311;
float NP333;
float NP222;
float NP233;
}SIX={2.768114,0.01401,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,};

update(M6,Z6,X6,Y6,AD6,BD6,CD6,DD6,ED6,FD6,SIX)
float M6,Z6,X6,Y6,AD6,BD6,CD6,DD6,ED6,FD6;
struct var_const *SIX;
{
SIX->M6 = M6;
SIX->Z6 = Z6;
SIX->X6 = X6;
SIX->Y6 = Y6;
SIX->AD6 = AD6;
SIX->BD6 = BD6;
SIX->CD6 = CD6;
SIX->DD6 = DD6;
SIX->ED6 = ED6;
SIX->FD6 = FD6;
SIX->MC6 = M6;
SIX->MC5 = SIX->MC6;
SIX->MC4 = SIX->MC5;
SIX->MC3 = SIX->MC4;
```

```
SIX->MC2 = SIX->MC3;
SIX->MC1 = SIX->MC2;
SIX->FP42 = SIX->MC4 * KP42 ;
SIX->FP32 = SIX->MC3 * KP32 ;
SIX->FP21 = SIX->MC2 * KP21 ;
SIX->NP411 = SIX->MC4 * (KP42*KP42) ;
SIX->NP433 = SIX->MC4 * (KP42*KP42) ;
SIX->NP311 = SIX->MC3 * (KP32*KP32) ;
SIX->NP333 = SIX->MC3 * (KP32*KP32) ;
SIX->NP222 = SIX->MC2 * (KP21*KP21) ;
SIX->NP233 = SIX->MC2 * (KP21*KP21) ;
}

#include <math.h>
dyn_robot(Q,DIJ,DI,SIX)
float Q[7],DIJ[7][7],DI[7];
struct var_const *SIX;
{

float GRAV=9.81;
float S1,S2,S3,S4,S5,S6;
float C1,C2,C3,C4,C5,C6;
float C23,S23;
float C6C6,S6C6,S6S6,C5C5,S5C5,S5S5,C4C4,S4C4;
float S4S4,C3C3,S3C3,S3S3,C2C2,S2C2,S2S2,C1C1;
float S1C1,S1S1;
float T3511,T3513,T3521,T3523,T1214,T1224,T1421,T1423;
float T1411,T1414,T1424,T1521,T1522,T1523,T1511,T1514;
float T1512,T1524;
float BS21,BS23,BS22,BA31,BS33,BS31,BS32,BA42;
float BS43,BS41,BS42,BA52,BS52,BS51,BS53,BS61;
float BS63,BS62;
float JS222,JS223,JS233,JA322,JA333,JS311,JS312,JS322;
float JS313,JS323,JS333,JA411,JA433,JS411,JS413,JS433;
float JS412,JS423,JS422,JA511,JA533,JS511,JS513,JS533;
float JS512,JS523,JS522,JS611,JS613,JS633,JS612,JS623;
float JS622;
float PS031,PS041,PS042,PS051,PS052,PS131,PS141,PS142;
float PS151,PS152,PS241,PS251,PS252;
S1 = sin(Q[1]);
C1 = cos(Q[1]);
S2 = sin(Q[2]);
C2 = cos(Q[2]);
```

```
S3 = sin(Q[3]);
C3 = cos(Q[3]);
S4 = sin(Q[4]);
C4 = cos(Q[4]);
S5 = sin(Q[5]);
C5 = cos(Q[5]);
S6 = sin(Q[6]);
C6 = cos(Q[6]);
C23 = cos(Q[2]+Q[3]);
S23 = sin(Q[2]+Q[3]);
C6C6 = C6 * C6;
S6C6 = S6 * C6;
S6S6 = S6 * S6;
C5C5 = C5 * C5;
S5C5 = S5 * C5;
S5S5 = S5 * S5;
C4C4 = C4 * C4;
S4C4 = S4 * C4;
S4S4 = S4 * S4;
C3C3 = C3 * C3;
S3C3 = S3 * C3;
S3S3 = S3 * S3;
C2C2 = C2 * C2;
S2C2 = S2 * C2;
S2S2 = S2 * S2;
C1C1 = C1 * C1;
S1C1 = S1 * C1;
S1S1 = S1 * S1;

T3511 = ((C4 * C5));
T3513 = (- (C4 * S5));
T3521 = ((S4 * C5));
T3523 = (- (S4 * S5));
T1214 = (C2 * A2);
T1224 = (S2 * A2);
T1421 = ((S23 * C4));
T1423 = ((S23 * S4));
T1411 = ((C23 * C4));
T1414 = (- (S23 * D4) + T1214);
T1424 = ((C23 * D4) + T1224);
T1521 = ((S23 * T3511) + (C23 * S5));
T1522 = (- (S23 * S4));
T1523 = ((S23 * T3513) + (C23 * C5));
```

```
T1511 = ((C23 * T3511) - (S23 * S5));
T1514 = (- (S23 * D4) + T1214);
T1512 = (- (C23 * S4));
T1524 = ((C23 * D4) + T1224);

BS62 = ((S6 * SIX->X6) + (C6 * SIX->Y6));
BS63 = (SIX->Z6);
BS61 = ((C6 * SIX->X6) - (S6 * SIX->Y6));
BS53 = (- BS62);
BS51 = ((C5 * BS61) - (S5 * BS63));
BS52 = ((S5 * BS61) + (C5 * BS63));
BA52 = ((Y4 + BS52) + SIX->FP42);
BS42 = ((S4 * BS51) - (C4 * BS53));
BS41 = ((C4 * BS51) + (S4 * BS53));
BS43 = (BA52);
BA42 = (BS42 + SIX->FP32);
BS32 = ((S3 * BS41) + (C3 * BS43));
BS31 = ((C3 * BS41) - (S3 * BS43));
BS33 = (- BA42);
BA31 = (BS31 + SIX->FP21);
BS22 = ((S2 * BA31) + (C2 * BS32));
BS23 = (BS33);
BS21 = ((C2 * BA31) - (S2 * BS32));

JS622 = ((S6S6 * SIX->AD6)
+ ((2 * S6C6) * SIX->DD6) + (C6C6 * SIX->BD6));
JS623 = ((S6 * SIX->FD6) + (C6 * SIX->ED6));
JS612 = ((S6C6 * (SIX->AD6 - SIX->BD6)) + ((- S6S6 + C6C6) * SIX->DD6));
JS633 = (SIX->CD6);
JS613 = ((C6 * SIX->FD6) - (S6 * SIX->ED6));
JS611 = ((C6C6 * SIX->AD6) - ((2 * S6C6) * SIX->DD6) + (S6S6 * SIX->BD6));
JS522 = ((S5S5 * JS611) + ((2 * S5C5) * JS613) + (C5C5 * JS633));
JS523 = (- (S5 * JS612) - (C5 * JS623));
JS512 = ((S5C5 * (JS611 - JS633)) + ((- S5S5 + C5C5) * JS613));
JS533 = (JS622);
JS513 = (- (C5 * JS612) + (S5 * JS623));
JS511 = ((C5C5 * JS611) - ((2 * S5C5) * JS613) + (S5S5 * JS633));
JA533 = (CD4 + JS533 - SIX->NP433);
JA511 = (AD4 + JS511 - SIX->NP411);
JS422 = ((S4S4 * JA511) - ((2 * S4C4) * JS513)
+ (C4C4 * JA533) + (2.0 * (D4 * BS43)));
JS423 = ((S4 * JS512) - (C4 * JS523) - (D4 * BS42));
JS412 = ((S4C4 * (JA511 - JA533)) + ((S4S4 - C4C4) * JS513));
```

```
JS433 = (JS522);
JS413 = ((C4 * JS512) + (S4 * JS523) - (D4 * BS41));
JS411 = ((C4C4 * JA511) + ((2 * S4C4) * JS513)
+ (S4S4 * JA533) + (2.0 * (D4 * BS43)));
JA433 = (JS433 - SIX->NP333);
JA411 = (JS411 - SIX->NP311);
JS333 = (JS422);
JS323 = (- (S3 * JS412) - (C3 * JS423) - (D3 * BS32));
JS313 = (- (C3 * JS412) + (S3 * JS423) - (D3 * BS31));
JS322 = ((S3S3 * JA411) + ((2 * S3C3) * JS413) + (C3C3 * JA433) +
 (2.0 * (D3 * BS33)));
JS312 = ((S3C3 * (JA411 - JA433)) + ((- S3S3 + C3C3) * JS413));
JS311 = ((C3C3 * JA411) - ((2 * S3C3) * JS413) + (S3S3 * JA433)
+ (2.0 * (D3 * BS33)));
JA333 = (JS333 - SIX->NP233);
JA322 = (JS322 - SIX->NP222);
JS233 = (JA333 + (2.0 * ((T1224 * BS22) + (T1214 * BS21))));
JS223 = ((S2 * JS313) + (C2 * JS323) - (T1224 * BS23));
JS222 = ((S2S2 * JS311) + ((2 * S2C2) * JS312) + (C2C2 * JA322)
+ (2.0 * (T1214 * BS21)));

PS031 = ((C23 * D3));
PS041 = ((T1411 * D3) + (S4 * T1414));
PS042 = (- (S23 * D3));
PS051 = ((T1511 * D3) + (T3521 * T1514));
PS052 = ((T1512 * D3) + (C4 * T1514));
PS131 = (- (C23 * T1224) + (S23 * T1214));
PS141 = (- (T1411 * T1424) + (T1421 * T1414));
PS142 = ((S23 * T1424) + (C23 * T1414));
PS151 = (- (T1511 * T1524) + (T1521 * T1514));
PS152 = (- (T1512 * T1524) + (T1522 * T1514));
PS241 = (- (C4 * D4));
PS251 = (- (T3511 * D4));
PS252 = ((S4 * D4));

DIJ[1][1] = ((JS222) + IA1);
DIJ[1][2] = (JS223);
DIJ[1][3] = ((S2 * JS313) + (C2 * JS323));
DIJ[1][4] = ((S23 * JS413) + (C23 * JS522) - (BS42 * PS031) + (BS41 *
T1214));
DIJ[1][5] = ((T1421 * JS513) + (C23 * JS523)
+ (T1423 * JS622) - (BS52 * PS041) + (BS51 * PS042));
DIJ[1][6] = ((T1521 * JS613) + (T1522 * JS623)
```

```
+ (T1523 * SIX->CD6) - (BS62 * PS051) + (BS61 * PS052));
DIJ[2][2] = ((JS233) + IA2);
DIJ[2][3] = (JS422 + (BS31 * A2));
DIJ[2][4] = (- JS423 - (BS42 * PS131));
DIJ[2][5] = (- (S4 * JS513) + (C4 * JS622) - (BS52 * PS141) + (BS51 *
PS142));
DIJ[2][6] = (- (T3521 * JS613) - (C4 * JS623)
- (T3523 * SIX->CD6) - (BS62 * PS151) + (BS61 * PS152));
DIJ[3][3] = ((JS422) + IA3);
DIJ[3][4] = (- JS423);
DIJ[3][5] = (- (S4 * JS513) + (C4 * JS622) - (BS52 * PS241));
DIJ[3][6] = (- (T3521 * JS613) - (C4 * JS623)
- (T3523 * SIX->CD6) - (BS62 * PS251) + (BS61 * PS252));
DIJ[4][4] = ((JS522) + IA4);
DIJ[4][5] = (JS523);
DIJ[4][6] = ((S5 * JS613) + (C5 * SIX->CD6));
DIJ[5][5] = ((JS622) + IA5);
DIJ[5][6] = (- JS623);
DIJ[6][6] = ((SIX->CD6) + IA6);
DI[1] = (0);
DI[2] = (GRAV * (BS21));
DI[3] = (GRAV * (- (BS32 * S2) + (BS31 * C2)));
DI[4] = (GRAV * (- (BS42 * S23)));
DI[5] = (GRAV * (- (BS52 * T1421) + (BS51 * C23)));
DI[6] = (GRAV * (- (BS62 * T1521) + (BS61 * T1522)));
}
```

The difference in this program is the procedure "update" that permits to update the values corresponding to the last link. These canges are stored on a structure "var_const"' that is passed as a parameter to the procedure "dyn_robot".

To generate the "C" program that calculates the velocity torques for the case of the PUMA 260 without load, we used the LISP machine command "(LAG:principa "[alberto.gene]input260v.lisp" [alberto.lagrange]output260v.lisp")". The listing of the file "upenn:usr[alberto.lagrange]output260v.lisp" is the following :

```
#define M6   2.768114
#define Z6   0.01401
#define Y4   -0.38219
#define AD4   0.079781
#define CD4   0.077761
#define IA1   0.091631
```

```
#define IA2  0.136312
#define IA3  0.030843
#define IA4  0.001781
#define IA5  0.006759
#define IA6  0.001262
#define RS1  0.787033
#define RS2  1.38928
#define RS3  0.650706
#define RS4  0.256854
#define RS5  0.036607
#define RS6  0.106594
#define RD1  0.575662
#define RD2  0.94467
#define RD3  0.417502
#define RD4  0.066791
#define RD5  0.101721
#define RD6  0.030363
#define A2 0.2032
#define D3 0.12624
#define D4 0.2032
#define KP21 A2
#define KP32 (- D3)
#define KP42 D4

#include <math.h>
vel_robot(Q,QD,TORQUE)
float Q[7],QD[7],TORQUE[7];
{


float S1,S2,S3,S4,S5,S6;
float C1,C2,C3,C4,C5,C6;
float WV22,WV21,WV31,WV32,WV33,WV41,WV42,WV43;
float WV51,WV52,WV53,WV61,WV63,WV62;
float WP211,WP221,WP222,WP321,WP322,WP333,WP331,WP431;
float WP432,WP443,WP441,WP541,WP542,WP551,WP651,WP652;
float WP662,WP661;
float VP21,VP22,VP23,VP31,VP32,VP33,VP41,VP42;
float VP43,VP51,VP53,VP62,VP61;
float DV233,DV222,DV212,DV213,DV312,DV311,DV333,DV323;
float DV412,DV411,DV433,DV423,DV413,DV613,DV623,DV611;
float DV622;
float UV211,UV231,UV312,UV322,UV332,UV412,UV422,UV432;
```

```
float UV613,UV623,UV633;
float FP61,FP62,FP63,FP43,FP41,FP42;
float NP41,NP43,NP42;
float FL62,FL61,FL51,FL52,FL43,FL41,FL42,FL32;
float NL61,NL62,NL51,NL52,NL41,NL43,NL42,NL31;
float NL32,NL22;
S1 = sin(Q[1]);
C1 = cos(Q[1]);
S2 = sin(Q[2]);
C2 = cos(Q[2]);
S3 = sin(Q[3]);
C3 = cos(Q[3]);
S4 = sin(Q[4]);
C4 = cos(Q[4]);
S5 = sin(Q[5]);
C5 = cos(Q[5]);
S6 = sin(Q[6]);
C6 = cos(Q[6]);
WV22 = (C2 * QD[1]);
WV21 = (S2 * QD[1]);
WV31 = ((C3 * WV21) + (S3 * WV22));
WV32 = (- (QD[2] + QD[3]));
WV33 = ((- (S3 * WV21)) + (C3 * WV22));
WV41 = ((C4 * WV31) + (S4 * WV32));
WV42 = (WV33 + QD[4]);
WV43 = ((S4 * WV31) - (C4 * WV32));
WV51 = ((C5 * WV41) + (S5 * WV42));
WV52 = (- (WV43 + QD[5]));
WV53 = ((- (S5 * WV41)) + (C5 * WV42));
WV61 = ((C6 * WV51) + (S6 * WV52));
WV63 = (WV53 + QD[6]);
WV62 = ((- (S6 * WV51)) + (C6 * WV52));

WP211 = (QD[1] * QD[2]);
WP221 = (C2 * WP211);
WP222 = (- (S2 * WP211));
WP321 = (WP221 + (WV22 * QD[3]));
WP322 = (WP222 - (WV21 * QD[3]));
WP333 = ((- (S3 * WP321)) + (C3 * WP322));
WP331 = ((C3 * WP321) + (S3 * WP322));
WP431 = (WP331 + (WV32 * QD[4]));
WP432 = (- (WV31 * QD[4]));
WP443 = ((S4 * WP431) - (C4 * WP432));
```

```
WP441 = ((C4 * WP431) + (S4 * WP432));
WP541 = (WP441 + (WV42 * QD[5]));
WP542 = (WP333 - (WV41 * QD[5]));
WP551 = ((C5 * WP541) + (S5 * WP542));
WP651 = (WP551 + (WV52 * QD[6]));
WP652 = ((- WP443) - (WV51 * QD[6]));
WP662 = ((- (S6 * WP651)) + (C6 * WP652));
WP661 = ((C6 * WP651) + (S6 * WP652));

DV233 = (- (QD[2] * QD[2]));
DV222 = (- (WV22 * WV22));
DV212 = (WV21 * WV22);
DV213 = (WV21 * QD[2]);
DV312 = (WV31 * WV32);
DV311 = (- (WV31 * WV31));
DV333 = (- (WV33 * WV33));
DV323 = (WV32 * WV33);
DV412 = (WV41 * WV42);
DV411 = (- (WV41 * WV41));
DV433 = (- (WV43 * WV43));
DV423 = (WV42 * WV43);
DV413 = (WV41 * WV43);
DV613 = (WV61 * WV63);
DV623 = (WV62 * WV63);
DV611 = (- (WV61 * WV61));
DV622 = (- (WV62 * WV62));

UV211 = (DV233 + DV222);
UV231 = (DV213 - WP222);
UV312 = (DV312 - WP333);
UV322 = (DV311 + DV333);
UV332 = (DV323 + WP331);
UV412 = (DV412 - WP443);
UV422 = (DV411 + DV433);
UV432 = (DV423 + WP441);
UV613 = (DV613 + WP662);
UV623 = (DV623 - WP661);
UV633 = (DV611 + DV622);

VP21 = (UV211 * A2);
VP22 = (DV212 * A2);
VP23 = (UV231 * A2);
VP31 = (((C3 * VP21) + (S3 * VP22)) - (UV312 * D3));
```

```
VP32 = ((- VP23) - (UV322 * D3));
VP33 = (((- (S3 * VP21)) + (C3 * VP22)) - (UV332 * D3));
VP41 = (((C4 * VP31) + (S4 * VP32)) + (UV412 * D4));
VP42 = (VP33 + (UV422 * D4));
VP43 = (((S4 * VP31) - (C4 * VP32)) + (UV432 * D4));
VP51 = ((C5 * VP41) + (S5 * VP42));
VP53 = ((- (S5 * VP41)) + (C5 * VP42));
VP62 = ((- (S6 * VP51)) - (C6 * VP43));
VP61 = ((C6 * VP51) - (S6 * VP43));

FP61 = ((M6 * VP61) + (UV613 * Z6));
FP62 = ((M6 * VP62) + (UV623 * Z6));
FP63 = ((M6 * VP53) + (UV633 * Z6));
FP43 = (UV432 * Y4);
FP41 = (UV412 * Y4);
FP42 = (UV422 * Y4);

NP41 = ((WP441 * AD4) + (DV423 * CD4));
NP43 = ((WP443 * CD4) - (DV412 * AD4));
NP42 = (DV413 * (AD4 - CD4));

FL62 = ((S6 * FP61) + (C6 * FP62));
FL61 = ((C6 * FP61) - (S6 * FP62));
FL51 = ((C5 * FL61) - (S5 * FP63));
FL52 = ((S5 * FL61) + (C5 * FP63));
FL43 = (FL52 + FP42);
FL41 = ((C4 * (FL51 + FP41)) + (S4 * ((- FL62) + FP43)));
FL42 = ((S4 * (FL51 + FP41)) - (C4 * ((- FL62) + FP43)));
FL32 = ((S3 * FL41) + (C3 * FL43));

NL61 = ((- (C6 * (Z6 * VP62))) - (S6 * (Z6 * VP61)));
NL62 = ((- (S6 * (Z6 * VP62))) + (C6 * (Z6 * VP61)));
NL51 = (C5 * NL61);
NL52 = (S5 * NL61);
NL41 = ((C4 * (NL51 + (NP41 + ((D4 * ((- FL62) + FP43)) + (Y4 * VP43))))))
+ (S4 * ((- NL62) + (NP43 + ((- (D4 * (FL51 + FP41))) - (Y4 * VP41))))));
NL43 = (NL52 + NP42);
NL42 = ((S4 * (NL51 + (NP41 + ((D4 * ((- FL62) + FP43)) + (Y4 * VP43))))))
- (C4 * ((- NL62) + (NP43 + ((- (D4 * (FL51 + FP41))) - (Y4 * VP41))))));
NL31 = ((C3 * (NL41 - (D3 * FL43))) - (S3 * (NL43 + (D3 * FL41))));
NL32 = ((S3 * (NL41 - (D3 * FL43))) + (C3 * (NL43 + (D3 * FL41))));
NL22 = ((S2 * NL31) + (C2 * (NL32 + (A2 * FL42))));
```

```
TORQUE[1] = NL22 + RS1 * sgn(QD[1]) + RD1 * QD[1];
TORQUE[2] = ((- NL42) + (A2 * FL32)) + RS2 * sgn(QD[2]) + RD2 * QD[2];
TORQUE[3] = (- NL42) + RS3 * sgn(QD[3]) + RD3 * QD[3];
TORQUE[4] = (NL52 + NP42) + RS4 * sgn(QD[4]) + RD4 * QD[4];
TORQUE[5] = (- NL62) + RS5 * sgn(QD[5]) + RD5 * QD[5];
TORQUE[6] = 0 + RS6 * sgn(QD[6]) + RD6 * QD[6];
}
```

The procedure "vel_robot" computes the velocity torque in the array $TORQUE$ in $N-m$, and the input arrays Q and QD contain the angles and velocities of the joints in rad and rad/sec respectively.

To generate the "C" program to calculate the velocity torques for the case of the PUMA 260 with arbitrary load, we used the LISP machine command "(LAG:principa "[alberto.gene]input260allv.lisp" [alberto.lagrange]output260allv.lisp")". The listing of the file "upenn:usr[alberto.lagrange]output260allv.lisp" is the following :

```
#define Y4  -0.38219
#define AD4  0.079781
#define CD4  0.077761
#define IA1  0.091631
#define IA2  0.136312
#define IA3  0.030843
#define IA4  0.001781
#define IA5  0.006759
#define IA6  0.001262
#define RS1  0.787033
#define RS2  1.38928
#define RS3  0.650706
#define RS4  0.256854
#define RS5  0.036607
#define RS6  0.106594
#define RD1  0.575662
#define RD2  0.94467
#define RD3  0.417502
#define RD4  0.066791
#define RD5  0.101721
#define RD6  0.030363
#define A2 0.2032
#define D3 0.12624
#define D4 0.2032
#define KP21 A2
```

```
#define KP32 (- D3)
#define KP42 D4
struct var_const{
float M6;
float Z6;
float X6;
float Y6;
float AD6;
float BD6;
float CD6;
float DD6;
float ED6;
float FD6;
}SIX={2.768114,0.01401,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,};

update(M6,Z6,X6,Y6,AD6,BD6,CD6,DD6,ED6,FD6,SIX)
float M6,Z6,X6,Y6,AD6,BD6,CD6,DD6,ED6,FD6;
struct var_const *SIX;
{
SIX->M6 = M6;
SIX->Z6 = Z6;
SIX->X6 = X6;
SIX->Y6 = Y6;
SIX->AD6 = AD6;
SIX->BD6 = BD6;
SIX->CD6 = CD6;
SIX->DD6 = DD6;
SIX->ED6 = ED6;
SIX->FD6 = FD6;
}

#include <math.h>
vel_robot(Q,QD,TORQUE,SIX)
float Q[7],QD[7],TORQUE[7];
struct var_const *SIX;
{


float S1,S2,S3,S4,S5,S6;
float C1,C2,C3,C4,C5,C6;
float WV22,WV21,WV32,WV31,WV33,WV42,WV41,WV43;
float WV52,WV51,WV53,WV62,WV63,WV61;
float WP211,WP221,WP222,WP321,WP322,WP331,WP333,WP431;
```

```
float WP432,WP441,WP443,WP541,WP542,WP551,WP553,WP651;
float WP652,WP661,WP662;
float VP21,VP22,VP23,VP31,VP32,VP33,VP41,VP42;
float VP43,VP53,VP51,VP62,VP61;
float DV233,DV222,DV212,DV213,DV312,DV311,DV333,DV323;
float DV412,DV411,DV433,DV423,DV413,DV623,DV612,DV613;
float DV633,DV622,DV611;
float UV211,UV231,UV312,UV322,UV332,UV412,UV422,UV432;
float UV621,UV631,UV632,UV612,UV613,UV623,UV611,UV622;
float UV633;
float FP61,FP62,FP63,FP43,FP41,FP42;
float NP61,NP62,NP63,NP41,NP43,NP42;
float FL62,FL61,FL51,FL52,FL43,FL41,FL42,FL32;
float NL61,NL63,NL62,NL51,NL52,NL41,NL43,NL42;
float NL31,NL32,NL22;
S1 = sin(Q[1]);
C1 = cos(Q[1]);
S2 = sin(Q[2]);
C2 = cos(Q[2]);
S3 = sin(Q[3]);
C3 = cos(Q[3]);
S4 = sin(Q[4]);
C4 = cos(Q[4]);
S5 = sin(Q[5]);
C5 = cos(Q[5]);
S6 = sin(Q[6]);
C6 = cos(Q[6]);
WV22 = (C2 * QD[1]);
WV21 = (S2 * QD[1]);
WV32 = (- (QD[2] + QD[3]));
WV31 = ((C3 * WV21) + (S3 * WV22));
WV33 = ((- (S3 * WV21)) + (C3 * WV22));
WV42 = (WV33 + QD[4]);
WV41 = ((C4 * WV31) + (S4 * WV32));
WV43 = ((S4 * WV31) - (C4 * WV32));
WV52 = (- (WV43 + QD[5]));
WV51 = ((C5 * WV41) + (S5 * WV42));
WV53 = ((- (S5 * WV41)) + (C5 * WV42));
WV62 = ((- (S6 * WV51)) + (C6 * WV52));
WV63 = (WV53 + QD[6]);
WV61 = ((C6 * WV51) + (S6 * WV52));

WP211 = (QD[1] * QD[2]);
```

```
WP221 = (C2 * WP211);
WP222 = (- (S2 * WP211));
WP321 = (WP221 + (WV22 * QD[3]));
WP322 = (WP222 - (WV21 * QD[3]));
WP331 = ((C3 * WP321) + (S3 * WP322));
WP333 = ((- (S3 * WP321)) + (C3 * WP322));
WP431 = (WP331 + (WV32 * QD[4]));
WP432 = (- (WV31 * QD[4]));
WP441 = ((C4 * WP431) + (S4 * WP432));
WP443 = ((S4 * WP431) - (C4 * WP432));
WP541 = (WP441 + (WV42 * QD[5]));
WP542 = (WP333 - (WV41 * QD[5]));
WP551 = ((C5 * WP541) + (S5 * WP542));
WP553 = ((- (S5 * WP541)) + (C5 * WP542));
WP651 = (WP551 + (WV52 * QD[6]));
WP652 = ((- WP443) - (WV51 * QD[6]));
WP661 = ((C6 * WP651) + (S6 * WP652));
WP662 = ((- (S6 * WP651)) + (C6 * WP652));

DV233 = (- (QD[2] * QD[2]));
DV222 = (- (WV22 * WV22));
DV212 = (WV21 * WV22);
DV213 = (WV21 * QD[2]);
DV312 = (WV31 * WV32);
DV311 = (- (WV31 * WV31));
DV333 = (- (WV33 * WV33));
DV323 = (WV32 * WV33);
DV412 = (WV41 * WV42);
DV411 = (- (WV41 * WV41));
DV433 = (- (WV43 * WV43));
DV423 = (WV42 * WV43);
DV413 = (WV41 * WV43);
DV623 = (WV62 * WV63);
DV612 = (WV61 * WV62);
DV613 = (WV61 * WV63);
DV633 = (- (WV63 * WV63));
DV622 = (- (WV62 * WV62));
DV611 = (- (WV61 * WV61));

UV211 = (DV233 + DV222);
UV231 = (DV213 - WP222);
UV312 = (DV312 - WP333);
UV322 = (DV311 + DV333);
```

```
UV332 = (DV323 + WP331);
UV412 = (DV412 - WP443);
UV422 = (DV411 + DV433);
UV432 = (DV423 + WP441);
UV621 = (DV612 + WP553);
UV631 = (DV613 - WP662);
UV632 = (DV623 + WP661);
UV612 = (DV612 - WP553);
UV613 = (DV613 + WP662);
UV623 = (DV623 - WP661);
UV611 = (DV633 + DV622);
UV622 = (DV611 + DV633);
UV633 = (DV611 + DV622);

VP21 = (UV211 * A2);
VP22 = (DV212 * A2);
VP23 = (UV231 * A2);
VP31 = (((C3 * VP21) + (S3 * VP22)) - (UV312 * D3));
VP32 = ((- VP23) - (UV322 * D3));
VP33 = (((- (S3 * VP21)) + (C3 * VP22)) - (UV332 * D3));
VP41 = (((C4 * VP31) + (S4 * VP32)) + (UV412 * D4));
VP42 = (VP33 + (UV422 * D4));
VP43 = (((S4 * VP31) - (C4 * VP32)) + (UV432 * D4));
VP53 = ((- (S5 * VP41)) + (C5 * VP42));
VP51 = ((C5 * VP41) + (S5 * VP42));
VP62 = ((- (S6 * VP51)) - (C6 * VP43));
VP61 = ((C6 * VP51) - (S6 * VP43));

FP61 = ((SIX->M6 * VP61) + (((UV611 * SIX->X6)
+ (UV612 * SIX->Y6)) + (UV613 * SIX->Z6)));
FP62 = ((SIX->M6 * VP62) + (((UV621 * SIX->X6)
+ (UV622 * SIX->Y6)) + (UV623 * SIX->Z6)));
FP63 = ((SIX->M6 * VP53) + (((UV631 * SIX->X6)
 + (UV632 * SIX->Y6)) + (UV633 * SIX->Z6)));
FP43 = (UV432 * Y4);
FP41 = (UV412 * Y4);
FP42 = (UV422 * Y4);

NP61 = (((((WP661 * SIX->AD6) + (DV623 * (SIX->CD6 - SIX->BD6)))
+ (UV621 * SIX->FD6)) - (UV631 * SIX->DD6)) + ((DV633 - DV622) * SIX->ED6));
NP62 = (((((WP662 * SIX->BD6) + (DV613 * (SIX->AD6 - SIX->CD6)))
+ (UV632 * SIX->DD6)) - (UV612 * SIX->ED6)) + ((DV611 - DV633) * SIX->FD6));
NP63 = (((((WP553 * SIX->CD6) + (DV612 * (SIX->BD6 - SIX->AD6)))
```

```
+ (UV613 * SIX->ED6)) - (UV623 * SIX->FD6)) + ((DV622 - DV611) * SIX->DD6));
NP41 = ((WP441 * AD4) + (DV423 * CD4));
NP43 = ((WP443 * CD4) - (DV412 * AD4));
NP42 = (DV413 * (AD4 - CD4));

FL62 = ((S6 * FP61) + (C6 * FP62));
FL61 = ((C6 * FP61) - (S6 * FP62));
FL51 = ((C5 * FL61) - (S5 * FP63));
FL52 = ((S5 * FL61) + (C5 * FP63));
FL43 = (FL52 + FP42);
FL41 = ((C4 * (FL51 + FP41)) + (S4 * ((- FL62) + FP43)));
FL42 = ((S4 * (FL51 + FP41)) - (C4 * ((- FL62) + FP43)));
FL32 = ((S3 * FL41) + (C3 * FL43));

NL61 = ((C6 * (NP61 + ((SIX->Y6 * VP53) - (SIX->Z6 * VP62))))
- (S6 * (NP62 + ((SIX->Z6 * VP61) - (SIX->X6 * VP53)))));
NL63 = (NP63 + ((SIX->X6 * VP62) - (SIX->Y6 * VP61)));
NL62 = ((S6 * (NP61 + ((SIX->Y6 * VP53) - (SIX->Z6 * VP62))))
+ (C6 * (NP62 + ((SIX->Z6 * VP61) - (SIX->X6 * VP53)))));
NL51 = ((C5 * NL61) - (S5 * NL63));
NL52 = ((S5 * NL61) + (C5 * NL63));
NL41 = ((C4 * (NL51 + (NP41 + ((D4 * ((- FL62) + FP43))
+ (Y4 * VP43))))) + (S4 * ((- NL62)
+ (NP43 + ((- (D4 * (FL51 + FP41))) - (Y4 * VP41))))));
NL43 = (NL52 + NP42);
NL42 = ((S4 * (NL51 + (NP41 + ((D4 * ((- FL62) + FP43))
+ (Y4 * VP43))))) - (C4 * ((- NL62)
+ (NP43 + ((- (D4 * (FL51 + FP41))) - (Y4 * VP41))))));
NL31 = ((C3 * (NL41 - (D3 * FL43))) - (S3 * (NL43 + (D3 * FL41))));
NL32 = ((S3 * (NL41 - (D3 * FL43))) + (C3 * (NL43 + (D3 * FL41))));
NL22 = ((S2 * NL31) + (C2 * (NL32 + (A2 * FL42))));

TORQUE[1] = NL22 + RS1 * sgn(QD[1]) + RD1 * QD[1];
TORQUE[2] = ((- NL42) + (A2 * FL32)) + RS2 * sgn(QD[2]) + RD2 * QD[2];
TORQUE[3] = (- NL42) + RS3 * sgn(QD[3]) + RD3 * QD[3];
TORQUE[4] = (NL52 + NP42) + RS4 * sgn(QD[4]) + RD4 * QD[4];
TORQUE[5] = (- NL62) + RS5 * sgn(QD[5]) + RD5 * QD[5];
TORQUE[6] = (NP63 + ((SIX->X6 * VP62) - (SIX->Y6 * VP61)))
 + RS6 * sgn(QD[6]) + RD6 * QD[6];
}
```

The difference with the previous program is in the procedure "update" that permits one to update the values of the parameters corresponding to the last link as explained before.

B Identification results

In this appendix we show the results by fitting torques with the dynamic model. The directory that contains the programs to fit the dynamic model are in "grasp:/usr/alberto/DYNAMIC".

In figure 1 we show the joint position corresponding to the first trajectory. In figure 2 we show the velocity of the first trajectory, calculated by the formula $v_i = (p_{i+1} - p_i)/(\Delta t)$, where v_i is the estimated velocity at the sample i, p_i is the joint position at the saple i, and Δt is the increment in time between two sample periods. In our case the sample period is 28 msec. In figure 3 we show the velocity calculated by using the formula $v_i = (p_{i+1} - p_{i-1})/(2.0 * \Delta t)$. This formula produce a more accurate and smoother estimation of the velocity and has been used for the identification procedure.

In figure 4 we show the acceleration calculated for the same trajectory by using the formula $a_i = (v_{i+1} - v_i)/(\Delta t)$, where a_i is the acceleration at the sample i, v_i is the velocity shown in Figure 5. In figure 6 we show the acceleration calculated by using the formula $a_i = (v_{i+1} - v_{i-1})/(2.0 * \Delta t)$, where v_i is the velocity shown in Figure 7. The last calculation produces a more accurate and smoother estimation of the acceleration, and was used in the identification of the constants of the dynamic coefficients.

Figure 8 contains the measured and estimated torques for the first trajectory, using all the 52 linearly independent parameters. The fitting is very accurate for all six joints. Figure 9 contains the measured and estimated torques for the first trajectory, using 23 significant parameters. The fitting is very accurate for the first three joints, having small differences in the last three joints.

Finally, figures 10 to 15 contains plots of measured and estimated torques for six different trajectories, using the average of the 23 significant parameters. The fitting for the three first joints is very accurate. The fitting for the last three joints is less accurate, although the errors are less than 20 percent of the torque. This may be due to the fact that the torque in the three joints is small, but also to the fact that the three last joints are coupled. Effectively, rotation of joint 4 affects rotation on joints 5 and 6, as well as rotation of joint 5 affects the rotation of joint 6, and these effects are not taken into account in the dynamic model.

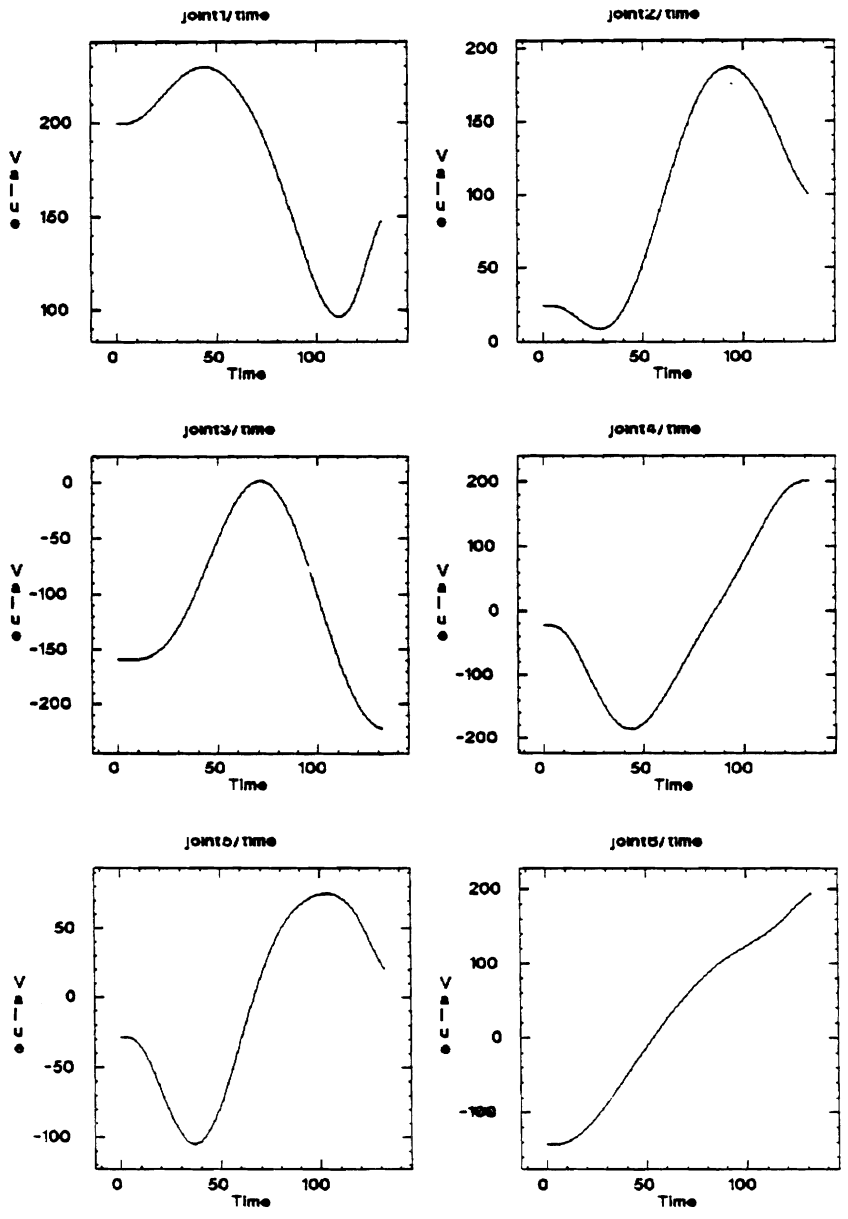


Figure 1: Joint position for the first trajectory

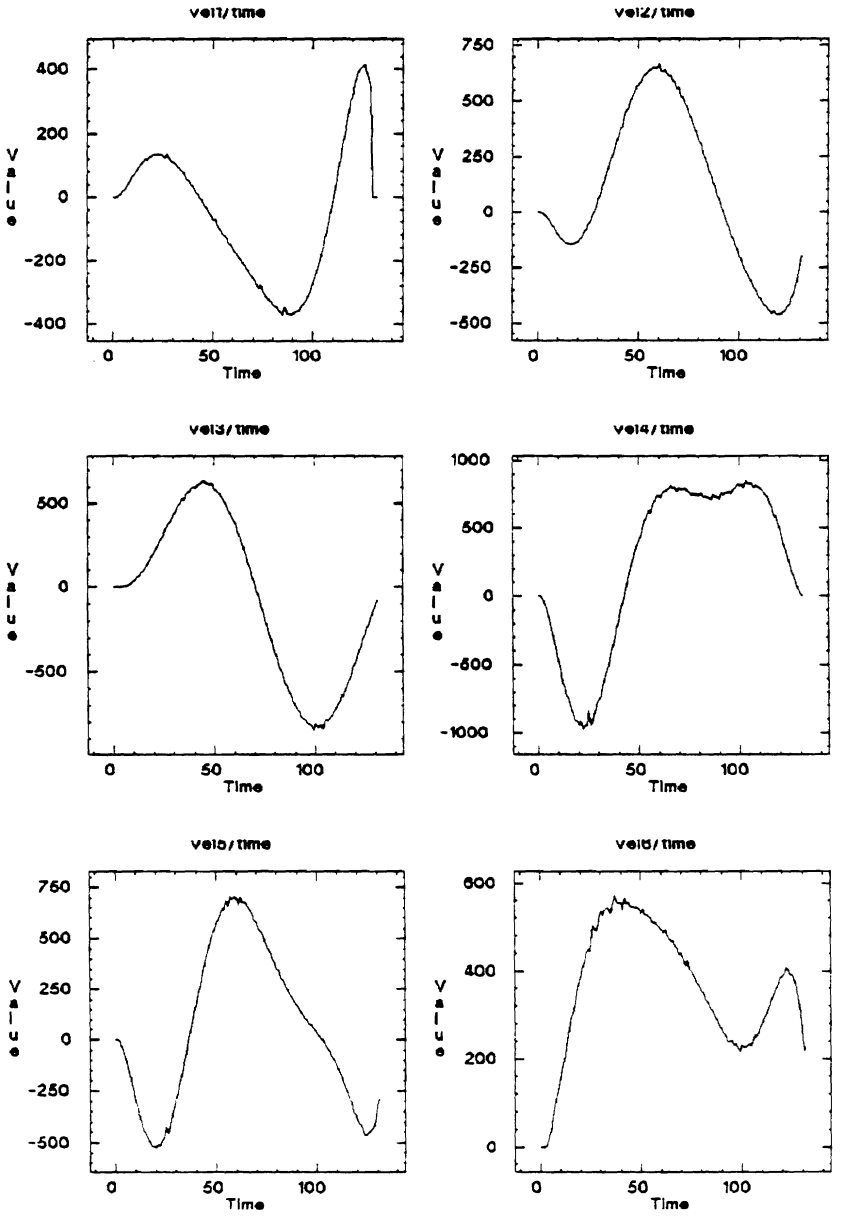


Figure 2: Velocity of the joints calculated by the first method and first trajectory

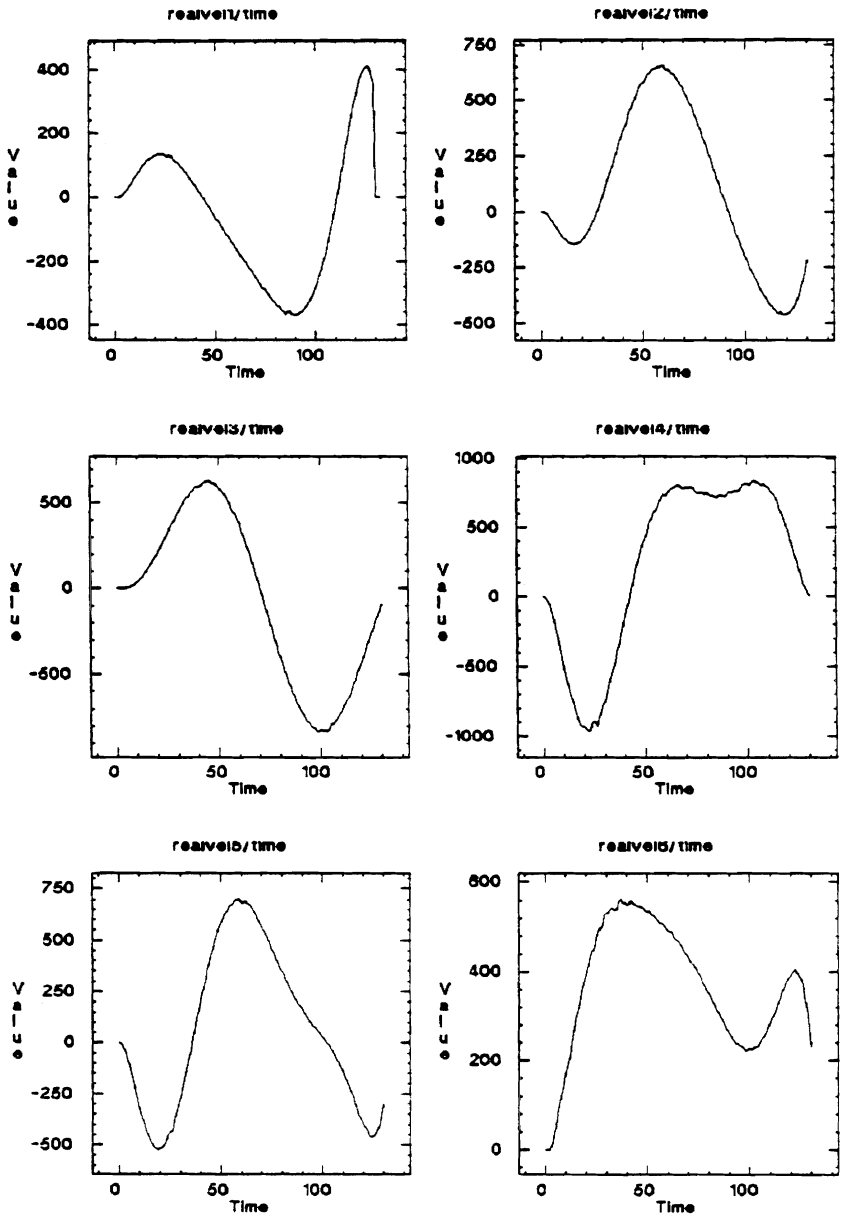


Figure 3: Velocity of the joints calculated by the second method and first trajectory

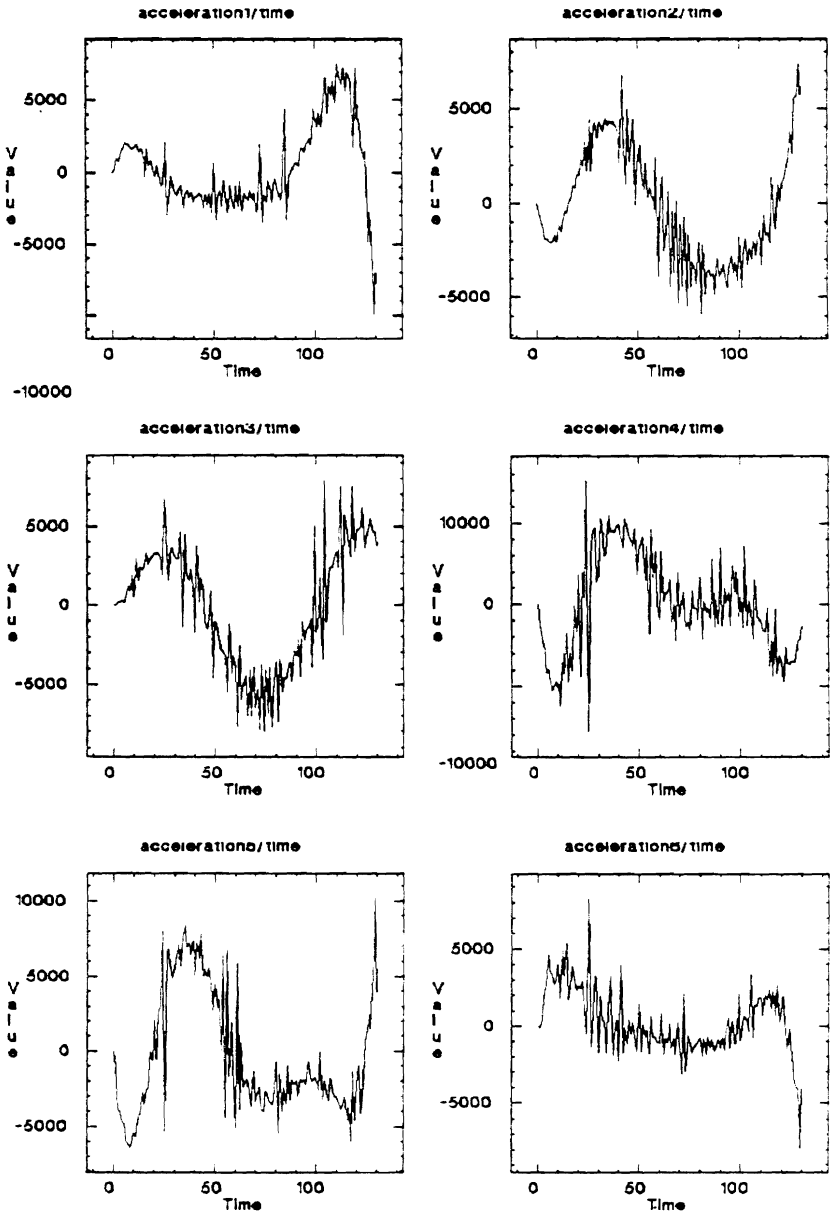


Figure 4: Acceleration of the joints calculated by the first method and first trajectory

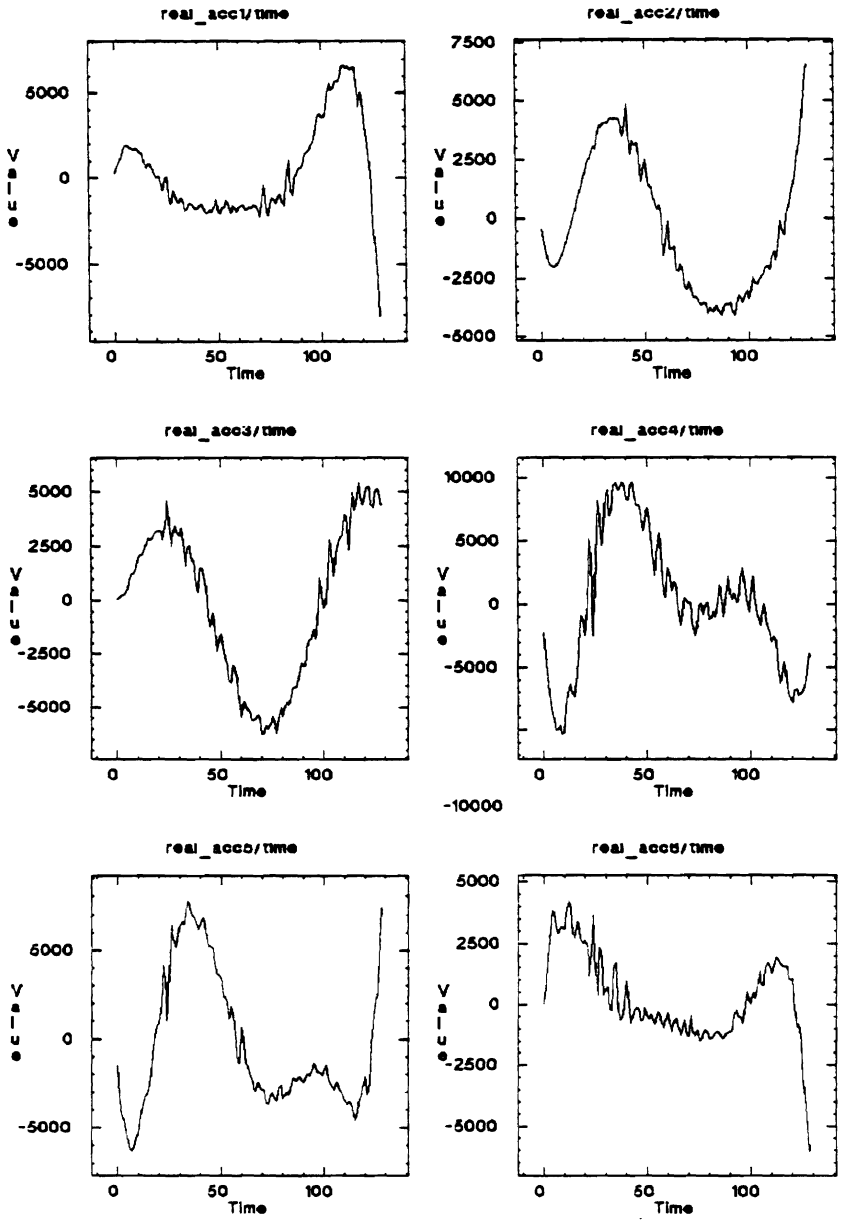


Figure 5: Acceleration of the joints calculated by the second method and first trajectory

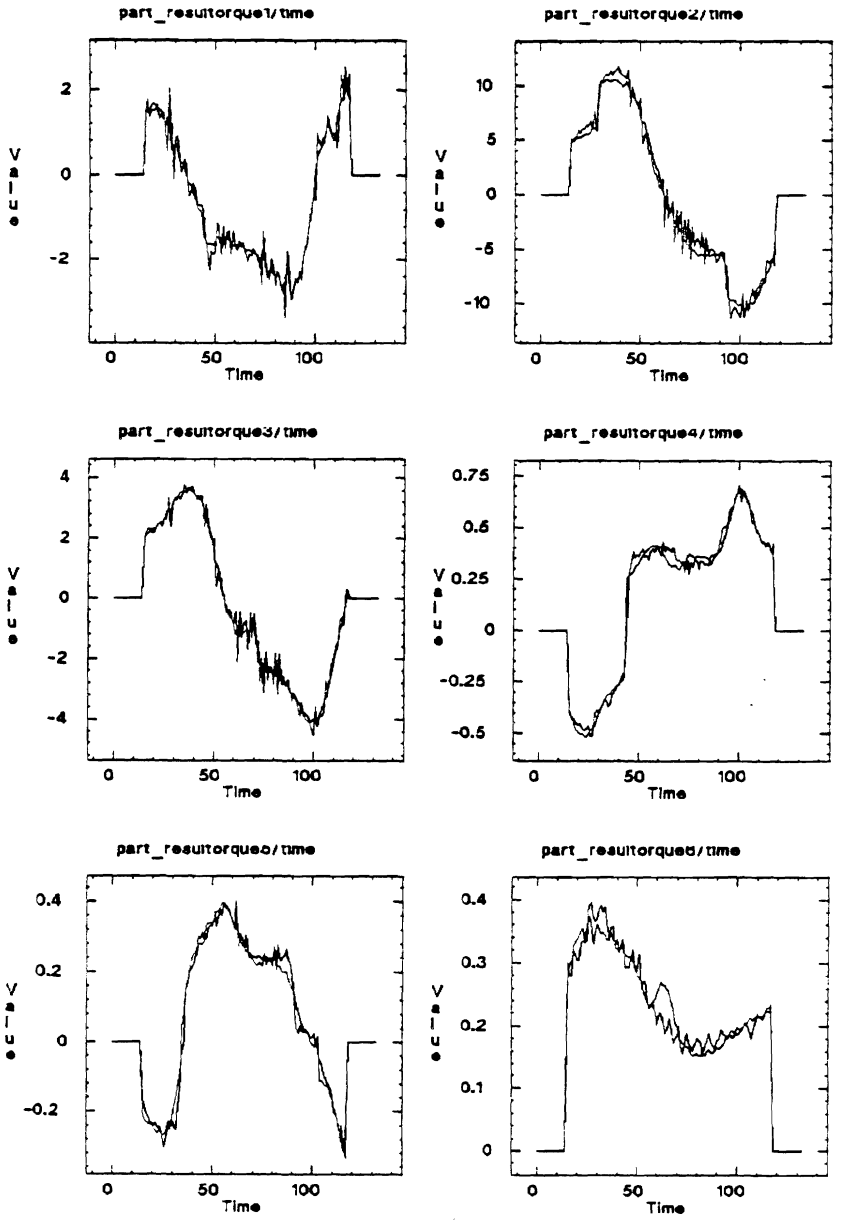


Figure 6: Fitting of the torques for 6 joints, 52 parameters

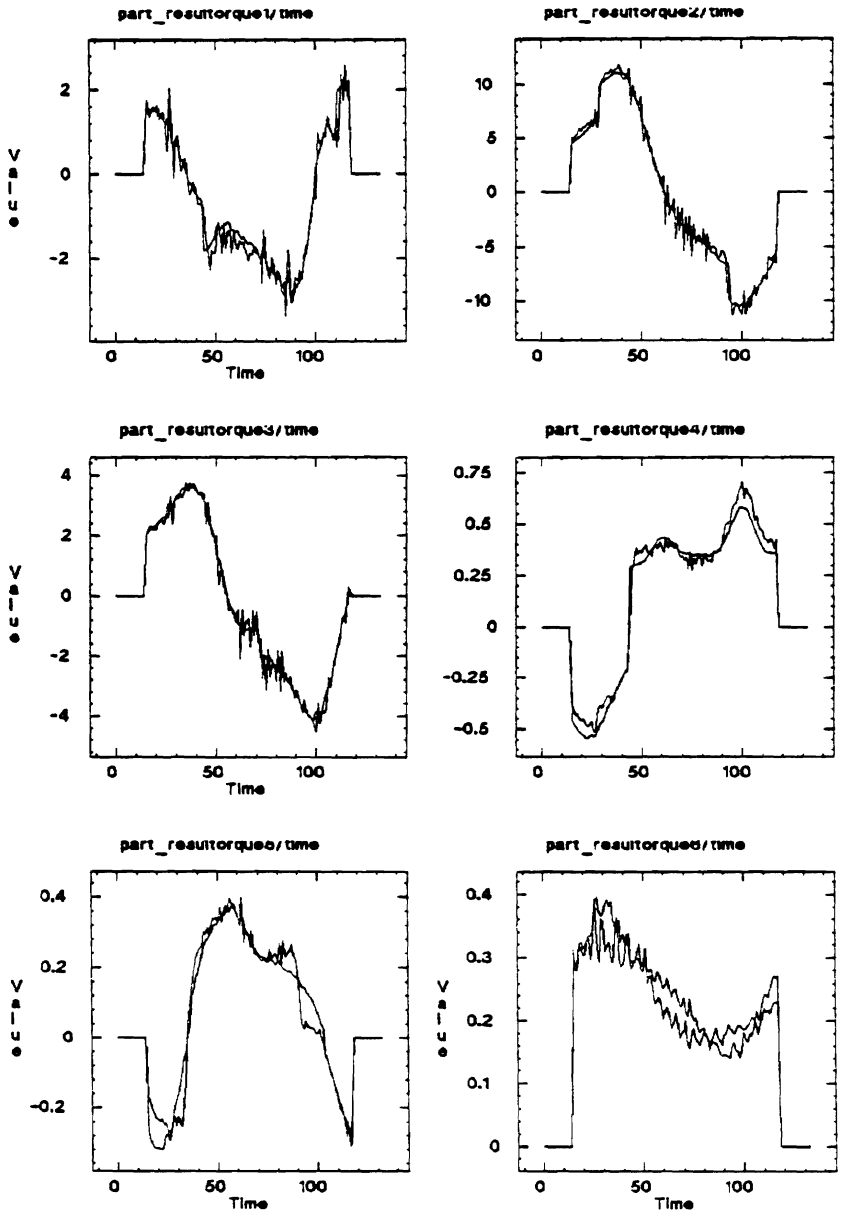


Figure 7: Fitting of the torques for 6 joints, 23 parameters

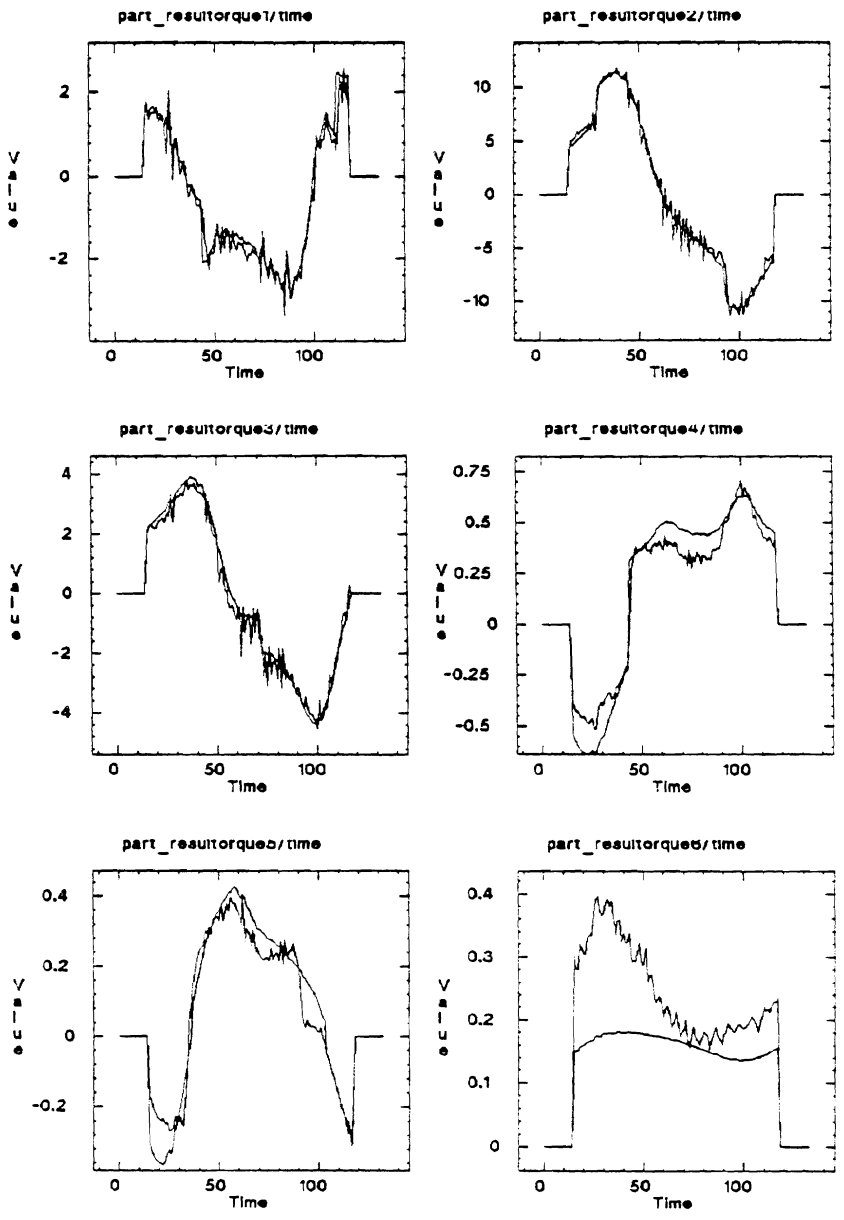


Figure 8: Fitting of the torques for 6 joints, 23 "averaged" parameters

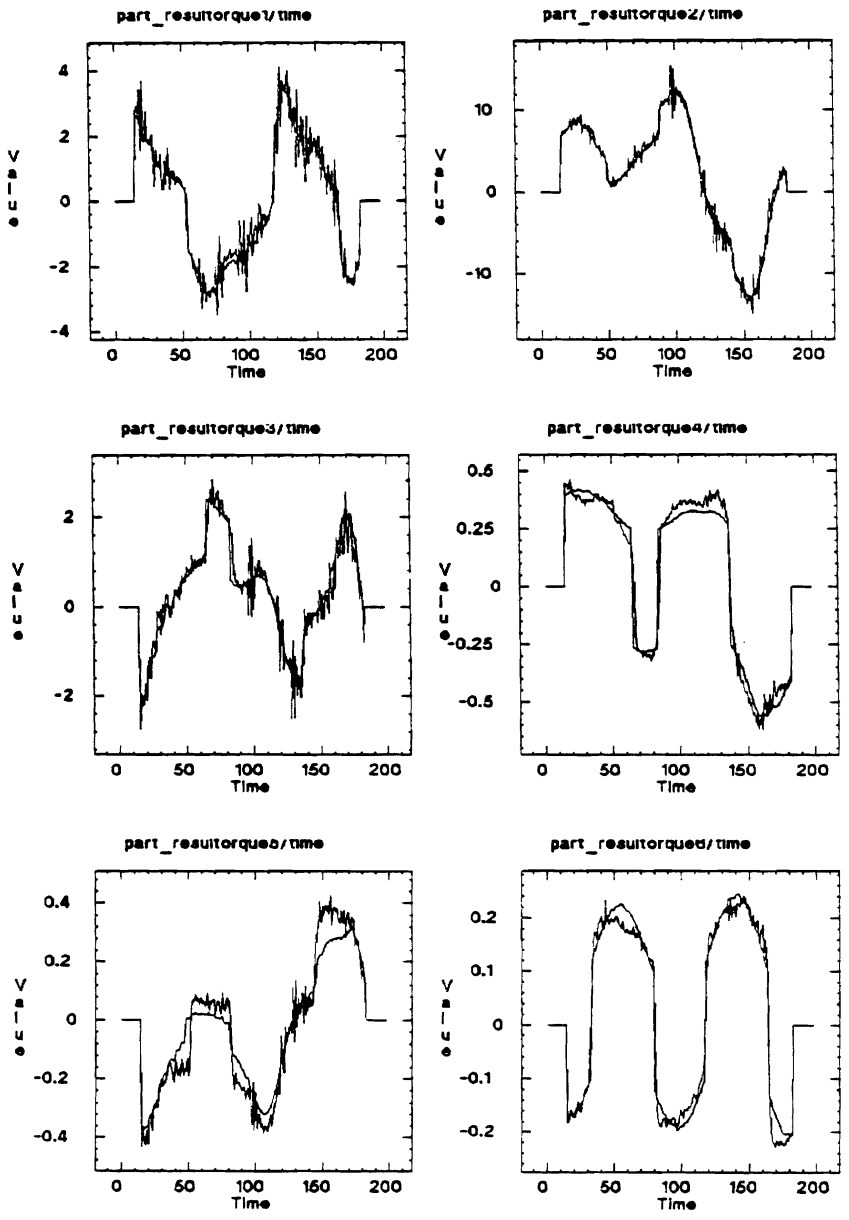


Figure 9: Fitting of the torques for 6 joints, 23 parameters

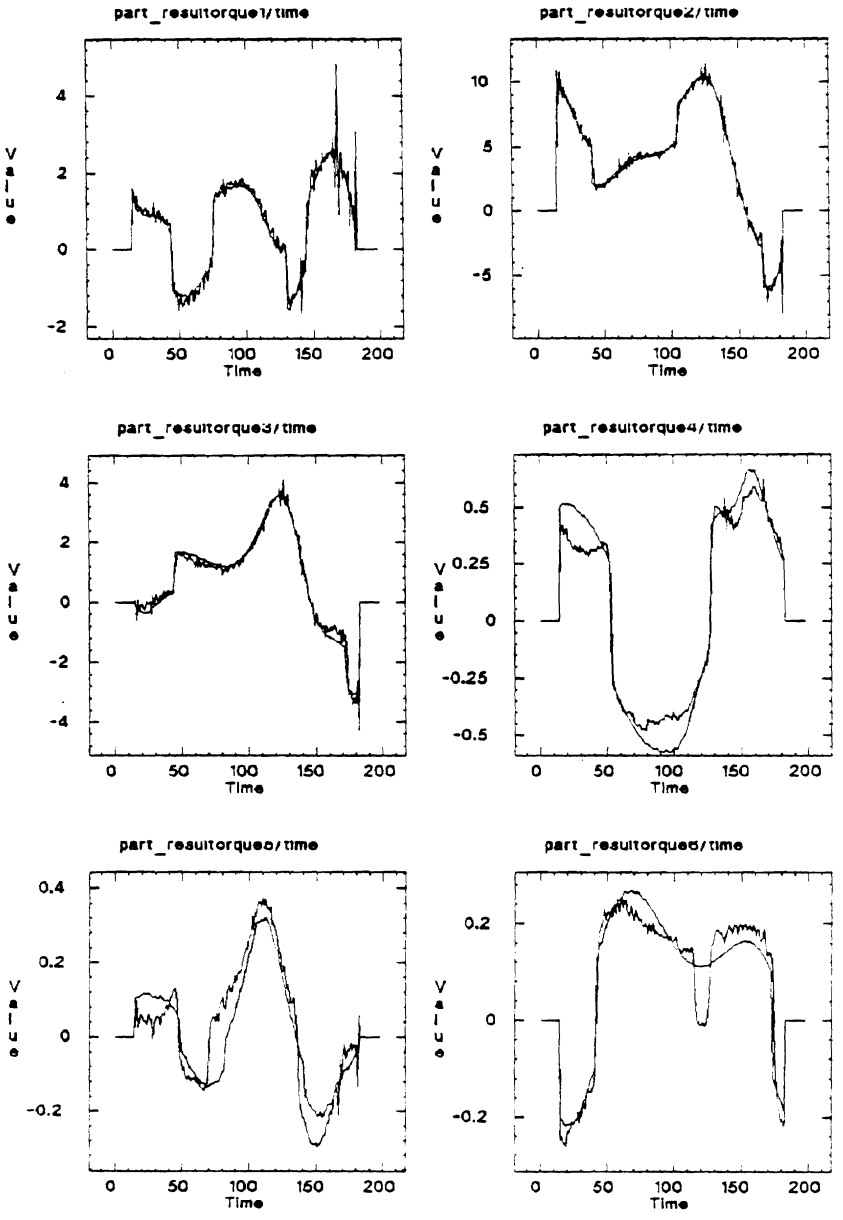


Figure 10: Fitting of the torques for 6 joints, 23 parameters

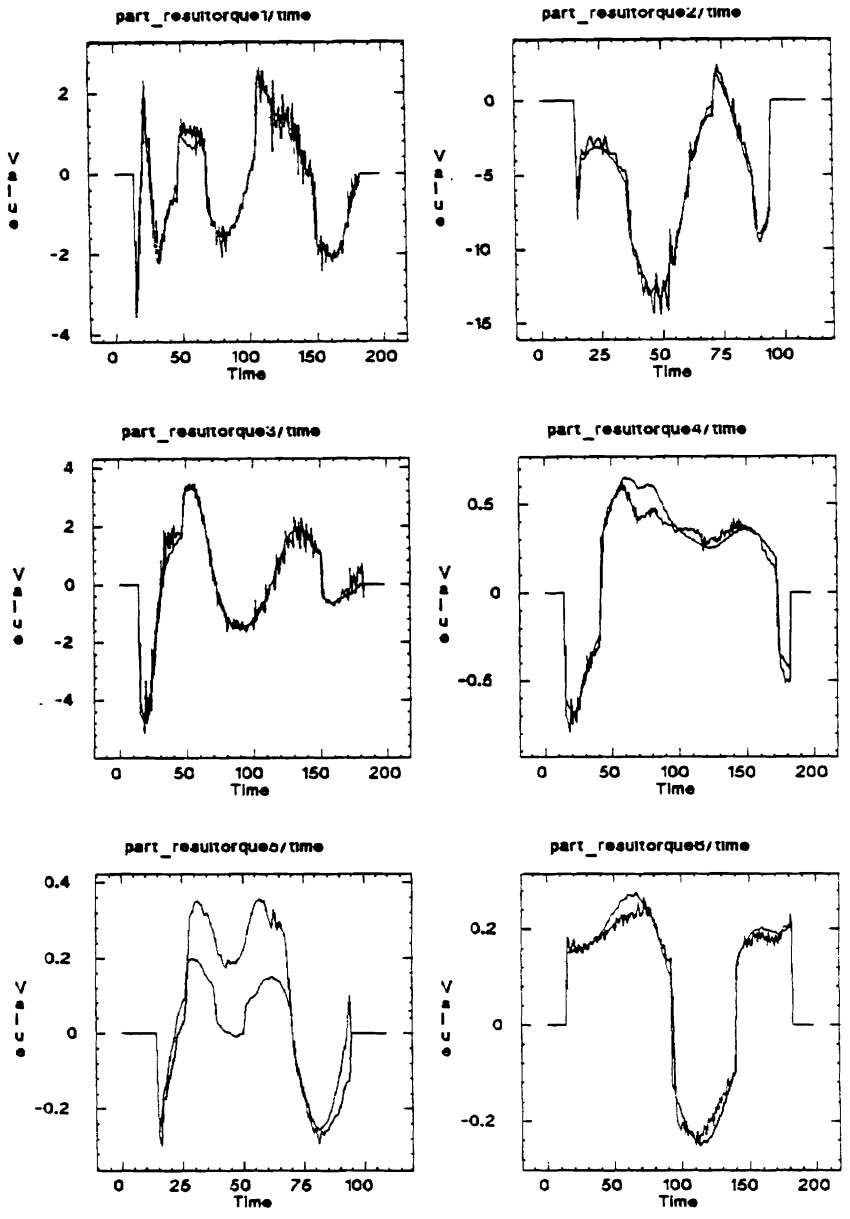


Figure 11: Fitting of the torques for 6 joints, 23 parameters

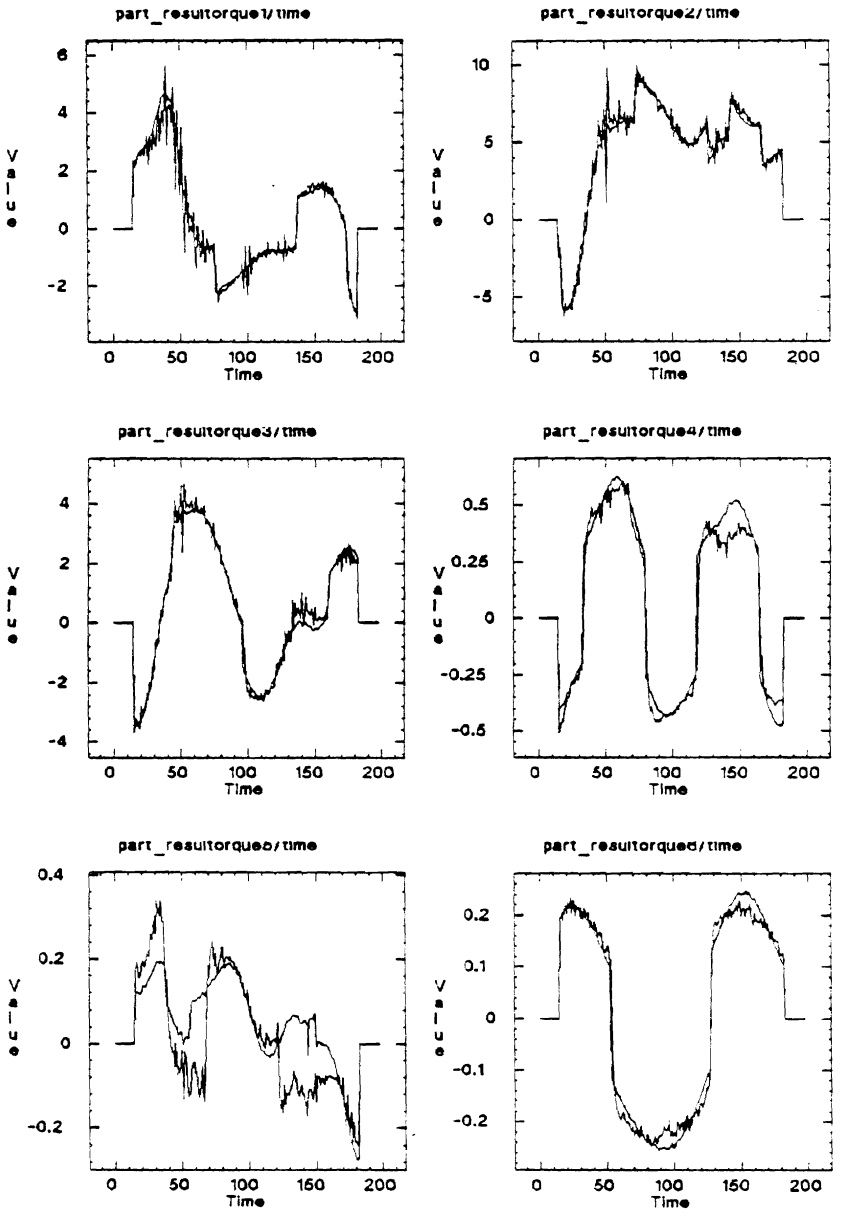


Figure 12: Fitting of the torques for 6 joints, 23 parameters

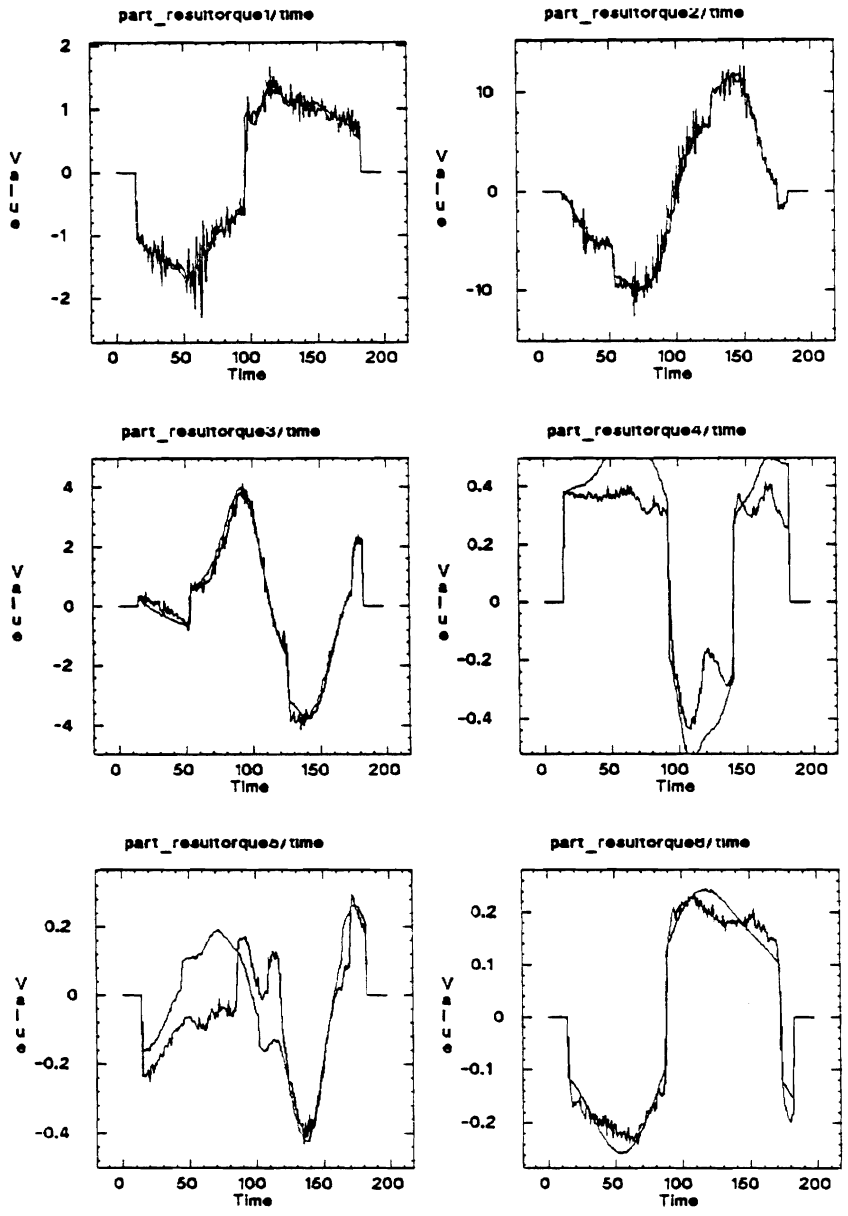


Figure 13: Fitting of the torques for 6 joints, 23 parameters

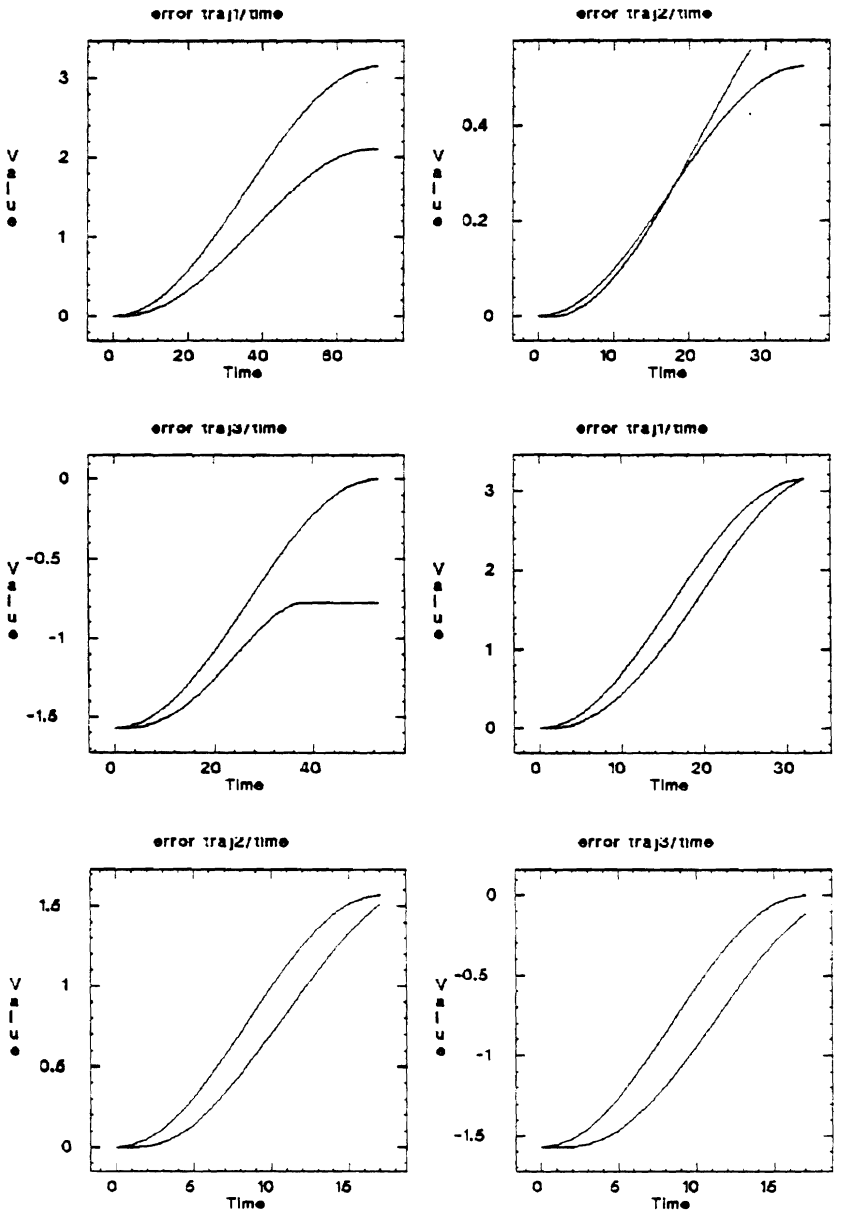


Figure 14: Input and real trajectory for joints 1,2,3 slow and fast motion

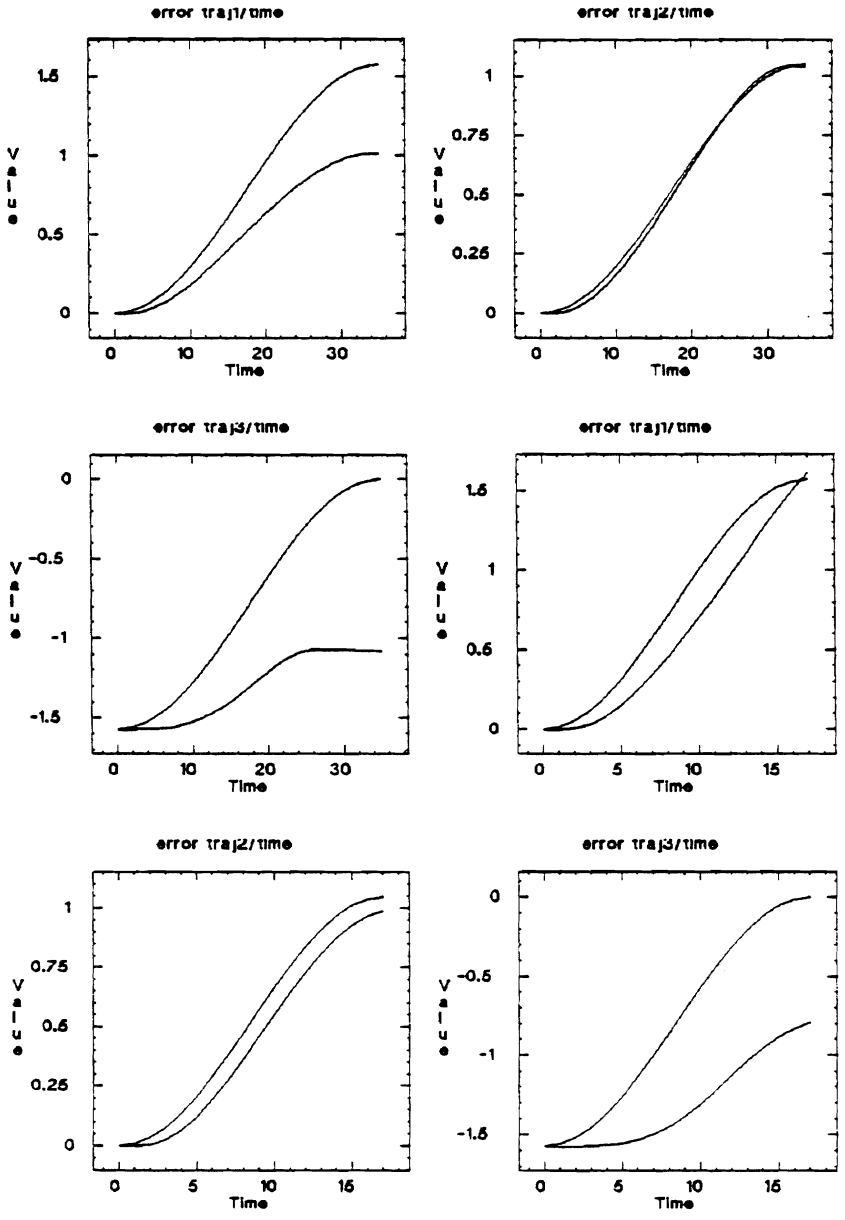


Figure 15: Input and real trajectory for joints 1,2,3 together slow and fast motion

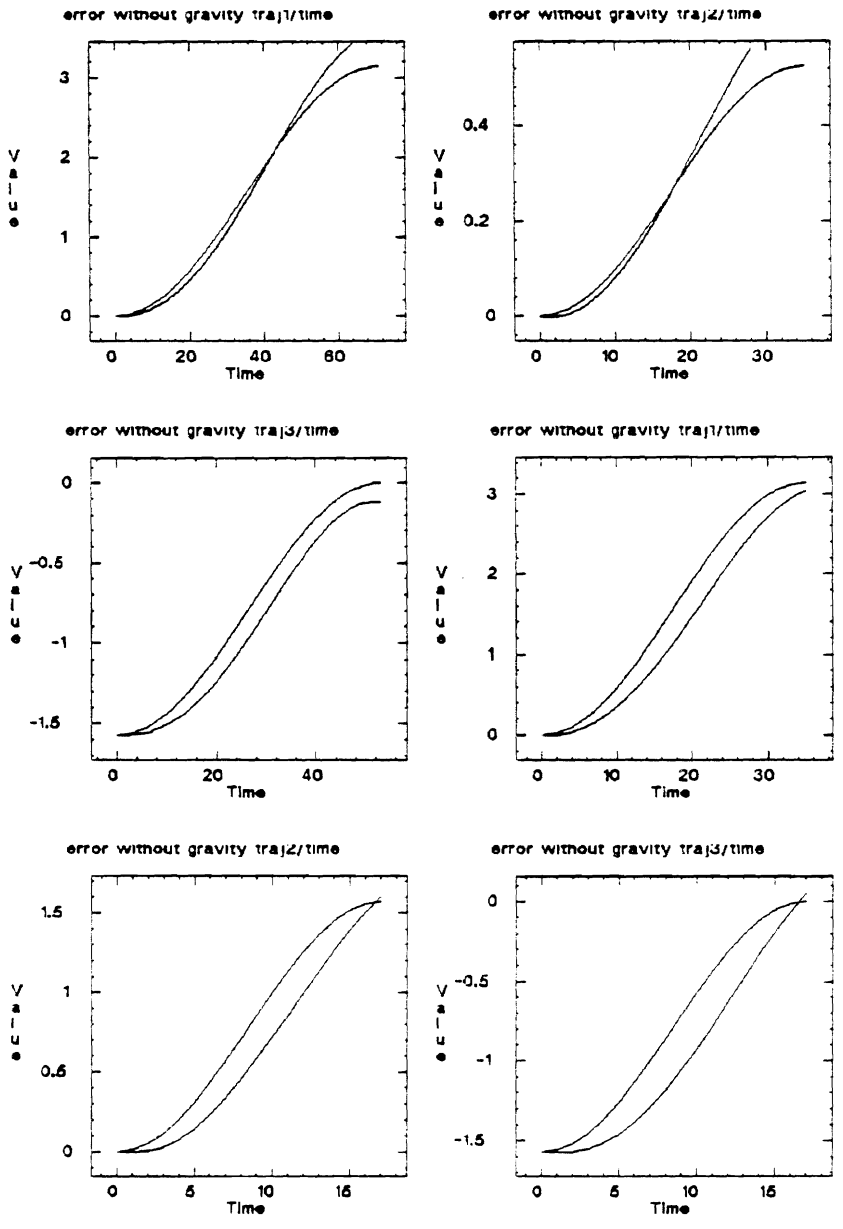


Figure 16: Input and real trajectory for joints 1,2,3 slow and fast motion, with compennsation of gravity

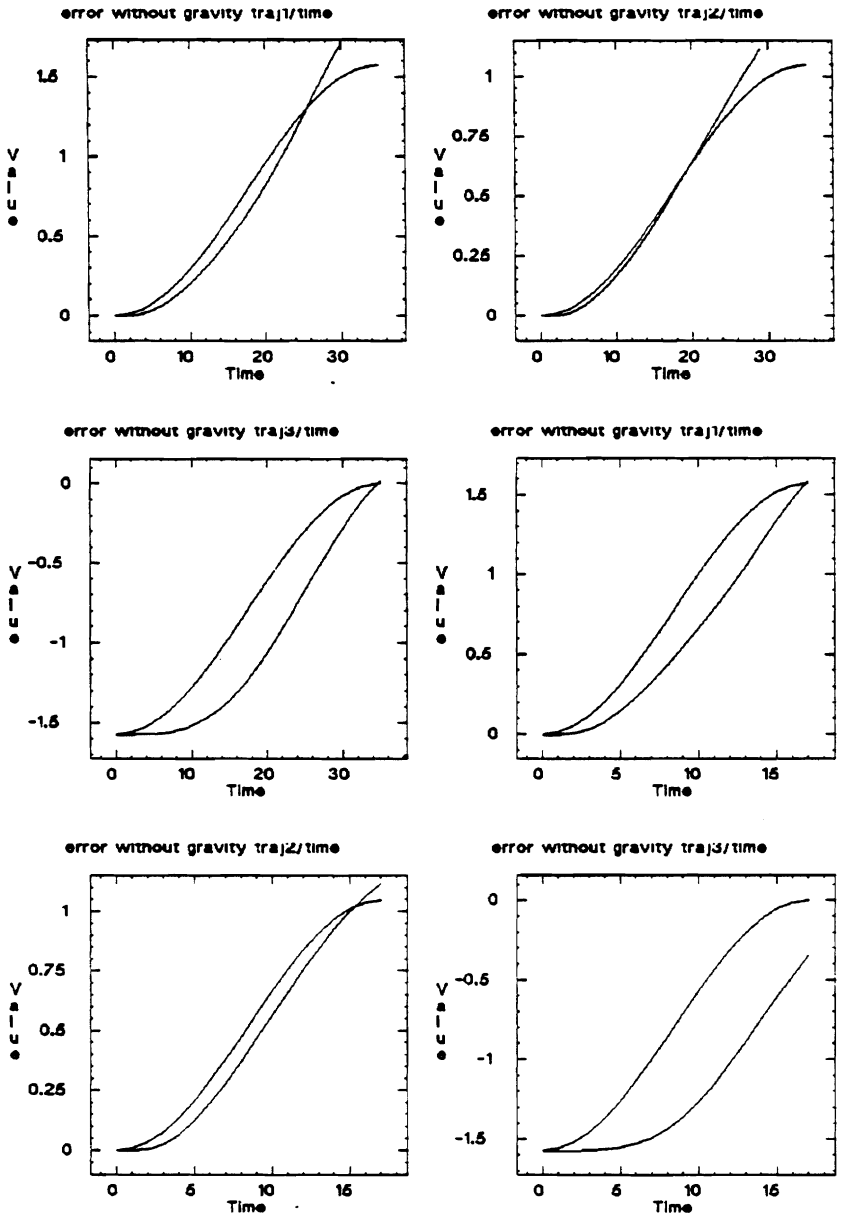


Figure 17: Input and real trajectory for joints 1,2,3 together slow and fast motion

A.5 RFMS Software Reference Manual

CONTENTS

APPENDICES

RFMS SOFTWARE REFERENCE MANUAL

Hong Zhang
Department of Computer and Information Science
The University of Pennsylvania

1. Introduction

This manual explains the software of the Robot Force and Motion Server (RFMS)[1], a high performance robot control system designed and implemented in the GRASP laboratory. In this system, the robot manipulator is considered a force/motion server to the robot and a user application is treated as a request for the service of the manipulator. The user application is created on one of the Unix/VAX machines in 'C' programming language as a set of function calls. The application is carried out in a multi-processor controller, which consists of Intel single board computers and provides computing power necessary for computationally intensive tasks. The VAX machine and the Intel controller communicate through Ethernet, a local area network, which also allows interaction between the user and sensors. Design principles of the system can be found in [2].

The software of the system involves a variety of computers: the user interface is written to be executed on a Unix/VAX machine; the control software is written to be executed on Intel 8086-based single board computers; and the network software is written to be executed on a Unix/VAX machine on one end and Intel processor on the other. The rest of the documentation will be organized according to where the execution of the program is. Section Two will discuss user interface, and for those who intend to only use the system for specific applications, it is adequate to read this section. Section Three will discuss the implementation the Ethernet software. This section is useful only if one would like to make changes to the communication protocols between the user and the Intel controller. Section Four will discuss the software written for the robot controller which consists of Intel single board computers to control the robot manipulator, a PUMA 260 in our case. It is important for one to understand this section if what is provided in the system is insufficient to carry out his applications.

This material is based on work supported by the National Science Foundation under Grant No. ECS-8411879. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

We would like to mention that the system is yet to be finalized, for we have been using it for research and thus need to constantly make changes. Several versions of the system exist among the people who have used and modified the system for their own needs. We will try to be consistent throughout this documentation, though confusion may occur from time to time. The programs are organized by the processor on which they are executed, with one directory per processor and common include files in two separate directories. The following table roughly explains the contents of the directories under */usr/users/hz on robo.cis.upenn.edu* and */usr/users/hz/robo* on *grasp.cis.upenn.edu*.

Directory	*Content*
/VAX	user interface and Ethernet driver on the VAX side
/include	include files for /VAX directory
/186	Ethernet driver on the Intel side
/h	include files for the Intel controller
/super	programs written for the supervisor of the Intel controller
/J*i*	programs on the *i*th joint
/math	programs for the math processor
/sys	library functions for 8086 (I/O, interrupt control, vector operations, etc.)
/c86	cross compiler for 8086, loader, and optimizers
/c186	cross comipler for 80186

Table 1. RFMS Directories

All source files will be *underlined* and all functions will be *italicized*.

2. User Interface

From a user's point of view, the available functions can be classified into three categories: world-model definition, motion record definition, and motion requests. Another category, task synchronization, enables the user to wait until the completion of a sub-task before the next one starts. Although it is not available at this time, it can be easily added. Sensor input is another area yet to be integrated into the system, and all the mechanisms exist. The structure of the program is similar to that of an RCCL

program in spirit, whose underlining principles can be found in [3]. A user requests the service of the robot controller by making function calls from a 'C' function named *pumatask()*.

2.1. Programs of the User Interface

A total of eight programs constitute the user interface of the system. Since the emphasis of the system is not to construct a comprehensive robot programming system, effort made to create the user process is kept at minimum. We have used this part of the system only for testing the robot controller.

A user defines a task by making calls to the system functions. A task defines the world model in terms of the transformations (relationships between coordinate frames of interest) and position equations (definitions of points in the work space to which the manipulator is to move). The fashion in which a move to a position is conducted such as segment time, compliance specification, etc., is defined by a motion record. Upon any call to create one of these, the created data structure is first stored in the corresponding symbol table and then a copy of it is sent to the RFMS through the Ethernet. To initiate an action, a move is called with two parameters: a pointer to the destination position and a pointer to a motion record. Fundamental to the user interface are the three symbol tables storing transformations, position equations, and motion records that have been created. The move requests are not stored in a symbol table because they are not referred to by other variables. This may change, however, once task synchronization is needed for the system has to keep track of the move requests have been issued. Once the application is created and compiled, one can run the application like any other 'C' programs by *a.out*.

The *main.c* allocates memory for static symbol tables for the user process, initializes the communication link between the user process and the RFMS, and then calls *pumatask()* defined in, say, *myapp.c*, by the user, which contains a stream of function calls to the system. After defining an application, the user may call the function *debug()*, which logs data coming from the Intel controller in real-time and store them in six different files, corresponding to six joints of the robot manipulator. The nature of the data is entirely up to the user, but there must be an agreement in what the Intel controller sends and what the user interprets. This function call is optional and has been used as a debugging tool so far. One can expect to log one set of data every four to five sampling periods.

There are currently a number of ways to create a transformation: a transformation with pure translation and no rotation by *gentr_trsl()*, a rotation transformation defined in terms of either Euler angles or roll-pitch-yaw angles by *gentr_eul()* or *gentr_rpy()*. All functions related to transformation creation are defined in *trans.c*.

A position equation is created by a call to *makepst()* in *pst.c*. One must provide a name to the position as a string of characters in the first argument and three constants for the three configurations *lefty*, *up*, and *flipped*, associated with the PUMA 260. Since a position equation may contain a number of transformations on either side, *makepst()* must be able to handle variable number of arguments[4]. The last argument of *makepst()* when defined is declared to as a pointer to a transformation, the same data type as the rest of the arguments that follow it when the actual call is made. Two key words, EQ and TL in the actual call help interpret where left-hand side ends and which transformation is the tool transformation [5].

A motion record specifies how a motion is to be executed. and it contains such attributes as segment time, acceleration time, mode of the motion, and compliance specification. These attributes then become the four input arguments to a call to *makemot()*, which is contained in the program *mot.c*. Both segment time and acceleration time are in seconds, and mode of the motion can be either Cartesian or joint. Compliance uses a bit pattern as in Figure 1 to indicate the physical constraints to the motion

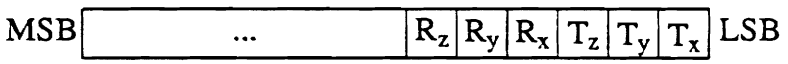


Figure 1. Bit Pattern Representing Compliance

where R_i represents rotational compliance along a certain Cartesian direction and T_i translational compliance along a certain Cartesian direction. In this example, four motion records are defined. The first simply defines a joint motion with a segment time of 2 seconds an acceleration time of 0.2 seconds. The third motion records defines a Cartesian motion with a 20 second segment time, a 0.5 second acceleration time, and compliance along *z* direction.

The program *move.c* contains the function *move()*. The function uses the two input arguments, a pointer to position and a pointer to the motion record, to issue a move request.

At the end of each function call, a message is issued to the RFMS. Functions in the file *mess.c* handle packet preparation. Currently, the user application is not receiving any messages, even though the software could handle it. The format of the messages is defined in *msgs.h*. The message type identifies the content and interpretation of the message. A message is written into the buffer, *msg*, before function *mess()* is called, which prepares the Ethernet packet and invokes Ethernet function *Send()* in *comm.c* to send it.

All floating point numbers are modified before being sent, since the VAX machine and Intel computers represent a floating point number differently, as illustrated in Figure 2.

low word	sign	exponent (8 bits)	high significant (7 bits)
	low significant (16 bits)		

Intel Floating Point Representation

low word	low significant (16 bits)		
	sign	exponent (8 bits)	high significant (7 bits)

DEC Floating Point Representation

Figure 2. Floating Point Representation

We choose to convert floating point numbers on the VAX machine since it is faster than any Intel computer and time on the Intel computers is more valuable. The function *convert()* in *mess.c* performs the conversion.

2.2. An Example

The following example further illustrates how an application program is created.

```
# include "../include/datdef.h"
# include "../include/extdef.h"
# include "../include/condef.h"

pumatask()
{
        TRSF *t2;
        PST *home_pst;
        MOT *mjnt, *mwait, *mcart, *mcwait, *mcartcz, *mcartcx;

        t2 = gentr_trsl("t2", 203.2, -126.23, 203.2);        /* home */
        home_pst = makepst("home", RIGHT, DOWN, FLIP, t6, EQ, t2, TL, t6);
        mjnt = makemot(2.0, 0.2, JNT, 0);
        mcart = makemot(4.0, 0.3, CAR, 0);
        mcartcz = makemot(20.0, 0.5, CAR, 0x4);
        mcartcx = makemot(15.0, 0.5, CAR, 0x1);

        move(home_pst, mcartcz);
        move(home_pst, mjnt);
```

}

The three include files in the beginning are necessary for the user to define local variables of the data types created for robot programming *(h/datdef.h)*, to make function calls to the system *(h/extdef.h)*, and to make use of the constants defined in the system *(h/condef.h)*. *TRSF*, *PST*, and*MOT* represent data type transformation, position equation, and motion request, respectively. In the instruction section of *pumatask()*, a transformation is first created by providing function *gentr_trsl()* with three translational components of the *p* vector in the order of x, y, and z.

The Function call, *makepst()*, creates a position equation with transformations either known to the system or defined by the user. In our case, it has *t6*, which is known to the system, on one side and *t2*, which is defined by the user, on the other. Configurations of this position are specified as right, down and flip. Four motion records are defined in this program, with one joint motion, and three Cartesian motion, of which two require compliance.

Two motions are requested in this task. The arm will move to the same position as the initial position (i.e., remain stationary), while complying along *z* direction. Once this is finished, the arm will move back to home position.

Once the application is created, it can be compiled and linked with the rest of the system. The application is executed in the same fashion as any other Unix executable file, when the Intel controller is initialized and ready to accept tasks.

3. Ethernet Interface

The user and the Intel controller communicate through Ethernet, a local area network. The implementation details of this interface can be found in [6] and in Appendix A. Here we only outline some of its features users need to know in order to use it.

The interface on the users' side is performed on a Unix/VAX machine. Unix supports Ethernet and, for robot control, our software is built as the data link layer by making use of the Data Link Interface (DLI). The interface on Intel's side is built from scratch and has two layers, the data link and logic link. The protocol used between the two machines is *one-bit-sliding window and positive acknowledgement with retransmission*, which means the machine sending a message keeps trying until it receives acknowledgement or the number of trials exceeds a limit. A token exists which determines who can send a message at any given moment. It is usually held by the VAX machine and the Intel machine has it only when the VAX machine requests a message from the Intel controller. Typically the VAX machine sends a message to the Intel machine whenever it wants and the arrival of a message creates an interrupt to the Ethernet board 186/51[14] of the Intel controller, which then reads the message in its interrupt handling procedure. The Intel controller, on the other hand, cannot send a

message to the VAX unless it is explicitly asked to do so. This is caused by the fact that the software on the VAX side is not written as an interrupt handler, but rather as a listener and therefore can not deal with any unexpected incoming messages.

Two primitives on the VAX for sending and receiving a message have the syntax:

Send (buffer, size)

and

Recv(buffer, size).

The counterpart on Intel side employs two primitives:

Recv_Frame(buffer)

and

Send_Ack() or *Ans_Send_Req()*.

Which one to use to send a packet depends upon if the message just received is a real message or a request for a message to be sent to the VAX. Once messages are received by the 186/51, they are queued in an array, waiting to be processed by the supervisor of the Intel controller.

The communication software for VAX is contained in one file *comm.c*, and for the Intel controller there are three 'C' files in the directory */186*, *dld.c*, *llc.c*, and *main.c*. The program *dld.c* contains the data link layer, and the program *llc.c* contains the logic link layer. The program *main.c* first initializes the data link layer by *Init_586()*, sets up a linear array of messages in which the incoming messages are stored, and inform the supervisor of the array address by storing it at a fixed memory location accessible to both supervisor. Two other assembly programs in this directory, *reint.a86* and *handler.a86*, deal with the interrupt control of the 186/51.

There is only limited memory space on the 186/51 and, therefore, the size of the message queue can be of only a finite length. Currently, a total of 100 messages can be stored, of which each has a fixed size of RBUF_SIZE bytes. Since the supervisor keeps looking in the queue for available new entries, overflow never occurs if we assume the speed of processing messages by the supervisor is faster than the that of the incoming messages. The system fails if this assumption is not valid. A dirty bit in the last byte of a message buffer indicates if the buffer contains an unprocessed message.

There are currently two 186/51 computers of different models: one is an ES and the other an S. In additional to their difference in jumper locations and notations, the only software difference one needs to know is the Ethernet address defined for the Ethernet chip 82586. The S model has an address of

0x08, 0x00, 0x2b, 0x02, 0x89, 0xfc,

and the ES model has an address of

0x08, 0x00, 0x2b, 0x02, 0x96, 0x74.

4. Intel Controller

This part of the software runs on Intel single board computers, and it is developed on a VAX machine where the user process is and cross-compiled and down-loaded to the targets via a serial line. (The information on the cross-compiler can be found in Appendix B) The controller is a multi-computer system with shared memory and a common bus, through which data communication and control signals are transmitted. Each computer in the system contains dual-ported memory, of which part is defined as global so that other computers in the system can access it as well. Information exchange takes place in the form of mail boxes and system synchronization is achieved by interrupts. There are currently nine computers running in parallel, six joint processors, a supervisor, a math processor, and an Ethernet computer. There is a real-time synchronized interrupt driven process on each of the joint processors, the supervisor and the math processor. In addition, there is a background process on the supervisor and the math processor. 186/51 runs asynchronously with the rest of the system.

Supervisor, joints and the rest of the system need to communicate with each other and exchange information. Also the kind of data each one requires of any other is known a priori. To facilitate such communication, *mail-box*es are created on each computer with their addresses stored at pre-defined memory locations. These addresses are currently stored in the topmost part of the memory from segment *0xff00* so as not to interfere with the code, data, or stack segments. During the initialization process, supervisor waits until ready flags are cleared in all processors before it picks up addresses of the mail-boxes where it will either drop or pick up mails. Most of the global memory access is done by the supervisor. Currently the only access by the joints is during the compliance when every joint needs to collect other joints' errors. Two system functions, *rblock()* and *wblock()* facilitate global memory access. The sources and destinations of the mail boxes are summerized in the following table.

data type	*source buffer (origin)*	*destination buffer*	*description*
S_MAIL	MAIL (supervisor)	MAIL	one copy to each joint to instruct what actions to take
M_MAIL	MMAIL (supervisor)	MMAIL	information math processor needs to compute Jacobian matrices and dynamics
J_MAIL	JMAIL (joints)	JMAIL*i* *i=1...n*	one from each joint to the supervisor to return the status of the joint
PARCEL	PARi (math)	PARC*i* *i=1...n*	results computed by math and collected by supervisor for one of the joints
PARCEL	PARC*i* *i=1...n* (supervisor)	PARC	one on each joint distributed by the supervisor

Table 2. Mailbox Description

Both trajectory generation and inverse kinematics are performed on this parallel processor and a lot of efforts have been devoted to computation distribution. Trajectory generation at Cartesian level, *i.e.*, calculation of the end effector position and orientation, is performed on the supervisor. Joints, on the other hand, plan their individual trajectories given the end effector coordinates. The dependency exists among the inverse kinematics of the joints, for the ith joint requires solutions of all prior i - 1 joints. This dependency, however, can be eliminated when each joint uses other joints' solutions in the previous period. This scheme is approximate, but it allows the system to compute the kinematics in parallel thus speeding up the system substantially. The details of the trajectory trajectory can be found in [7] and the details of the parallel inverse kinematics can be found in [8].

4.1. Supervisor

Two concurrent processes, one being interrupt driven and the other in the background, are executed on the supervisor. The background process reads the messages stored in the 186/51 and sets up data structures, which the second interrupt driven process uses to coordinate the operation of the controller and the generation of motion trajectories. Supervisor runs on an iSBC 86/30 computer[22].

The program, *main.c*, initializes the system and interacts with the user to go through the *manual* mode, the *calibration* mode, and then onto the *set-point* mode. Its serial port is connected to a terminal where the user operates for the purpose of down-loading the code and monitoring the controller operation during system development. Eventually, the interactive session should take place between the VAX machine where the user really is and the control system through the Ethernet.

4.1.1. Background Process

The background process program is stored in *bkgd.c*. To process messages stored on the 186/51 (refer to Section 3), the supervisor maintains a pointer to the next available message in the message queue. Depending upon the type of the message, different action is taken. The format of the messages are defined in the include file *h/msgs.h*. Data structure definitions in this file must agree with those in *include/datdef.h*, if the supervisor is to interpret the messages correctly. When there is no message in the queue, the background process simply waits.

Upon the arrival of a message, the type of a message is determined, and a corresponding data structure may be created and added to the world model. Currently, there are six possible types, INIT, STOP, TTR, TPOS, TMD, TREQ. The first two simply are signals for the beginning and end of a task definition. The rest are for a transformation, a position, a mode, and, motion request message, respectively. The definitions of these data structures can be found in *h/datadef.h*.

These data structures refer to or are linked with each other. For example, a position contains pointers to transformations defined previously. If the messages came from the same machine as the one that receives it, the addresses could be used as pointers. Unfortunately this is not the case. A linked structure must be sent piece by piece and the receiving machine must be able to resolve all the cross references. In order to be able to locate the dependencies, we associate each message of a given type with an identification number. To facilitate a fast search, four symbol talbes, *ttbl*[], *mtbl*[], *ptbl*[], *rtbl*[], are set up to store the pointers to the data structures and the id numbers are indeces in the symbol tables.

When a position equation message arrives, a ring structure is created[5]. The program, *pstn.c*, contains functions necessary to create the structure. A ring consists of a number of *items* representing transformations in the equation, of which each contains a pair of *atoms* containing the forward and inverse transformation. Function *Atom()* allocates memory for one atom, *NewTerm()* links a pair of atoms, *Listn()*'s link *n* terms, and *MakePos()* takes two lists of terms as left and right hand sides of the equation and forms the ring.

Processing of other messages requires much less work and is dealt without any primitive functions.

4.1.2. Real-time Process

The real-time process is executed upon a periodic interrupt signal generated by the programmable timer on the supervisor. The entire process runs like a finite state machine and action taken in each period depends on two state variables. The variable, *rtstate*, in program *rtisr.c*, changes among eight possible states, IDLE, FREE, MANU, CALIB, HOLD, SETP, STOP, and EMGCY. These constants are defined in file *comm.h*. The state the system may fall in is illustrated by the following graph.

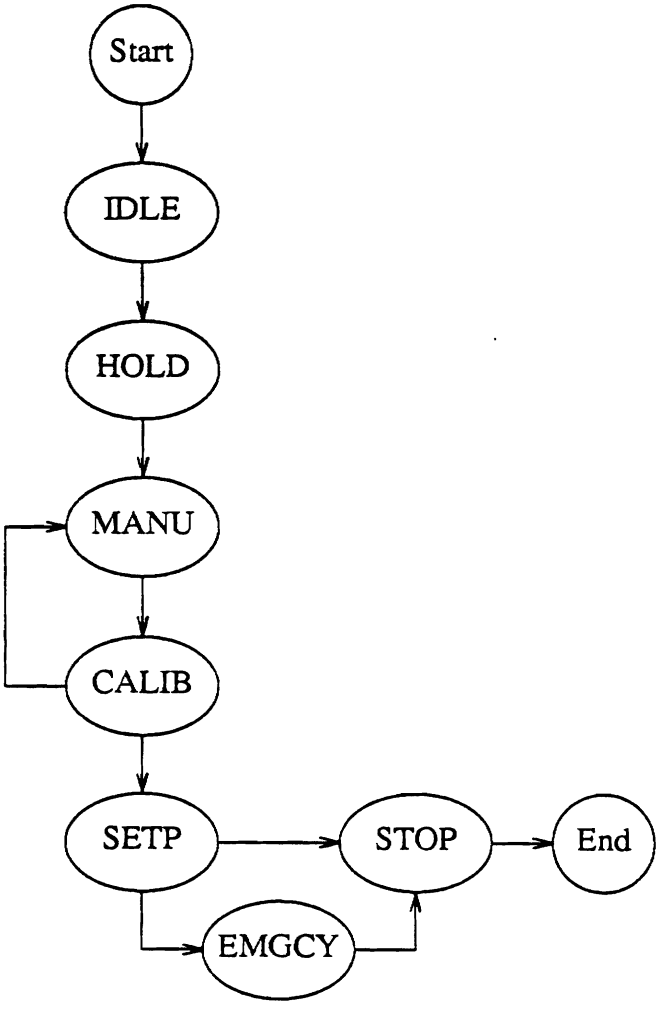


Figure 1: State Diagram of the RFMS

The interpretation of each state is summerized in the following table.

state	*action*
IDLE	get current position and keep the power off
FREE	get current position and send current compensating for gravity
CALIB	keep incrementing joint position until zero index is observed
SETP	call jsetp() and derive encoder position and compute observed sin and cos
HOLD	turn the power on
MANU	increment desired encoder position by 4 counts either clockwise or counterclockwise

Table 3. Real-Time State

In IDLE state, the system is in the initialization process. The free state is one in which the all joints are freed and compensate only for the gravity. This state is useful when we check the gravity loading constants we compute from the dynamics equations. In MANU state, the joints can be controlled manually in order to position the manipulator. The state CALIB indicates that the joints are going through a calibration procedure by looking for the zero indeces while making incremental moves. The state SETP is entered once the calibration is finished. Finally states STOP and EMGCY represent when the joints should stop and when the joints have detected abnormal conditions and need to come to a stop, respectively.

If the system is in SETP state, another variable *state*, in file *setp.c*, determines the stage in which the trajectory generation is. The number of states correspond to the number of cases in the motion control summary in [7], plus two additional states for the stationary case when there is no next motion command and for the case when the manipulator is coming to a stop. The state diagram in *state* is given in Figure 2.

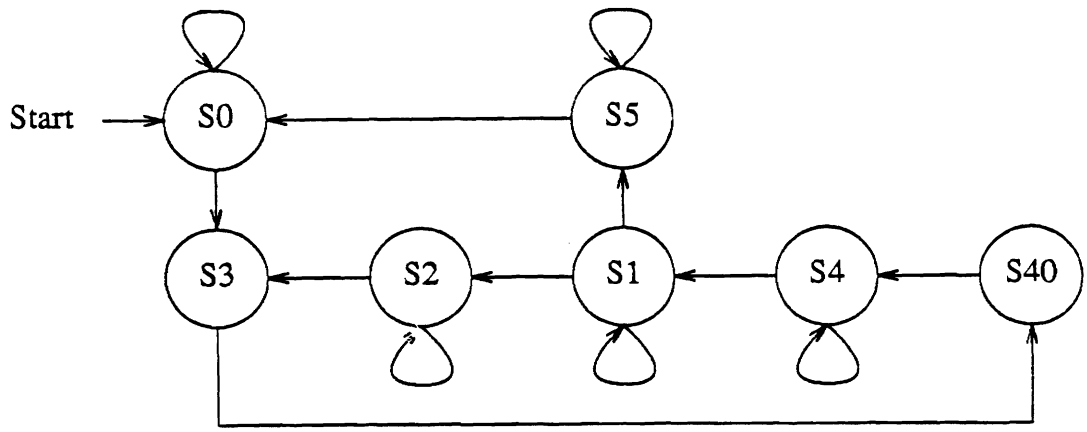


Figure 2. State Diagram of the Trajectory Generator

where the states are defined in Table 4.

state	*definition*
S0	wait for a new move request
S1	straight line motion segment
S2	one sampling period before the transition
S3	beginning period of the transition
S40	initialization of the transition
S4	during the transition
S5	end of a motion with no next move

Table 4. Definition of *state*

Whichever state the system is in, supervisor exchanges information with and for the rest of the system. Four data structures, also defined in *h/comm.h*, function as buffers holding information to be exchanged. The structure, S_MAIL, contains what to be shipped to the joints from the supervisor, J_MAIL, contains what to be shipped from the joints to the supervisor, and M_MAIL, contains information updated by the math processor for the joints. Another structure, PARCEL, contains information related to manipulator kinematics, such as dynamics and Jacobian matrices, that is provided to the joints at a low rate. In fact, each joint receives its new PARCEL every n periods, where n is the number of joints.

A few points concerning mails need to be clarified. First, there are three sets of sines and cosines returned from each joint in J_MAIL. The first two sets are expressed in terms of a sine and the sign the the cosine. They correspond to the sines and cosines

of the current and the next destination and positions, respectively. The third set is sine and cosine of the observed joint position. Secondly, The interpretation of the integer for the sign of the cosine is illustrated by the following figure where a clear bit in the corresponding position represents positive and set bit negative.

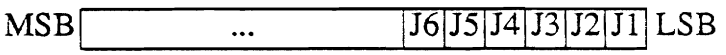


Figure 3. Bit Pattern Representing Signs of the Cosines

Thirdly, the fields in S_MAIL *Csigns* and *CsignsC* are simply the oring of the corresponding signes from all the joints.

The program *setp.c* depends on a number of functions. Functions *Dequeue()* and *Unqueue()* either take next motion request out of or and put back a fetched motion request to the motion request queue. *GetEX()* and *GetLDR()* compute the next T_6 in joint motion and Cartesian motion, respectively. All these functions are stored in file *expr.c*. Another function *InitD()*, defined in file *drive.c*, initializes the constant parts of the drive transformation for the next segment of Cartesian motion.

4.2. Joint Process

We describe the joint processes by showing how one joint works, since other joints are simply replicates of this example and differ mainly in the constants used in the programs. There are two joint independent programs, *jsetp.c* and *jrtc.c*, In addition, there is one joint dependent program in each joint directory, *jnti.c*, where *i* refers to the joint number, in each joint directory. The executable file of each joint is made up of the joint dependent and independent files. Joints share only the source code, not the executable code.

The program, *jnti.c*, contains the entry point, *main()*, that initializes joint dependent global variables and calls *rtc()* in *jrtc.c* to begin joint's operation. Two other functions in *jnti.c*, *InvKine()*, and, *InvKineC()*, compute inverse kinematics from two different set of parameters provided in supervisor mail. *ManuInc()* is used during manual mode to compute the amount of position increment. *IsReady()* determines if the joint should start calibrating or wait. This is necessary to overcome the mechanical coupling among joints during calibration. *AngToEng()* performs conversion between the encoder count and the joint angle in radians. *WriteEnc()* writes the change in its joint angle to the other joints that are coupled with this joint in order for them to make compensation. *ReadChgs()* copies the changes in other joints written in its memory into 'C' variables so as to be refered to later. Function *PID* calculates the control law.

Finally *StartS()* informs the supervisor of the completion of the joint's initialization.

The interrupt handler *Rtisr()* in *jrtc.c* is dictated by the same *rtstate* variable as on the supervisor to determine what the joint should do. It is executed at the same rate as the supervisor's interrupt service routine and computes the desired joint position in encoder count. According to the current *rtstate*, the fashion in which the desired position is computed varies. The result is passed on to the function *Servo()*, which actually performs servoing of the joint with the position computed in the previous sampling period. Currently it is either a PD or a PID control with gravity and friction compensations. Should the compliance be required, the servo error is adjusted in *Adjust()* before used to compute reacting torque.

JSetp() in *jsetp.c* computes the joint set-point. The variable *state* drives the process. There are several worth-noting points. First, all information needed by the joints is assumed to be available in the data structure MAIL, the buffer sent by the supervisor. Secondly, since in general the kinematic solution for *i*th joint requires the solution of inner *i*-1 joints, values of those joints computed in the previous sampling period are used in order for the joint not to wait for solutions to be computed, as has been mentioned previously. Finally, the joints should not have to wait for the supervisor to finish before they can start doing inverse kinematics. Instead, the T_6 is pipelined so that supervisor and joints start computing at the same time.

4.3. Math Process

The purpose of this process is to compute dynamic coefficients and Jacobian-relation matrices. Current computed joint angles are passed to this process as input and it provides gravity loadings and the compliance matrix as output to one joint per sampling period cyclically. The reason for only one joint per period is that the update of the parameters takes place at a much slower rate than the sampling rate and there is no point of sending XX The incoming information is deposited in MMAIL, the mail box for the math process from the supervisor and the output is returned in the buffer PARi, whose content applies to the joint specified in MMAIL.joint.

Again there is a real-time interrupt driven process that handles interaction with the supervisor and there is a background process that computes in an endless loop. The calculation of the dynamic coefficients is based on equations in [9], which uses Lagrangian mechanics to express dynamic terms explicitly and determines the constants in the coefficient from experiments. Procedures in [10] are used for the calculation of the compliance matrix. In order to prevent from happening the situation where the real-time interrupt service routine copies results partially updated by the interrupted process, a binary variable is used to indicate which of the two copies of a particular quantity, such as Jacobian matrix, is valid.

Currently only the Jacobian matrices from the base of the robot to the end-effector are considered. Should a tool be added to the system, modification would be necessary. Further, velocity dependent dynamic coefficients as well as the effects of a load at the robot end effector on the dynamics are not considered.

5. Postscript

One of the lessons we have learned from the RFMS project is that it is extremely difficult to program a multiprocessor system without a powerful development system. It is then predictably difficult to try to explain the system to someone wishing to understand and modify the system. To fully master the system requires a lot of time. It is however not as overwhelming to simply use the existing software to program the robot. This single document provides but a portion of the knowledge one must learn before he can feel comfortable working with the controller. It is strongly recommended that one read other related documentations and the hardware reference manual being prepared for this system for a better understanding.

Appendix A

RoboNet: A Local Area Network for Robot Systems

This documentation is about RoboNet, an Ethernet-based local area network that the we have designed and implemented. This documentation serves two purposes: as a user's guide to give robot system users a brief description on how to use the network software to transfer data from one machine to another, and as a system programmer's manual for those who maintain this network and those who are interested in customizing part of this network or extending it for other applications.

The remainder of this documentation is organized in four sections. Section two describes the network software function calls, their usage, and the results of those calls; Section three describes the network and its layers; Section four describes the logical link layer of RoboNet; and Section 5 describes the data link layer of RoboNet. Two appendixes describe how to compile the network software, where to find the files, and how to maintain the network software. For those who are interested only in using the software, we suggest that you read section two and three. For system programmers, we suggest that you read the entire documentation.

A.1. User's Guide

Currently only Grasp (VAX 11/785), Robo (Microvax II) and Intel 186/51 have RoboNet software. These machines are physically all attached to the Ethernet cable. We use RoboNet to transfer messages from the VAX machines to the Intel 186/51 and vise versa. Exchanges of messages among VAX machines are performed by software already available on these machines running Unix. The RoboNet is illustrated in Figure 1.

The VAX users can send messages to the Intel machines by invoking the network software. If the VAX user desires a particular piece of information from the Intel, he must send a message request to the Intel. The Ethernet communication on the Intel side is not accessible at the user level. A user can assume that process exists on the 186/51 that handles the messages and message request.

This appendix is an edited and revised version of the reference manual, *"RoboNet: A Local Area Network for Robot Systems"*, prepared by Pearl Pu, the Department of Computer and Information Science, the University of Pennsylvania.

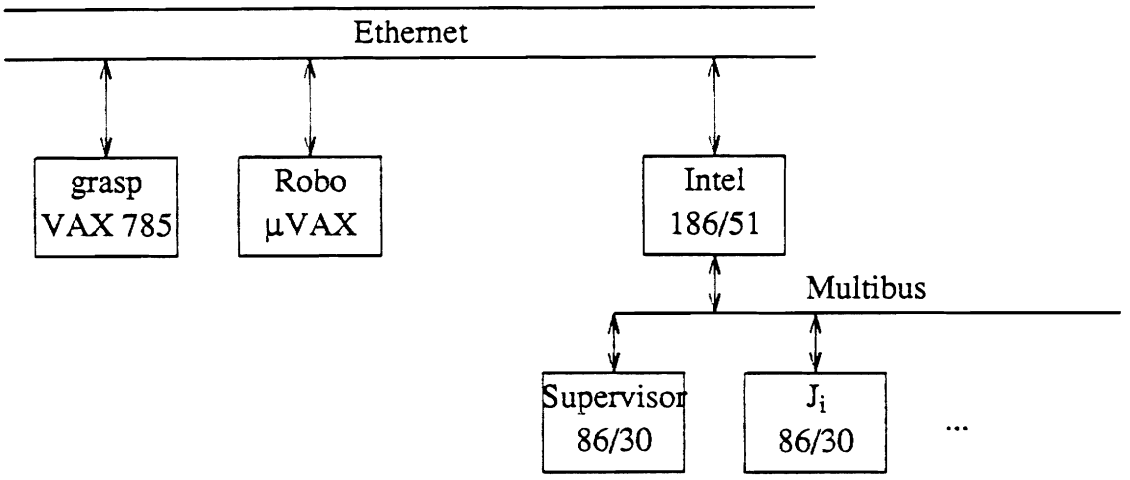


Figure A.1. RoboNet

A.1.1. The Network Software Function Calls for the VAX users

To be able to use these function calls, you have to have a Grasp, or Robo account. You have to know how to program in C. And finally you have to know what you are doing with these messages on the Intel side.

In order to use the software, you have to do the following:

1. Include *vax.h* in your program.
2. Compile your program with *vax_llc.o*.

The network software, seen at the user level, consists of the following C function calls: *Init_Comm_Link()*, *Sync()*, *Send()*, and *Recv()*.

Init_Comm_Link():

This function initializes the communication link between the host where the user is located and the Intel 186/51. The Intel Ethernet address is specified in this routine automatically as the destination address in sending and source address in receiving. Note that if the Intel address changes, one needs to notify the system programmer to modify this address accordingly.

Sync():

This routine synchronizes certain variables between the user process on the VAX and the communication process on the Intel 186/51.

Send(msgsptr, length):

msgsptr is a pointer to the buffer which contains the message you want to send, and *length* is an integer that specifies the length of the message string. Note that *length* can not be greater than *MAX_FRAME* or less than *MIN_FRAME* in *vax.h*.

Recv({type, msgsptr, length}):

type specifies what type of information you would like to receive from the Intel side. There are ten types of such information. *msgsptr* points to the buffer area where you want to receive the message. *Length* returns the actual length of message received. For certain reasons, all messages coming from Intel must be of one size. That size is specified by *R_SIZE* (receive packet size) in *vax.h.*

An example program, which illustrates how to use the network software on the VAX side, is shown in Figure 2.

```
main()
{
        int i;
        char msgs[100], buffer[R_SIZE];
        int length;

        /* fill up the msgs to be sent out */
        for(i=0; i<100; i++)
                msgs[i]= 'a' + ( i% 10);


        Init_Comm_Link();
        Sync();

        /* send the same message 10 times */
        for (i=0; i<10; i++)
                Send(msgs, sizeof(msgs));

        Recv(type2,buffer, &length);   /* receive type2 message */
        buffer[length]= NULL;
        printf("The received message is %s", buffer);
}
```

Figure 2. An Example Program

A.1.2. The Network Software on the Intel Side

Currently user support on the Intel side is entirely tailored to the need of the robot controller, which is a multiprocessor system based on Intel 86/30s with a supervisor handling message bookkeeping. All the messages sent from a user process on any of the VAXes or Microvaxes are queued up in a large buffer area on the 186/51. The

beginning address of the large buffer area is stored in the RAM of the 186/51 at 0x1ff00. The robot controller decides where each message finally goes. If the user requests a piece of information to be sent back to the VAX side, the network software on the Intel side will take care of this request.

To bring up the network process on the Intel 186/51, you have to ask the system programmer to do so. This process, once brought up, should be running continuously.

A.2. RoboNet: An Overview

RoboNet is a research effort to investigate the feasibility of designing a tailored local area network for robot systems, and stimulate further interest in this area. The current trend for robot systems is to distribute user tasks and robot tasks on different processors to increase computation speed. This introduces, however, communication problems between the users and the robot controller. To solve the communication problems, there are two solutions: one is to use existing software; the other is to design new software.

The reason we designed and implemented our own communication network stemmed from the observation that existing local area network protocols[11][12] are for large data file transfers. The header in each packet is usually complicated and the data large. If we use these protocols for transferring messages of small sizes, which is the situation with communication in robot systems, the system will be inefficient.

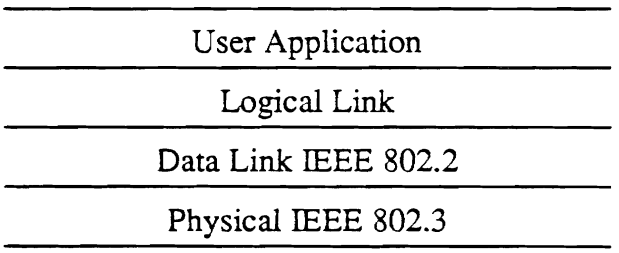


Figure A.2. Layers in RoboNet

RoboNet is designed with four layers as shown in Figure 3. The lowest layer, the physical layer, is an IEEE 802 standard. The data link layer is an IEEE compatible layer. IEEE 802.2 consists of data link and logical link layers. We only chose to implement the data link layer with the standard. It is hoped that RoboNet will be adaptable, should there be more suitable protocols. For instance, MAP (Manufacturing Automation Protocol) is another IEEE 802 standard. It is claimed that MAP is more efficient than Ethernet, and it does not degenerate when the load of the network becomes heavy. Therefore, if MAP is found to be more suitable for our application and affordable, we

can replace Ethernet by MAP without changing anything above. Another advantage of a standard implementation of the lower layers is to support heterogeneous machines. The robot system we have here contains VAX 11/785s, Microvax IIs, Intel microprocessors. In the future, it may also have Lisp machines. Since most computer manufactures now make Ethernet chips available to most of their machines, in order to install RoboNet on a machine we only have to install the upper three layers.

In the next two sections, we will describe the three lower layers. Section two is a description of the user application layer. Currently RoboNet is installed on Grasp (VAX 11/785), Robo (Microvax II), and Intel 186/51. As mentioned earlier, since this part of documentation is for system programmers, we will concentrate on not only design issues but also implementation details.

A.3. The Physical and Data Link Layers in RoboNet

As shown in Figure 3, the physical layer is the IEEE 802.3 (Ethernet) standard. On the VAX machines (VAXes, Microvaxes), this layer comes with the machine. On the Intel 186/51, there is a network coprocessor called the 82586, which is essentially an Ethernet chip that handles low level packet sending, receiving, framing, etc. For a detailed description of the 82586, refer to [13][14]. The 82586 is the coprocessor to the main CPU 80186.

The data link layer on the VAX machines uses the data link interface (DLI) from the Digital Equipment Cooperation. All packets sent out from the DLI are Ethernet packets. The DLI only takes care of damaged packets by verifying the check sum. Lost, duplicated, and out-of-order packets, however, are not taken care of.

On the Intel 186/51 microprocessor, the data link layer has to be implemented since there is no existing software. Fortunately, there is a manual[13] which describes how to program the 82586 coprocessor. We largely adopted an example from this manual as the data link layer. According to the manual, this example implements an IEEE 802.2 compatible data link layer.

Some differences between the example and our implementation are worth mentioning.

1. Multicast is not supported in our implementation.
2. The address for ISCP is found to be different in our case from that specified in the example. The correct ISCP address on our board is 0xff0 (absolute) instead of 0xfff0.
3. The interrupt from 82586 is the zeroth interrupt instead of the third.
4. Broadcast mode is disable, i.e., no broadcast messages from the Ethernet will be received.

A.4. The Logical Link Control Layer in RoboNet

We designed this layer. The principal mechanism used to prevent the network from losing, duplicating, and sending out-of-order packet is called one-bit-sliding window and positive acknowledgement with retransmission protocol[15]. We describe the characteristics of the logical link control (LLC) in RoboNet by describing the LLC packets and the algorithms used on both the Intel and the VAX sides.

A.4.1. The LLC Packet Types

SYNC:

This type of LLC packets take care of synchronization problems between the two sides. A network process runs on the Intel 186/51 continuously, whereas network processes come and go on the VAX side. Synchronization of sequence numbers is a problem if not taken care properly. We solve this problem by sending a SYNC packet every time a network process comes up on the VAX side. Upon receiving this packet, the Intel network process will initialize the sequence number.

ACK:An acknowledgement packet is sent out whenever the network process receives a good packet (i.e., with good check sum) other than an acknowledgement packet, that is, we do not acknowledge ACK packets.

REG:A regular packet will be passed to the host for processing if the sequence number matches expected frame number (specified by FrameExpected in *llc.c*). This is to ensure that no duplicated packet, from retransmission, is passed to the host.

SendReq:

A packet of this type can only be sent out from the VAX machines. This type of packet will cause a message to be sent out from the Intel to the network process on the VAX. For instance, a SendReq packet with T6 specified in the first byte will cause the T6 matrix, which is stored and kept updated on the Intel 186/51, to be sent to the VAX. This way, the robot system users can be updated with information from the Intel machines.

A.4.2. The Algorithm for the LLC on the VAX Side

procedure Send(type, msgsptr, length):
/ type: one of (ACK, REG, SYNC, SendReq)*
msgsptr: points to data to be sent
length: the length of message
Functionality: this routine prepares a LLC header for each
message pointed by msgsptr by adding the type, sequence fields,
then sends out the message. If an acknowledgement does not

```
  arrive within the timeout period, this routine will send out
  again the same message. It keeps doing so until either an ack
  arrives, or exceeds the allowed trial limit (maxtimeout).
*/

var f:frame;

if (type== SendReq)
        sendreq= TRUE;
f.type = type;                  /* specify packet type */
f.seq = NextFrameToSend;        /* append sequence number  */
f.data = msgsptr;

Acked= FALSE;
timeoutcnt=0;

/* keep trying if no ack, and # of tries has not exceeded the limit */

while( timeoutcnt < maxtimeout AND Acked== FALSE) do
        begin
                sendf(f);       /* transmit a frame */
                Timeout=FALSE;
                StartTimer();
                Recv_Ack;  /* timer can timeout in this routine */
        end;

if (timeoutcnt >= maxtimeout)
        write("Error: a frame is lost.");
Inc(NextFrameToSend); /* invert sender seq number */

end; /* end of Send */

procedure Recv_Ack():
/* Functionality: this routine waits for an acknowledgement to
  arrive from the other side. If timer times out, it will stop
  waiting and return to Send, which will resend the same message
  If an ack comes, it will set the flag to indicate so.
*/
```

```
var r : frame;      /* place to put received frame */

While (Acked== FALSE AND Timeout== FALSE) do
      begin
            wait(event); /*note: timer can timeout while waiting */
            if (event== FrameArrival AND r.seq== NextFrameToSend)
                  Acked== TRUE;

                  /* if the packet sent out was a sendreq,
                   * then acknowledge packet contains info. */

                  if (sendreq == TRUE )
                        To_Host(r);    /* pass message to host */
                  Inc(FrameExpected);
      end;

end; /* end of Recv_Ack */

procedure Isr_Timer():
/* Functionality: this routine will be called when the timer times out.
*/
Timeout=TRUE;
timeoutcnt= timeoutcnt+1;

end;  /* end of Isr_Timer */

/* type specifies what type of information to be sent back
  msgsptr returns the address of received message
  length returns the length of received message
  Functionality: Receiving a message is similar to sending a message.
  The requested message is sent back from the Intel in the Acknowledge
  packet. This is called piggybacking.
*/

var   req : frame;

req.data[0] = type;      /* specify what information to receive */
Send(SendReq, req, sizeof(req));      /* send a request frame */
```

```
end;  /* end of Recv */

procedure Sync():
/* Functionality: This routine sends out a packet to synchronize
   sequence numbers on both VAX and Intel side.
*/

var f:frame;

Send(SYNC, f, sizeof(f));

end;  /* end of SYNC. */
```

A.4.3. The Algorithm for the LLC on the Intel Side

```
procedure Recv_Frame(f):
/* f points the received frame
   Functionality: This procedure is invoked when 82586 receives a frame
   and issues an interrupt to CPU. It does different things according
   to the type of messages it received.
*/

case f.type

        ACK:  /* there will be no ACK frame on the Intel side */

        REG:        Send_Ack (f.seq);
              if (f.seq == FrameExpected)
                   putf(f.data);        /* put f in big buffer */
                    Inc(FrameExpected);    /* invert seq */

        SYNC: Send_Ack( f.seq );
            FrameExpected=0;  /* reinitialize */
            NextFrameToSend;  /* reinitialize */

        SendReq:  Ans_Send_Req();    /* answer send request */
                if (f.seq == FrameExpected )
                 Inc(FrameExpected);
```

```
end;  /* of case */

end;  /* of Recv_Frame */

procedure Send_Ack( seq ):
/* Functionality: this routine sends out an acknowledgement packet.
*/
var f: frame;

f.type= ACK;
f.seq= seq;
sendf (f);     /* transmit a frame */

end; /* of Send_Ack */

procedure Ans_Send_Req (seq);
/* Functionality: this routine piggyback the requested information
   in the acknowledgement packet.
*/

var f : frame;

f.type= ACK;
f.seq= seq;
f.data = getf(data);
sendf(f);      /* transmit a frame */

end; /* of Ans_Send_Req  */
```

A.5. Miscellaneous

The data link layer for the 186/51 is contained in file *dld.c*. The packet size from the Intel controller to the VAX can be changed by modifying constant *R_SIZE* in *vax.h*, in *llc.h*, and the field in *so_addr.choose_addr.dli_eaddr.dli_protype* in *vax_llc.c*. If you get errors like "ERROR: enable toget CB, TBD, or FD", you should consider to increase the size of the CB, or TBD, or FD queues by changing the *CB_CNT*, *TBD_CNT*, or *FD_CNT* in *dld_llc.h*.

Appendix B

Use of 8086 Cross Compiler Under Unix

B.1. Introduction

This document is interesting to those who intend to program an 8086/87-based single board computer under a VAX/Unix environment. The compiler introduced here was initially obtained from MIT Laboratory for Computer Science; however, it was written for an IBM-PC/MS-DOS environment. Modification to this compiler is mostly done to the I/O library and math library. In addition, Intel's iSDM (System Debug Monitor) is incorporated to the system to allow both down-loading of users' programs and debugging of them. Efforts have been made to optimize the intermediate assembly programs generated by the compiler so that a 15 to 30 percent better performance can be achieved after running the optimizer.

This document serves as a users' manual of the cross compiler without elaborating on the details. It assumes a user to have experience with C language and Unix. Knowledge of 8086/87 assembly language is necessary for debugging a program.

Throughout the discussion, host computer refers to the one where you develop your programs. The target computer is the 8086-based single board computer. Unix C compiler is simply called compiler and the cross compiler is explicitly qualified.

Running a C program consists of several steps. First, you should properly connect the hardware. The search path of your account should be set up correctly so that you can access the library files. The compilation of your C program using cross compiler follows similar syntax as to those of the C compiler. Before running an executable file, it should be down loaded to the target computer. Finally, you can run your program with the help of Intel's System Debug Monitor (referred to as SDM from now on).

B.2. Cable Hook-up

Your interface to both Unix and the target computer is all done from a single terminal. Normally, your terminal acts just like a regular Unix terminal and the target computer is simply another tty to the same host computer. You should connect your terminal to the a tty line and the serial port of the target to another tty line, both using standard RS232. After the lines are connected and power plugged in, turn on the switch of the the target system and initialize its line to Unix by

```
% stty 9600 raw -echo > /dev/ttyxy
```

where xy is the target's tty number.

B.3. Down Loading the Loader via SDM

Setting up your path on Unix correctly is important because your program need to find the libraries and you need to access several executable files. The directory of these files is machine dependent, but on Upenn-GRASP, the following in your .cshrc or .tcshrc is adequate:

set path=($path /usr/users/hz/c86/lib86)

If you are a shell user, use in .profile

PATH = $PATH:/usr/users/hz/c86/lib86

export PATH

Initiate the communication with the target by kermit function of Unix which changes your Unix terminal to a virtual terminal of the target. Kermit is invoked by the following command:

% kermit clb /dev/ttyxy 9600

You are then communicating to the target through the SDM from this point on. The SDM responds with the following message followed by either a dot (.) or asterisks(*), the latter indicating that SDM has not been booted and you are talking to it for the first time.

iSDM 86 Monitor Vx.y
Copyright 1983 Intel Corporation

To boot, type capital U and you will see the monitor respond with a dot indicating it has been booted. To exit kermit thereby exiting SDM upon completion of your job, type ^ followed by a letter c and message "C-kermit Disconnected" will be printed.

Although you could use SDM to down load your application program, the slow loading speed prohibits development of any large program. Alternatively, a fast loader is available to directly read your program from serial port and store it into memory without going through SDM. The idea is then to load the fast loader with SDM and to load your program with the fast loader. To load the fast loader, type:

% ldld

You will then asked if the tty of the target is the right one such as

ttyh3? (y/n)

You should answer accordingly. The loaded data and the corresponding addresses will echo on the screen. This fast loader is invoked later by the dl command to load the

application program.

B.4. Cross Compiler

As a C programmer, you may be used to writing programs under Unix and not aware of what is C and what is Unix. Therefore, it is important that you read through this document before attempting to write any C program. Basically, C is a high level language that allows you to express your algorithms in terms of C functions, whereas Unix is an operating system which provides C with an environment. Many things you use in the form of function calls are intrinsic to Unix, such as multi-processes, file systems, and I/O interface. When your program is intended for an 8086/87, many utilities on Unix are no longer available on your target board. For example, you can not open files or write to a file. Any library with which your program is linked must be created for 8086/87.

Theoretically, the language definition of the C cross compiler is 100% compatible with Unix C, i.e., all variable types, data structures, operations, type specifications, etc. follow the conventions in [16]. However, there are major differences between this compiler and Unix C compiler in the Unix interface and I/O libraries. In fact, the only system calls you can make are limited to those of standard I/O (see in Appendix A), although they may expand in the future. The reason for not implementing them is obviously that your single board computer does not contain a sophisticated operating system which actually provides these system functions. Our thought on I/O library support was that a total compatibility would require a major undertaking which may not be necessary although not impossible.

The options accepted by the compiler are the following:

-P	run only the C preprocessor (cpp) and leave the result in prog.i, where prog.c was the input file.
-S	do not run the assembler, leaving the assembly language output file in prog.a86, where prog.c was the input file.
-c	compile, assemble, but do not create a .com file, leaving binary file in prog.b, where pr.c was the input file.
-o name	changes the name of the generated default a.abs file to "name.abs".
-lm	links the program with the mathematics library
-lr	links the program with the RFMS library
-Ilib	specifies a directory to be searched when processing #include statements during preprocessor stage.

To cross compile your programs for the 8086/87 target system, use the shell script cc86 as

% cc86 [options] ...file ...

Unless -o option is specified, the default name of the output is a.abs, instead of a.out, where abs stands for absolute file. It has a format understandable by the fast loader and, apparently, it can not be executed on the host computer. The input to cc86 can be more than one file; it can be a combination of assembly programs, object files, and C programs. There are two standard libraries: I/O library, which is always linked with your programs, and the math library. Read Appendix B for the math functions provided by the math library.

As usual, there are three parts to this cross compiler: a compiler that produces assembly programs from input C programs, a assembler that reads the output of the compiler and the input assembly programs and assembles them to the object files, and a linker that links everything together. Unfortunately, the intermediate assembly language, A86, is not standard ASM-8086 assembly language but a hybrid between ASM-8086 and VAX-11 assemblers; nor is it equivalent to ASM-8086 particularly in its instructions dealing with data allocation and the floating point stack. Therefore, if you need to write assembly programs, the best you could do is using -S option of the cross compiler to generate sample assembly programs and figure your way out, with the help of 8086/87 and VAX-11 literatures [17][18][19][20]. Appendix C contains a table of encodings of 8087 stack arithmetic instructions, which may be useful when you need to program 8087 and would like to achieve efficiency.

Because of the nature of the program execution, the main program can no longer have arguments argc and argv, which are usually handled by the operating system. Also be warned that you are at your own risk if you do not initialize variables, local or global. Your target computer does not do everything the Unix does such as initializing memory. Failure to comply to this may result in meaningless outcomes. We have also found that the cross compiler can not handle functions which return a float; you must define these functions to return a double. Further, when a function is declared double, it must have a return statement to avoid underflow of the float stack on 8087. Finally, an integer variable on 8086 is 16 bits long rather than 32 as on VAX and a double is eight bytes.

B.5. Down Loading Your Application

The next step is to load your program to the memory of the target. The default location of the starting address of your program is at hex 1000 or 4 kilobytes from the beginning. This information is useful later when you debug your program. To load the program, simply type:

% dl <abs file>

The down loading speed is about one kilobytes per second, or 9600 baud. You may examine the size of your program to figure out how long a down loading takes.

The location of the code segment and data segment can be at any 16-byte boundary by changing two constants in the down loading program. Currently, the memory format of the target is set to the following diagram:

```
---------------------- 0x0000

  Reserved for SDM

---------------------- 0x0900
    Fast Loader
---------------------- 0x1000 CS (code segment)

       Code

---------------------- 0x10000 DS, SS, ES

   Data + Stack

---------------------- 0x1ff00 = top of the stack
                SP (stack pointer) = 0xff00
---------------------- End of 128 K Ram
```

The size of your programs is limited to almost an 8086 segment and can be as large as 60 kilobytes. Data and stack may take another 64K segment less 256. The sizes are examined by the linker and warnings are issued when the actual sizes exceed or approach the limits.

B.6. SDM - System Debug Monitor

SDM is an assembly language level debugger that offers such features as disassembling code, single step, changing register and memory contents, break point, and displaying register and memory contents. You can monitor your program on the target directly from your Unix terminal with the help of the on board SDM through kermit which changes a Unix terminal to that of your target computer. As mentioned above, this can be done by

% kermit clb /dev/ttyxy 9600

and you will also see SDM respond as before. In case it has crashed for any reason, push the reset button of the target and type capital U to reboot the system.

We will try to explain a few commands that are particularly useful in executing your program. It is strongly recommanded that you read [21] if you really want to learn how to use SDM. This section gives just a tiny subset of the rich debugging commands of SDM.

B.6.1 X Command

This command allows you to examine and modify registers.

.x

will display all the 8086 registers.

To modify a register, do

.x register = value

where value can be a hexadecimal number, another register, or an expression of the sum or difference of numbers and registers.

.xn

displays the 8087 registers and stack registers and you can change the values of stack registers by

.xst(i) = real number

where i is the stack register number from 0 through 7 and real number is represented in exponential notation such as 1.23 e-4

B.6.2 D Command

This command displays memory contents in a given data type which can be integer(i), long integer(li), long real(lr), short integer(si), short real(sr), binary code decimals(t), temporary real(tr, ten bytes), word(w), or disassembled instruction(x). Address is represented as segment:offset. The default segment is code segment(cs) and default offset is instruction pointer (ip). For example,

.14dx
displays 14 disassembled instructions from location cs:ip.
.d ds:5#16t
displays 16 decimal bytes in both hexadecimal and ASCII format, beginning at ds:5.
.5dtr 10
displays five temporary real values, beginning at cs:10 in both temporary real hexadecimal and decimal format.

B.6.3 G Command

This command instructs the monitor to begin executing your program at the current cs:ip. It can be followed by a starting address and addresses where you want to break the program. For example,
.g 7fa, 1f0:e20
will stop either at cs:7fa or 1f0:e20, whichever comes first.
.g 2d0:113, ip
tells the monitor to begin execution instructions at 2d0:113 and continue until it gets to the current cs:ip.

When the program stops at a break point, the following message is printed.
*BREAK at xxxx:yyyy

B.6.4. Bugs

As usual, there are bugs associated with SDM package. The single step feature is shaky at times when you use 8087. For example, to step through a program by G command may generate a different result from that you obtain to go all the way by G command; or when you single step, the board may not do what the next instruction says it will do, etc. We have no solutions to this and encourage you to ask Intel for help.

B.7. Miscellaneous

In lib86 directory, there exist several utility programs to convert files from one format to another.

abshex - converts an abs file to a hex file,
ldabs - converts an ld file (output of MIT compiler) to an abs file,
ldhex - converts an ld file to a hex file.

B.8. An Example

In this section, we will go through an example to demonstrate how the cross compiler and the debugger work. Suppose you have created the following program on Unix:

```
# include <math.h>
# define RAD_TO_DEG 57.29578

main()
{
        double x, y;
        int i;

        x = 0.1;
        for(i = 0; i < 10; i++) {
                y += x;
                printf("sin(%4.1f) = %f\n\r", y*RAD_TO_DEG, sin(y));
        }
}
```

First compile the program using the C compiler and test it on Unix as

```
% cc prog.c -lm
% a.out
sin( 5.7) = 0.099833
sin(11.5) = 0.198669
sin(17.2) = 0.295520
sin(22.9) = 0.389418
sin(28.6) = 0.479426
sin(34.4) = 0.564642
sin(40.1) = 0.644218
sin(45.8) = 0.717356
sin(51.6) = 0.783327
sin(57.3) = 0.841471
```

Of course, on Unix we can only test the portion of the program not dependent on the target hardware.

After making sure the program is free of errors as far as you can go on Unix, you can then cross compiler your program:

```
% cc86 prog.c -lm
```

An a.abs is created at this point for you to down load. You are then ready to try it out on your target computer. As the first step, properly connect the Unix tty (e.g. ttyh3) line to your target computer and turn on the power. A typical sequence of commands may look like:

```
% stty 9600 raw -echo > /dev/ttyh3
% kermit clb /dev/ttyxy 9600

iSDM 86 Monitor Vx.y
Copyright 1983 Intel Corporation

*** (capital U is pressed here)

.(^c)
C-Kermit Disconnected
% ldld
ttyh3 ? (y/n) y
S 0090:0000
0090:0000 00 - b8,
0090:0001 00 - 90,
0090:0002 00 - 00,
0090:0003 00 - 8e,
   .
   .
   .
0000:007F FF - 00,
0090:0080 FF -
```

```
% dl a.abs (wait approximate 5 seconds)
% kermit clb /dev/ttyxy 9600
iSDM 86 Monitor Vx.y
Copyright 1983 Intel Corporation

.x
AX = 0006 CS = 0100 IP = 0000 FL = F046 O0 D0 J0 T0 S0 Z1 A0 P1 C0
BX = 1AE3 SS = 1000 SP = 0000 BP = 0000
CX = 0000 DS = 009B SI = 0000
DX = 00D8 ES = 0000 DI = 0000
.x ip=0
.np,

0100:0000 FA          CLI -,
0100:0001 B83F13      MOV AX, 133FH        ;I = +4927-,
0100:0004 B104        MOV CL, 4

.g
sin( 5.7) = 0.099833
sin(11.5) = 0.198669
sin(17.2) = 0.295520
sin(22.9) = 0.389418
sin(28.6) = 0.479426
sin(34.4) = 0.564642
sin(40.1) = 0.644218
sin(45.8) = 0.717356
sin(51.6) = 0.783327
sin(57.3) = 0.841471

*BREAK at 0100:002B
.x
AX = 0006 CS = 0100 IP = 0020 FL = F046 O0 D0 J0 T0 S0 Z1 A0 P1 C0
BX = 1AE3 SS = 1000 SP = FF00 BP = 0000
CX = 0000 DS = 1000 SI = 0081
DX = 00D8 ES = 1000 DI = 0000
.(^c)
C-Kermit Disconnected
%
```

You are now at the end of a debugging session.

B.9. I/O Library

Only standard input and output functions are provided by the library, i.e., input to the program and output from the program can only go through your terminal. Furthermore, I/O functions are restricted to the following. Attempt to invoke any other will result in an undefined function error.

```
char getchar();
char *gets();
putchar(ch) char ch;
putw(word) int word;
puts(s) char *s;
printf(s, arg) char *s;
```

It should be pointed out that the line feed character '\n', when used to obtain a new line, must be accompanied by a carriage return '\r' in order to move the cursor back to the beginning of the next line. This second character is put out by Unix automatically so that your printing program need not use it explicitly.

B.10. Math Library

The following math functions are provided in the math library.

```
double fabs(), ldexp(), modf();
double sqrt();
double sin(), cos(), tan(), asin(), acos(), atan(), atan2();
double sc(sc_p, angle)
struct sncs *sc_p; double angle;
where sncs is
struct sncs {
        float sin;
        float cos;
};
```

B.11. 8087 Floating Point Stack Programming

The compiler does not make use of the floating point stack registers one through seven for the sake of simplicity. On the other hand, at times you may desire to achieve better efficiency by programming in A86 and taking advantage of the floating registers. Unfortunately, the A86 does not provide instructions which handle the float stack registers except for the top, it is necessary to program in 8087 machine code directly. The following table provides some of the frequently used arithmetic instructions to

manipulate on the float stack. An example is also presented to illustrate the idea and the technique.

Instructions	i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7
fadd st, s(i)	0xc0d8	0xc1d8	0xc2d8	0xc3d8	0xc4d8	0xc5d8	0xc6d8	0xc7d8
fadd st(i), st	0xc0dc	0xc1dc	0xc2dc	0xc3dc	0xc4dc	0xc5dc	0xc6dc	0xc7dc
faddp st(i), st	0xc0de	0xc1de	0xc2de	0xc3de	0xc4de	0xc5de	0xc6de	0xc7de
fsub st, st(i)	0xe0d8	0xe1d8	0xe2d8	0xe3d8	0xe4d8	0xe5d8	0xe6d8	0xe7d8
fsubr st, st(i)	0xe8d8	0xe9d8	0xead8	0xebd8	0xecd8	0xedd8	0xeed8	0xefd8
fsub st(i), st	0xe8dc	0xe9dc	0xeadc	0xebdc	0xecdc	0xeddc	0xeedc	0xefdc
fsubr st(i), st	0xe0dc	0xe1dc	0xe2dc	0xe3dc	0xe4dc	0xe5dc	0xe6dc	0xe7dc
fsubp st(i), st	0xe8de	0xe9de	0xeade	0xebde	0xecde	0xedde	0xeede	0xefde
fsubrp st(i), st	0xe0de	0xe1de	0xe2de	0xe3de	0xe4de	0xe5de	0xe6de	0xe7de
fmul st, st(i)	0xc8d8	0xc9d8	0xcad8	0xcbd8	0xccd8	0xcdd8	0xced8	0xcfd8
fmul st(i), st	0xc8dc	0xc9dc	0xcadc	0xcbdc	0xccdc	0xcddc	0xcedc	0xcfdc
fmulp st(i), st	0xc8de	0xc9de	0xcade	0xcbde	0xccde	0xcdde	0xcede	0xcfde
fdiv st, st(i)	0xf0d8	0xf1d8	0xf2d8	0xf3d8	0xf4d8	0xf5d8	0xf6d8	0xf7d8
fdivr st, st(i)	0xf8d8	0xf9d8	0xfad8	0xfbd8	0xfcd8	0xfdd8	0xfed8	0xffd8
fdiv st(i), st	0xf8dc	0xf9dc	0xfadc	0xfbdc	0xfcdc	0xfddc	0xfedc	0xffdc
fdivr st(i), st	0xf0dc	0xf1dc	0xf2dc	0xf3dc	0xf4dc	0xf5dc	0xf6dc	0xf7dc
fdivp st(i), st	0xf8de	0xf9de	0xfade	0xfbde	0xfcde	0xfdde	0xfede	0xffde
fdivrp st(i), st	0xf0de	0xf1de	0xf2de	0xf3de	0xf4de	0xf5de	0xf6de	0xf7de
fld st(i)	0xc0d9	0xc1d9	0xc2d9	0xc3d9	0xc4d9	0xc5d9	0xc6d9	0xc7d9
fxch st(i)	0xc8d9	0xc9d9	0xcad9	0xcbd9	0xccd9	0xcdd9	0xced9	0xcfd9
fst st(i)	0xd0dd	0xd1dd	0xd2dd	0xd3dd	0xd4dd	0xd5dd	0xd6dd	0xd7dd
fstp st(i)	0xd8dd	0xd9dd	0xdadd	0xdbdd	0xdcdd	0xdddd	0xdedd	0xdfdd

Table A.1. Encodings of 8087 Float Stack Arithmetic Instructions

Suppose you would like to program a partial sinus function using 8087's partial tangent call. It may look like:

```
        .globl _psin
|       double psin(x) x double; compute sinus of x in radians
_psin:  mov bx, sp
        fldd   *2(bx)
       fptan
       fwait
       .word  0xc8d8            | fmul st, st(0)
       fwait
       .word  0xc1d9            | fld   st(1)
       fwait
       .word  0xc8d8            | fmul st, st(0)
       fwait
       .word  0xc1de            | faddp       s(1), st(0)
       fsqrt
       fwait
       .word  0xf9de            | fdivp       st(1), st(0)
       ret
```

Note that every instruction must be preceded by a float wait o instruction to assure normal function of the hardware. Also, if you are serious about programming 8087, always remember to clean up the float stack before exiting a function, with the return value of the function on the stack if there is any. Pushing too many things on to the saturated float stack leads to unexpected result as the values at the bottom of the stack will not drop out as one would think.

REFERENCES

[1] *Paul, R.P. and Zhang, H.* 1985. "Design of a Robot Force/Motion Server". Proceedings of IEEE International Conference on Robotics and Automation, St.Louis, MO.

[2] *Paul, R.P., Zhang, H., Hashimoto, M., Durrant-Whyte, H., Izaguirre, A., Trinkle, J., Zhang, Y., Fuma, F., Ulrich, N., and Donham, M.* 1986. "A Distributed System for Robot Manipulator Control", Department of Computer and Information Science, the University of Pennsylvania. 1986.

[3] *Hayward, V. and Paul, R.* 1984. "Introduction to RCCL: A Robot Control C Library", Proceedings of IEEE International Conference on Robotics and Automation, Atlanta, GA.

[4] *Pu, P.* 1986. "RoboNet: A Local Area Network for Robot Systems", Department of Computer and Information Science, University of Pennsylvania.

[5] *Paul, R.P.* 1981. "Robot Manipulators: Mathematics, Programming, and Control", MIT Press.

[7] *Paul, R.P. and Zhang, H.* 1984. "Robot Motion Trajectory Specification and Generation", ISRR Proceedings , Japan.

[8] *Zhang, H. and Paul, R.P.* 1988. "A Parallel Solution to Robot Inverse Kinematics", Proceedings of IEEE International Conference on Robotics and Automation, Philadelphia, PA.

[9] *Izaguirre, A., Hashimoto, M., and Paul, R.* 1987. "A New Computational Structure for Real-time Dynamics". Proceedings of International Workshop on Robotics: Trends, Technology, and Applications, Madrid, Spain.

[10] *Paul, R. P. and Zhang, H.* 1986. "Computationally Efficient Kinematics for Manipulators with Spherical Wrists Based on the Homogeneous Transformation Representation". *International Journal of Robotics Research* 5(2):32 - 44.

[11] *Postel, J.*, 1980. "User Datagram Protocol", RFC 768, Information Sciences Institute.

[12] *Postel, J.*, 1982. "TCP-IP Implementations", Network Information Center, SRI Int.

[13] *Intel* 1985. "Local Area Networking (LAN) Component User's Manual", 230814-002, Intel Corporation.

[14] *Intel* 1984. "iSBC 186/51 COMMputer Board Hardware Reference Manual," 122136-002, Intel Corporation.

[15] *Tanenbaum, A.*, 1981. "Computer Networks", Englewood Cliffs, N.J., Prentice-Hall,

[16] *Kernighan, B.W and Ritchie, D.M.* 1978. "The C Programming Language", Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632.

[17] *Intel Corporation,* "iSBC 337 Multimodule Numeric Data Processor Hardware Reference Manual", Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

[18] *Rector, R. and Alexy, G,* "The 8086 Book", Osborne/McGraw-Hill, 630 Bancroft Way, Berkeley, California 94710.

[19] *Levy, H.M. and Eckhouse, R.H.*, "Computer Programming and Architecture", Digital Equipment Corporation, Bedford, MA 01730.

[20] *Intel Corporation,* "ASM86 Language Reference Manual", Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

[21] *Intel Corporation,* "iSDM 86 System Debug Monitor Reference Manual", Hardware Reference Manual", Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

[22] *Intel* 1982. "iSBC 86/14 and iSBC 86/30 Single Board Computer Hardware Reference Manual," 14404-002, Intel Corporation.

A.6 A Parallel Solution to Robot Inverse Kinematics

A Parallel Solution to Robot Inverse Kinematics

Hong Zhang
Richard P. Paul

Department of Computer and Information Sciences
University of Pennsylvania
Philadelphia, PA 19104

Abstract — **In this paper, we introduce an algorithm by which the inverse kinematics of a robot manipulator with closed-form solution can be computed in parallel to reduce the computational complexity roughly by a factor of *n*, the number of joints of the manipulator. Further, we study the errors introduced by the algorithm statistically to demonstrate that the algorithm is stable, well behaved and, for all practical purposes, it produces satisfactory results. Comparison with other methods employing approximation is made to show the superiority of the algorithm. Finally, we briefly describe its implementation on a multiprocessor system.**

I. INTRODUCTION

A robot task is specified in the Cartesian space, while the robot manipulator is actuated in the joint space. The inverse kinematics problem is defined as the mapping from the Cartesian space to the joint space,

$$(\mathbf{R},\mathbf{p}) \rightarrow \boldsymbol{\theta} \tag{1}$$

i.e., given the position, **p** and orientation, **R**, of the end effector of the robot manipulator, solve for the joint coordinates which will result in the desired position and orientation.

Typically, a robot manipulator is designed as a six-joint mechanical linkage with the last three joints intersecting each other, forming a wrist. In this case, it has been repeatedly shown that a closed form solution exists to the inverse kinematics problem. The value of joint i can be expressed in terms of the end effector position and orientation and values of prior i-1 joints. If we represent the manipulator position and orientation by a homogeneous transformation called $\mathbf{T}_6$, we have the following general equation,

$$\boldsymbol{\theta} = \Lambda(\mathbf{T}_6) \tag{2}$$

This material is based on work supported by the National Science Foundation under Grant No. ECS-8411879. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

or, in a more detailed fashion, as

$$\theta_i = \Lambda_i(\mathbf{T}_6, \theta_{i-1}, \cdots, \theta_1) \tag{3}$$

for $i=1$ through 6.

The inverse kinematics as solved above requires a number of multiplications and divisions, additions and subtractions, and trigonometry functions. Depending upon the processor on which the computations are performed and the way programming is done (e.g., in assembler or in a high level language), its computational complexity lies typically between two to ten milliseconds. Using the PUMA 260 robot manipulator as an example, the inverse kinematics requires 40 multiplications and divisions, 21 additions and subtractions, 1 square root, eight inverse trigonometry calls, and four sets of sine/cosine calls [1]. If this is to be programmed in 'C' and computed on an 8086/8087 based system in floating point with the overhead of the compiler considered, it will take about seven milliseconds [2].

Of course a manipulator control system must compute more than just the inverse kinematics. It must, for example, derive the next $\mathbf{T}_6$ from the task specification before inverse kinematics can be solved. In case of a Cartesian motion, this may involve a number of matrix operations. Adding the time required to compute $\mathbf{T}_6$, it takes well over ten milliseconds to derive each point along a motion trajectory.

Robot control is a real-time process, the output of which provides to the joint servos a sequence of positions, called *setpoints*, that are evenly spaced in time and separated by one sampling period, Δt, which must be small enough to ensure a smooth and stable motion. If a robot task requires a straight line motion, inverse kinematics must be performed periodically. However, it is usually impossible to compute the inverse kinematics at the same rate as the sampling rate due to its computational complexity. Instead, a new setpoint is generated only every $T^G_{upd} > \Delta t$, and one often resorts to one of two solutions to fill the missing setpoints between two updates: either generating the setpoints off line or employing numerical methods such as polynomial interpolation. The off-line programming would be fine if the trajectory were not to be modified while being followed, which is not the case in many tasks such as compliance or a sensor-driven motion. Therefore, off-line programming is of value only in simple pick-and-place operations with fixed and known task geometry.

The numerical interpolation is widely applied to reduce the computational load, but it ought to be used with an understanding of its consequences. First, since in general a linear motion in Cartesian space requires a non-linear motion in joint space, the interpolated intermediate points will only generate an approximately straight line motion in the Cartesian space. Suppose that between two computed positions of joint i $\theta_i(t_k)$ at t_k and $\theta_i(t_{k+1})$ at t_{k+1}, nine more points, one every Δt, are interpolated linearly, the result is illustrated in Fig. 1, where the arc is the trajectory that the joint must follow to generate a straight line Cartesian motion and the line segment is the result of the interpolation of two computed joint positions The difference between the arc and straight line would lead to errors in the Cartesian space. In general, this

piece-wise linear motion of joint i contributes to errors in all Cartesian directions.

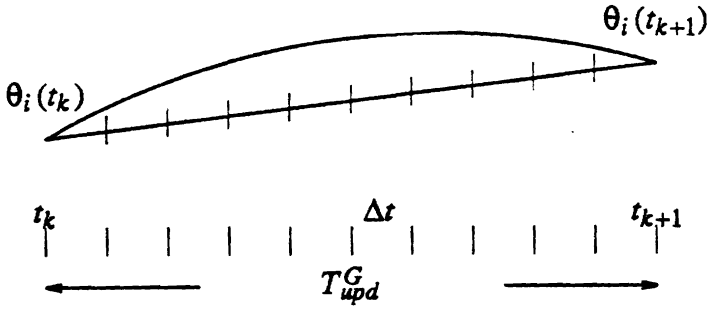


Fig. 1. Error due to joint space interpolation

The second and more important reason why one would like to minimize T^G_{upd} is related to robot tasks such as tracking, in which the trajectory is modified while being executed. If the rate at which the points are generated falls short of that at which trajectory is modified, tracking inaccuracy will result. This would be the case where the changes occur faster than the system can compute, even though mechanically the manipulator may still be able to react. Fig. 2 illustrates a bad case of tracking error when the interpolating process tells the joint to go one way (solid straight line) while the modification tells it to go the other (dotted curve).

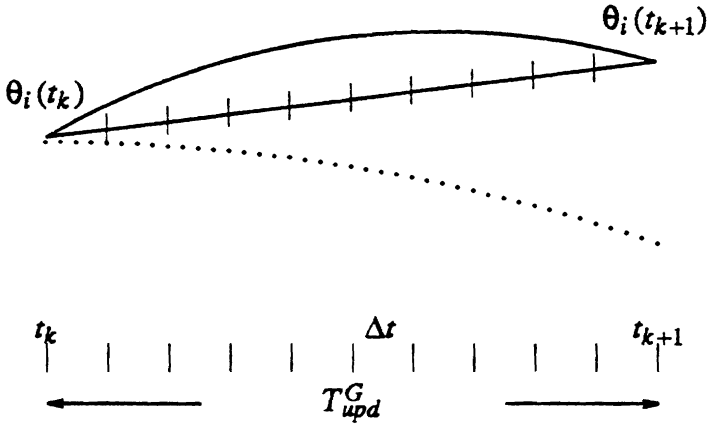


Fig. 2. Error due to path modification during execution

Such a characteristic associated with numerical interpolation limits the range of applications a robot manipulator can perform, creating a situation where the controller limits the performance of the manipulator. In general, if the update takes place every T^G_{upd}, the modification to motion, whether by a tracking camera or by contact with the environment during compliance, must take place at a rate slower than $1/T^G_{upd}$ in order for the modification to be meaningful.

The key question now becomes how we can minimize T^G_{upd}, ideally, to the level when the manipulator can still respond. If we break the process of generating a new setpoint into two steps, first computing the next desired Cartesian position $\mathbf{T}_6$, then performing inverse

kinematics, we can reduce T^G_{upd} by decreasing the time spent on solving the inverse kinematics, denoted here as T^K_{upd}. In the next few sections, we will describe a parallel solution to the inverse kinematics problem to achieve the above goal, discuss the implications of the solution, and show an actual implementation of the solution on a multi-processor system.

II. PROPOSED PARALLEL INVERSE KINEMATICS (PIK)

The advance in micro-electronics technology has made a multi-processor system a solution to many problems that were not feasible before. The use of multi-processors is also justified economically by the fact a uni-processor system usually costs more than the multiprocessor system with the same throughput. Though the technology has been widely applied to robot control problems, we have yet to see the inverse kinematics problem be solved with a multi-processor system. As argued above, if the time spent on solving inverse kinematics is reduced, the entire system can be driven and respond at a higher rate, thereby improving its performance.

The inverse kinematics is generally viewed as a serial process, since it is solved from the first joint of the manipulator up to the last joint one after another due to the θ_i's dependency on the prior i-1 joints. Mathematically for a six-joint robot manipulator this can be expressed as

$$\begin{aligned}
\theta_1 &= \Lambda_1(\mathbf{T}_6) \\
\theta_2 &= \Lambda_2(\mathbf{T}_6, \theta_1) \\
\theta_3 &= \Lambda_3(\mathbf{T}_6, \theta_1, \theta_2) \\
\theta_4 &= \Lambda_4(\mathbf{T}_6, \theta_1, \theta_2, \theta_3) \\
\theta_5 &= \Lambda_5(\mathbf{T}_6, \theta_1, \cdots, \theta_4) \\
\theta_6 &= \Lambda_6(\mathbf{T}_6, \theta_1, \cdots, \theta_5)
\end{aligned} \tag{4}$$

Notice that since the process of generating setpoints is repetitive, there is a time implicitly associated in Eq(4). To be explicitly in time, we have in general,

$$\theta_i(t_k) = \Lambda_i(\mathbf{T}_6(t_k), \theta_1(t_k), \cdots, \theta_{i-1}(t_k)) \tag{5}$$

Joint three, for example, cannot be computed until joints one and two are finished to make $\theta_1(t_k)$ and $\theta_2(t_k)$ available.

This serial process, however, can be *parallelized* approximately by recognizing the dynamic nature of the process and the continuity of joint trajectories. Rewrite Eq(5) to the general form:

$$\tilde{\theta}_i(t_k) = \Lambda_i(\mathbf{T}_6(t_k), \theta_1(t_{k-1}), \cdots, \theta_{i-1}(t_{k-1})) \tag{6}$$

Inverse kinematics for joint three, for example, becomes

$$\tilde{\theta}_3(t_k) = \Lambda_3(\mathbf{T}_6(t_k), \theta_1(t_{k-1}), \theta_2(t_{k-1})) \tag{7}$$

In effect, the i th joint uses the values of the prior i-1 joints in the previous sampling period. Obviously Eq (7) will not generate the same values as the original due to the approximation, one can however argue that since the difference in time between two neighboring points is small, one expects only a *reasonably* small error in θ_3. (We will better quantify *reasonably* in the next section.) What comes out of the conversion process is a set of six parallel processes so that when one processor is assigned to each joint, all six processors can start computing the inverse kinematics of their respective joint angles at the same time. Assuming when inverse kinematics is solved serially, the computational complexity is

$$T_{serial} = \sum_{i=1}^{6} T_i \tag{8}$$

where T_i is the computation time of Eq(4), then the complexity of the parallel inverse kinematics is *roughly*

$$T_{para} = \max\left\{T_1, \cdots, T_6\right\} \tag{9}$$

We say roughly since T_i may change when we change a serial process to a parallel process, as certain intermediate results used in a serial process no longer exist in the parallel process.

To illustrate the method, we use a simple two-link manipulator in Fig. 3. It has two revolute joints and both links are of unit length.

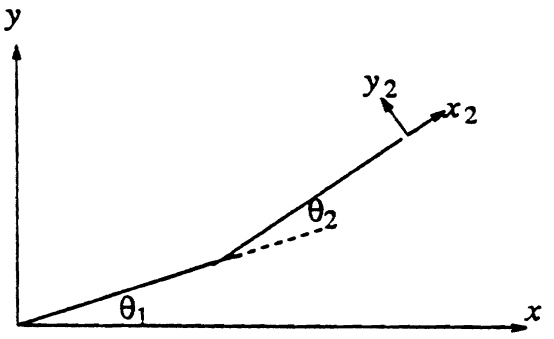


Fig. 3. A simple manipulator

Suppose we are interested in positioning the manipulator arbitrarily on the x–y plane, the direct kinematics has the form:

$$\begin{aligned} x &= \cos(\theta_1 + \theta_2) + \cos(\theta_1) \\ y &= \sin(\theta_1 + \theta_2) + \sin(\theta_1) \end{aligned} \tag{10}$$

And the inverse kinematics has the form:

$$\theta_1 = \tan^{-1}\frac{y}{x} + \cos^{-1}\frac{\sqrt{x^2+y^2}}{2}$$
$$\theta_2 = \tan^{-1}\frac{y-\sin(\theta_1)}{x-\cos(\theta_1)} - \theta_1 \tag{11}$$

Using the parallel algorithm in Eq(6), we employ two processors. On the first processor, we compute first half of Eq.(11), which can be rewritten with time variable expressed explicitly:

$$\tilde{\theta}_1(t_k) = \tan^{-1}\frac{y(t_k)}{x(t_k)} + \cos^{-1}\frac{\sqrt{x^2(t_k)+y^2(t_k)}}{2} \tag{12}$$

and on the second processor, we compute,

$$\tilde{\theta}_2(t_k) = \tan^{-1}\frac{y(t_k)-\sin(\theta_1(t_{k-1}))}{x(t_k)-\cos(\theta_1(t_{k-1}))} - \theta_1(t_{k-1}) \tag{13}$$

Notice in Eq(13) that θ_2 at t_k is solved in terms of the current Cartesian position (x, y) and the θ_1 at time t_{k-1}.

The difference between Eq(8) and Eq(9) in terms of computational complexity can be considerable. In the ideal case when all T_i are equal, the complexity would be reduced by a factor of six, the number of joints of the robot. More realistically, we use PUMA 260 as an example to calculate the difference in computational complexity on a per-processor basis. The costs of various operations listed in Table I are determined based on the actual times of execution of those operations when executed on an Intel 8087 floating point processor, taking into account the overhead for fetching and storing data.

operation	*time of exec*
adds/subs	40μs
multiply/divide	53μs
inverse trig	350μs
sin/cos pair	360μs
square root	100μs

Table I. Time of Execution of Different Operations

Table II lists the computational complexity of the serial inverse kinematics in terms of the weighted cost. The weighted cost of an operation is one-tenth of the actual time of execution in milliseconds. The complexity of the entire inverse kinematic equals the sum of the complexities of the individual joints if the inverse kinematics is solved serially as in Eq (4).

	+ or -	* or /	$trig^{-1}$	sin/cos	sqrt	weighted cost
θ_1	1	2	2	1	0	120.6
θ_2	5	7	2	1	1	173.1
θ_3	5	8	1	0	0	97.4
θ_4	3	7	1	0	0	84.1
θ_5	2	5	1	0	0	69.5
θ_6	7	15	1	0	0	142.5
Σ	21	40	8	2	1	687.2

Table II. Computational Complexity When Executed in Serial

Table III lists the complexity when the inverse kinematics is solved in parallel as in Eq(6). In this case, the complexity of each joint solution may vary from the serial case, for as mentioned the joints can no longer share intermediate results, but overall the complexity of the inverse kinematics is reduced by a factor of about four, from a relative cost of 687.2 to 173.7, as we expect.

	+ or -	* or /	$trig^{-1}$	sin/cos	sqrt	weighted cost
θ_1	1	2	2	1	0	120.6
θ_2	5	7	2	1	1	173.1
θ_3	4	6	1	0	0	82.8
θ_4	5	11	1	0	0	113.3
θ_5	7	15	1	0	0	142.5
θ_6	9	19	1	0	0	171.7
max(= θ_2)	5	7	2	1	1	173.1

Table III. Computational Complexity When Executed in Parallel

The reduction in computational complexity, however, is not obtained without paying a price. First, a multiprocessor or parallel machine is more difficult to program, and there is overhead involved in data communication and system synchronization. Second, errors are introduced as the result of approximating the current joint angles by the previous ones. The fact that the update period T^G_{upd} is usually small assures to certain extent that the approximation we use will not yield a trajectory substantially different from the accurate one. However, it is

necessary to evaluate the error in both more qualitative and quantitative terms.

III. ERROR ANALYSIS

In general, the manipulator moves along a straight line from the initial to the final configuration defined by $(\mathbf{R}_i, \mathbf{p}_i)$ and $(\mathbf{R}_f, \mathbf{p}_f)$, respectively. To simplify the analysis, we ignore transition between path segments and assume that the position changes linearly with time by evaluating the equation

$$\mathbf{p} = \mathbf{p}_i + h^* (\mathbf{p}_f - \mathbf{p}_i) \tag{14}$$

where the motion parameter h linear with time varies from 0 to 1 to bring the manipulator from initial to final position. The orientation change in a Cartesian motion can be accomplished in a number of ways, as a linear rotation in space cannot be uniquely defined. In one commonly used approach, the orientation change, $\mathbf{R}$, takes place about the unit vector $\mathbf{n}$ which remains constant before and after the change. The vector can be defined by two Euler angles, ϕ and ψ, as

$$\mathbf{n} = C_\phi S_\psi \mathbf{i} + S_\phi S_\psi \mathbf{j} + C_\psi \mathbf{k} \tag{15}$$

and the rotation change by an angle θ. Those three variables can be found by solving the equation [5]

$$\mathbf{R}(\mathbf{n}, \theta) = \mathbf{R}_i^{-1} \mathbf{R}_f \tag{16}$$

which, when multiplied by the initial orientation, produces the final orientation. Similar to position change, we vary the amount of rotation successively by multiplying θ with motion parameter h to bring the manipulator from the initial to the final orientation.

A. Error Definition

Errors are defined as the difference between the nominal position and orientation of the manipulator and the position and orientation as the result of the approximation in our parallel inverse kinematic solution. As position error is decoupled from orientation error as far as inverse kinematics, they are considered separately.

Similar to [3], the deviation of the position vector, $\mathbf{e}_p$ is computed by

$$\mathbf{e}_p (t) = \mathbf{p}(t) - \mathbf{p}_p (t) \tag{17}$$

where $\mathbf{p}$ is the position vector for a given time and $\mathbf{p}_p$ the position vector corresponding to the joint angles that are computed with the parallel scheme from $\mathbf{p}$, and the norm of $\mathbf{e}$

$$\delta_p (t) = \| \mathbf{e}_p \| \tag{18}$$

we define as the position error.

It is less clear what we should define as the orientation error; one may favor one way or another depending upon the application. Here, we outline two conventions used in defining

orientation errors.

One popular approach [3] defines the error as the absolute value of the difference between the desired amount of rotation and the actual amount of rotation when inverse kinematics is computed in parallel, i.e.,

$$\begin{aligned}\delta_r(t) &= |\,angle\,(\mathbf{R}(\mathbf{n}, h^*\theta)) - angle\,(\mathbf{R}_a(t))\,| \\ &= |\,h^*\theta - angle\,(\mathbf{R}_a(t))\,|\end{aligned} \tag{19}$$

where the function *angle* returns the actual amount of rotation about the unit vector $\mathbf{n}$.

Another convention to define orientation error is based on Cartesian coordinates or differential rotations about the principle axes [4]. Since the actual rotation $\mathbf{R}_a(t)$ is close to the desired rotation $\mathbf{R}(t)$, the matrix multiply

$$d\mathbf{R} = \mathbf{R}^{-1}(t)\mathbf{R}_a(t) \tag{20}$$

is a valid differential rotation with the general form

$$d\mathbf{R} = \begin{bmatrix} 1 & \delta z & -\delta y \\ -\delta z & 1 & \delta x \\ \delta y & -\delta x & 1 \end{bmatrix} \tag{21}$$

where δx, δy, and δz represent errors of rotation about x, y, and z axes. We now can define the orientation error as the norm of the vector $(\delta x, \delta y, \delta z)$

$$\delta_r(t) = \sqrt{\delta x^2 + \delta y^2 + \delta z^2} \tag{22}$$

B. Statistical Models of Errors

Now that the error criteria have been specified for any given moment along a trajectory, it is yet another problem how to study the behavior of the error in order to reach conclusions valid over the entire robot workspace. Unfortunately, it is extremely difficult to come up with an analytical expression for Eq(18) or Eq(22) even for a simple manipulator. Therefore, we cannot derive the error analytically first and then base our evaluation of the method on the analytical form of the error.

To establish the fact the method produces results acceptable from a practical point of view, we can statistically investigate the errors due to the approximation by showing their characteristics such as bounds and averages. If our domain of trajectories covers the entire robot workspace and if the method is well behaved statistically even in the worse case, then we can be confident that the method is applicable in practice.

While we conduct such a statistical study, we should also consider the effect of time parameters on the error. Intuitively, for example, the shorter T^G_{upd} and slower the motion is, the smaller the errors, since the previous joint positions more closely approximate the current ones.

Given the fact that we deal with the manipulators with three positioning joints and an intersecting wrist, we can build the statistic model for position error as follows:

Let $\mathbf{p}_i$ and $\mathbf{p}_d$ be the initial and destination position vectors of a motion segment. The Cartesian trajectory planner G produces successive $\mathbf{p}(t_k)$ as

$$\mathbf{p}(t_k) = G(\mathbf{p}_i, \mathbf{p}_d, t_k) \tag{23}$$

and, if we apply Eq(6), the first three joints solve for their joint angles from the position vector $\mathbf{p}(t_k)$ by

$$\begin{aligned}
\tilde{\theta}_1(t_k) &= \Lambda_1(\mathbf{p}(t_k)) \\
\tilde{\theta}_2(t_k) &= \Lambda_2(\mathbf{p}(t_k), \theta_1(t_{k-1})) \\
\tilde{\theta}_3(t_k) &= \Lambda_3(\mathbf{p}(t_k), \theta_1(t_{k-1}), \theta_2(t_{k-1}))
\end{aligned} \tag{24}$$

The resultant vector, $\mathbf{p}_p(t_k)$ is computed using direct kinematics

$$\mathbf{p}_p(t_k) = \mathbf{\Lambda}^{-1}(\theta_1(t_k), \theta_2(t_k), \theta_3(t_k)) \tag{25}$$

The position error function is then

$$\delta_p(t_k) = |\mathbf{p}(t_k) - \mathbf{p}_p(t_k)| \tag{26}$$

Now the position error function is a function of the initial and destination positions, and manipulator kinematics. For a given manipulator, the kinematics is fixed. The statistic model of the errors can then be established by randomizing the initial and final positions, $\mathbf{p}_i$ and $\mathbf{p}_d$, the error function in Eq(26) becomes basically a stochastic process dependent on random variables, $\mathbf{p}_i$ and $\mathbf{p}_d$, and on time t. It can be interpreted as follows: at any given time t, δ_p is a random variable itself; and for any two chosen $\mathbf{p}_i$ and $\mathbf{p}_d$, $\delta_p(t)$ becomes an ordinary function of time.

At this point, we bring two other important variables into the error function, the motion segment time T_{seg} and sampling period T^G_{upd}. For a given robot control system, the sampling period, once chosen, usually remains unchanged. Segment times, however, change from motion to motion, but in a much more predictable fashion than the positions the manipulator may move to. Therefore, we can take this into account by considering the error function for a few representative and fixed values of T_{seg}. With the time parameters, Eq (23) is rewritten as

$$\mathbf{p}(t_k) = G(\mathbf{p}_i, \mathbf{p}_d, T_{seg}, t_k) \tag{27}$$

Orientation error function can be similarly computed. Given $\mathbf{R}_i$ and $\mathbf{R}_f$, we compute $\mathbf{n}$, θ that defines the rotation change. Applying Eq (24) we arrive at the first three joint solutions in PIK. We then carry out Eq (28) to complete the solution.

$$\bar{\theta}_4(t_k) = \Lambda_1(\mathbf{R}(t_k), \theta_1(t_{k-1}), \cdots, \theta_3(t_{k-1}))$$
$$\bar{\theta}_5(t_k) = \Lambda_2(\mathbf{R}(t_k), \theta_1(t_{k-1}), \cdots, \theta_4(t_{k-1}))$$
$$\bar{\theta}_6(t_k) = \Lambda_3(\mathbf{R}(t_k), \theta_1(t_{k-1}), \cdots, \theta_5(t_{k-1})) \tag{28}$$

Similar to Eq (26) the equivalent rotation of the above solution is computed by the direct kinematics

$$R_a(t_k) = \Lambda^{-1}(\bar{\theta}(t_k)) \tag{29}$$

By applying Eq (20) through (22) to compute orientation error $\delta_r(t_k)$, we can compute orientation error at each point along a motion trajectory. Furthermore, if we randomize the parameters ψ, ϕ, and θ that define orientation change over the entire robot orienting space, we can evaluate our algorithm by studying the characteristics of Eq (22).

So far we have introduced our parallel inverse kinematics solution and a method to construct statistical models of position and orientation error functions. Both the solution and the technique for constructing error models are applicable to a number of robot manipulators with a closed-form inverse kinematic solution. In the next section, we use a specific example to evaluate our algorithm.

IV. EVALUATION ON PUMA 260

We evaluate the algorithm on a PUMA 260 manipulator, which has six degrees of freedom with six revolute joints and a reach of approximately 40 centimeters. The symbolic inverse kinematics and Jacobian matrices we use here are based on those in [1]. We break the section into two parts - in the first part, we study the position errors; in the second, we study orientation errors. In each case, we vary the time parameters, T_{seg} segment time and T^G_{upd} the setpoint update period, and we examine distributions of two variables, the mean error $\bar{\delta}$ and maximum error δ^{max}, in both position and orientation. All simulation programs are written in 'C' using single precision floating point arithmetics.

A. Position Error Analysis

While generating random initial and final positions, one must make sure the Cartesian trajectory between the two positions lies inside the robot workspace and does not include any singularity points. While one could test the condition whether a singularity is reached after a new setpoint is generated, one could also use the following criteria to predict the presence of any position singularity along the trajectory without performing any inverse kinematics.

The position workspace of PUMA 260, which contains only an elbow position singularity, can be viewed as the space between a sphere and a cylinder as illustrated in Fig. 4.

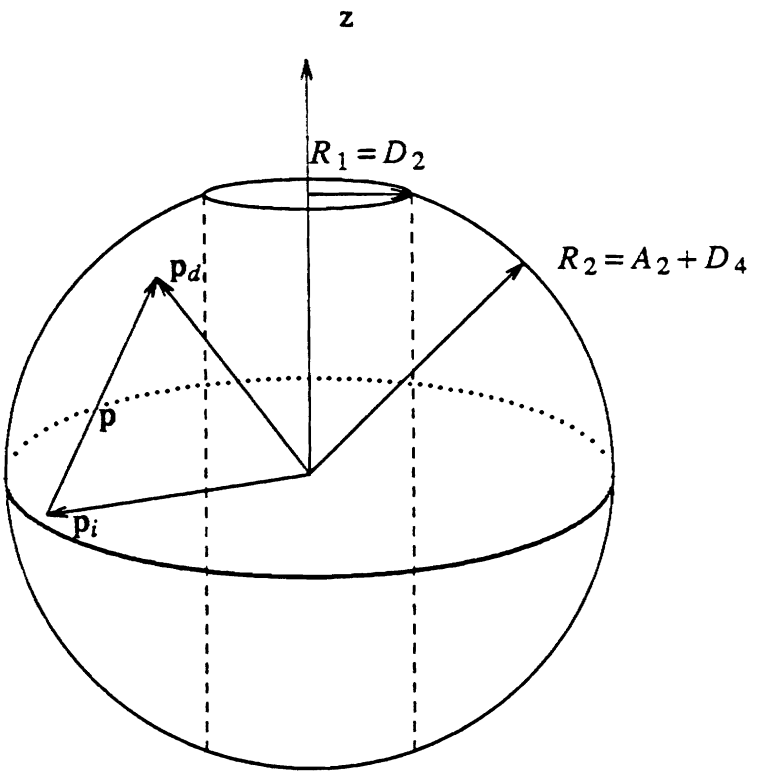


Fig. 4: Position workspace of PUMA 260

The sphere is the space swept by the arm when it is fully extended, and the cylinder is the space the arm cannot reach due to the shoulder offset, D_2. We ignore the joints' limits since they are irrelevant as far as our analysis is concerned. To determine if the two randomly chosed positions $\mathbf{p}_i$ and $\mathbf{p}_f$, where

$$\mathbf{p}_i = (x_i, y_i, z_i), \quad \mathbf{p}_d = (x_d, y_d, z_d) \tag{30}$$

form a Cartesian trajectory that does not contain the elbow singularity and that lies entirely in the workspace, the trajectory $\mathbf{p} = \mathbf{p}_d - \mathbf{p}_i$ must satisfy two conditions:

(i) The end points of two vectors, $\mathbf{p}_i$ and $\mathbf{p}_d$, must lie in the workspace. For this to be true, the it is necessary that

$$x_i^2 + y_i^2 > R_1^2, \quad x_d^2 + y_d^2 > R_1^2 \tag{31}$$

and that

$$|\mathbf{p}_i| < R_2, \quad |\mathbf{p}_d| < R_2 \tag{32}$$

(ii) If (i) is true, in order for every point on between $\mathbf{p}_i$ and $\mathbf{p}_d$ to be inside the workspace it must be true that when $\mathbf{p}_{\min}$ and λ are defined as

$$\mathbf{p}_{\min} = \mathbf{p}_i + \lambda(\mathbf{p}_d - \mathbf{p}_i) \tag{33}$$

$$\mathbf{p}_{min} \cdot (\mathbf{p}_d - \mathbf{p}_i) = 0 \tag{34}$$

for some $0 \leq \lambda \leq 1$, $\mathbf{p}_{min}$ must satisfy

$$x_{min}^2 + y_{min}^2 > R_1^2 \tag{35}$$

and

$$|\mathbf{p}_{min}| < R_2 \tag{36}$$

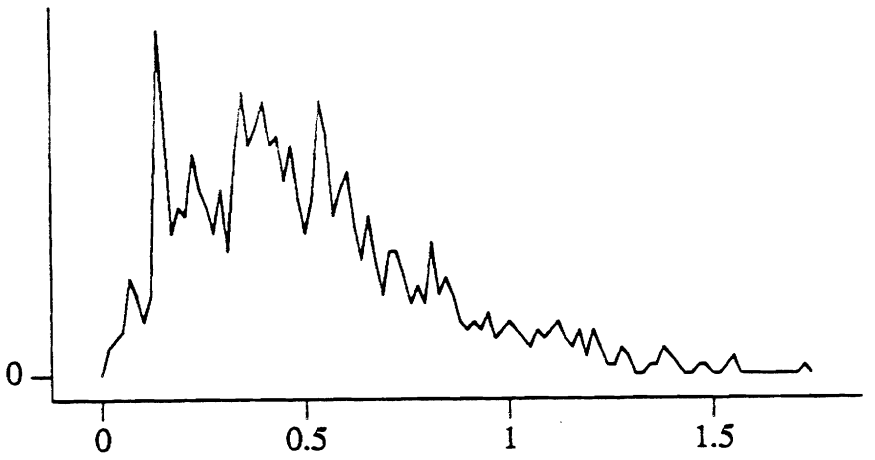
In case the solution leads to $\lambda > 1$ or $\lambda < 0$, the trajectory also lies entirely in the workspace. Geometrically, we shrink the sphere and grow the cylinder symmetrically, then make sure the line segment between the two end positions lies entirely inside the volume between the shrunk sphere and grown cylinder. Such a test can determine the feasibility of a Cartesian trajectory most efficiently without actually generating intermediate points.

In generating the initial and final vectors, the x,y and z coordinates are randomly generated by a random number generator with uniform distribution between $(0, R_2)$ for x, $(-R_2, R_2)$ for y, and $(-R_2, R_2)$ for z. The reason why x is chosen to be alway positive is that it is sufficient to study a semisphere as all other cases simply correspond to semispheres that can be obtained by rotating this one about the waist axis; such a rotation does not affect the nature of the problem.

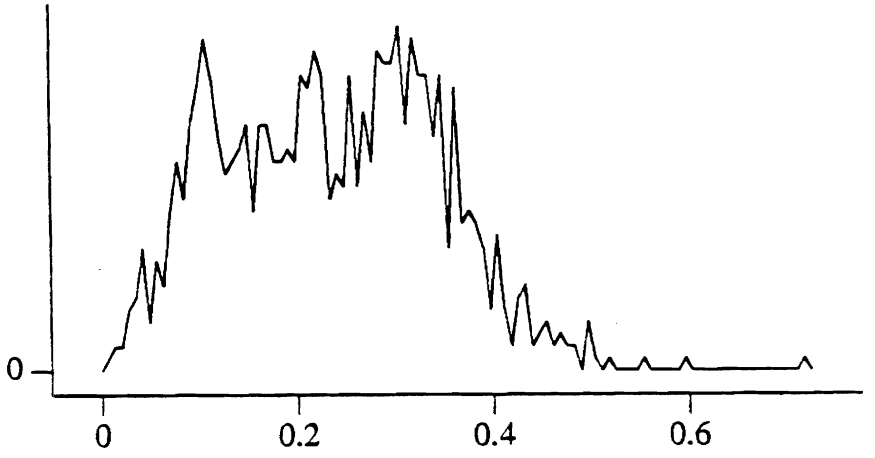
We choose to have segment times of 3 and 7 *seconds*, and T_{upd}^G of 3 *ms* 7 *ms*, which cover the range of values for these parameters in typical robot control systems. Unless otherwise indicated, every distribution is obtained from 1000 randomly selected trajectories.

Error δ_p *Case 1:* $T_{seg} = 7secs$ *and* $T_{upd}^G = 7ms$

In the first set of plots, we display the distributions of the maximum and mean of the position error function in Eq(26) for the time parameters given above.



(a)



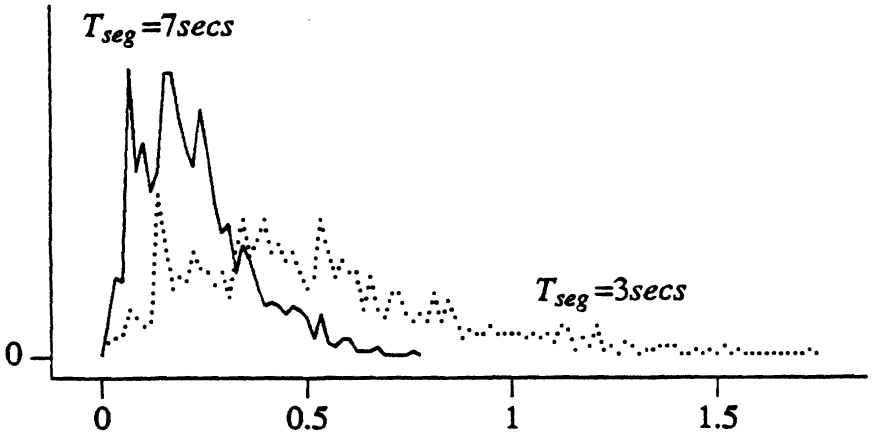
(b)

Fig. 5. Position Error Distributions (in *mm*). (a) $\delta_p^{\max}$. (b) $\overline{\delta}_p$.

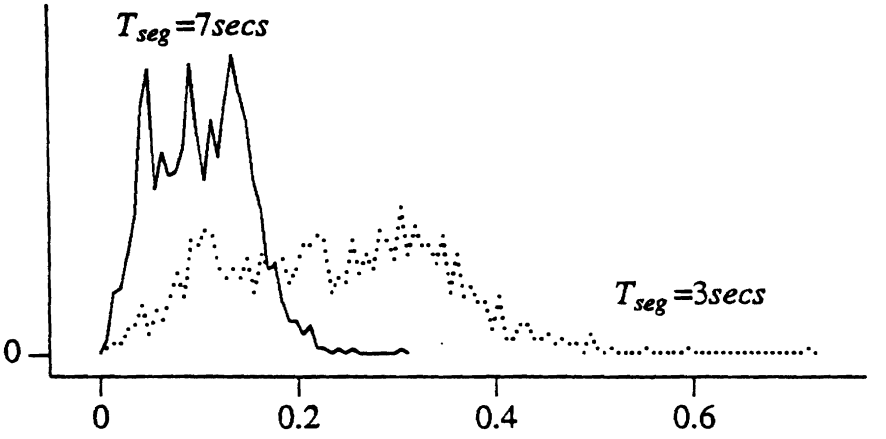
Each distribution is interpreted as a density function with the either mean error or the maximum error as the horizontal axis. Therefore, the probability for a randomly chosen trajectory to have a mean or maximum error less than x is given by the integration from 0 to x of the respective distribution. Bounds on errors can also be easily identified. In our first case, the maximum errors are bounded by about 1.2 *mm* and the average error by roughly 0.5 *mm*, for the specified set of time parameters. For the majority of the trajectories, the maximum error is less than 0.8 *mm* and the mean error less than 0.4 *mm*.

Error δ_p Case 2: Effect of the Segment Time

Intuitively, a smaller segment time with unchanged displacement requires higher Cartesian velocity, leading to larger joint errors. To verify the conjecture, we set the setpoint update time at 3 *ms* but vary the segment time from *3 secs* to *7 secs*. The number of trajectories over which distributions are computed remains unchanged.



(a)



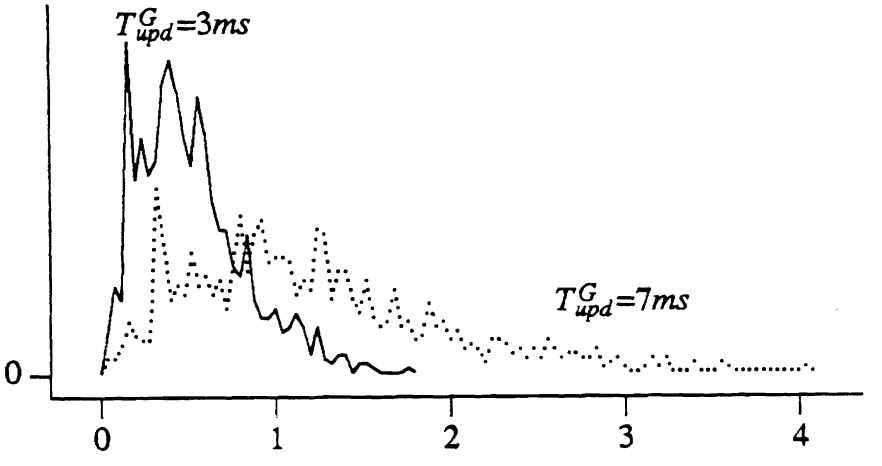
(b)

Fig. 6. Effect of T_{seg} at $T_{upd}^{G} = 3ms$. (a) $\delta_p^{\max}$ (in *mm*). (b) $\overline{\delta}_p$ (in *mm*).

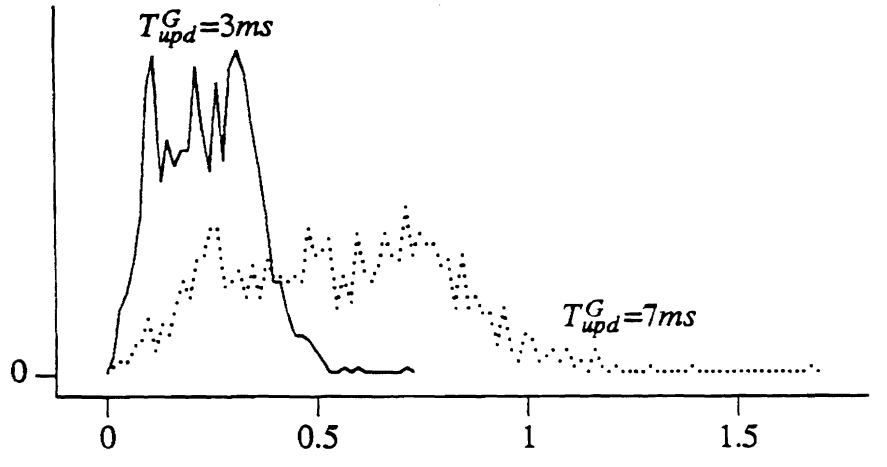
The curve corresponding to the distribution of smaller segment time is a compressed version of the one corresponding to larger segment time, with the envelope preserved. By comparing the peaks of the two curves, we can see that the change in the magnitude of the errors is approximately inversely proportional to that in the segment time. Upper bound on position error for T_{seg} = 7 *secs* is about 0.6 *mm*.

Error δ_p Case 3: Effect of Update Period

On the other hand, if we increase the rate at which the new setpoints are computed, we expect to see decrease in errors. The next two plots display the effect of the update periods on position errors. The error function is studied for $T^G_{upd} = 3ms$ and $T^G_{upd} = 7ms$.



(a)



(b)

Fig. 7. Effect of T^G_{upd} on δ^p Distribution at $T_{seg} = 3secs$ (a) $\delta_p^{\max}$ (in mm). (b) $\overline{\delta}_p$ (in mm).

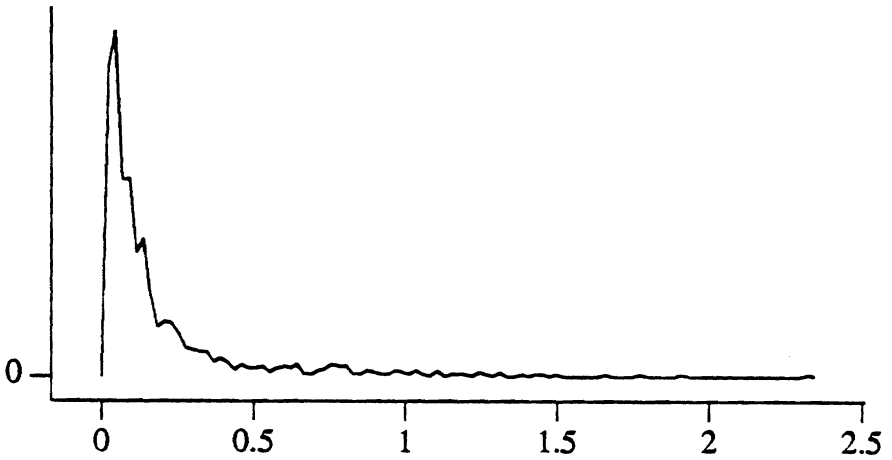
The update period has the similar effect on errors to the inverse of segment time. In all distributions there is a clear upper bound. Also notice that the distribution for $T^G_{upd} = 3ms$ and $T_{seg} = 3secs$ in Fig. 7 is almost the same as that for $T^G_{upd} = 7ms$ and $T_{seg} = 7secs$, implying that the determining factor of the error characteristics is the ratio between T_{seg} and T^G_{upd}, rather than the size of each parameter.

Orientation Error Analysis

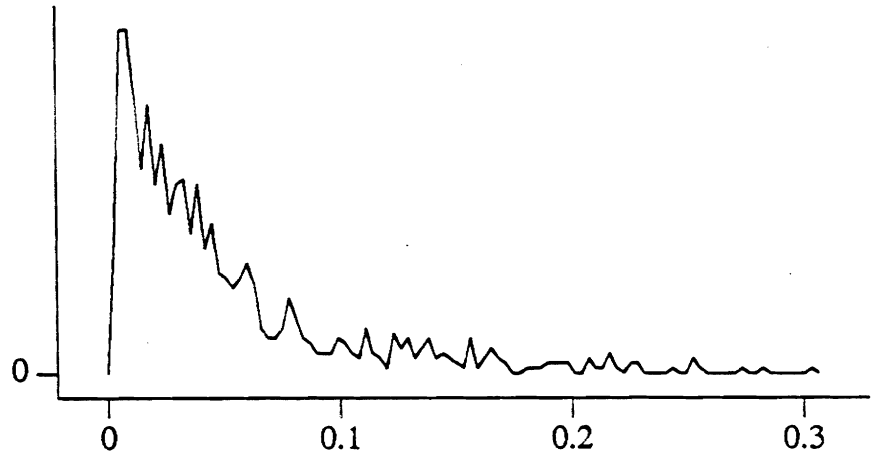
We now turn our attention to the analysis of orientation errors. To generate random orientation changes, we generate the first two Euler angles, ϕ and ψ, to create unit vector Eq(15), and then we generate a rotation random between 0.1 and 0.9 of one π for θ, which rotates about the unit vector. Additionally, the above random rotation is accompanied by a random translation change as described in the previous section. While generating the random rotations, unlike the case for position trajectory generation, there is no simple way such as Eq (25) through Eq (30) to predict if the orientation trajectory will reach any orientation singularity regions[8]. Instead we have to test if a point is inside the singularity region after it is generated.

Using the parallel inverse kinematics, we tested the solution over a large number of trajectories again to arrive at our statistical evaluation. Conclusions similar to position error analysis can be drawn from the results of our study on orientation errors. In our analysis we use Eq (22) to calculate orientation error on each point along a trajectory as the square root of the sum of squares of rotational errors about each principle axes. Since the conclusions are similar, here we simply display some of the plots to show the characteristics of the distributions, again of mean orientation errors and maximum orientation errors.

Error δ_r *Case 1:* $T_{seg} = 7secs$ *and* $T^G_{upd} = 7ms$



(a)



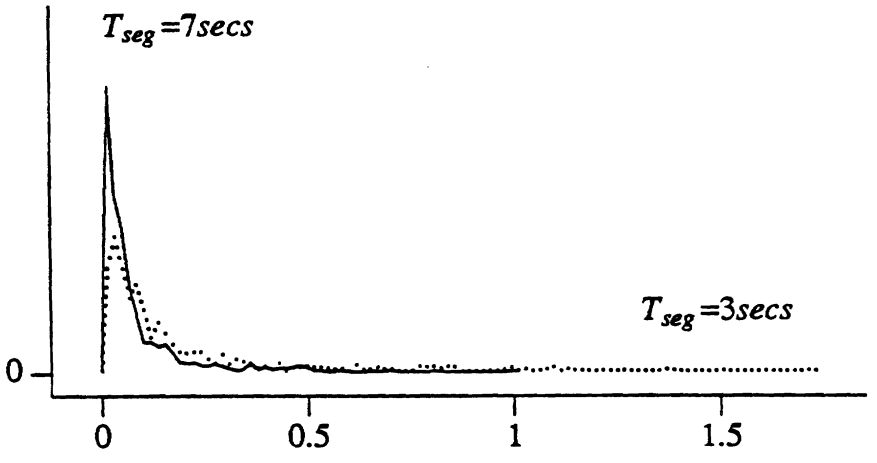
(b)

Fig. 8. Orientation Error Distribution (in *degs*). (a) δ_r^{max}. (b) $\overline{\delta}_r$.

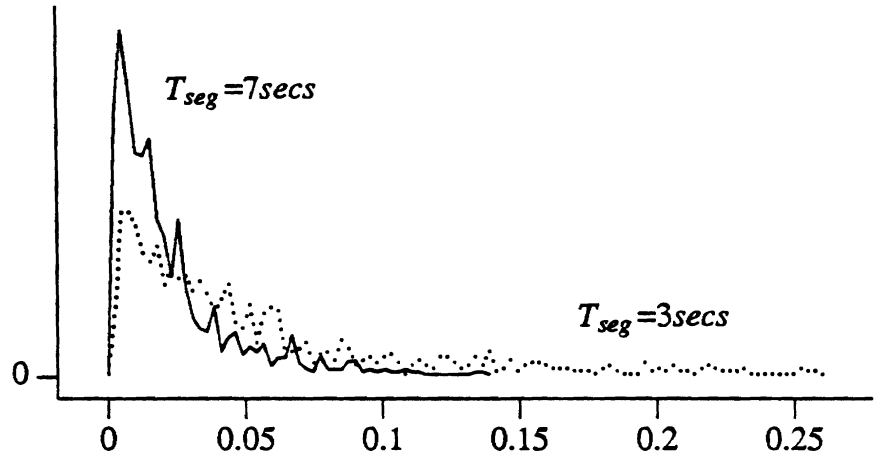
As we can see, the maximum errors for the given set of time parameters are almost always less than 1.5 degrees and mean errors less than 0.2 degrees. Compared with the position error distribution, the peak of the error distribution occurs more closely to zero. The upper bound corresponding to the worst case over the entire workspace in each case occurs at a relatively large value. This may be caused by the fact the orientation of $\mathbf{T}_6$ is determined by all six joints so that the outer joints' computation bares the noises due to approximation that are generated by all prior joints, leading to the drifting of errors to a large upper bound before they disappear. For position errors, since solution of the the positioning joints is independent of that of the orienting joints, they are less noisy than the orientation errors.

Error δ_r *Case 2: Effect of Segment Times*

Next we study the orientation errors by varying the segment times of motion trajectories between 3 and 7 seconds but maintaining the same update time at 3 *mm*.



(a)



(b)

Fig. 9. Effect of T_{seg} on δ_r at $T_{upd}^{G} = 3ms$. (a) δ_r^{max} (in *degs*). (b) $\overline{\delta}_r$ (in *degs*).

The decrease of the update time compresses the distribution, but preserves its envelope. Further, the decrease in error is roughly linear to the decrease in update period.

C. Comparison to Resolved Motion Control

Now we have statistical models of the mean and maximum errors, but we have not answered the question: how does the method compare with other similar approaches in terms of these errors? As we mentioned the parallel algorithm gains efficiency through approximation. Therefore, it is only fair to compare the method with other representative approximation methods used to generate Cartesian trajectories.

One widely applied approach is *Resolved Motion Rate Control* (RMRC) proposed by Whitney. The approach is based on the idea that the joint velocity vector can be derived from the Cartesian velocity vector by using the inverse Jacobian. However since the setpoint process is discrete in time, the desired joint velocity vector at the entire update period is approximated by that at the beginning of the period, ignoring second or higher order effects. Further, since the update of Jacobian matrix takes a considerable amount of time, it is usually computed in background at a slower rate, causing further deviation of the computed trajectory from the desired one. The effect of both factors on tracking error is illustrated in Fig. 10, where T^{J}_{upd} is usually a few times longer than T^{G}_{upd}.

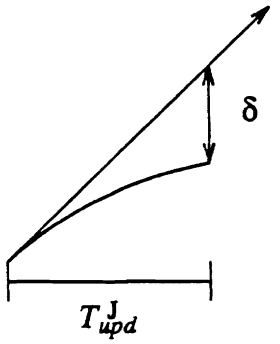


Fig. 10. Linear Approximation In Jacobian Control

Errors caused by the approximation can be readily computed. We denote $J(t)$ as the Jacobian matrix and desired Cartesian rate can be computed from the initial and final configurations as

$$\dot{\mathbf{x}} = (\mathbf{x}_f - \mathbf{x}_i)/T_{seg} \tag{37}$$

where each $\mathbf{x}$ is a vector of six Cartesian coordinates, three translation and three rotations. The desired joint velocity is then

$$\dot{\boldsymbol{\theta}} = \mathbf{J}^{-1}\,\dot{\mathbf{x}} \tag{38}$$

the joint position is computed by

$$\boldsymbol{\theta}(t) = \boldsymbol{\theta}(0) + \int_0^t \mathbf{J}^{-1}\dot{\mathbf{x}}dt \tag{39}$$

To calculate the error, we first perform the direct kinematics to find out the position of the end-effector corresponding to the computed joint positions,

$$\mathbf{T}_6(t) = \Lambda^{-1}(\boldsymbol{\theta}(t)) \tag{40}$$

The error is computed by comparing Eq (40) and the desired $\mathbf{T}_6$. One can easily get the desired $\mathbf{T}_6$ from the initial and final configurations and the time variable. Using Eq (40) we can calculate error in position $\delta_p(t)$ and error in orientation $\delta_r(t)$ as we did when studying PIK. By

randomizing the initial and final manipulator configurations, we obtain the statistical models of these error functions.

Another disadvantage of this formulation is that once the real trajectory deviates from the planned one, the deviation will remain without correction since the velocity is not adjusted according to where the end-effector is. To reduce the tracking error, one can continuously recompute the desired Cartesian velocity based on where the end-effector really is, $\mathbf{x}(t)$, where it is heading, $\mathbf{x}_d$, and how much time there is left for the current trajectory $T_{seg}-t$, by the equation

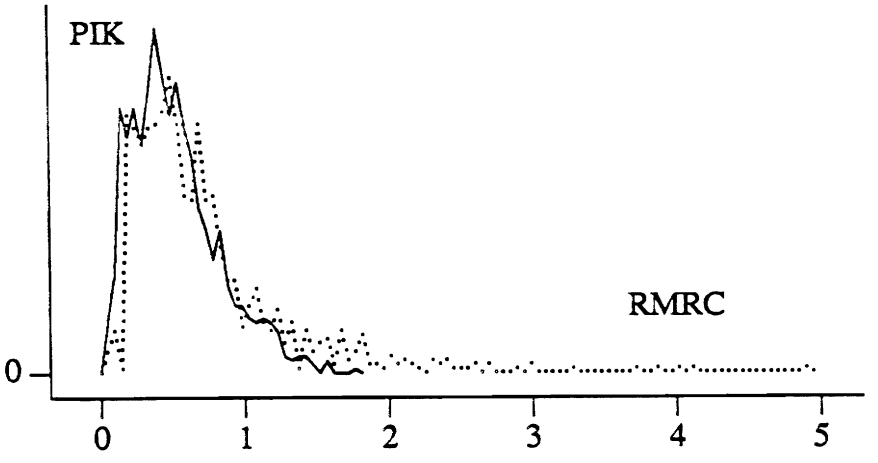
$$\dot{\mathbf{x}}(t) = \frac{\mathbf{x}_d - \mathbf{x}(t)}{T_{seg}-t} \tag{41}$$

which is no longer constant over $[0, T_{seg}]$.

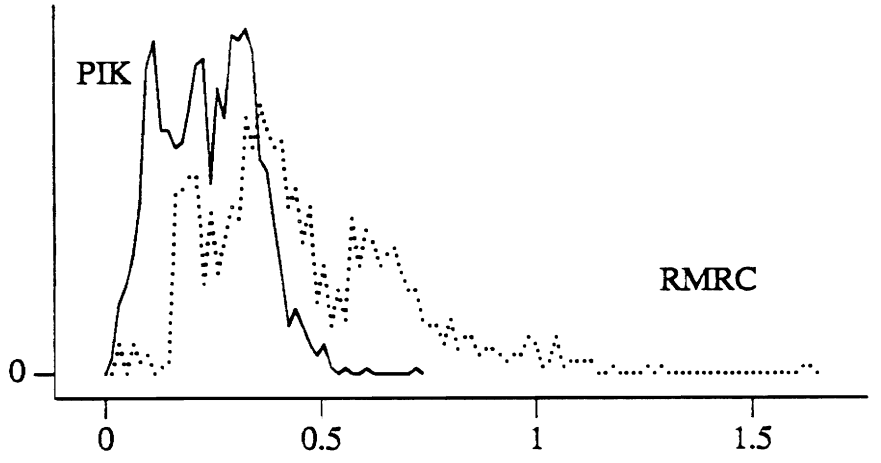
We evaluate the error function again using PUMA 260 as the representative manipulator. The performance of RMRC is studied over 1000 randomly chosen trajectories and the results are are compared with those of the parallel algorithm under various time conditions.

Comparison of δ_p *Case 1:* $T_{seg} = 7secs$, $T^{\mathbf{J}}_{upd} = 14\ ms$ *and* $T^{G}_{upd} = 7ms$

Here we assume the Jacobian matrix update takes place at one half of the the rate of the inverse kinematics computation, a more optimistic ratio favoring Jacobian control than actuality when they are performed on the same processor. On the PUMA robot for example[1], symbolic evaluation of the inverse Jacobian matrix requires 118 multiplications, 50 additions, and 6 trigonometry function calls, with a weighted cost of 1035.4; on the other hand, the parallel inverse kinematics has a weighted cost of 173.1 (Table III), excluding the overhead, corresponding to a ratio of 1 to 7. Even in the serial solution, the relative cost is 687.2 (Table II), corresponding to a ratio of 1 to 1.5. In the following plots, the dotted curve represents the maximum position error distribution of RMRC and the solid curve the maximum position error distribution of PIK.



(a)



(b)

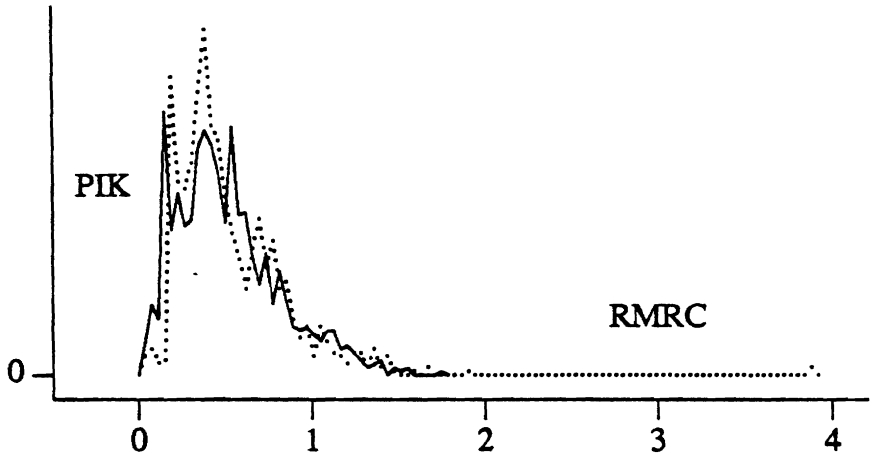
Fig. 11. Comparisons of δ_p Distributions (in *mm*). (a) $\delta_p^{\max}$. (b) $\overline{\delta}_p$.

The comparison of $\delta^{\max}$ shows remarkable similarity in the envelopes of the two distributions, with the distribution of RMRC shifted toward right, implying larger maximum errors. Further, the distribution for RFMS is much more sparse and worse behaved than PIK. Distribution for RMRC does not have a clear cut-off error, the value above which no more errors would be observed. However, in the case of PIK in each case we have studied, position error is well bounded by a cut-off value. In our experiments, this characteristic is maintained regardless the number of trajectories over which the distribution is computed.

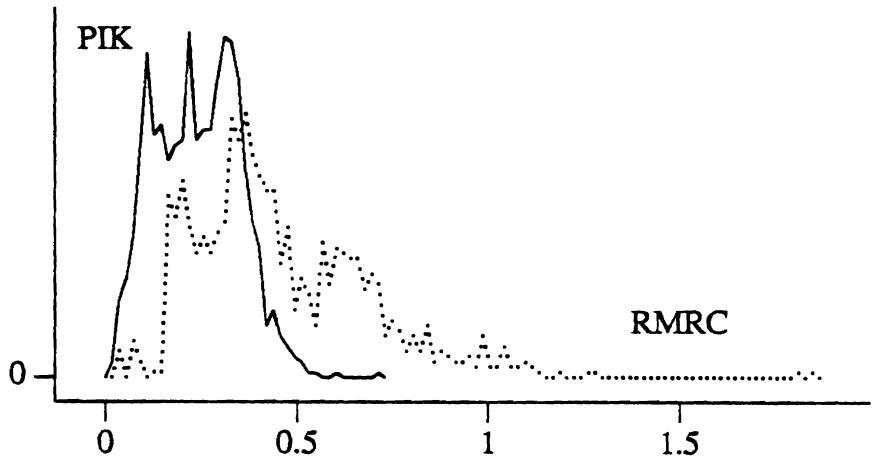
The disparity becomes more apparent in the distributions of mean position errors with the RMRC's distribution shifting further right and drifting to zero at much larger value than PIK.

Comparison of δ_p *Case 2:* $T_{seg} = 3secs$, $T^J_{upd} = 6\ ms$, *and* $T^G_{upd} = 3\ ms$

In the next set of plots, we cut both the segment time and sampling period by one half and maintain a two-to-one ratio between the Jacobian matrix update period and the sampling period. This corresponds to having the Jacobian update period of 6 milliseconds, a figure hard to approach even by the best microprocessor on the market. The comparative results are displayed in Fig. 12 with the distribution for RMRC in dashed curve and that for PIK in solid curve.



(a)



(b)

Fig. 12. Comparisons of δ_p Distributions for Smaller T_{seg} and T^G_{upd} (in *mm*). (a) $\delta_p^{\max}$. (b) $\bar{\delta}_p$.

Similar conclusions can be drawn from the above figures. Again the PIK is superior to RMRC in terms of tracking accuracy by a large margin. Also we can note that the distributions do not change much from the Fig. 11, indicating the determining factor is the ratio between the segment time and T^G_{upd}. *D. Comparison to VAL*

Comparison of PIK can also be made with the control system of PUMA 260, VAL[8], provided by the manufacturer (Unimation). VAL is typical of industrial robot controllers with a long T_{upd}^{G} but a small Δt. In particular, VAL computes a new position every 28 milliseconds and a faster servo loop, running at about a kilohertz, performs a linear interpolation between two neighboring points. In our study, we set the segment time at 7 seconds, the update period for PIK at 2 milliseconds, which is about about the best achievable time in an 8086/8087-based multiprocessor (Table II). The update period for VAL is set at 28 *ms*, with 13 interpolated points for each computed point. The result of comparison is displayed in Fig. 13.

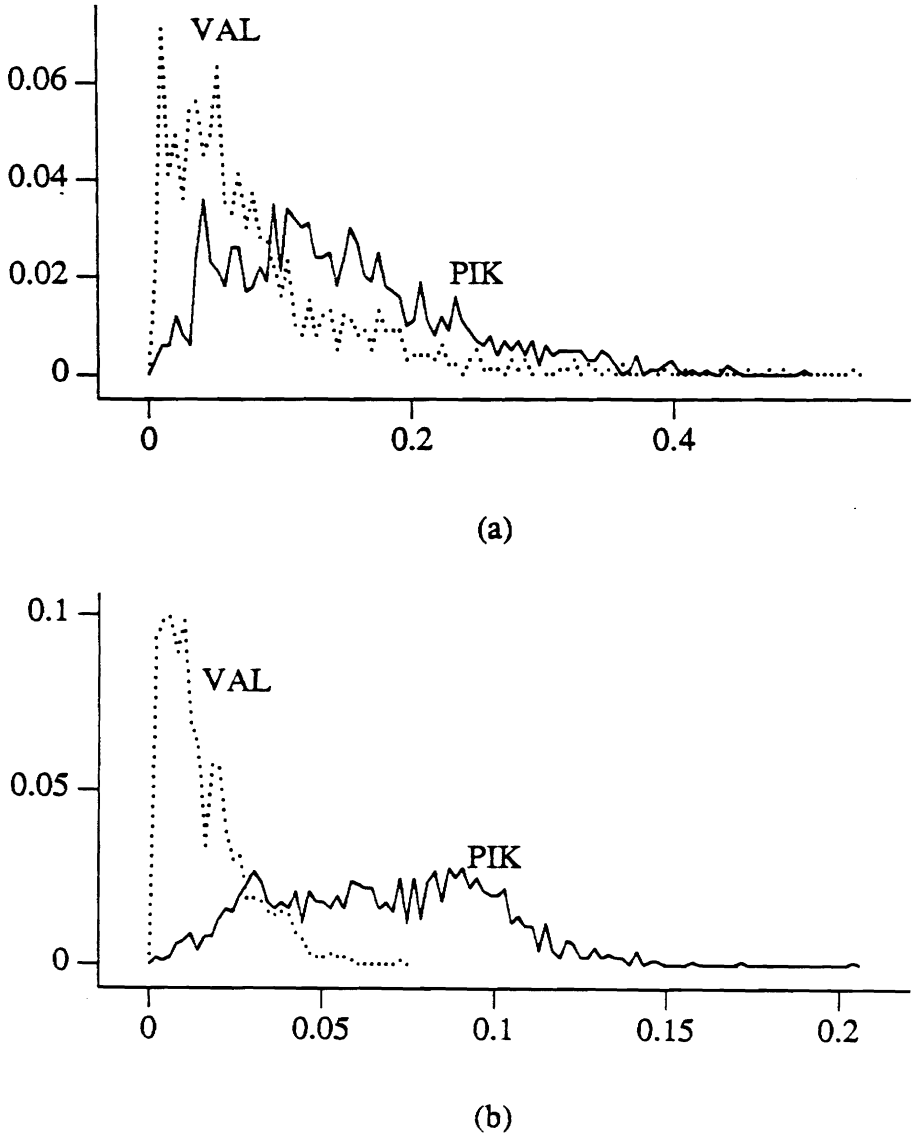


Fig. 13. Comparison between VAL and PIK (in *mm*).(a) $\delta_p^{\max}$. (b) $\overline{\delta}_p$.

In terms of the mean error distribution, VAL has a better performance than PIK with a upper bound of 0.07 *mm*. PIK's mean errors are bounded by 0.2 *mm*. In terms of the two maximum error distributions, the more important comparison of the two, it is not clear which

method is preferred to the other, since VAL has a larger upper bound but PIK a larger mean of maximum errors. When comparing the two methods by their response times to modification, PIK is much more superior to VAL: PIK modifies trajectory every 2 *ms*, enabling the robot to react to input signals at up until 500 hertz, whereas VAL can handle the input signals only at up until 35 hertz.

V. IMPLEMENTATION

The parallel solution outlined above is implemented using Intel single board computers as part of a robot control system to control a PUMA 260 manipulator [9]. The system is illustrated in Fig. 14. Each joint employs an 8086-based 86/30 and is equipped with an 8087 co-processor so that computations can be performed in floating point. The system is based on the Multibus to enable joints to communicate with each other.

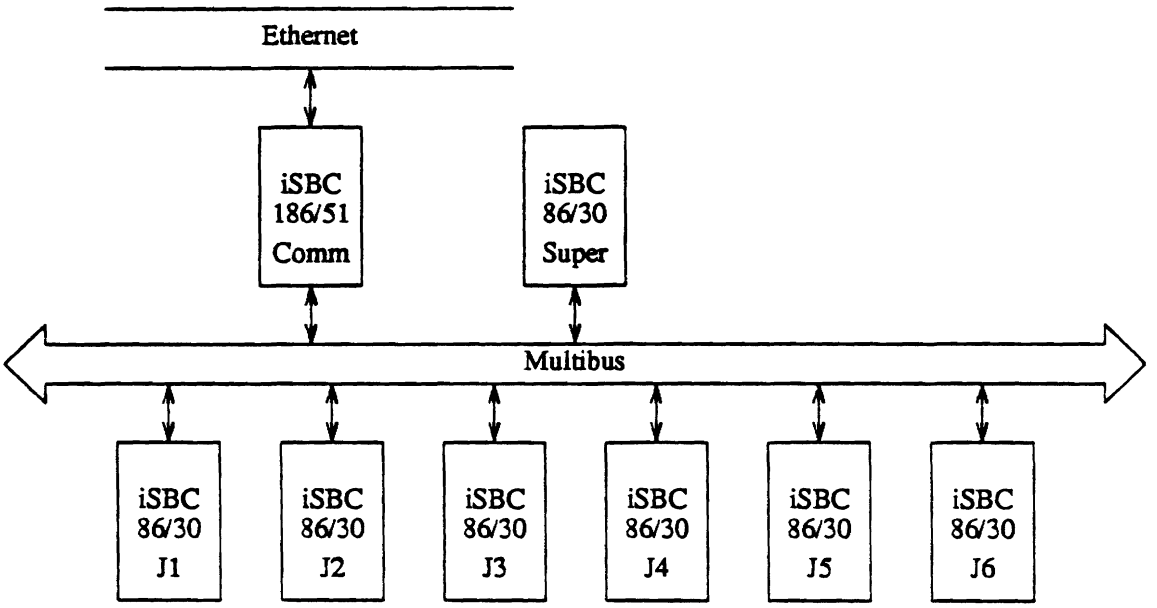


Fig. 14. System Implementation

The trajectory generation algorithm used is based on [5] which is formulated to take advantage of a multiprocessor system. The process is interrupt driven with an even update period T^G_{upd}, which we set at 6 *ms* due to various considerations other than inverse kinematics, although it can be as small as 3.0 *ms*. The system supervisor computes the desired Cartesian positions $\mathbf{T}_6$ and joints compute inverse kinematics in parallel. Between motion segments, joints perform transitions also in parallel, to remove any discontinuities in position and velocity between the current and next segment of motion. At the servo level, the system runs a PID control loop with a sampling period of 1.5 *ms*. The servo derives the position commands by linearly interpolating setpoints computed by the trajectory generation process.

Joints and supervisor communicate via the Multibus through shared memory. At the end of each sampling period, each joint stores, to known locations or mail boxes, the computed results, which then are collected, and distributed to other joints if necessary, by the supervisor. The overhead is dominated by such data collection and distribution among supervisor and the joints. The supervisor sends a buffer of about 100 bytes to the joints, the buffer containing one of vectors of the current T_6, sines and cosines of other joints, etc., which each joint needs for inverse kinematics. A joint sends a buffer of 36 bytes back to the supervisor, reporting what was computed in the last period. The amount of data transfer totals 816 bytes in each sampling period. Since there can be only one bus master at any given moment, the data transfers occur in serial. Consequently, the system spends about 0.8 *ms* on exchanging data.

The controller communicates with the external world through an Ethernet communication processor 186/51. Task definitions are first sent to the controller through this process. While the task is being executed, task geometry can be modified through this communication. We are able to handle input changing at up to 50 hertz.

VI. CONCLUSION

A parallel algorithm for the inverse kinematics of a robot manipulator with closed-form solution has been described in this work. The method is highly efficient as demonstrated by the four-fold reduction in the computational complexity when applied to solving the inverse kinematics of PUMA 260. Such an improvement in computation efficiency allows the robot to react to external modifications at a high rate, which is critical for execution of real-time sensor driven tasks.

We have also introduced a method to study the behavior of the algorithm from the statistical point of view. The method allows us to arrive at statistical models of error functions, on which the evaluation of the algorithm can be based to study such important parameters as mean errors and upper bounds, over the entire workspace of the robot. Further, the method can be applied effectively as well to other problems in which it is difficult to derive symbolic expressions for the variables of interest.

The parallel inverse kinematics is applied to a specific robot manipulator, PUMA 260, and the error analysis technique is used to evaluate the algorithm. The study shows that the method introduces errors much in the acceptable range; the errors disappear in at most six sampling periods at the end of the motion. When the update period is seven milliseconds and segment time is seven seconds, for example, the position errors are upper-bounded by 1.5 millimeters and the average errors by 0.6 millimeters (Fig. 5). Even though the method has about the same performance as the interpolation method normally employed in industrial robots in terms of upper bound on errors, the algorithm can be executed at a speed 5 to 10 times faster, thereby minimizing errors due to modification to the trajectory.

The algorithm we have introduced can be implemented using special architecture on a single-board multiprocessor. The device should be made programmable so that it can solve kinematics of various robot manipulators. The single-board implementation cuts down the communication overhead, the physical size, and, hopefully, the cost. With today's microprocessor technology, it is conceivable that such an inverse kinematics processor can run at about 1 Kilohertz, a rate at which the errors are upper-bounded at about 0.2 *mm*, small enough for virtually any application.

REFERENCES

[1] R. P. Paul and H. Zhang, "Computationally efficient kinematics for manipulators with spherical wrists based on the homogeneous transformation representation," *Int. J. Robotics Res.*,, vol. 5, no. 2, pp. 30-42, Summer, 1986.

[2] H. Zhang, "Use of the C/8086 cross compiler," *Internal Memo.*, Department of Computer and Information Sciences, University of Pennsylvania, Philadelphia, PA., 1985.

[3] R. H. Taylor, "Planning and execution of straight line manipulator trajectories," *IBM Journal of Research and Development*, vol.23, no.4. July, 1979.

[4] R. P. Paul, *Robot Manipulators: Mathematics, Programming and Control.* Cambridge, MA: MIT 1981.

[5] R. P. Paul and H. Zhang, "Robot motion trajectory specification and generation". *Proc. 2nd International Symposium of Robotics Research*, pp: 373-380, August 20-23, Kyoto, JAPAN. 1984.

[6] Whitney, D. E. "The mathematics of coordinated control of prostheses and manipulators," *J. Dynamic Systems, Measurement, Control*, pp: 303-309, December, 1972.

[7] R. P. Paul and C.N. Stevenson, "Kinematics of robot wrists," *Int. J. Robotics Res.*, vol. 2, no. 1, pp.31-38, Spring, 1983.

[8] Unimation Inc., *"Breaking away from VAL or how to use your PUMA without using VAL,"* Unimation Inc., 1982.

[9] H. Zhang and R. P. Paul, "A robot force and motion server," *Proc. 1986 ACM/IEEE Computer Society Fall Joint Computer Conference*, Dallas, Texas, November 1986.