MANIPULATOR CONTROL WITH OBSTACLE AVOIDANCE
USING LOCAL GUIDANCE

by

KOICHI FUNAYA

B.E. Aeronautical Engineering, University of Tokyo
(1987)

SUBMITTED TO THE DEPARTMENT OF
AERONAUTICS AND ASTRONAUTICS
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 1988

© Massachusetts Institute of Technology

Signature of Author ____________________
Department of Aeronautics and Astronautics
August 19, 1988

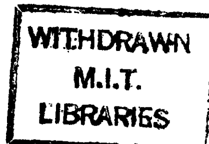
Certified by ____________________
Professor David L. Akin
Thesis Supervisor, Department of Aeronautics and Astronautics

Accepted by ____________________
Professor Harold Y. Wachman
Chairman, Department Graduate Committee

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

SEP 07 1988

LIBRARIES



Aero

MANIPULATOR CONTROL WITH OBSTACLE AVOIDANCE
USING LOCAL GUIDANCE

by

KOICHI FUNAYA

Submitted to the Department of Aeronautics and Astronautics on August 19, 1988
in partial fulfillment of the requirements for the Degree of Master of Science

ABSTRACT

For costly and repetitive operations in manufacturing, and for operations in hostile environments such as space, undersea, or nuclear reactors, robots are considered to be potentially advantageous over human beings in terms of productivity and safety. The current obstacle to this prospect is the robot's lack of autonomy. In this thesis, the basic concept of the obstacle avoidance trajectory planning and its tracking control method are developed.

Among the prior studies in obstacle avoidance trajectory planning problem, Udupa, Lozano-Perez, etc. proposed algorithms which avoid obstacles by computing an explicit representation of the manipulator configurations that would bring about a collision. However, this is a higher control level in hierarchical robot control systems, so that it is computationally intensive and limited to off-line trajectory planning.

Klein, Kirćanski, etc. took another approach. The trajectories are generated according to lower-level (local) control. In this approach, redundant degrees of freedom are utilized to keep the nearest link away from the obstacle, while performing the required end-effector motion by using the pseudo-inverse of the manipulator Jacobian matrix. This is advantageous because it is possible for this to be performed in real-time.

In this thesis, the pseudo-inverse method and the steepest descent method are discussed as trajectory generation schemes. The pseudo-inverse of the manipulator Jacobian matrix is one which relates desired velocity (or rate) at a certain point on the arm to the corresponding joint rate. By using pseudo-inverse matrices, several tasks, such as goal-reaching movement of the end-effector and obstacle avoidance movement of the arm, are somewhat performed simultaneously at each time instant. In the steepest descent method, an index function is set up first. This should be chosen such that a decrease in the function corresponds to an increase in the obstacle clearance and a decrease in the distance to the goal. Then the joint rate is chosen so as to reduce this function as rapidly as possible.

Both of the two trajectory generation methods, the pseudo-inverse method and the steepest descent method, use local schemes rather than global schemes. Because of this, the methods take less computation time at each cycle, thus making it possible to apply them to real-time systems. The pseudo-inverse method has an advantage in its good performance of higher priority tasks. However, this method has a singularity problem. The steepest descent method is often more robust and requires less computation time.

The trajectories generated would be tracked by a tracking control system. In this thesis, the computed torque method and the sliding surface control method are explained as possible candidates. It is shown that sliding surface control has better accuracy and more robustness against parameter uncertainties.

Thesis Supervisor: David L. Akin
Professor of Aeronautics and Astronautics

ACKNOWLEDGEMENTS

I would like to thank Professor David L. Akin for offering me the opportunity to study at the MIT Space Systems Laboratory.

I would also like to thank John Spofford and Robert Sanner and other people at the Space Systems Laboratory for invaluable advice.

This work was sponsored by the office of Aeronautics and Space Technology, NASA Headquarters, under NASA Grant NAGW-21.

1. INTRODUCTION 1
2. TRAJECTORY PLANNER 4
2.1 Local Schemes 4
2.2 Trajectory Planning Using Task-priority 5
2.2.1 Pseudo-inverse Matrix 5
2.2.3 Local Optimal Trajectory Planning Based on Task Priority 11
[1] End-effector Motion 11
[2] Obstacle Avoidance Velocity 12
[3] Joint Rate 13
2.2.4 Simulation 15
2.2 Steepest Descent Method 21
2.2.1 Steepest Descent Method 21
2.2.2 Application to Manipulator Arm Trajectory Planning 24
2.3.3 Simulation 28
2.4 Comparison of the Two Methods 42
2.5 Dynamically Changing Environment 44
3. TRACKING CONTROL 49
3.1 Tracking Control Using Computed Torque Method 49
3.2 Tracking Control Using Sliding Surfaces 51
4. CONCLUSION 59
REFERENCES 61
APPENDIX 63
A.1 Calculation of the Gradient 63
A.2 Dynamics Equation 67
A.3 Computer Listings 72

1. INTRODUCTION

For costly and repetitive operations in manufacturing, and for operations in hostile environments such as space, undersea, or nuclear reactors, robots are considered to be potentially advantageous over human beings in terms of productivity and safety. The current obstacle to this prospect is the robot's lack of autonomy. Up to now, a human operator has performed the lower-level work for a robot, such as the control of its manipulator arm movement, which includes:

1) the task of planning the robot manipulator joint movements so that the end-effector [hand] and arm can achieve the specified objective within imposed constraints, and

2) the task of calculating and generating joint torques which best realize the trajectories given above.

Currently, this lower-level work requires human labor, thereby decreasing the advantage of robots. A robot needs to be automated to assist the operator with such work, so that it can act more flexibly in the presence of changing goals and unmodeled environments.

This paper deals with the topic of joint coordinate trajectory planning and tracking for a manipulator arm when given an obstacle configuration and an end-effector goal position. Among the prior studies in this area, Udupa[11], Lozano-Perez[5], etc. proposed algorithms which avoid obstacles by computing an explicit representation of the manipulator configurations that would bring about a collision. Any set of joint motions that attains the goal configuration without collision is considered a satisfactory result. The task in this case can be defined in terms of known initial and final configurations for the end-effector. Therefore this algorithm is suited for applications such as the pick-and-place problem. However, this is a higher control level in hierarchical robot control systems, so that it is computationally intensive and limited to off-line trajectory planning.

Klein[6], Kirćanski[4], etc. took another approach. The trajectories are generated according to lower-level (local) control. In this approach, redundant degrees of freedom are utilized to keep the nearest link away from the obstacle, while performing the required end-effector motion. This could be achieved by using the pseudo-inverse of the manipulator Jacobian matrix at the end-effector and at the nearest point on the arm to the obstacles. Unlike the configuration space method, which gives global trajectories, this gives the joint rate at each instant. This is advantageous because it is possible for this to be performed in real time, in dynamically varying environments.

In this thesis, the basic concept of the obstacle avoidance trajectory planning and its tracking control method are developed. First two different trajectory generation schemes are discussed:

1) the pseudo-inverse method

2) the steepest descent method.

The trajectories generated are tracked by manipulators with a tracking control systems, which is discussed in the subsequent chapter.

The pseudo-inverse of the manipulator Jacobian matrix is one which relates desired velocity (or rate) at a certain point on the arm to the corresponding joint rate. The manipulator model here has redundant degrees of freedom. This indicates the possibility that several tasks could be performed, such as goal-reaching movement of the end-effector and obstacle avoidance movement of the arm. Here, the word "task" means to specify a velocity (and possibly a rate) at a certain point on a manipulator arm. It is shown that, by using pseudo-inverse matrices, those tasks are somewhat performed simultaneously at each time instant. In order for the trajectory planner to perform the tasks, a priority has to be allocated among them. Once the priority is determined, the joint rate is calculated which best performs the top priority task. Then this rate is modified to accomplish the second task to the extent that it does not affect the first. Again the rate is modified to take the third into account, to the extent it does not affect the first and the second ones, etc. Also

explained in this section are two examples of tasks: velocity at reaching a goal, and obstacle avoidance vecocity.

In the steepest descent method, an index function is set up first. This function contains the distance between the arm and the obstacle, and the distance between the end-effector of the arm and the goal. The index function should be chosen such that a decrease in the function corresponds to an increase in the obstacle clearance and a decrease in the distance to the goal. Then the joint rate (or acceleration) is chosen so as to reduce this function as rapidly as possible.

The joint coordinate trajectory generated by either of the above planners have to be tracked by a manipulator to realize the objective. As a tracking control method, the computed torque method is developed first. However, this method lacks robustness. To improve robustness, the sliding surface control method is developed. Conceptually, this method consists of two different tactics. At each time instant, the desired trajectory is represented by a point in a state space. The sliding surface is the one in the state space which contains this point. The surface also has a characteristic that , once the system state is on it, it converges to the desired trajectory point exponentially. So the problem in this tracking control method is how to move the system state towards the sliding surface. This is explained in section 3.2. It is also explained why this method could be robust against parameter uncertainties.

2. TRAJECTORY PLANNER

2.1 Local Schemes

In this chapter, trajectory generation programs are discussed. In certain tasks, such as assembling of parts, robots are required to pick a part from a certain point and place it at a specified location. For such tasks, the role of the trajectory planner is to find the joint coordinate rate (or acceleration) which realizes the desired end-effector position (and orientation), starting from a given arm configuration, while avoiding collision with obstacles. In other words, the problem is to find a manipulator trajectory which reduces d_{eg} to zero, while keeping d_{co} above certain value. There are basically two different approaches to this problem. These are:

1) Global optimization

2) Local optimization

In global schemes, the whole trajectory from the start configuration to the end is computed off-line. The assumption here is that the environment around the manipulator is known beforehand. Because it only chooses the trajectory which attains the goal while avoiding obstacles, conceptually there should be no mistake in this approach. However, this method is not adequate for a dynamically changing environment, such as real-time applications. An example of this is the geometric approach studied by Lozano-Perez[5].

In local schemes, small increments of joint coordinates are calculated at each time step. As might be expected, local schemes require much less computation than global ones, so that it becomes possible to apply them to real-time systems. Because it calculates the trajectory according to the surrounding conditions at each time step, it can also be applied to dynamically changing environments (e.g. moving obstacles, mobile robots and moving targets). One of the drawbacks is that this might fall into a local optimum, which might prevent the manipulator from reaching the goal.

In this thesis research, local schemes are tested for manipulator arm trajectory planning. First the pseudo-inverse method, then the steepest descent method are discussed. In the pseudo-inverse method, two separate goals are set up: one is to reduce the goal distance (d_{eg}), and the other is to increase the obstacle clearance (d_{co}). Those tasks are implemented by setting priority between them. It is also shown in a later section that more than two tasks are incorporated by using this pseudo-inverse method. In the steepest descent method, an index function is set up, which contains the distance between the arm and the obstacle (d_{co}), and the distance between the end-effector of the arm and the goal (d_{eg}). The index function should be chosen so that a decrease in the function corresponds to an increase in the obstacle clearance and a decrease in the distance to the goal. Then the joint rate (or acceleration) is chosen so as to reduce this function the most. This can be considered as an optimization problem, where the joint coordinate rate is considered as an conversion vector. The joint rate is calculated by taking the gradient of the index function.

2.2 Trajectory Planning Using Task-priority

2.2.1 Pseudo-inverse Matrix

The position (and the orientation) of a certain point on the manipulator arm, $p \in R^m$, is uniquely defined by the manipulator configuration, i.e., by the joint coordinate vector q. The relationship between p and q are described by the kinematic model

$$p(t) = f\,q(t) \tag{2.1}$$

where

$$f : R^n \dashrightarrow R^m \tag{2.2}$$

is a nonlinear, continuous differentiable vector function. The case considered here is where n>m, that is, where the manipulator arm has redundant degrees of freedom beyond those required to achieve specified position and orientation. In such a case, it is generally hard to derive an explicit solution to the inverse problem,

$$q(t) = f^{-1}(p(t)) \tag{2.3}$$

So the problem is linearized locally for convenience.

$$\delta p = J(q)\ \delta q \tag{2.4}$$

where

$$J = \frac{\partial f}{\partial q} \in R^{mxn} \tag{2.5}$$

is the manipulator Jacobian matrix specified by q(t). δp and δq are small deviations of p and q. By dividing both sides of equation (2.4) by a small time increment dt, the following equation can be derived.

$$\dot{p} = J\,(q)\ \dot{q} \tag{2.6}$$

$\dot{p}$ and $\dot{q}$ are vectors of external and joint velocities. The equation (2.6) can be regarded as a linear mapping from n-dimensional vector space V^n to m-dimensional space V^m. For the following argument, some definition has to be made.

Definition

The sub-space R(J) in V^m is the range of J(q).

R(J) and its dimension are called the manipulatable space and the degrees of manipulatability (d.o.m.) at q, respectively.

Definition

The sub-space N(J) in V^n is the null space of J(q).

N(J) and its dimension are called the redundant space and the degrees of redundancy (d.o.r.) at q, respectively. According to linear algebra, the range and null space of matrix J satisfy the following

$$n = \dim R(J) + \dim N(J) \tag{2.7}$$

Therefore,

$$\text{d.o.m.} + \text{d.o.r.} = n \tag{2.8}$$

for any q. This suggests that d.o.r is calculated by subtracting m from n, unless the robot manipulator is in singular condition[7].

Assume the d.o.r. is greater than zero. Let $\dot{q}^*$ be a solution of equation (2.6) and $\dot{q}_o$ be a vector in the null space. Then the vector of the form $\dot{q}^* + k\,\dot{q}_o$ is also a solution, where k is an arbitrary scalar quantity. Namely,

$$J\,\dot{q} = J\,\dot{q}^* + J\,k\,\dot{q}_o = J\,\dot{q}^* = \dot{p} \tag{2.9}$$

Therefore there is an infinite number of solutions for equation (2.6). From this unlimited set of solutions, one that minimizes the following performance index could be selected.

$$\Omega(\dot{q}) = \frac{1}{2}\dot{q}^T M \dot{q} \tag{2.10}$$

In this definition, $M \in R^{nxn}$ is a symmetric, positive definite matrix. This problem can be solved by using Lagrange multipliers. To this end, the cost function is modified to

$$\Omega(\dot{q}, \lambda) = \frac{1}{2}\dot{q}^T M \dot{q} - \lambda(J\,\dot{q} - \dot{p}) \tag{2.11}$$

where λ is an mx1 unknown vector of Language multipliers. The optimal conditions are

$$\frac{\partial \Omega}{\partial \dot{q}} = M\dot{q} - J^T\lambda = 0 \tag{2.12}$$

and

$$\frac{\partial \Omega}{\partial \lambda} = -J\,\dot{q} + \dot{p} = 0 \tag{2.13}$$

which is identical to (2.6) As matrix M is positive definite, we obtain from equation (2.12),

$$\dot{q} = M^{-1} J^T \lambda \tag{2.14}$$

Substituting (2.14) into (2.13), we obtain

$$\dot{p} = (J\,M^{-1} J^T)\,\lambda \tag{2.15}$$

Since J has full rank, $J\,M^{-1}\,J^T$ can be inverted. By eliminating the Lagrange multiplier vector λ, we obtain

$$\dot{q} = M^{-1} J^T (J M^{-1} J^T)^{-1} \dot{p} \tag{2.16}$$

The solution above satisfies the equation (2.6). When the matrix M is the m×m identity matrix, the solution reduces to

$$\dot{q} = J^T (J J^T)^{-1} \dot{p} \tag{2.17}$$

The matrix

$$J^+ = J^T (J J^T)^{-1} \tag{2.18}$$

is known as the pseudo-inverse matrix[9]. The solution (2.17) minimizes the error,

$$E = |\dot{p} - J\dot{q}|$$

that is,

$$|\dot{p} - J (J^+\dot{p})| < |\dot{p} - J\dot{q}| \qquad \text{for } \forall\, \dot{q} \tag{2.19}$$

Therefore the solution

$$\dot{q} = J^+\dot{p} \tag{2.20}$$

best realizes the given task $\dot{q}$. If the manipulator has enough degrees of freedom, that is, $n \geq m$, and it is not in singularity condition, then the solution (2.20) is one of the exact solutions of (2.6).

$$J\dot{q} = JJ^+\dot{p} = \dot{p}$$

$$JJ^+ = I \tag{2.21}$$

Thus an inverse kinematic problem could be solved by using pseudo-inverse matrix.

2.2.2 Joint rate for multiple tasks

The way to derive the joint coordinate velocity for a single task is shown above. Next, consider the case where N_t (>1) tasks are required. Let the sub-task with the i-th priority be specified by i-th manipulation variable, $p_i \in R^{m_i}$, such that,

$$p_i = f_i (q) \tag{2.22}$$

$$\dot{p}_i = J_i (q)\, \dot{q} \tag{2.23}$$

$$(i = 1, 2, ..., N_t)$$

where

$$J_i(q) = \frac{\partial f_i}{\partial q} \tag{2.24}$$

This p_i can be the position (and the orientation) of a point on the manipulator arm, which is specified by the sub-task i. Usually, a task is given as a desired velocity (rate) of that point ($\dot{p}_i$). A vector in a sub-space $N(J_i)^{\perp}$, the orthogonal complement of $N(J_i)$, contributes to the i-th manipulation variable, $\dot{p}_i$ and is called manipulatable space of J_i. For example, $N(J_i)^{\perp}$ is the sub-space which contributes to the first manipulation variable. $N(J_1) \cap N(J_2)^{\perp}$ is the sub-space which contributes to the second manipulation without disturbing the first. $N(J_1) \cap N(J_2)$ means the remaining degrees of freedom which can be used for performing the third manipulation variable. In general,

$$N(J_1) \cap N(J_2) \cap \ldots \cap N(J_{k-1}) \cap N(J_k)^{\perp}$$

is the sub-space where $\dot{q}$ contributes to the k-th priority task without affecting the first to the k-1-th. And

$$N(J_1) \cap N(J_2) \cap \ldots \cap N(J_{k-1}) \cap N(J_k)$$

are the remaining degrees of freedom.

Next, the way to derive those sub-spaces is shown. As explained before, the solution (2.20) belongs to a sub-space $N(J)^{\perp}$. It is also mentioned that (2.20) is one of the solutions for the index function (2.19). It turns out that deriving the null spaces and the manipulatable spaces is equivalent to solving the least squares problem of the form (2.19). The argument can be extended to the solution for multiple tasks as follows.

The least squares solution of

$$C_1 = \|\dot{p}_1 - J_1 \dot{q}\|^2 \tag{2.25}$$

is

$$\dot{q} = J_1^{\dagger}\dot{p}_1 + (I - J_1^{\dagger} J_1)\, y_1$$
$$\forall y_1 \in R^n \tag{2.26}$$

At the right hand side of the equation (2.26),

$$J_1^{\dagger} \dot{p}_1$$

denotes a vector in $N(J_1)^{\perp}$ sub-space, which satisfies the equation (2.26) and minimizes the index function (2.25). And,

$$(I - J_1^{+} J_1) y_1$$

means a vector in $N(J_1)$ sub-space. Among the solutions of (2.26), the ones that minimize

$$C_2 = \| \dot{p}_2 - J_2 \dot{q} \|^2$$
$$= \| \dot{p}_2 - J_2 (J_1^{+} \dot{p}_1 + (I - J_1^{+} J_1) y_1) \|^2$$

are needed in order to best execute the second sub-task. This could be achieved by letting

$$y_1 = \tilde{J}_2^{+} (\dot{p}_2 - J_2 J_1^{+} \dot{p}_1) + (I - \tilde{J}_2^{+} \tilde{J}_2) y_2 \tag{2.27}$$

where,

$$\tilde{J}_2 = J2 (I - J_1^{+} J1) \tag{2.28}$$

By substituting (2.27) into (2.26), the resulting joint rate, $\dot{q}$, is derived as follows.

$$\dot{q} = J_1^{+} \dot{p}_1 + (I - J_1^{+} J_1) \tilde{J}_2^{+} (\dot{p}_2 - J_2 J_1^{+} \dot{p}_1) + (I - J_1^{+} J_1) \; (I - \tilde{J}_2^{+} \tilde{J}_2) y_2 \tag{2.29}$$

In this equation,

$$J_1^{+} \dot{p}_1 \qquad \in N(J_1)^{\perp}$$
$$(I - J_1^{+} J_1) \tilde{J}_2^{+} (\dot{p}_2 - J_2 J_1^{+} \dot{p}_1) \qquad \in N(J_1) \cap N(J_2)^{\perp}$$
$$(I - J_1^{+} J_1) \; (I - \tilde{J}_2^{+} \tilde{J}_2) y_2 \qquad \in N(J_1) \cap N(J_2)$$

By repeating the same procedure, the joint coordinate q which best performs i sub-tasks can be derived recursively as follows.

$$\dot{q}_i = \sum_{j=1}^{i} \dot{q}_j^{*}$$

$$\dot{q}_i^{*} = A_i v_i$$

$$A_i = \dot{p}_i - J_i \dot{q}_{i-1}^{*}$$

$$C_i = J_i B_{i-1}$$

$$B_i = B_{i-1} (I - C_i^{+} C_i)$$

$$i = 1,2,\ldots,N_t \tag{2.30}$$

where the initial value of B_i and $\dot{q}_i^*$ are

$$B_0 = I$$
$$\dot{q}_0^* = 0 \tag{2.31}$$

In this equation, $\dot{q}_j^*$ belongs to the sub-space $N(J_1) \cap N(J_2) \cap \ldots \cap N(J_{j-1}) \cap N(J_j)^{\perp}$, and given $\dot{q}_{i-1} = \dot{q}_1^* + \dot{q}_2^* + \ldots + \dot{q}_{j-1}^*$,

$$\dot{q} = \dot{q}_i = \dot{q}_{i-1} + \dot{q}_i^* \tag{2.32}$$

minimizes the index

$$C_i = \| \dot{p}_i - J_i \dot{q} \|^2 \tag{2.33}$$

Thus, N_t sub-tasks can be performed as long as the degrees of redundancy allow.

2.2.3 Local Optimal Trajectory Planning Based on Task Priority

Next the application of pseudo-inverse method to the trajectory planning problem is explained. As described above, the problem is divided into two sub-tasks. The first is to move the end-effector towards the goal position. The other is to move the arm away from an obstacle. In the following sections, the end-effector motion, collision avoidance velocity, and the resulting joint rate are discussed, in that order.

[1] End-effector Motion

Often the primary goal for a manipulator arm is to move its end-effector in a desired direction. Here, the end-effector is moved toward a goal position, while avoiding collision with obstacles. To this end, the velocity vector at each instant of time is calculated as a vector which is identical to the gradient of the criteria function

$$f(\underline{x}_e) = \alpha\, d_{eg}^2(\underline{x}_e) + \frac{\beta}{d_{eo}(\underline{x}_e)} \tag{2.34}$$

where

$$d_{eg} = | \underline{x}_g - \underline{x}e | \tag{2.35}$$

d_{eg} : the distance between the end-effector and the goal

$\underline{x}_g$: the goal position

$\underline{x}_e$: the end-effector position

and d_{eo} is the distance between the end-effector and the obstacle. By differentiating (1), the desired end-effector velocity is derived as,

$$\underline{v}_e = -\frac{\partial f}{\partial x_e} = \alpha(\underline{x}_g - \underline{x}_e) + \frac{\beta}{d_{eo}^2}\underline{v}_h \tag{2.36}$$

where $\underline{v}_h$ is a unit vector in the direction away from the nearest point on the obstacle. The first term moves the end-effector toward the goal, while the second one moves it away from the obstacle.

[2] Obstacle Avoidance Velocity

The secondary goal considered here is to keep the arm away from the obstacle. This is identical to keeping the distance between the arm and the obstacle above a certain value. Therefore, if the distance becomes too short, $\underline{x}_c$ (the closest point on the arm to the obstacle) has to be moved away from the obstacle. This, again, is stated as giving u_c a velocity, $\underline{v}_c$, in a direction opposite to the obstacle (Fig.2.1(a)). The magnitude of $\underline{v}_c$ should be a function of the distance d_{co}. The function used here are described in Fig.2.1(b). If the manipulator has more degrees of freedom than necessary for this task, another sub-task can be executed. For this third sub-task, $\underline{x}_{c2}$ (another close point on the arm to obstacle), the point is given a velocity $\underline{v}_{c2}$, in a direction away from the obstacle.

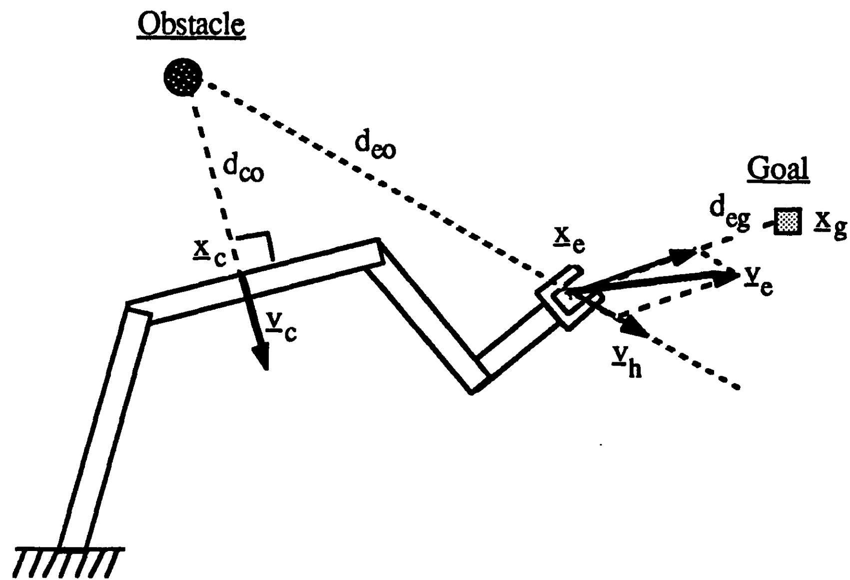


Fig. 2.1 (a) End-effector and Obstacle Avoidance Velocity

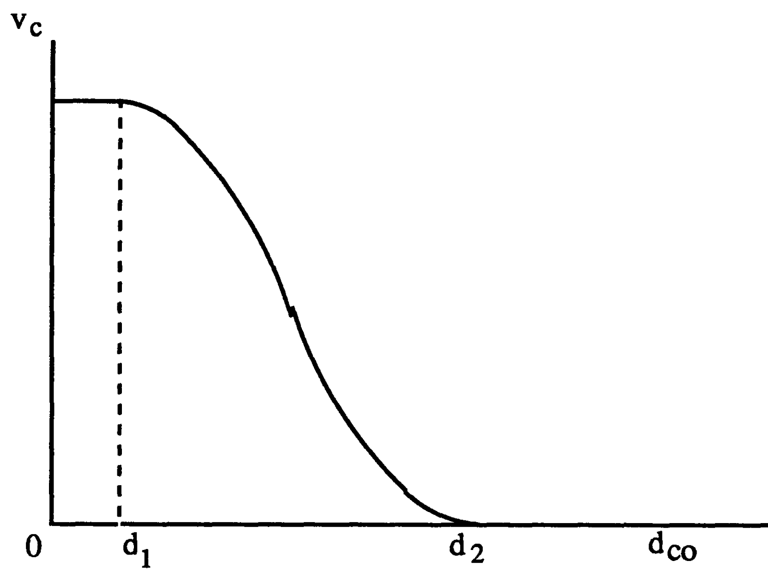


Fig. 2.1 (b) Obstacle Avoidance Velocity

[3] Joint Rate

Let us consider the case where two sub-tasks $\dot{p}_1 = v_e$ and $\dot{p}_2 = v_c$ are required. Denote the Jacobian matrix corresponding to each of the tasks as J_e and J_c, respectively. Then (2.31) becomes

$$B_0 = I$$
$$\dot{q}_0^* = 0$$

$$C_1 = J_e B_0 = J_e$$
$$A_1 = C_1^+ = J_e^+$$
$$B_1 = B_0 (I - C_i^+ C_i) = I - J_e^+ J_e$$
$$v_1 = \dot{p}_1 - J_e \dot{q}_0^* = v_e$$
$$\dot{q}_1^* = A_1 v_e = J_e^+ v_e$$

$$C_2 = J_c B_1 = J_c(I - J_e^+ J_e)$$
$$A_2 = C_2^+ = [J_c(I - J_e^+ J_e)]^+$$
$$v_2 = \dot{p}_2 - J_c \dot{q}_1^* = v_c - J_c J_e^+ v_e$$
$$\dot{q}_2^* = A_2 v_2 = [J_c(I - J_e^+ J_e)]^+ (v_c - J_c J_e^+ v_e)$$

$$\dot{q} = \dot{q}_2 = \sum_{j=1}^{2} \dot{q}_j^*$$
$$= J_e^+ v_e + [J_c(I - J_e^+ J_e)]^+ (v_c - J_c J_e^+ v_e) \tag{2.37}$$

The following equation is derived by simple calculation.

$$(I - J_e^+ J_e) [J_c(I - J_e^+ J_e)]^+ = [J_c(I - J_e^+ J_e)]^+ \tag{2.38}$$

This result agrees with equation (2.29). The equations (2.30) and (2.31) give the joint coordinate velocity which best realizes the N_t tasks required.

The trajectory generation program using the pseudo-inverse matrix (2.37) has a potential weakness. According to this equation, two different matrices have to be inverted. These are:

$$(C_1C_1^T)^{-1} = (J_eJ_e^T)^{-1}$$
$$(C_2C_2^T)^{-1} = \{[J_c(I - J_e^+J_e)][J_c(I - J_e^+J_e)]^T\}^{-1} \tag{2.39}$$

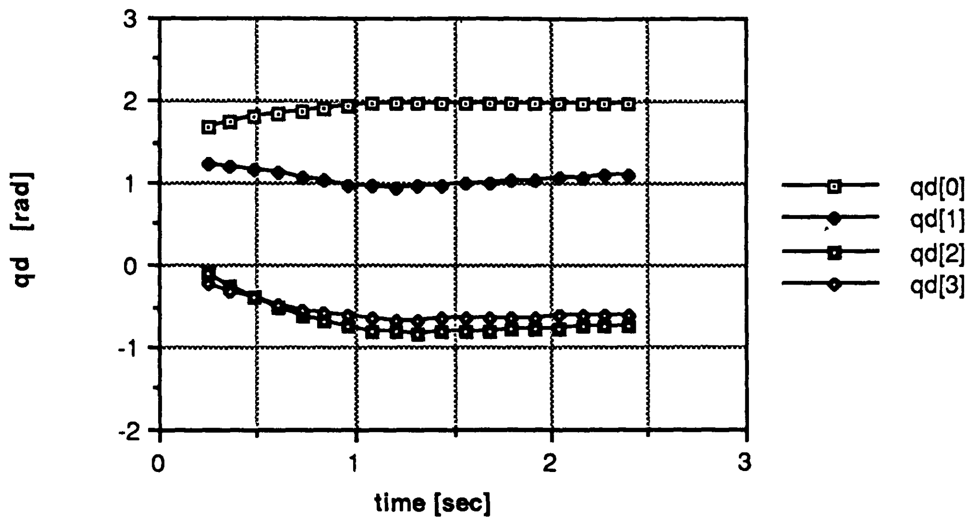
This indicates that there is a singularity problem if the determinent of the matrices become too small.

2.2.4 Simulation

Simulations were conducted to check the performance of the pseudo-inverse method. Fig. 2.2 shows examples of the performance. As can be seen from these figures, this method has a singularity problem. When the determinant $\det(CC^T)$ is too small, the joint rate jumps and there a discontinuity occurs. In this method, matrix inversion occurs twice. These are:

$$(C_1C_1^T)^{-1} = (J_eJ_e^T)^{-1}$$
$$(C_2C_2^T)^{-1} = \{[J_c(I - J_e^+J_e)][J_c(I - J_e^+J_e)]^T\}^{-1} \tag{2.40}$$

The determinant of these two matrices are described in Fig 2.3. Also an example of the determinants of an eight-link manipulator is shown in Fig 2.4. According to these figures, the deteminant of $(C_2C_2^T)$ is much smaller than that of $(C_1C_1^T)$, and is more likely to get into a singularity.



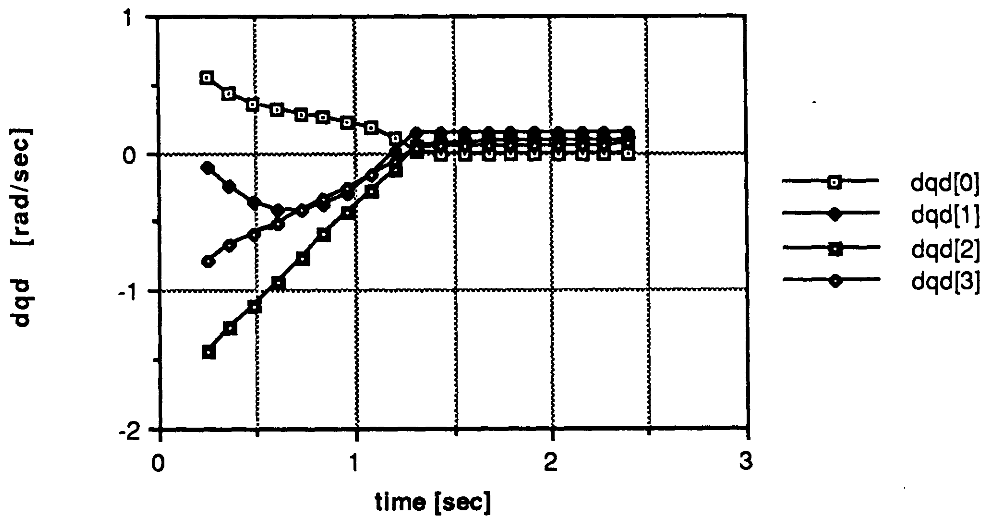
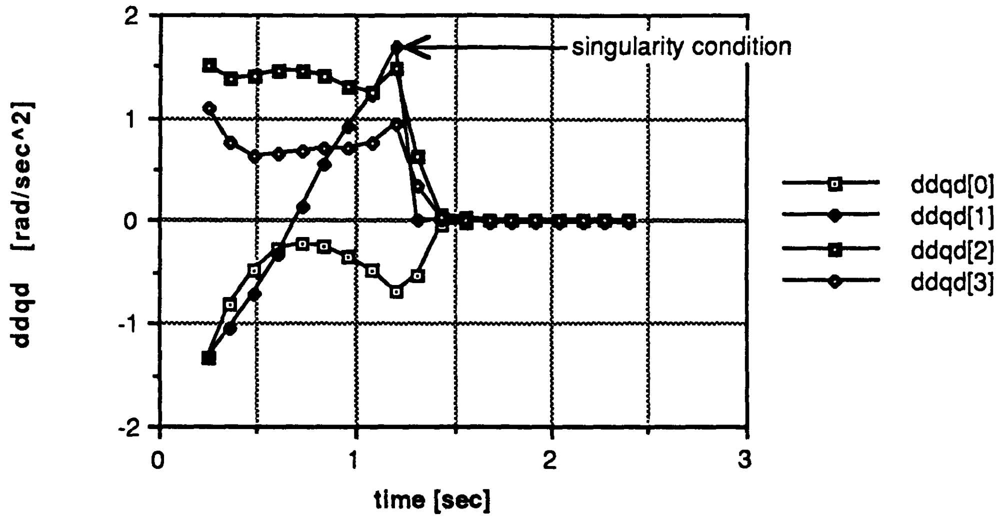


Fig 2.2



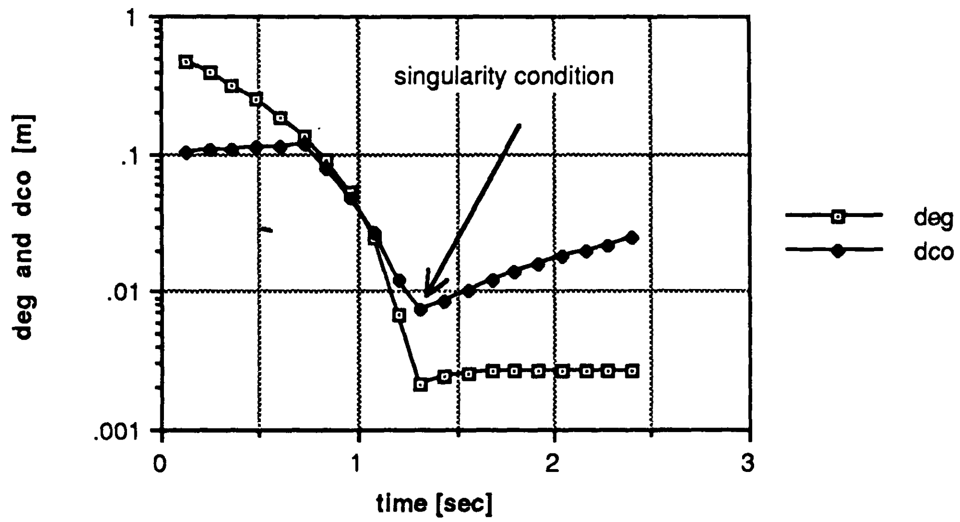
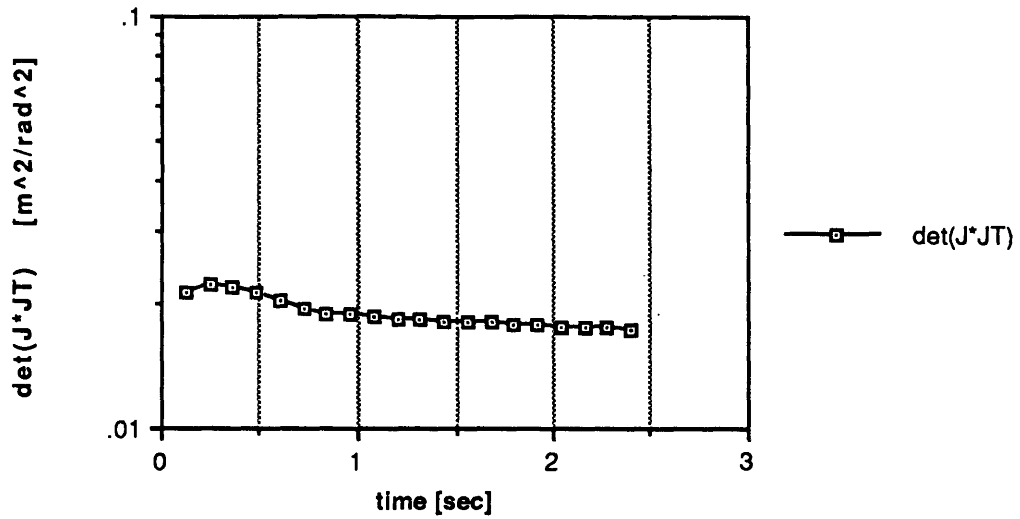


Fig 2.2



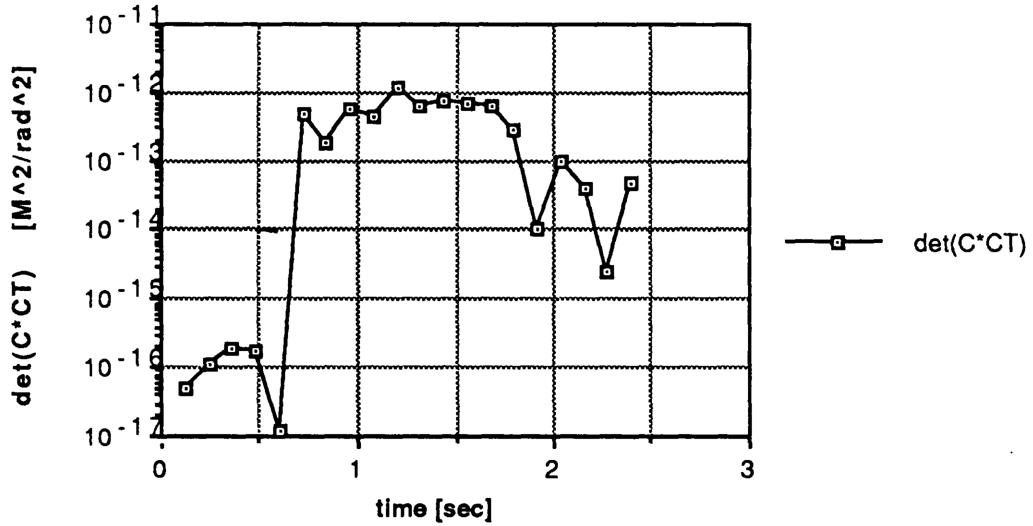
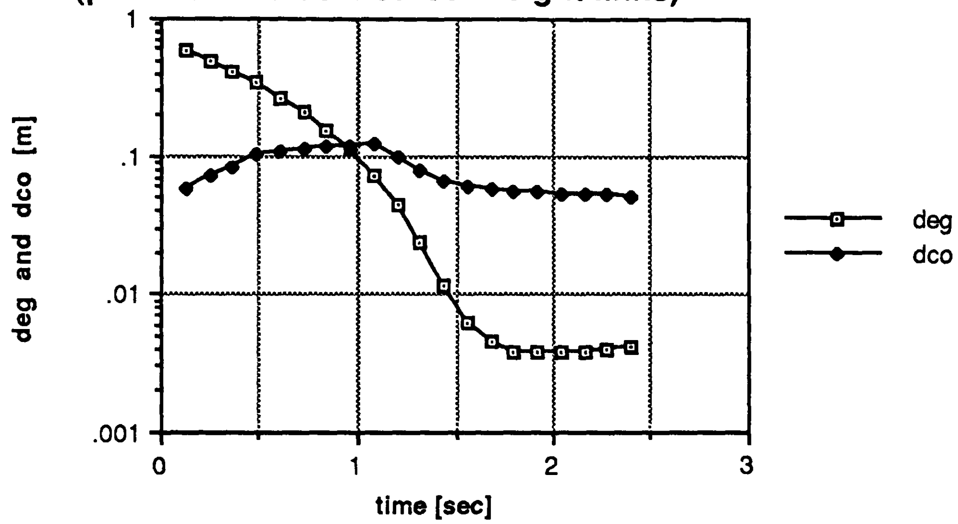


Fig 2.3



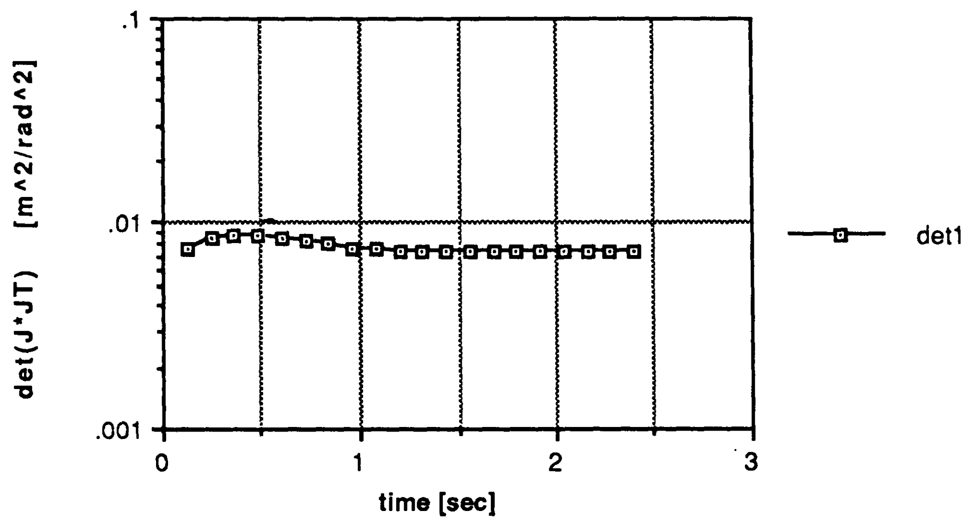
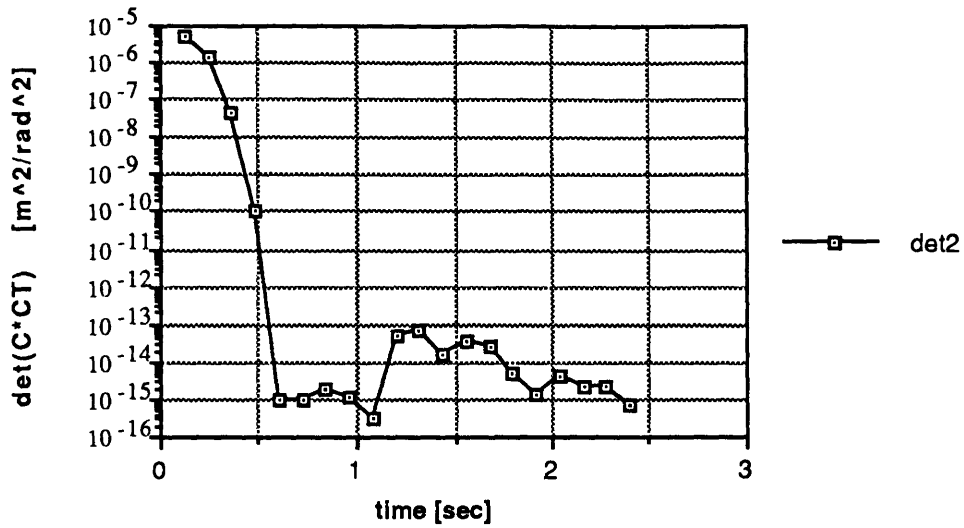


Fig 2.4



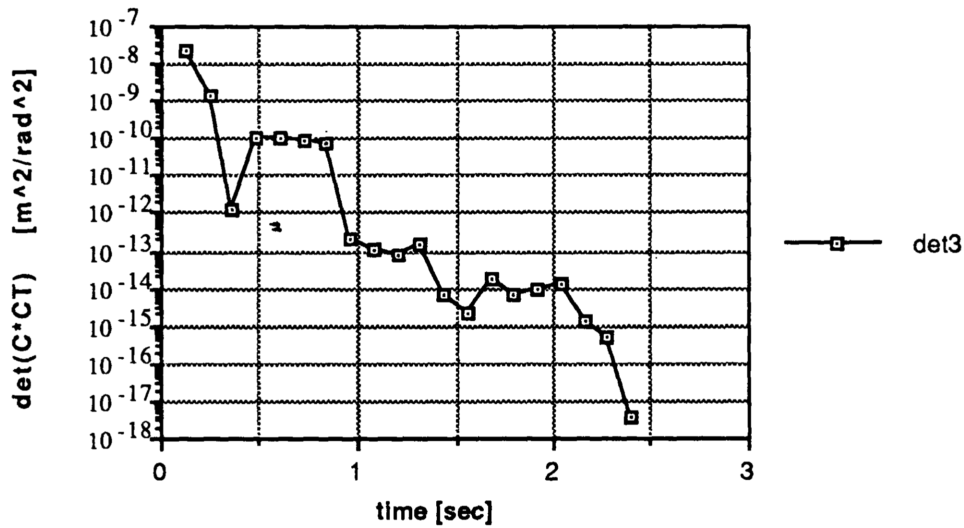


Fig 2.4

2.2 Steepest Descent Method

2.2.1 Steepest Descent Method

The partial derivatives of a function f, with respect to each of n variables are collectively called the gradient of the function and are denoted by ∇f:

$$\underset{nX1}{\nabla f} = [\partial f/\partial x_1, \partial f/\partial x_2, ..., \partial f/\partial x_n] \quad (2.41)$$

The gradient is a n-component vector and it has a very important property. The function value increases at the fastest rate along the gradient direction from any point in n-dimensional space. Hence the gradient direction is called the direction of steepest ascent. Note, however, that the direction of steepest ascent is a local property and not a global one. Since the gradient vector represents the direction of steepest ascent, the negative of the gradient vector denotes the direction of steepest descent. The steepest descent method is an optimization scheme which utilizes this property of the gradient vector, in order to find a minimum point of the function f in n-dimensional space[10]. In this method, a search point is moved towards a minimum point at each step, by using the negative of the gradient vector (Fig 2.5). Since this is the direction of the steepest descent, this method can be expected to give the minimum point faster than the one which does not make use of the gradient vector.

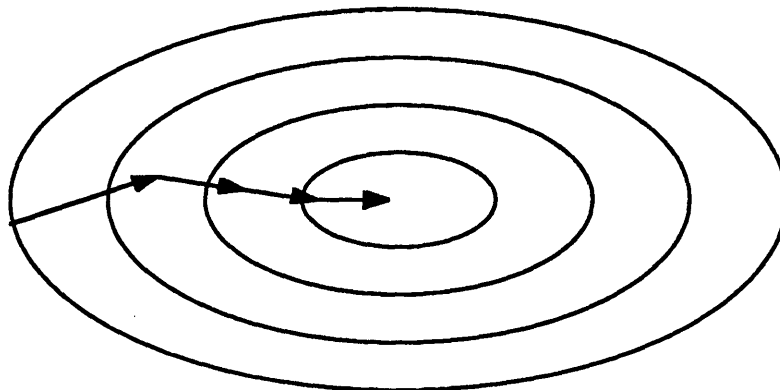


Fig 2.5 Steepest descent method

However there are some problems to this method.

(i) The gradient ∇f may not be defined at all points.

(ii) The method may settle into an n-dimensional zig-zag and the process will be hopelessly slow.

(iii) It may settle into a local optimum point, which is not the global minimum point.

The problem (i) is illustrated in Fig 2.6. Here, the gradient ∇f is not defined at $x=x_m$. In this case the descent method does not work. The problem (ii) is illustrated in Fig 2.7 for a two-dimensional case. In this case, the method creates undesirable oscillation and does not converge to the optimal point as fast as it should. The problem (iii) is illustrated in Fig 2.8. Once the search point is caught in the local optimum, the method does not converge to the optimum point. All of the problems above can happen when this method is applied to the manipulator arm trajectory planning problem, as discussed in the following section.

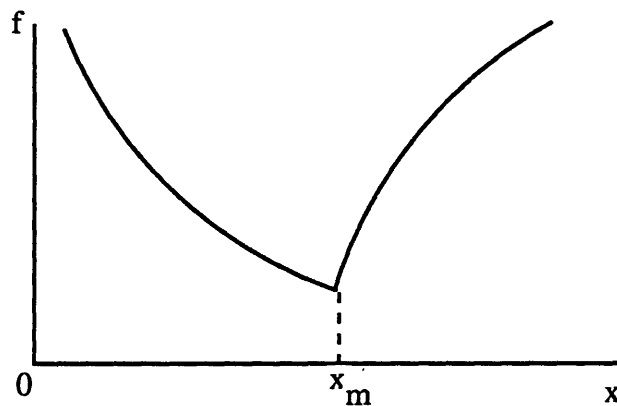


Fig 2.6 The gradient ∇f is not defined at $x=x_m$

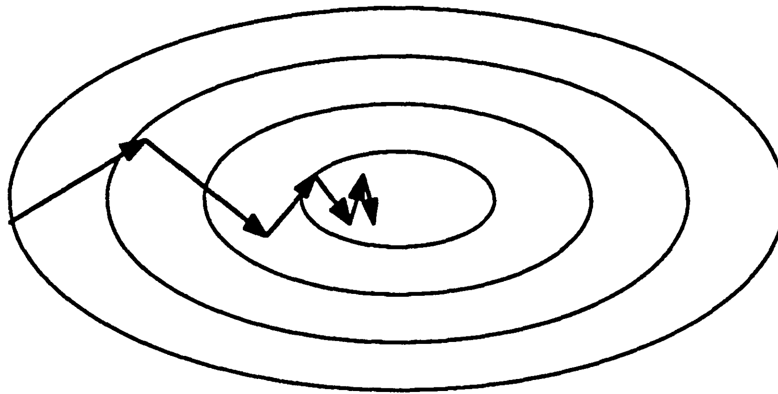


Fig 2.7 Two-dimensional Zig-zag Path

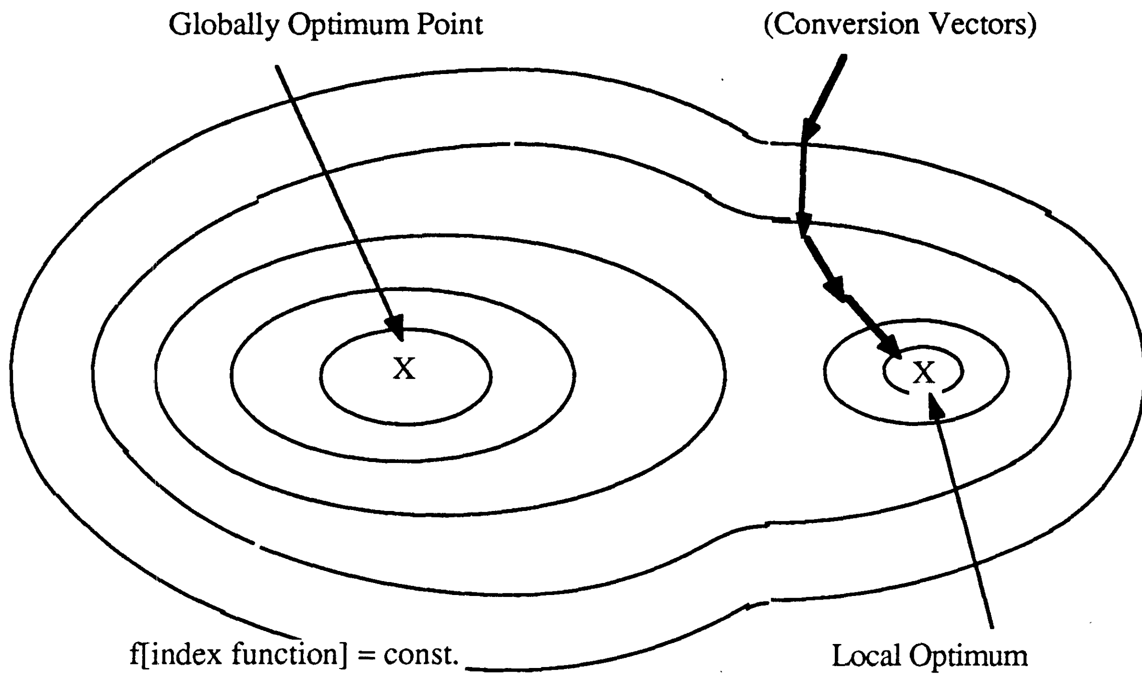


Fig. 2.8 Local Optimum

2.2.2 Application to Manipulator Arm Trajectory Planning

As in the previous section, trajectory planning with the following requirements is considered in this section.

(1) To move the end-effector towards the goal position.

(2) To avoid collision with obstacles.

This problem can be solved by an optimization scheme, such as the steepest descent method. The index function in this case should be an increasing function of d_{eg} (goal distance) and decreasing function of d_{co} (obstacle clearance). Then each joint rate $\dot{q}_i$ is chosen so that the vector $\left[\dot{q}_1, \dot{q}_2, \ldots, \dot{q}_n\right]^T$ is in the direction of steepest descent. Joint rate, in this case, corresponds to a conversion vector in the optimization scheme. In order to use the steepest descent method, a proper index function has to be chosen. Consider the following as an index function:

$$f(q) = k_{eg} d_{eg}(q) + \frac{k_{co}}{d_{co}(q)}$$

q : joint coordinate vector

k_{eg}, k_{co} : gains (2.42)

In this equation, d_{eg} and d_{co} are determined by the arm configuration, hence by the joint coordinate q. Then the joint coordinate rate is chosen to be the negative of the gradient of the index function:

$$\dot{q} := - \nabla_q f(q) \qquad (2.43)$$

where ∇_q is an gradient operator, that is,

$$\nabla_q = \left(\frac{\partial}{\partial q_1}, \frac{\partial}{\partial q_2}, \ldots, \frac{\partial}{\partial q_n}\right)$$

The calculation procedure for the gradient is explained in the appendix. The steepest descent method has the advantage that it requires less computation time than other local optimization methods such as the pseudo-inverse method, because it requires only simple

calculations as shown in the appendix. This is beneficial when it is to be applied for real-time operations. However it also has some drawbacks. These are:

(i) The discontinuity of the function when only d_{eg} (goal distance) and d_{co} (minimum distance to the obstacle) are included in the index function.

(ii) The possibility of falling into a zig-zag path, especially near the goal (that is, when the end-effector is close to the goal position).

(iii) The possibility of being trapped in a local optimum.

To illustrate the problem (i), consider a two-link manipulator and two obstacles as shown in Figure 2.9.

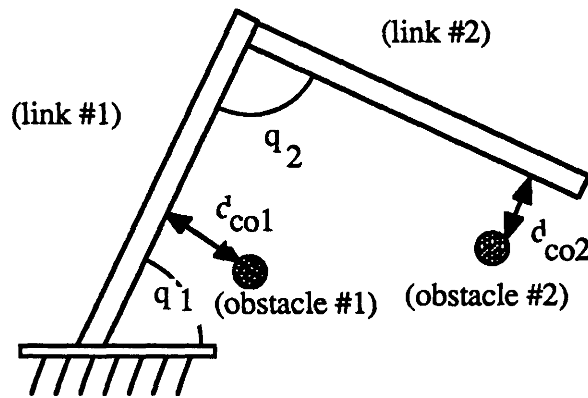


Fig 2.9 An example of two-link manipulator

Here, obstacle positions and joint angle q_2 are fixed. Then the obstacle clearance is the function of the joint angle q_1 alone. The curve for this function should look like Fig 2.10.

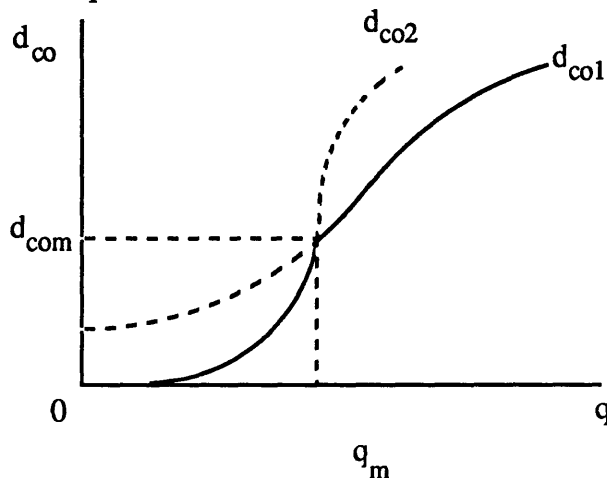


Fig 2.10 Unsmooth curve

The point (q_m, d_{com}) is one where the distance between link #1 and obstacle #1 ($= d_{co1}$) is the same as that between link #2 and obstacle #2 ($= d_{co2}$). And, when q_1 is smaller than q_m, d_{co1} is smaller than d_{co2}, and vice versa. Therefore there is a discontinuity in the partial derivative $\partial d_{co}/\partial q_1$ at $q_1 = q_m$. At this point, the steepest descent method does not work very well. What actually happens near this point, when applied to the manipulator arm trajectory planning problem, is that the joint rate just oscillates back and forth. To get around this problem, the index function is modified to include the distance between every link and obstacles:

$$f(q) = k_{eg} d_{eg}(q) + \sum \frac{k_i}{d_{coi}(q)} \tag{2.44}$$

d_{coi} : the obstacle clearance of link i.

This reduces the problem considerably. For example, the gradient can be defined practically everywhere in the case above.

The cause of the problem (ii) is considered to be

(1) n-dimensional zig-zag due to the property of the steepest descent method, as explained in the section above.

(2) the fact that the gain did not change near the goal, which caused chattering.

The cause (2) can be remedied by decreasing the gain when the end-effector approaches the goal position. So the method is modified to:

$$\dot{q} := - k_{eg}(d_{eg}) \nabla_q d_{eg} - k_{co}(d_{eg}) \nabla_q \sum \frac{k_i}{d_{coi}} \tag{2.45}$$

Here k_{eg} and k_{co} are functions of d_{eg}, which look like the figure 2.11 (a) and (b).

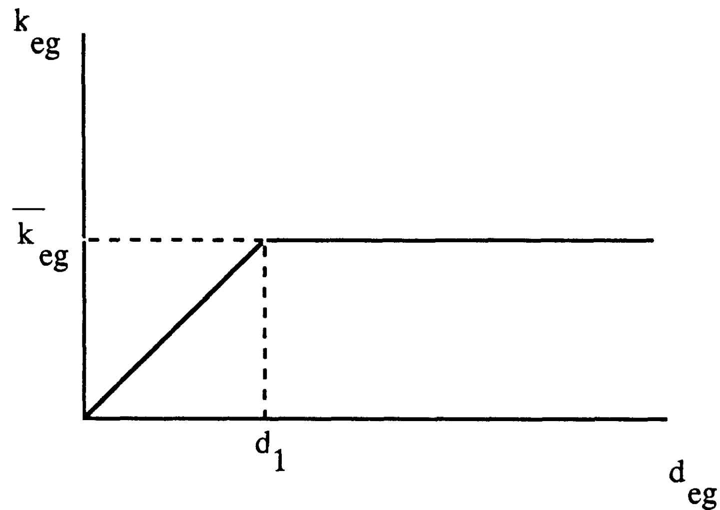


Fig 2.11 (a) $k_{eg} = k_{eg}(d_{eg})$

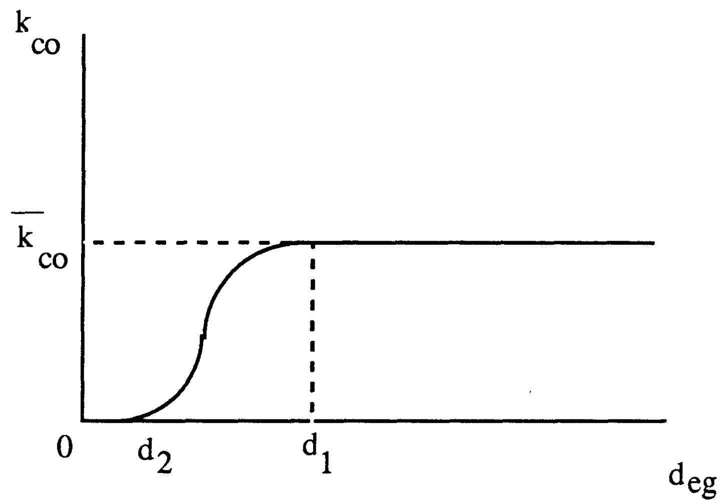


Fig 2.11 (b) $k_{co} = k_{co}(d_{eg})$

This modification reduces the noise to some extent, but it is not enough. This zig-zag movement, which is the remaining problem, is characterized by high frequency noise. So, the problem is further remedied by adding a low-pass filter. Thus the total algorithm look like Fig 2.12. Those reduces the oscillation considerably.

The third problem cannot be remedied by local schemes alone. The problem itself is a global one, in that it should be checked with higher-level control in hierarchical robot control systems, such as human monitoring.

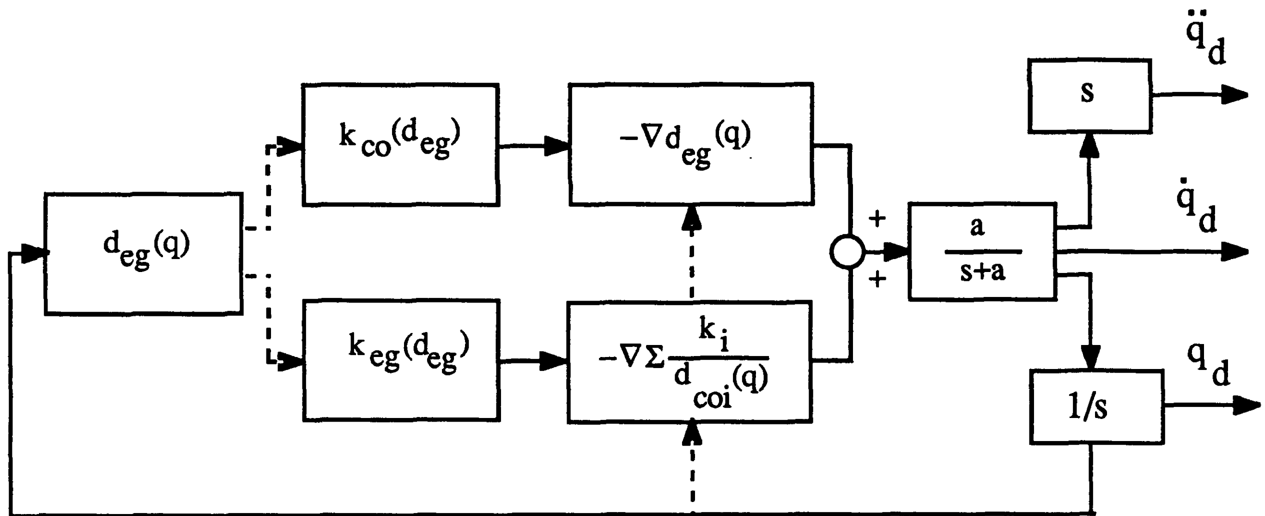


Fig 2.12 Modified steepest descent method

2.3.3 Simulation

Next, some sets of simulation were conducted to check the performance of the steepest descent method. Figures 2.13 (1)-(6) show the joint angle,joint rate, joint acceleration and criteria (goal distance and obstacle clearance) history of various steepest descent methods. Each index corresponds to the following condition.

(1) without a low-pass filter or modified index function

(2) with a low-pass filter, but without modified index function

(3) with a low-pass filter and the decreased gain near the goal configuration

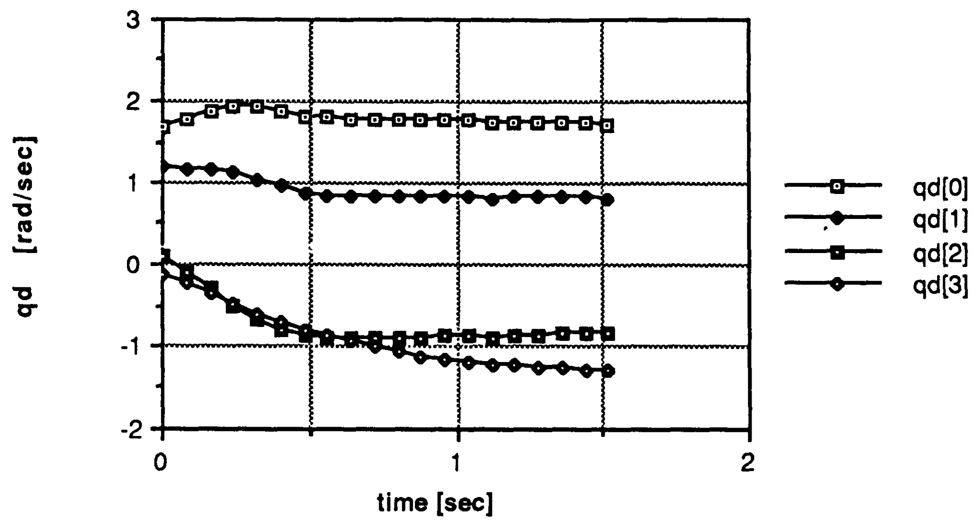
(4) without a low-pass filter, but with a modified index function that is differentiable everywhere

(5) with a low-pass filter and a modified index function

(6) with a low-pass filter, a modified function and a decreased gain near the goal configuration

The one without a low-pass filter nor modified index function does not work well as shown in Fig. 13 (a). This situation is not much improved simply by putting a low-pass filter (see Fig. 13 (b)). If the gain is decreased near the goal configuration, however, the oscillation is decreased considerably (see Fig. 13 (c)). Next, the index function is modified so that it becomes differentialble everywhere (see Fig. 13 (d)). This does not eliminate the oscillation near the goal configuration, but eliminates some oscillation (see Fig. 13 (a) or compare Fig 13 (c) and (f). Then a low-pass filter is added to reduce this oscillation (see Fig. 13 (e)). The magnitude of oscillation is reduced considerably, but it sill remains. Finally the gain near the goal configuration is decreased to reduce the chattering. This reduced the chattering by a satisfactory amount (see Fig. 13 (f)).

Joint angle history (steepest descent method #1)



Joint rate history (steepest descent method #1)

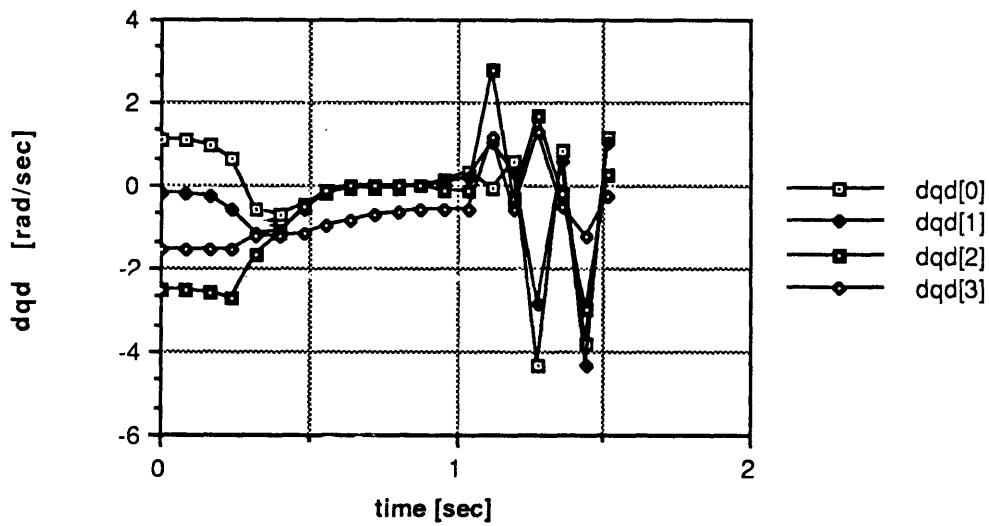
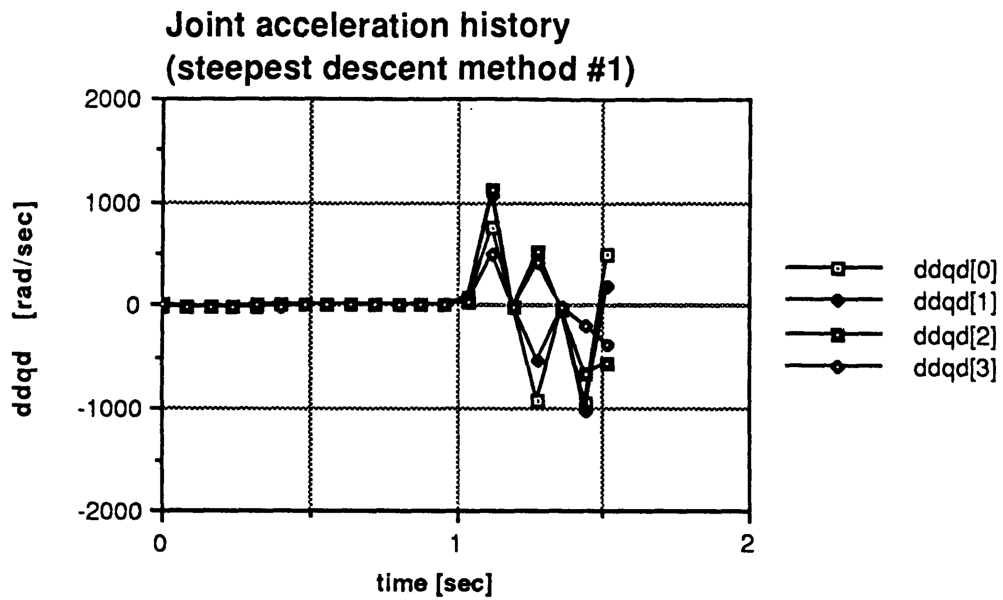


Fig 13 (a)



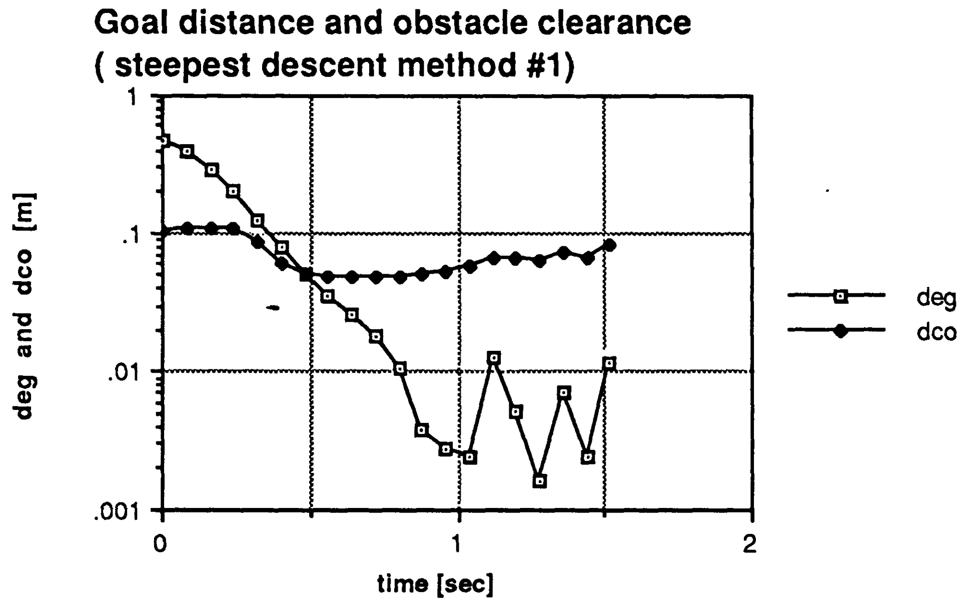
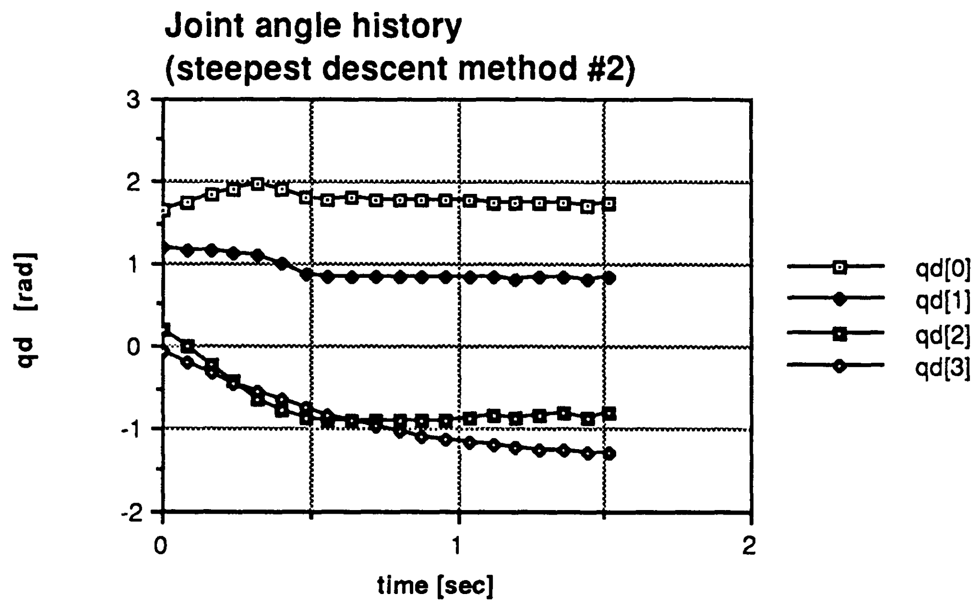


Fig 13 (a)



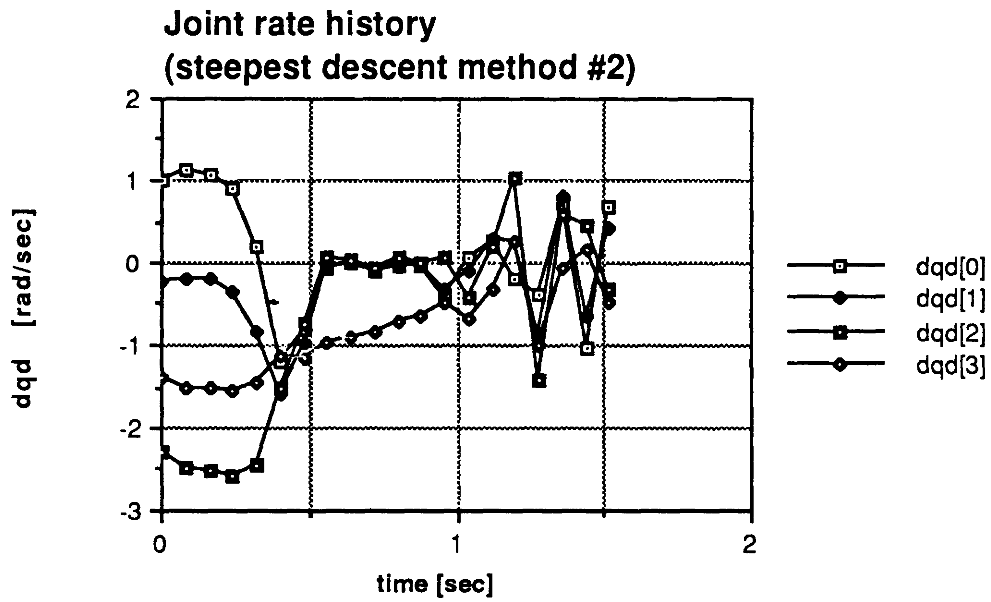
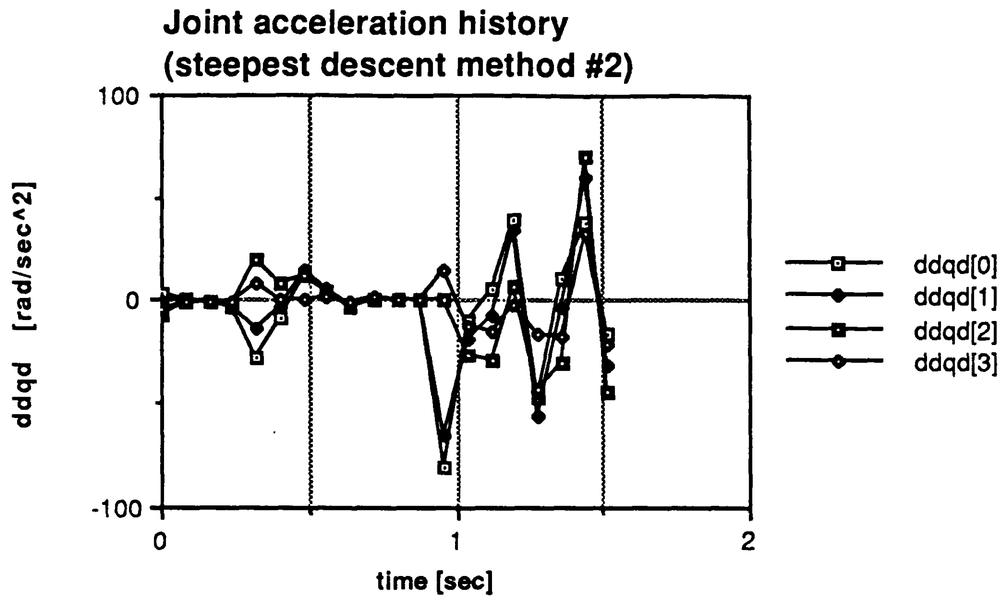


Fig 13 (b)



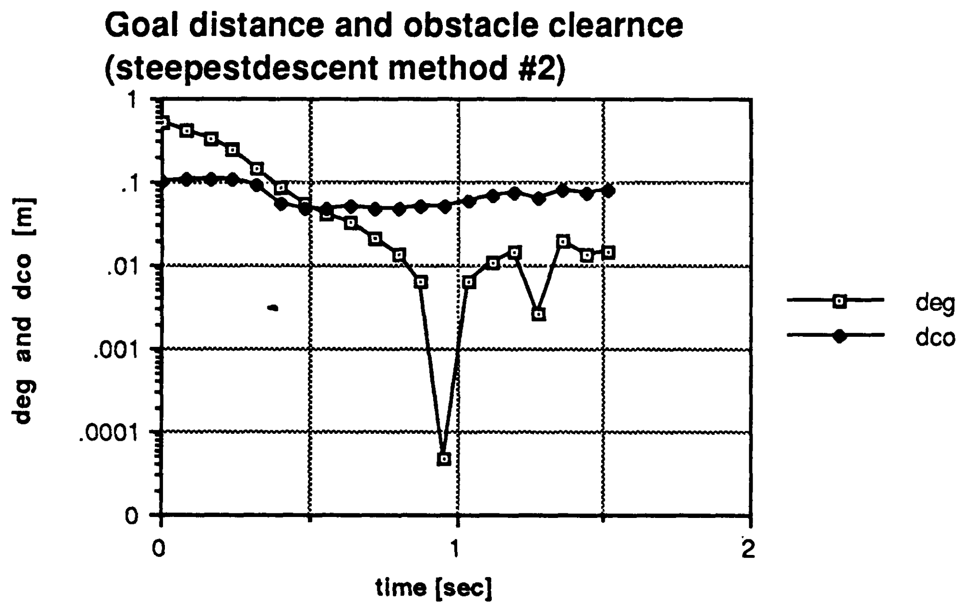
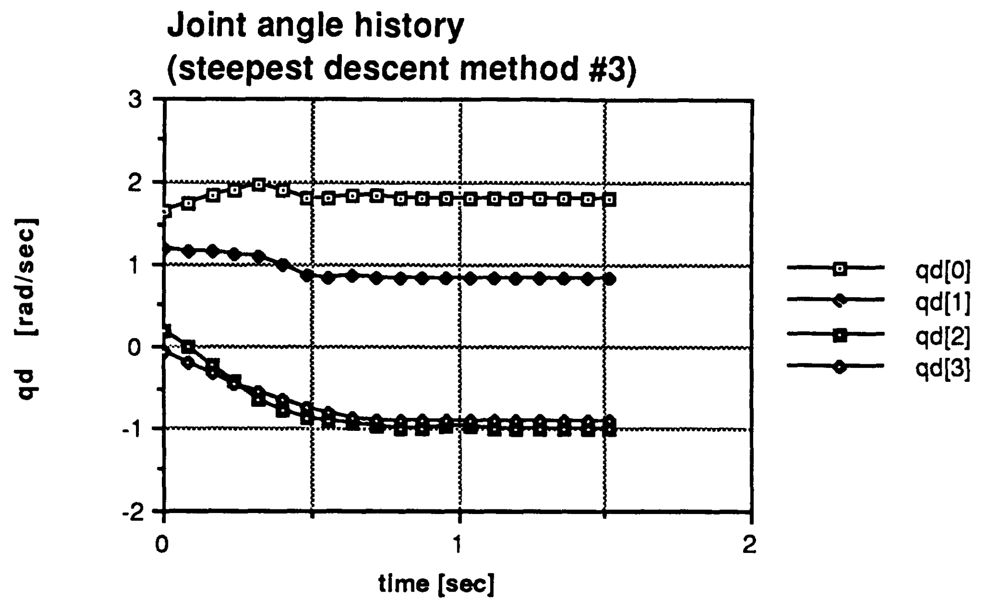


Fig 13 (b)



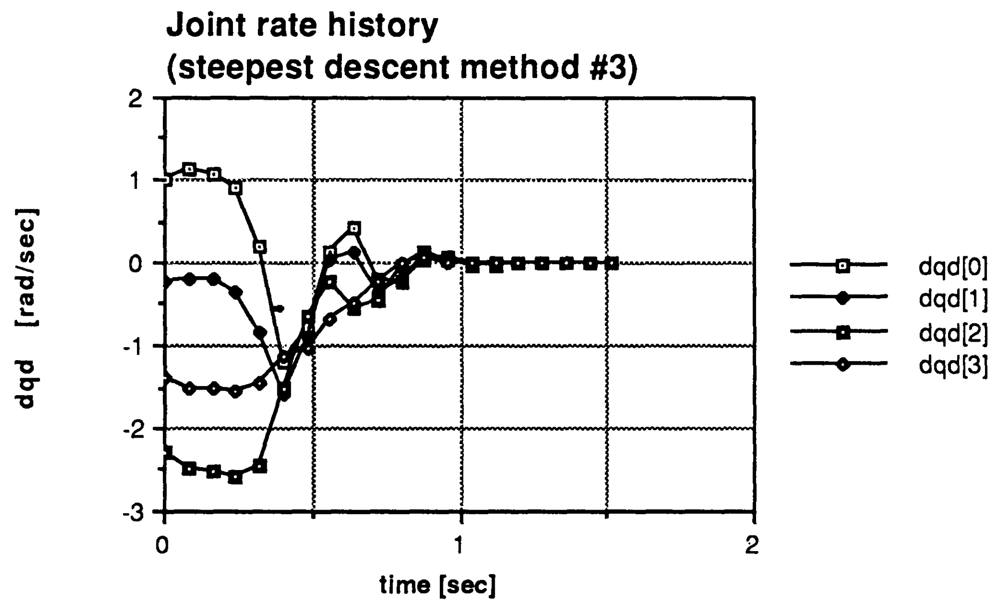
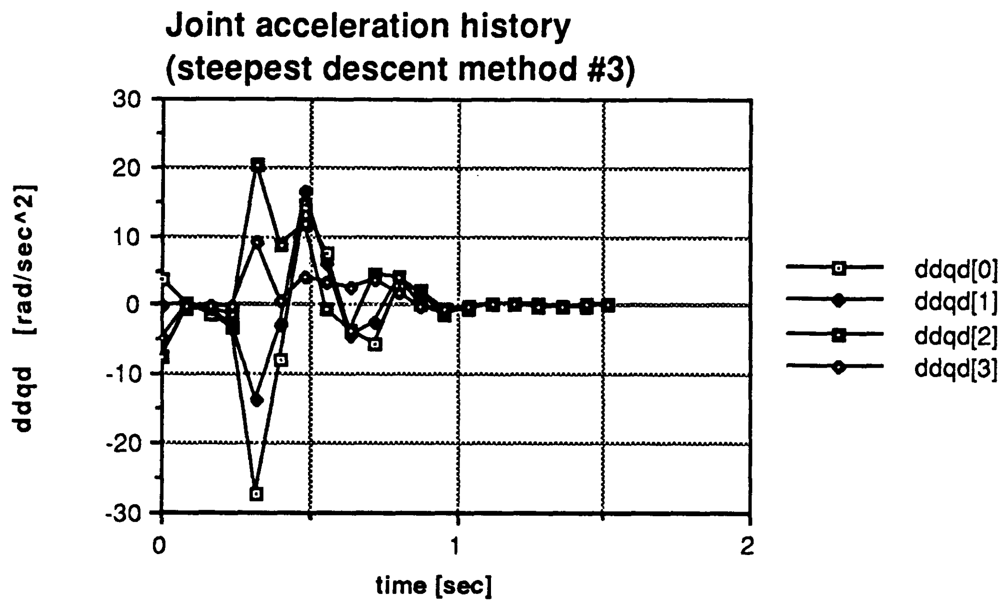


Fig 13 (c)



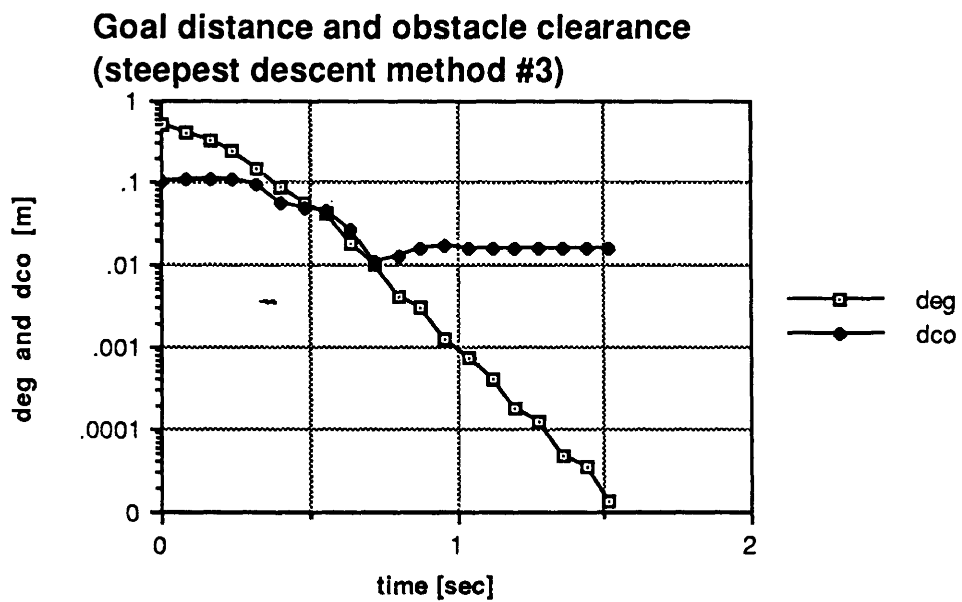
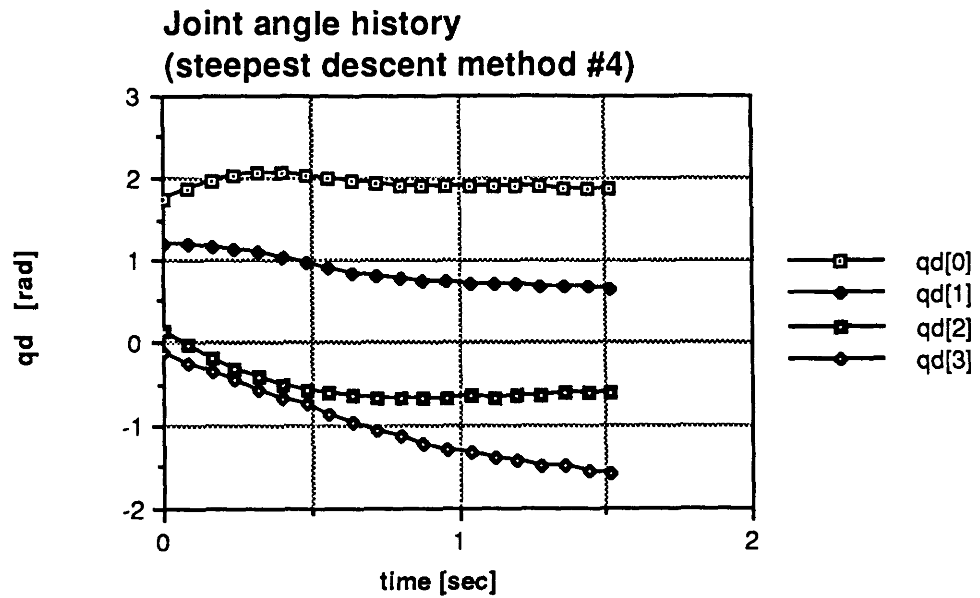


Fig 13 (c)



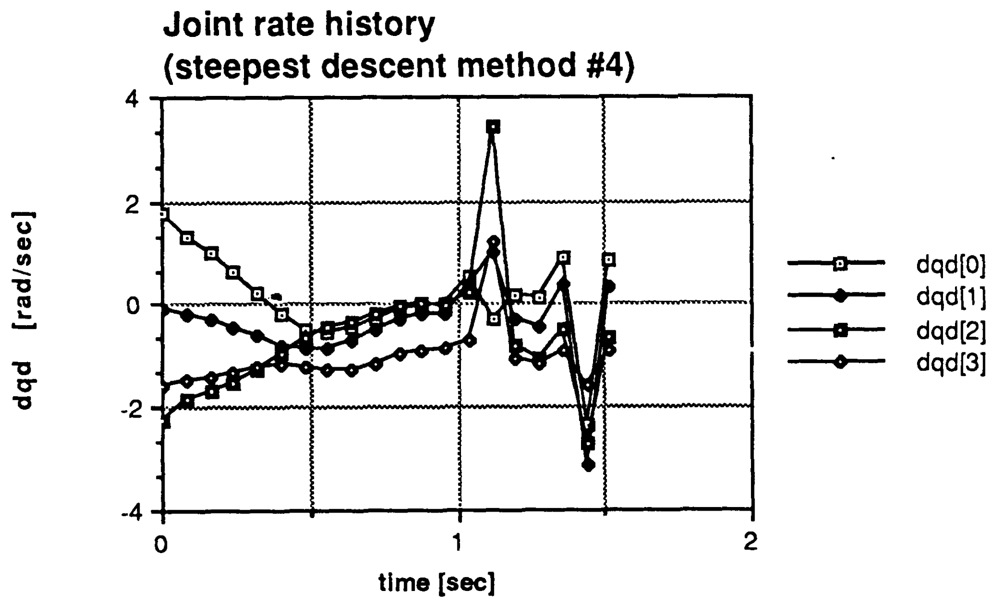
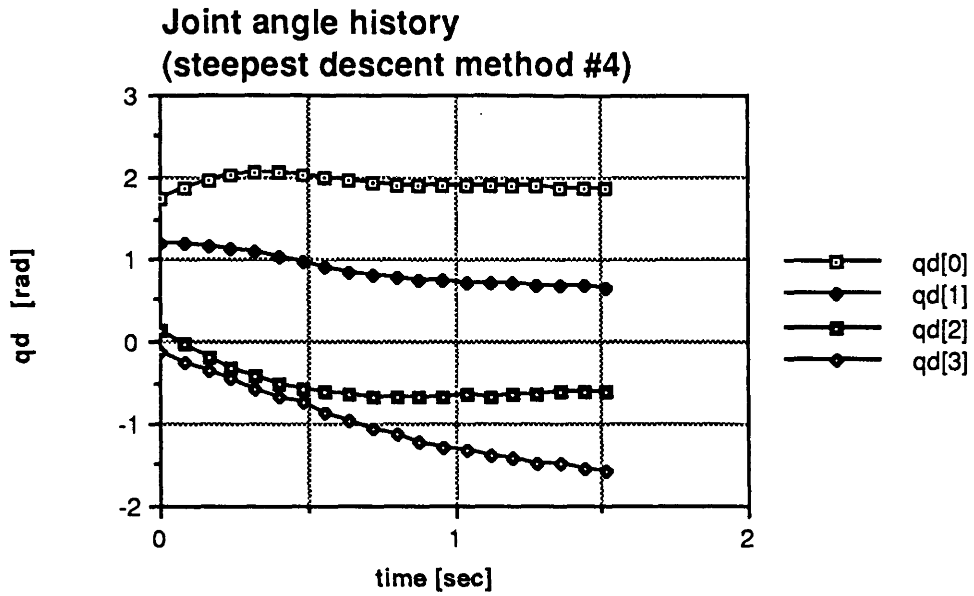


Fig 13 (d)



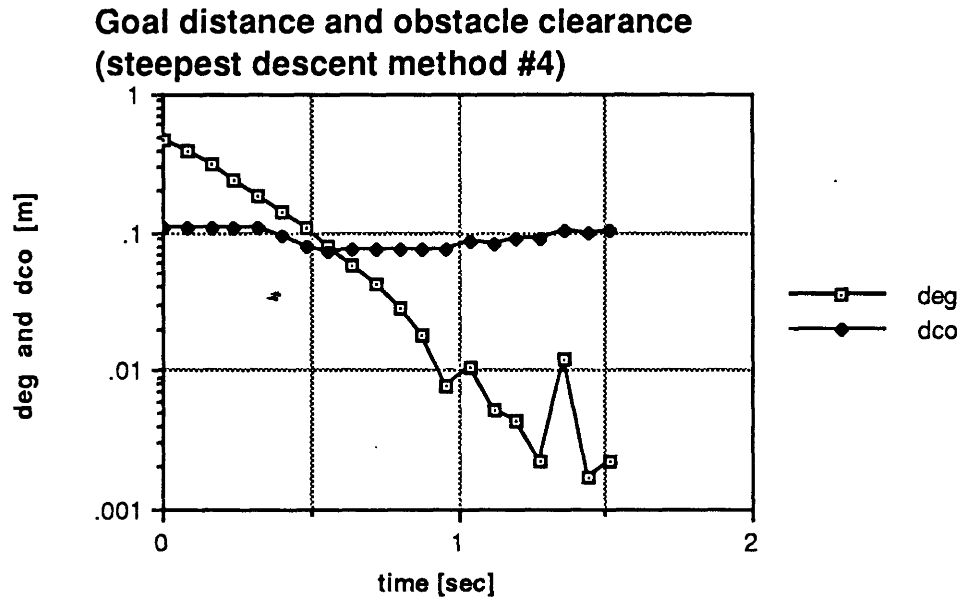
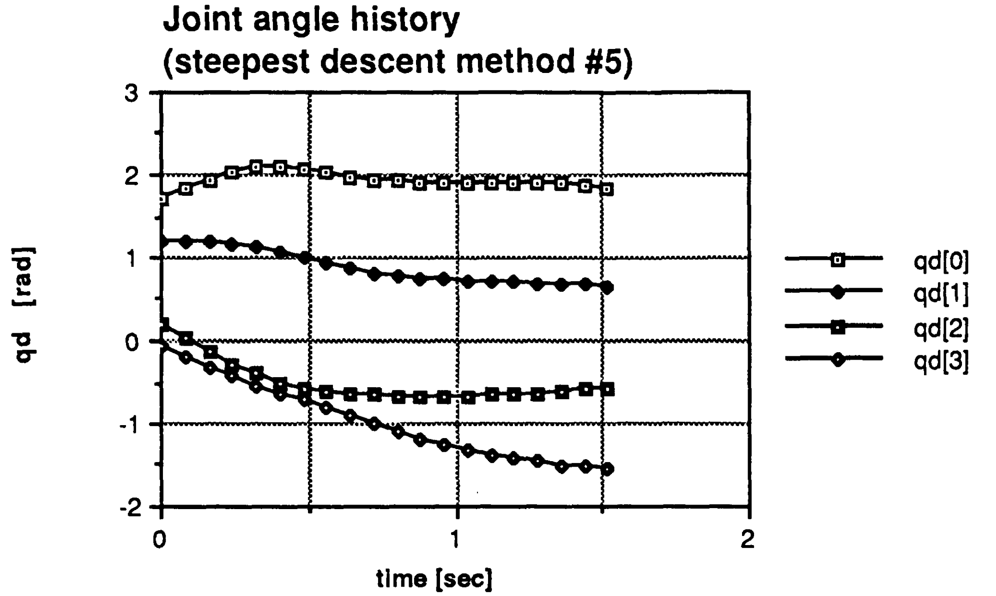


Fig 13 (d)



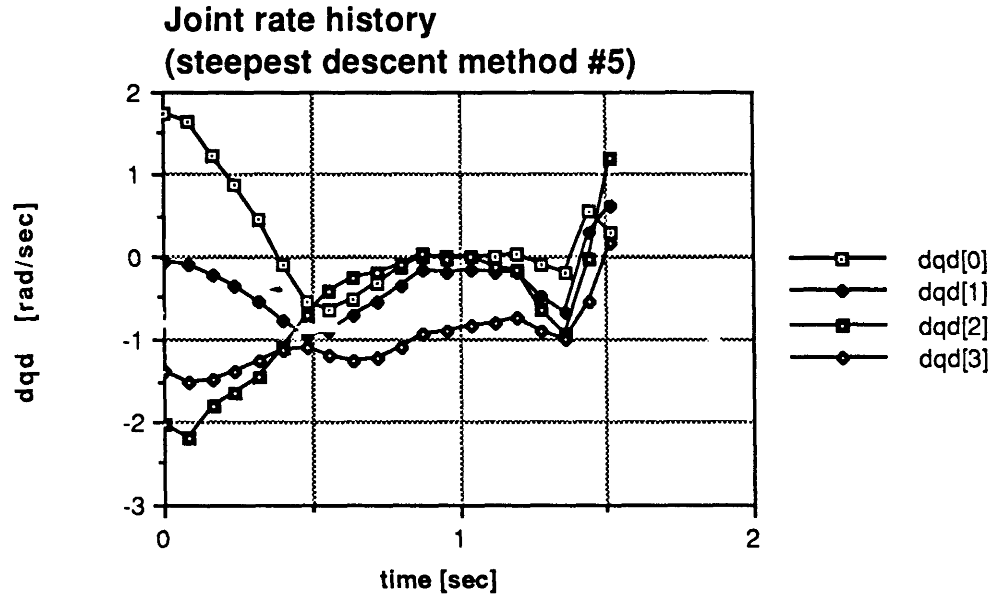
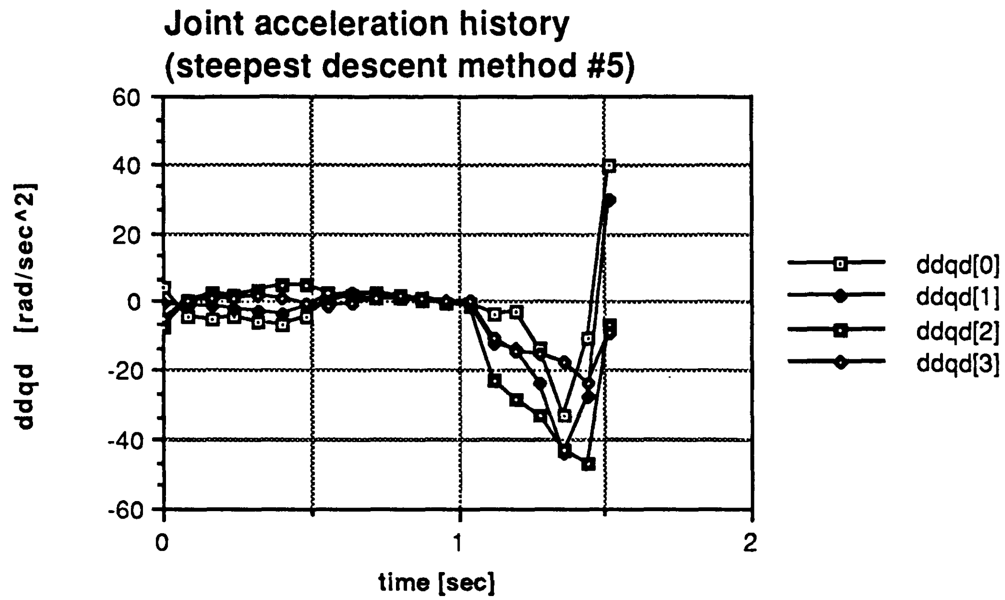


Fig 13 (e)



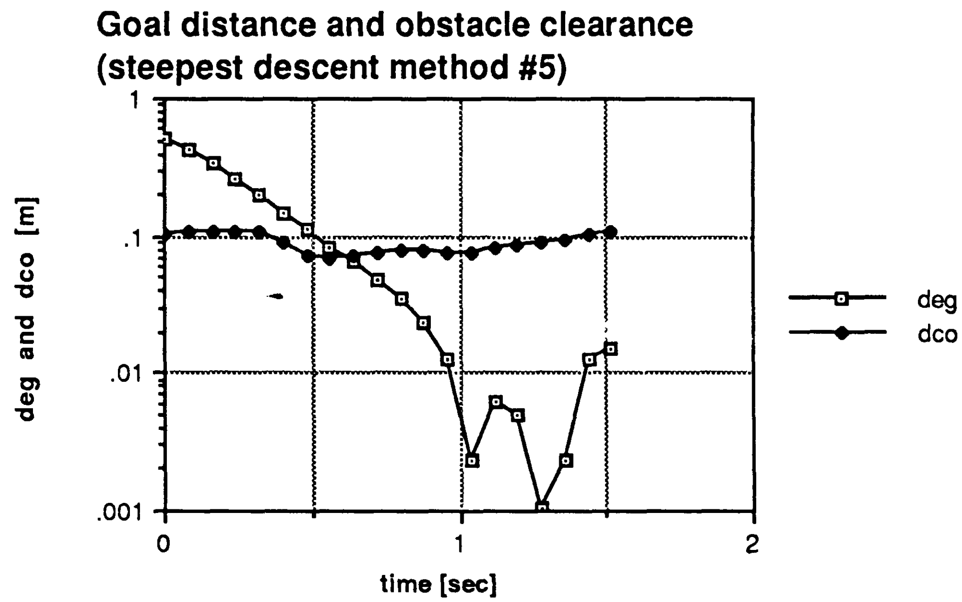
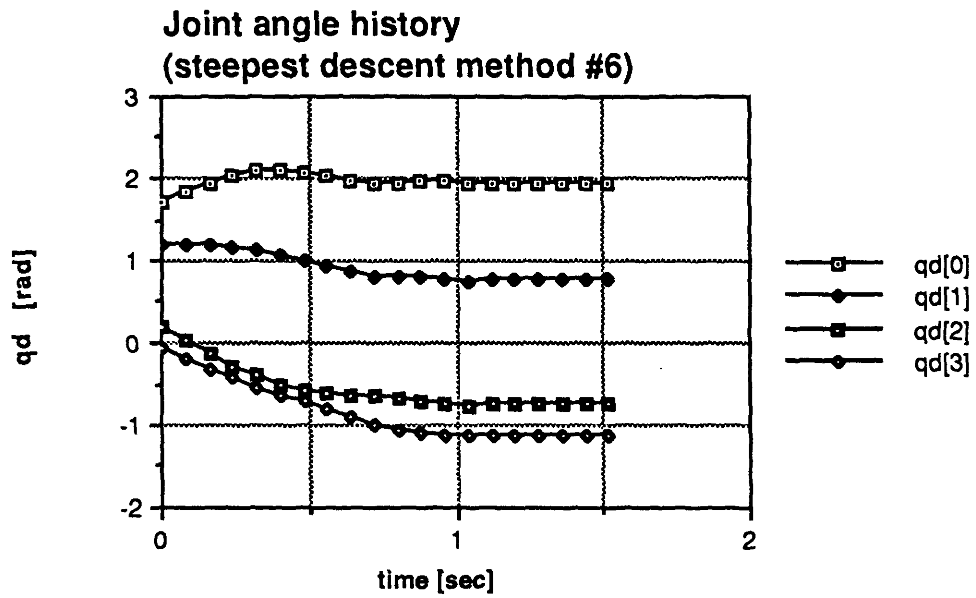


Fig 13 (e)



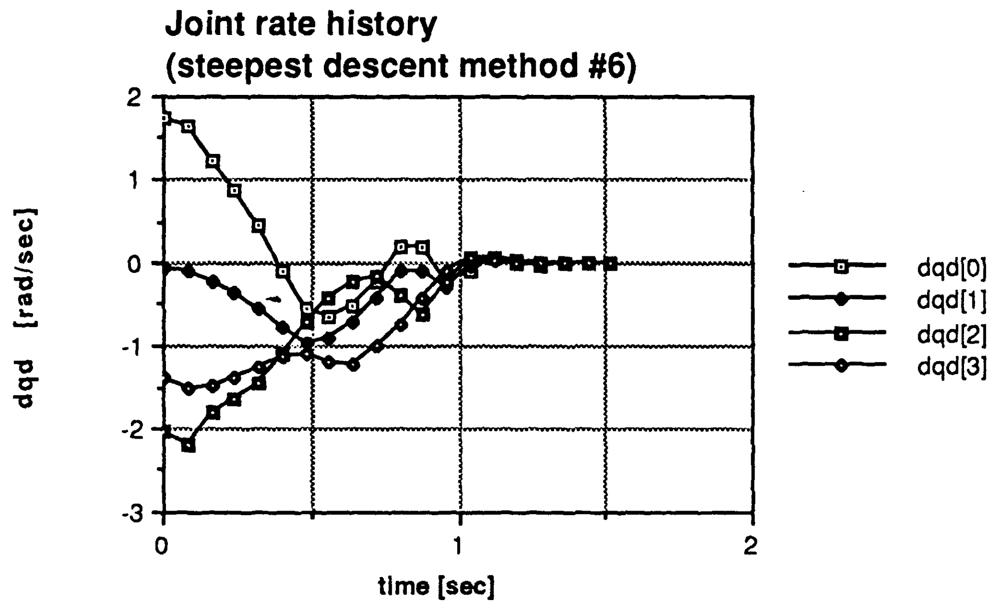
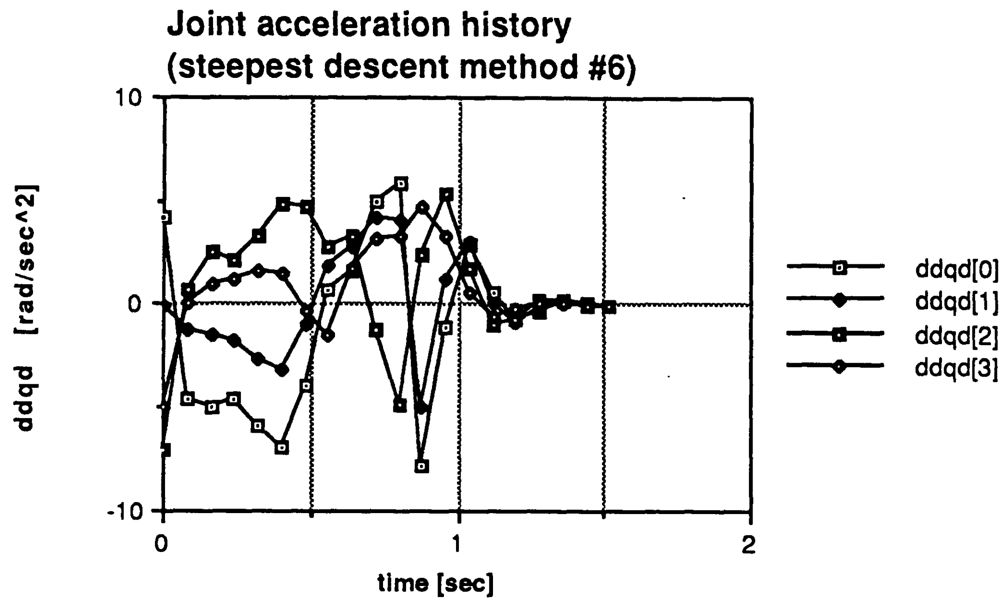


Fig 13 (f)



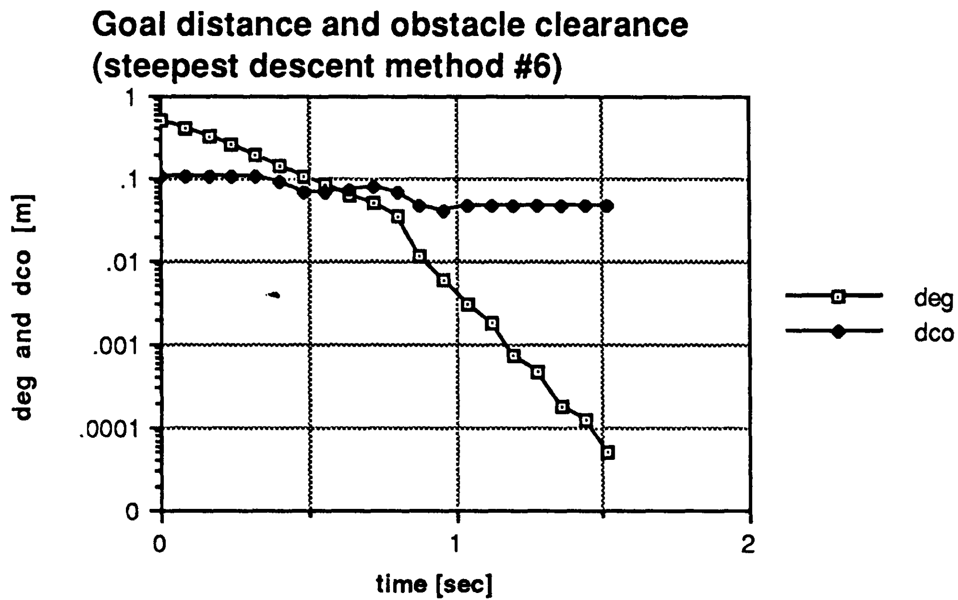


Fig 13 (f)

2.4 Comparison of the Two Methods

So far, two different trajectory planning methods have been discussed. In this section, these methods are compared with each other.

In general, it is better to have a trajectory planner program which takes less computation time, if it is to be useful in a real-time operation. The reason is that the performance of the tracking controller degrades noticeably when the bandwidth decreases. The bandwidth is the inverse of the calculation cycle time, in which both a planner and a tracking controller operates once. So both of the methods are advantageous over global schemes because they take much less computation time. The steepest descent method requires less computation time than the pseudo-inverse method as shown in the case of planar manipulator (Table 2.1). This is because pseudo-inverse method requires a number of matrix operations, as explained in the section 2.2. Also, as shown in the appendix, the steepest descent method has a relatively easier computation procedure. So in this sense, the steepest descent method is preferable to pseudo-inverse method as a real-time trajectory planner. Roughly speaking, it is desirable to have a tracking control algorithm which can operate at more than 100Hz of bandwidth. Therefore the calculation cycle time still needs to be reduced considerably. However, the simulation here indicates the good possibility of future implementation.

	The Pseudo-inverse Method	The Steepest Descent Method
Calculation Cycle Time [sec]	0.116	0.069

Table 2.1 Calculation Cycle Time (Microprocessor : 80286 + 80287)

The trajectories generated are satisfactorily smooth for both methods, without any undesired high-frequency oscillation. The one generated by the pseudo-inverse method is especially smooth, even without low-frequency oscillations. The steepest descent method originally had a higher-frequency oscillation problem, which was remedied by modification, such as a low-pass filter, gain adjustment near the goal and modification of the index function (2.3). But, compared to the pseudo-inverse method, it still creates some lower-frequency oscillation, which then causes higher joint acceleration. Those oscillation, however, are not so large as to cause a serious problem to the planner nor to the controller.

There is one serious problem with the pseudo-inverse method. The method occasionally falls into a singular configuration. In this case, the determinant of the matrix product of the Jacobian and its transpose becomes too small, so that it creates computational instability. It often creates joint rates which are extremely large, so that the joint angle just jumps in an uncontrollable manner. That means this method also requires some algorithm to check and avoid singular configuration. Such an algorithm might add to the computation time, as well as the complexity of the system. This might also degrade the performance, such as in obstacle clearance.

As to singularity avoidance program, several methods have been proposed. In one of those methods, the joint rates which avoids singularity configuration are calculated as a task[2]. If the determinant becomes low, this task is incorporated by using a technique similar to that of pseudo-inverse method.

Pseudo-inverse method has an advantage over steepest descent method in that its end-effector motion can be exactly as specified. For example, the end-effector can move on a straight line, as in welding. Because the steepest descent method simply adds the two sets of joint rates (i.e. the joint rates for end-effector motion and that for collision avoidance), neither of the tasks is done exactly as desired. Rather, the resulting joint rates are the compromise of the two conflicting tasks. However, in this method, the gain for each task is always adjustable, so that the gains can be chosen at each time instant

according to the surrounding environment. For example, if the obstacle clearance is large enough, the gain for obstacle avoidance can be dropped to zero, so that the end-effector can move as exactly desired.

In summary, both methods have their own advantages and drawbacks. It is more appropriate to remark that these two can serve for different purposes. The pseudo-inverse method should be applied to the situation where a specific end-effector trajectory has to be attained (e.g. welding and painting), and where there is less fear about singularity. The steepest descent method should be applied to the problem in which only the start and the final configuration of the end-effector are specified. In this kind of problem, the steepest descent method is often more robust.

2.5 Dynamically Changing Environment

One advantage of using a local scheme for trajectory planning is that it can be applied to a dynamically changing environment, such as moving obstacles and moving targets. The concept is briefly explained below.

Consider the following sets of points in the space.

$\Omega_o(t)$: set of points in space occupied by obstacles.

$\Omega_t(t)$: set of points in space occupied by the target object.

$\Omega_m(t)$: set of points in space occupied by manipulator itself.

Those sets are functions of time, dynamically changing over time. The trajectory planning schemes explained in this chapter generate joint rates as a function of current joint angles ($q_d(t)$), goal distance ($d_{eg}(t)$) and obstacle clearance ($d_{co}(t)$). By using the sets defined above,

$$q_d(t) = q_d(\Omega_m(t))$$

$$d_{eg}(t) = d_{eg}(\Omega_t(t), \Omega_m(t))$$

$$d_{co}(t) = d_{co}(\Omega_o(t), \Omega_m(t))$$

Therefore the trajectory planner can be expressed as,

$$
\begin{aligned}
\dot{q}_d &:= f\big(q_d(t),\, d_{eg}(t),\, d_{co}(t)\big) \\
&= f\Big(q_d(\Omega_m(t)),\, d_{eg}(\Omega_t(t),\, \Omega_m(t)),\, d_{co}(\Omega_o(t),\, \Omega_m(t))\Big) \\
&= f\Big(\Omega_o(t),\, \Omega_t(t),\, \Omega_m(t)\Big)
\end{aligned}
$$

This shows that the trajectory planner is capable of dealing with dynamically changing environment.

An example of trajectory planning problem in a dynamically changing environment is that of a dual arm manipulator. In planning the trajectory of one arm, the other arm is regarded as a moving obstacle to that arm. The only difference from normal moving obstacle is that the dynamics of the obstacle are governed by the movement of the arm itself.

Simulations were conducted to check the performance of the trajectory planner in dynamically changing environments. Figure 2.14 shows the manipulator trajectory avoiding moving an obstacle. Figure 2.15 shows a dual-arm trajectory. Both result show the advantage of the trajectory planner using local scheme.

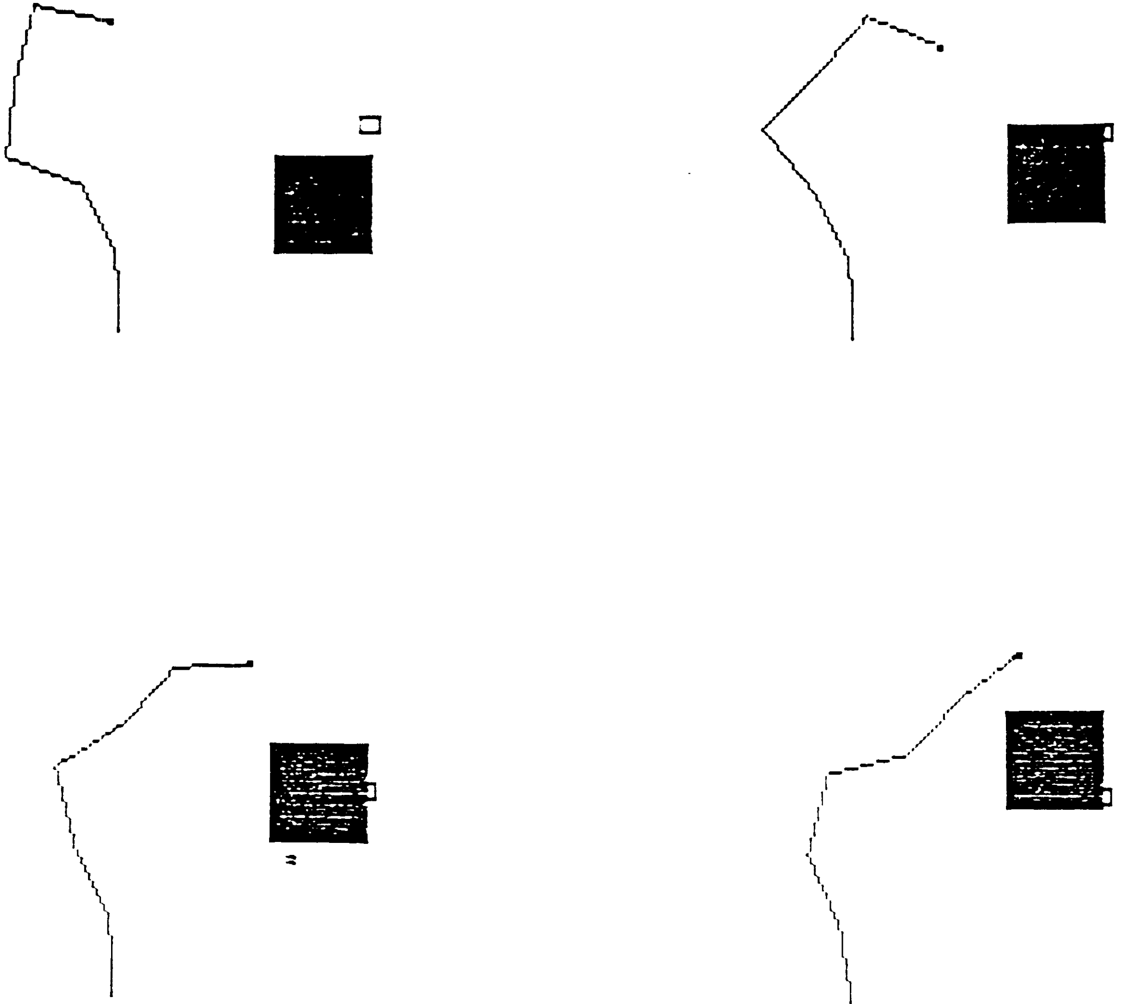


Fig. 2.14

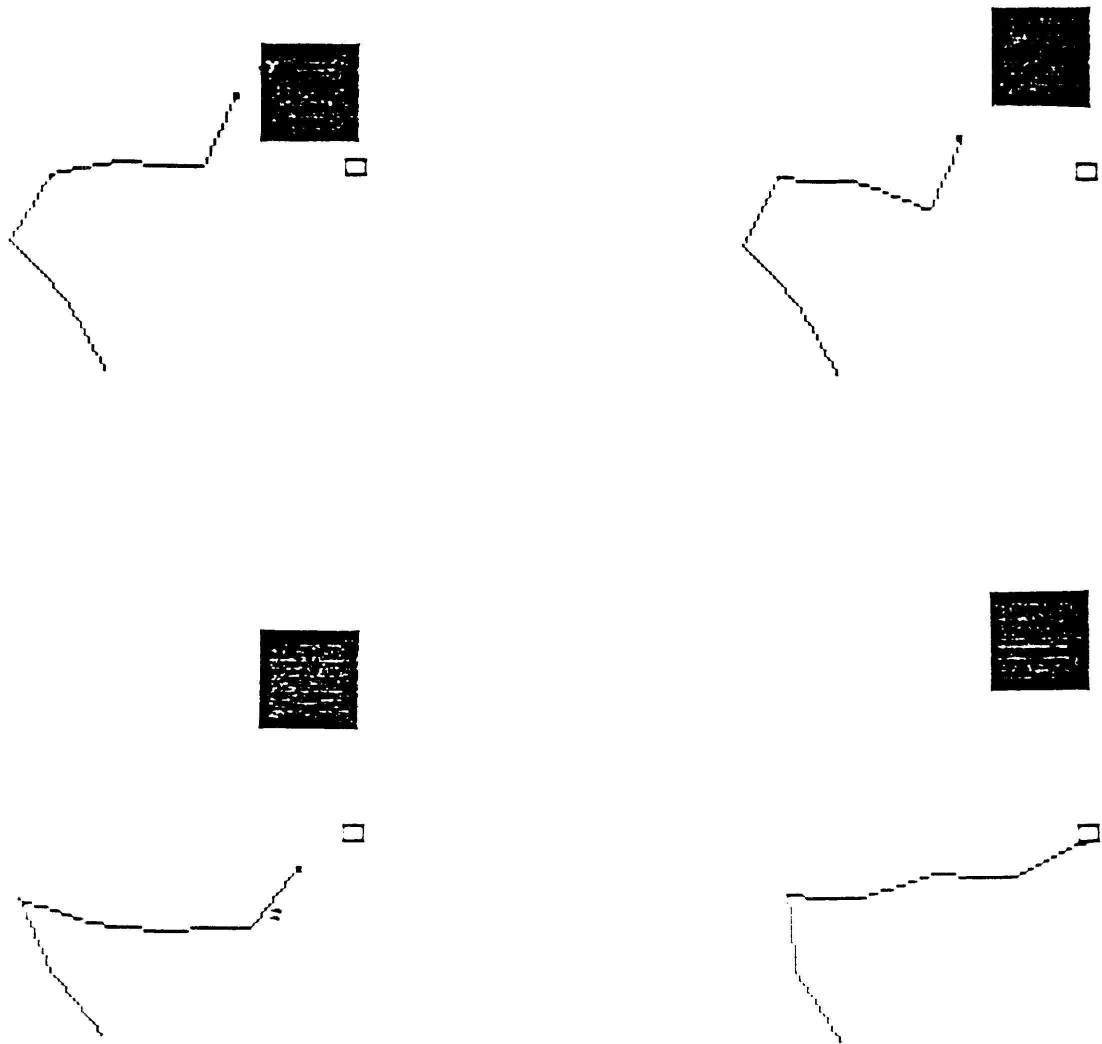


Fig. 2.14

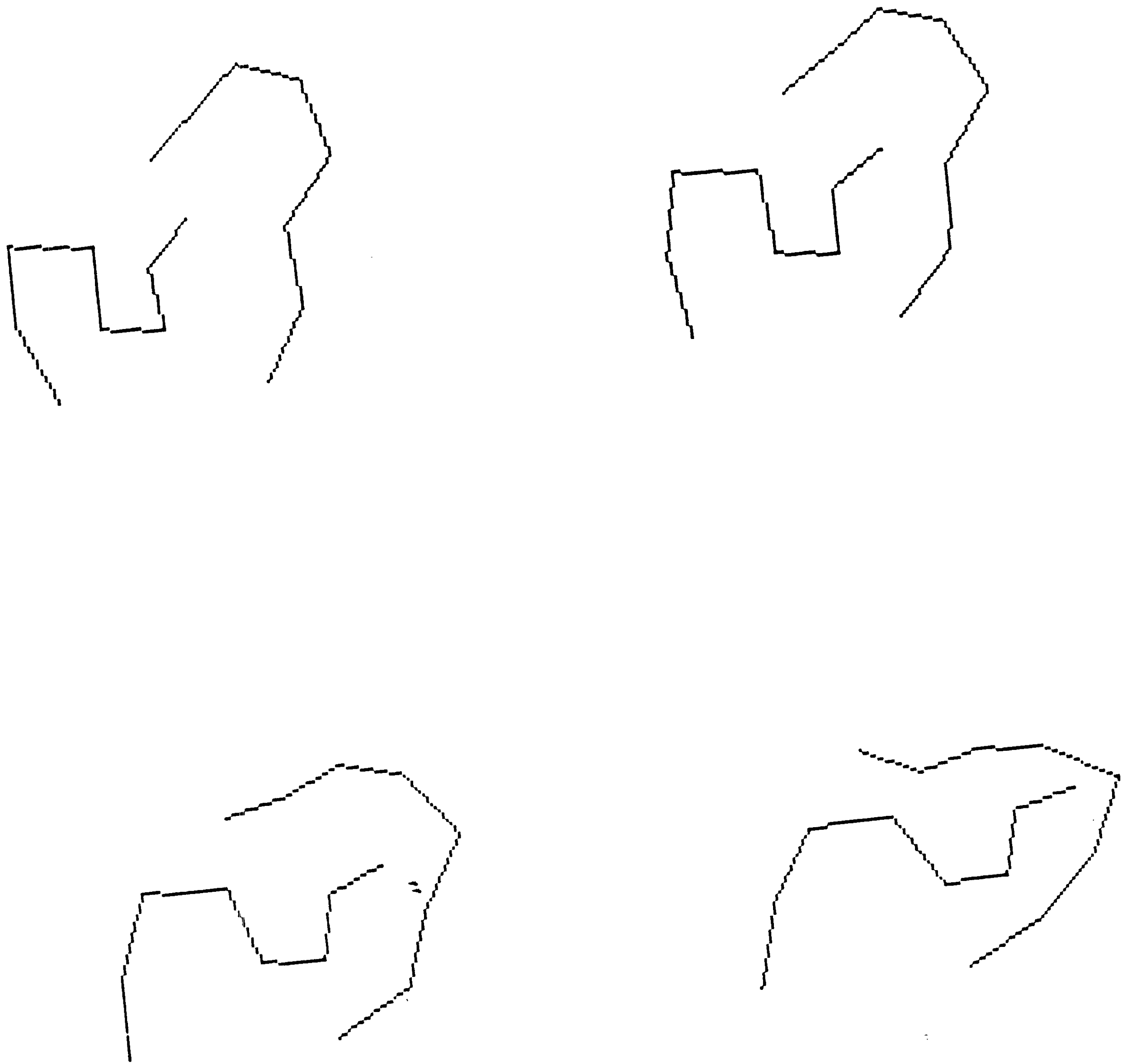


Fig. 2.15

3. TRACKING CONTROL

The desired trajectories generated have to be tracked by a manipulator to achieve the given objective. Therefore a tracking controller with satisfactory performance is necessary. The task of the tracking controller is to produce torque outputs which best minimize the errors between current joint state and that generated by the trajectory planner (Fig 3.1). In this chapter, the computed torque method and the sliding surface control method are discussed. It is shown that the sliding surface method improves robustness considerably from the computed torque method with a few modifications.

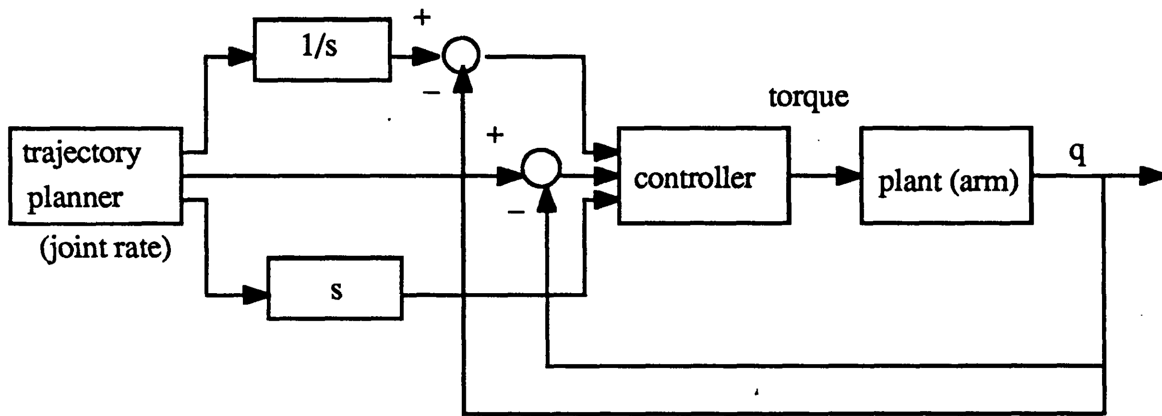


Fig. 3.1 Manipulator arm controller algorithm

3.1 Tracking Control Using Computed Torque Method

The simplest form of controller would be the individual joint PID controller, which uses the following rule.

$$\tau_j := \ddot{q}_{dj} - k_{jD}\dot{\tilde{q}}_j - k_{jP}\tilde{q}_j - k_{jI}\int_0^t \dot{\tilde{q}}_j dT$$

$$j = 1,\ldots,n \tag{3.1}$$

The scalar coefficient k_D, k_P and k_I are chosen to be positive and sufficiently large. This controller works for the position control problem. But because of the nonlinearity of the

dynamics of manipulators, this does not show good trajectory tracking performance. One way to tackle this problem is to make use of the dynamics equation,

$$H\underline{u} + \underline{h} + \underline{g} = \underline{\tau} \tag{3.2}$$

The idea of the computed toque method is to define control torque τ using a structure identical to that of the dynamics

$$\underline{\tau} := H\underline{u} + \underline{h} + \underline{g} \tag{3.3}$$

so that the problem ideally reduces to that of controlling

$$\ddot{\underline{q}} = \underline{u} \tag{3.4}$$

This represents a set of n decoupled double-integrators, each of which can be controlled independently using a simple P.D.

$$u_j := \ddot{q}_{dj} - k_{jD}\dot{\tilde{q}}_j - k_{jP}\tilde{q}_j$$

$$j = 1,\ldots,n \tag{3.5}$$

or P.I.D.

$$u_j := \ddot{q}_{dj} - k_{jD}\dot{\tilde{q}}_j - k_{jP}\tilde{q}_j - k_{jI}\int_0^t \tilde{q}_j dT$$

$$j = 1,\ldots,n \tag{3.6}$$

where $\ddot{q}_{dj}$ is the acceleration of the desired trajectory of joint j. Here the symbol (~) denotes an error. For instance,

$$\tilde{q} = q - q_d \tag{3.7}$$

The joint torque vector τ then can be computed from the given u.

The major problem to this approach is its robustness. This is because only estimates, $\hat{H}$ and $\hat{h}$, of H and h are available in practice. There is always parametric uncertainty due to inaccuracies on the manipulator mass properties, the torque constants of the actuators, the lack of a good model of friction, unknown loads, and so on. Thus we can only apply

$$\underline{\tau} := \hat{H}\underline{u} + \hat{\underline{h}} + \hat{\underline{g}} \tag{3.8}$$

so that

$$\ddot{\underline{q}} = (H^{-1}\hat{H})\underline{u} + H^{-1}(\hat{\underline{h}} - \underline{h}) + H^{-1}(\hat{\underline{g}} - \underline{g}) \tag{3.9}$$

This expression shows that the problem is not as simple as a pure linear control problem. The nonlinearity may degrade the controller system performance. Therefore its robustness, i.e. the performance sensitivity to model uncertainty, turns out to be low.

3.2 Tracking Control Using Sliding Surfaces

The idea of using "sliding surfaces" for control was first developed by Utkin [12] for stabilizing nonlinear systems and later by Slotine [1] for nonlinear tracking.

Although PID control works relatively well for position control applications, it has undesirable characteristics, such as overshooting or slow response.

Sliding mode control takes into account the following requirements:

1) parameter uncertainties

2) the presence of high-frequency unmodeled dynamics

It also explicitly quantifies the resulting modeling / performance trade-offs.

Manipulator dynamics typically has following form of dynamic equations.

$$H\ddot{\underline{q}} + C\dot{\underline{q}} + \underline{g} = \underline{\tau} \tag{3.10}$$

$\underline{q}$: joint coordinate

$\underline{g}$: gravity force

$\underline{\tau}$ torque

H : inertia matrix

C : Coriolis and centrifugal force matrix

The problem is to get the joint angle $\underline{q}$ and joint rate $\dot{\underline{q}}$ to track a desired state $\underline{q}_d$ and $\dot{q}_d$ in the presence of model imprecision on H, C and $\underline{g}$. Let

$$\tilde{\underline{q}} := \underline{q} - \underline{q}_d \tag{3.11}$$

be the tracking error. Define a time-varying sliding surface S(t) in the state-space by the equation $s(\underline{x}:t) = 0$, where

$$s(\underline{x}:t) = (\frac{d}{dt} + \Lambda)\tilde{q} \tag{3.12}$$

and λ is a positive constant. The design parameter can be interpreted as the desired control bandwidth. Suppose the initial condition is given as

$$\underline{q}(t=0) = \underline{q}_d(t=0)$$
$$\dot{\underline{q}}(t=0) = \dot{\underline{q}}_d(t=0) \tag{3.13}$$

The problem of tracking $(\underline{q},\dot{\underline{q}}) = (\underline{q}_d,\dot{\underline{q}}_d)$ is equivalent to that of remaining on the surface S(t) for all t>0. Thus the problem reduces to keeping the vector s at zero. This can be achieved by choosing a control law u such that outside of S(t),

$$\frac{d}{dt}\left(\frac{1}{2}\ \underline{s}^T H \underline{s}\right) \leq -\eta|\underline{s}| \tag{3.14}$$

where η is a positive constant. Inequality constrains the trajectories to point towards the surface S(t), and is referred to as the sliding condition. The idea behind this control design principle is to pick up a well-behaved function of the tracking error, $\underline{s}$, and then select the feed back control law u in that $\frac{1}{2}(\underline{s}^T H \underline{s})$ remains the Lyapnov function of the closed-loop system, despite the presence of imprecision and of disturbances. Further, if $(\underline{q},\dot{\underline{q}})|_{t=0}$ is off $(\underline{q}_d,\dot{\underline{q}}_d)|_{t=0}$, the surface S(t) will nonetheless be reached in a finite time smaller than $|s(t=0)|/\eta$. This can be shown by integrating the above equation. Once on the surface, tracking error tends exponentially to zero, with a time constant $1/\lambda$.

The control design procedure consists of two steps. First a feedback control law u is selected to verify the sliding condition. To illustrate this, consider the control design for a second-order system, specifically that of manipulator. The error in this case is

$$\begin{aligned} \underline{s}(t) &= (\dot{\underline{q}} - \dot{\underline{q}}_d) + \Lambda(\underline{q} - \underline{q}) \\ &= \dot{\tilde{\underline{q}}} + \Lambda\tilde{\underline{q}} \\ &= \dot{\underline{q}} - \dot{\underline{q}}_r \\ \dot{\underline{q}}_r &\equiv \dot{\underline{q}}_d - \Lambda\tilde{\underline{q}} \end{aligned} \tag{3.15}$$

As explained above, choose a Lyapnove function,

$$V = \frac{1}{2}\underline{s}^T H \underline{s} = \|\underline{s}\|_H^2 \tag{3.16}$$

By differentiating this function, the following equation is derived.

$$\dot{V} = \underline{s}^T H \dot{\underline{s}} + \frac{1}{2} \underline{s}^T \dot{H} \underline{s} \tag{3.17}$$

According to equation (3.15),

$$\dot{\underline{s}} = \ddot{\underline{q}} - \ddot{\underline{q}}_r \tag{3.18}$$

By substituting this into equation (3.17),

$$\dot{V} = \underline{s}^T H(\ddot{\underline{q}} - \ddot{\underline{q}}_r) + \frac{1}{2} \underline{s}^T \dot{H} \underline{s} \tag{3.19}$$

Again, by substituting the dynamic equation,

$$\begin{aligned} \dot{V} &= \underline{s}^T(\underline{\tau} - C\dot{\underline{q}} - \underline{g} - H\ddot{\underline{q}}_r) + \frac{1}{2} \underline{s}^T \dot{H} \underline{s} \\ &= \underline{s}^T(\underline{\tau} - C(\dot{\underline{q}}_r + \underline{s}) - \underline{g} - H\ddot{\underline{q}}_r) + \frac{1}{2} \underline{s}^T \dot{H} \underline{s} \\ &= \underline{s}^T(\underline{\tau} - C\dot{\underline{q}}_r - \underline{g} - H\ddot{\underline{q}}_r) + \frac{1}{2} \underline{s}^T (\dot{H} - 2C) \underline{s} \end{aligned} \tag{3.20}$$

C is chosen so that $\dot{H} - 2C$ becomes a skew-symmetric matrix. Therefore

$$\frac{1}{2} \underline{s}^T (\dot{H} - 2C) \underline{s} = 0$$

$$\therefore \ \dot{V} = \underline{s}^T(\underline{\tau} - C\dot{\underline{q}}_r - \underline{g} - H\ddot{\underline{q}}_r) \tag{3.21}$$

Choose $\underline{\tau}$ so that, as far as the estimated parameters are accurate,

$$\hat{H} = H$$

$$\hat{C} = C$$

$$\hat{\underline{g}} = \underline{g}$$

The symbol "^" denotes modeled quantities. Then, $\dot{V}$ becomes zero. Therefore the torque is

$$\hat{\underline{\tau}} := \hat{H}\ddot{\underline{q}}_r + \hat{C}\dot{\underline{q}}_r + \hat{\underline{g}} \tag{3.22}$$

Then, to assume the required condition ($\dot{V} \leq -\eta|\underline{s}|$), term $-\underline{k}(\underline{q},t)\mathrm{sgn}(\underline{s})$ is added.

$$\underline{\tau} := \hat{\underline{\tau}} - \underline{k}(\underline{q},t)\mathrm{sgn}(\underline{s}) \tag{3.23}$$

where

$$[\underline{k}(\underline{q},t)\mathrm{sgn}(\underline{s})]_i = k_i \mathrm{sgn}(s_i) \tag{3.24}$$

and the function sgn(x) is defined as follows.

$$\mathrm{sgn}(x) = \begin{cases} 1 & (x>0) \\ -1 & (x<0) \end{cases} \tag{3.25}$$

Then the equation (3.21) becomes

$$\begin{aligned} \dot{V} &= \underline{s}^T(\underline{\tau} - C\underline{\dot{q}}_r - \underline{g} - H\underline{\ddot{q}}_r) \\ &= \underline{s}^T\{(\hat{H} - H)\underline{\ddot{q}}_r + (\hat{C} - C)\underline{\dot{q}}_r + (\hat{\underline{g}} - \underline{g}) - \underline{k}\cdot\mathrm{sgn}(\underline{s})\} \\ &= -\underline{s}^T(\tilde{H}\underline{\ddot{q}}_r + \tilde{C}\underline{\dot{q}}_r + \tilde{\underline{g}}) - \sum k_i\cdot\mathrm{sgn}(s_i) \end{aligned} \tag{3.26}$$

where the symbol "~" denotes the error, that is,

$$\tilde{H} = H - \hat{H}$$

$$\tilde{C} = C - \hat{C}$$

$$\tilde{g} = g - \hat{g}$$

Choose k_i such that

$$k_i \geq |(\tilde{H}\underline{\ddot{q}}_r + \tilde{C}\underline{\dot{q}}_r + \tilde{\underline{g}})_i| + \eta_i \tag{3.27}$$

This control law satisfies the required condition,

$$\dot{V} \leq -\sum \eta_i \cdot |s_i| \tag{3.28}$$

Therefore the system is Lyapnov stable despite the presence of parameter uncertaintities and unmodeled dynamics. This is beneficial to a manipulator tracking control application, which requires robustness in its controller design.

However, the resulting design leads to control chattering across S(t). Thus, in the second step, the discontinuous control law u is suitably smoothed to achieve an trade-off between control bandwidth and tracking precision. This can be achieved by smoothing out the control discontinuity in a thin boundary layer neighboring the switching surface:

$$B(t) = \{\underline{x},\ |s(\underline{x}:t)| \leq \phi\}, \quad \phi > 0 \tag{3.29}$$

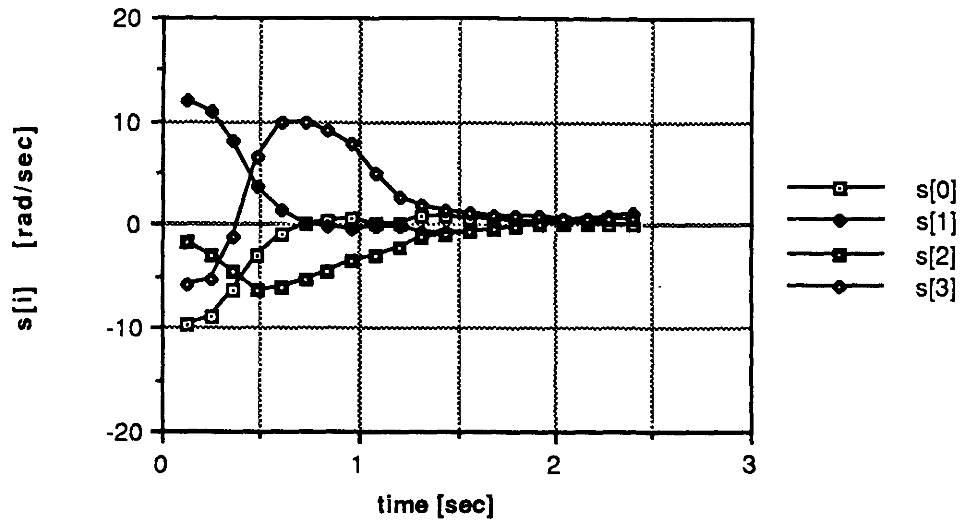
where ϕ is the boundary layer thickness, and $\varepsilon := \phi/\lambda$ is the boundary layer width. Outside of B(t), the same control law is chosen, which guarantees boundary layer attractiveness. Then the control output is interpolated inside the boundary layer. For instance, in the equation (3.23), sgn(s) can be replaced by s/ϕ inside B(t). Thus a control law which meets above condition is derived as follows.

$$\tau_i = \tau_i - k_i(\underline{q},t)\cdot sat(s_i/\phi_i) \tag{3.30}$$

and the "sat" function is

$$sat(x) = \begin{cases} 1 & x > 1 \\ x & 1 > x > -1 \\ -1 & -1 > x \end{cases} \tag{3.31}$$

Simulation are conducted to check the performance of the controllers (Fig 3.2 - 3.3). As can be seen in these figures, the sliding surface control has more accuracy and robustness than the computed torque method, which is as expected. There are a couple of remarks to be made here. First, the desired trajectory $\underline{q}_d$ itself must be chosen smooth enough not to excite the high-frequency unmodeled dynamics. Secondly, when there is a parameter uncertainty in the dynamics equation, it is sometimes better to use simplified model with less cycle time. In other words, there is a trade-off between model accuracy and bandwidth.



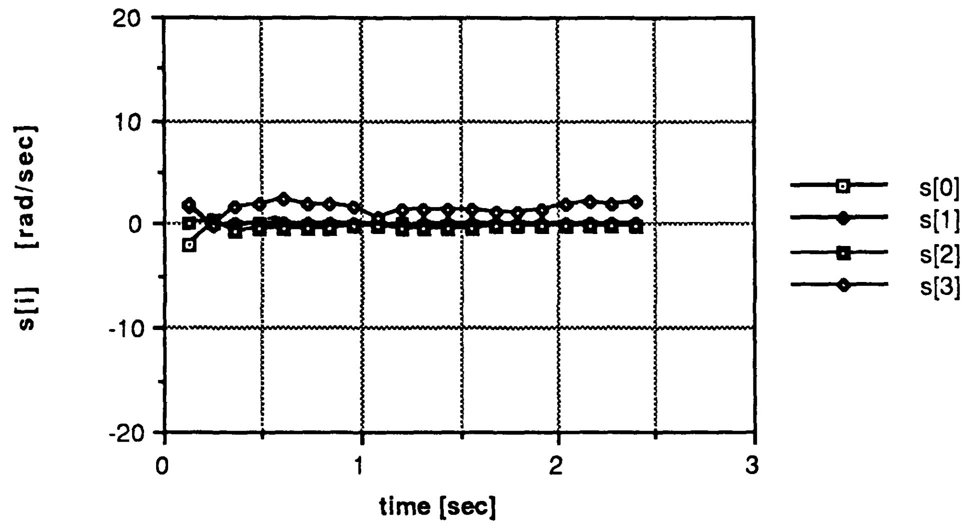
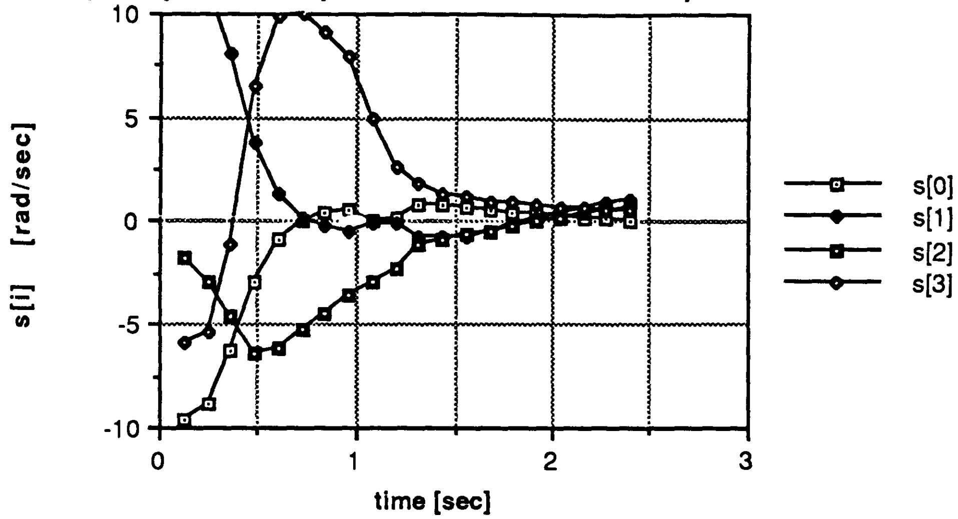


Fig 3.2



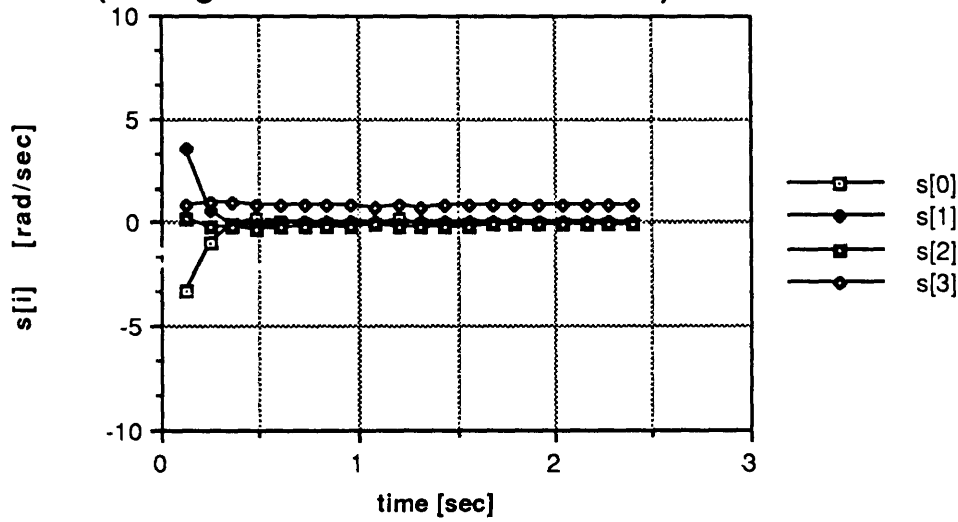
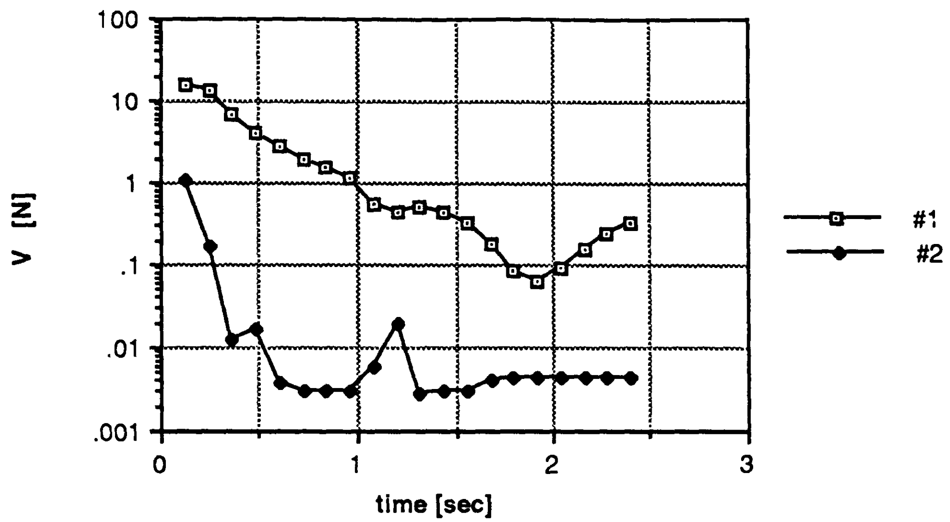


Fig 3.2



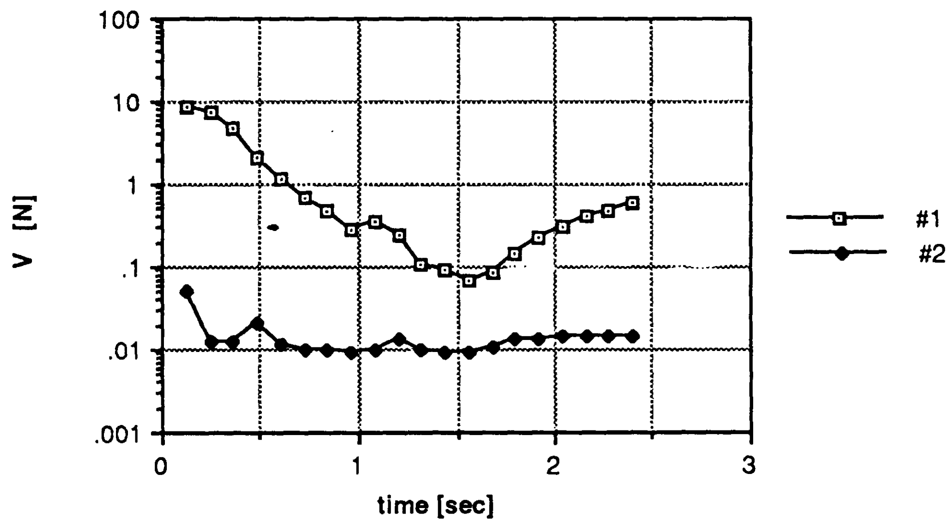


Fig 3.3

4. CONCLUSION

In this thesis, two trajectory generation schemes were developed. Both were shown to possess good capabilities for planning goal-reaching and obstacle avoidance trajectories for manipulators. The trajectories generated would be tracked by manipulators using a tracking control algorithm. The computed torque method and the sliding surface control method were discussed as tracking controllers.

Both of the two trajectory generation methods, the pseudo-inverse method and the steepest descent method, use local schemes rather than global schemes. Because of this, the methods are expected to take less computation time at each cycle, thus making it possible to apply them to real-time systems. Between them, the steepest descent method has a slight edge in this sense, due to its simpler calculation procedure, thus less computation time. In the simulation, the planner ran at 15Hz at the fastest, which is not satisfactory enough to be implemented as it is. However, further reduction in calculation time is possible, and this should be a task of future development.

Local schemes have another advantage in that they are expected to have good capabilities of coping with dynamically changing unforeseeable environments. This is because the planner only refers to the environment at each time instant, rather than referring to a predetermined model over time. Examples of dynamically changing environments include

(1) moving obstacles

(2) moving target

(3) dual arm collision avoidance

The steepest descent method originally produced some higher-frequency oscillation, which leads to poor performance. To remedy this problem, some modifications were proposed. Those were

(1) To add a low-pass filter

(2) To modify the index function

(3) To reduce the gain near the goal configuration

These worked fairly well, and most of the higher frequency oscillations were cut off.

The pseudo-inverse method has an advantage in its good performance of higher priority tasks. If the end-effector movement has the highest priority, then its motion can be exactly as specified. For example the end-effector can move on a straight line while other links avoid collision with obstacles. This feature is beneficial to such applications as welding and painting.

But the method also has a serious drawbacks. There is a possibility of its falling into a singular configuration. When in a singularity, the planner creates a joint rate which is extremely large, so that the trajectory becomes uncontrollable. Therefore a singularity avoidance algorithm may be additionally necessary.

The trajectories generated would be tracked by a tracking control system. In this thesis, the computed torque method and the sliding surface control method are explained as possible candidates. It was shown that sliding surface control has better accuracy and more robustness against parameter uncertainties. It was also remarked that there is a trade-off between model accuracy and bandwidth in a tracking control system.

REFERENCES

1. ASADA, H., SLOTINE, J-J. E., "Robot Analysis and Control", John Wiley & Sons, Inc., 1986.

2. HSU, P., HAUSER, J., SASTRY, S., "Dynamic Control of Redundant Manipulators", *Proceedings IEEE International Conference on Robotics and Automation*, pp. 183-187, 1988.

3. KHATIB, O., "Real-Time Obstacle Avoidance for Manipulators and Mobile Robots", *The International Journal of Robotics Research*, Vol. 5, No. 1, pp. 90-98, 1986.

4. KIRĆANSKI, M., VUKOBRATOVIĆ, M., "Contribution to Control of Redundant Robotic Manipulators in an Environment with Obstacles", *The International Journal of Robotics Research*, Vol. 5, No. 4, pp. 112-119, 1986.

5. LOZANO-PÉREZ, T., "Automatic Planning of Manipulator Transfer Movements", *IEEE Transaction on Systems, Man, and Cybernetics*, SMC-11:10, pp. 681-698, 1981.

6. MACIEJEWSKI, A. A., KLEIN, C. A., "Obstacle Aroidance for Kinematikally Redundant Manipulators in Dynamically Varying Environments", *The International Journal of Robotics Research*, Vol. 4, No. 3, pp. 109-117, 1985.

7. NAKAMURA, Y., HANAFUSA, H., "Task Priority Based Redundancy Control of Robot Manipulators", *Robotics Research - The Second International Symposium*, Cambridge, MA: MIT Press, pp. 155-161, 1985.

8. PAUL, R. P., "Robotic Manipulators: Mathematics, Programming, and Control", Cambridge, MA: MIT Press, 1981.

9. RAO, C. R., MITRA, S. K., "Generalized Inverse of Matrices and Its Applications", New York: John Wiley & Sons, Inc., pp. 19-27, 1971.

10. RAO, S. S., "Optimization, Theory and Applications", New York: John Wiley & Sons, Inc., pp. 293-330, 1979.

11. UDUPA, S. M., "Collision Detection and Avoidance in Computer Controlled Manipulators", *5th International Joint Conference on Artificial Intelligence*, Cambridge, MA: MIT, 737-748, 1977.

12. UTKIN, V. I., "Variable Structure Systems with Sliding Modes", *IEEE Transaction on Automation and Control,* AC-22:2, pp. 212-222, 1977.

APPENDIX

A.1 Calculation of the Gradient

Here, the computation procedure to derive the gradient of the index function is explained. The absolute joint angle and relative joint angle are defined as in Fig A.1. The index function is either of the following.

$$f(q) = k_{eg} d_{eg}(q) + \frac{k_{co}}{d_{co}(q)} \quad \text{(A.1)}$$

or

$$f(q) = k_{eg} d_{eg}(q) + \sum \frac{k_i}{d_{coi}(q)} \quad \text{(A.2)}$$

Therefore the gradient is

$$\nabla_q f(q) = k_{eg} \nabla_q d_{eg}(q) - \frac{k_{co}}{d_{co}(q)^2} \nabla_q d_{co} \quad \text{(A.3)}$$

or

$$\nabla_q f(q) = k_{eg} \nabla_q d_{eg}(q) - \sum \frac{k_i}{d_{coi}(q)^2} \nabla_q d_{co} \quad \text{(A.4)}$$

This means that we only have to calculate the gradients of the goal distance and the obstacle clearance (∇d_{eg} and ∇d_{co} (or ∇d_{coi})). First the gradient calculation for a planar manipulator arm in absolute joint angles will be explained. Next it will be shown that the gradient for a three-dimensional arm can be calculated in a similar manner, by using relative joint angles.

(1) In absolute joint angle (λ)

First, the gradient vector is calculated in absolute joint angle space. In this case, planar manipulators are considered instead of three-dimensional ones. Define a vector from the end-effector to the goal as $\underline{x}_{eg}$. Also define $\underline{x}_{co}$ as a vector from a point on a link to a obstacle such that $\|\underline{x}_{co}\|=d_{co}$ (Fig A.1).Suppose that λ_i is increased by $\delta\lambda_i$ and other absolute joint angles are fixed (Fig A.3). The $\underline{x}_{eg}$ and $\underline{x}_{co}$ increase by $\delta\underline{x}_{eg}$ and $\delta\underline{x}_{co}$, where

$$\delta\underline{x}_{eg} = \delta\underline{x}_{co} = \begin{bmatrix} l_i\delta\lambda_i\sin\lambda_i \\ -l_i\delta\lambda_i\cos\lambda_i \end{bmatrix} \tag{A.5}$$

Then the increment in d_{co} is

$$\begin{aligned} \delta d_{eg} &= \|x_{eg}+dx_{eg}\| - \|x_{eg}\| \\ &= \sqrt{(\|x_{eg}\|^2 + 2x_{eg}\delta x_{eg} + \|\delta x_{eg}\|^2)} - \|x_{eg}\| \\ &= \|x_{eg}\|(1+\frac{1}{2}\frac{2x_{eg}dx_{eg}}{\|x_{eg}\|^2}) - \|x_{eg}\| \\ &= \frac{\underline{x}_{eg}\cdot\delta\underline{x}_{eg}}{d_{eg}} \\ &= l_i\cdot(x_{eg}\sin\lambda_i - y_{eg}\cos\lambda_i)\cdot\delta\lambda_i \end{aligned} \tag{A.6}$$

Similarly,

$$\begin{aligned} \delta d_{co} &= \|x_{co}+dx_{co}\| - \|x_{co}\| \\ &= \frac{\underline{x}_{co}\cdot\underline{\delta x}_{co}}{d_{co}} \\ &= l_i\cdot(x_{co}\sin\lambda_i - y_{co}\cos\lambda_i)\cdot\delta\lambda_i \end{aligned} \tag{A.7}$$

By dividing both sides of above equations by $\delta\lambda_i$, one gets

$$\begin{aligned} \frac{\partial d_{eg}}{\partial\lambda_i} &= l_i\cdot(x_{eg}\sin\lambda_i - y_{eg}\cos\lambda_i) \\ \frac{\partial d_{co}}{\partial\lambda_i} &= l_i\cdot(x_{co}\sin\lambda_i - y_{co}\cos\lambda_i) \end{aligned} \tag{A.8}$$

Therefore the gradient can be derived.

(2) In joint angle (θ)

Similarly, the gradient vector can be calculated in relative joint angles. Define $\hat{\underline{b}}_i$ as a unit vector parallel to joint axis i. Also define the vector $\underline{x}_{ci}$ and $\underline{x}_{ei}$ as in Fig A.2. Then

$$\begin{aligned} \delta\underline{x}_{eg} &= (\delta\theta_i\hat{\underline{b}}_i) \times \underline{x}_{ei} \\ \delta\underline{x}_{co} &= (\delta\theta_i\hat{\underline{b}}_i) \times \underline{x}_{ci} \end{aligned} \tag{A.9}$$

where x denotes a vector product. The increment in d_{eg} and d_{co} in this case are

$$\delta d_{co} = \frac{\underline{x}_{co}\cdot\underline{\delta x}_{co}}{d_{co}}$$

$$= \frac{(\hat{\underline{b}}_i \times \underline{x}_{ci}) \cdot \underline{x}_{co}}{d_{co}} \delta\theta_i \tag{A.10}$$

$$\delta d_{eg} = \frac{\underline{x}_{eg} \cdot \delta \underline{x}_{eg}}{d_{eg}}$$

$$= \frac{(\hat{\underline{b}}_i \times \underline{x}_{ei}) \cdot \underline{x}_{eg}}{d_{eg}} \delta\theta_i \tag{A.11}$$

By dividing both sides by $\delta\theta_i$, each component of the gradient are derived as follows.

$$\frac{\partial d_{eg}}{\partial \theta_i} = \frac{(\hat{\underline{b}}_i \times \underline{x}_{ei}) \cdot \underline{x}_{eg}}{d_{eg}}$$

$$\frac{\partial d_{co}}{\partial \theta_i} = \frac{(\hat{\underline{b}}_i \times \underline{x}_{ci}) \cdot \underline{x}_{co}}{d_{co}} \tag{A.12}$$

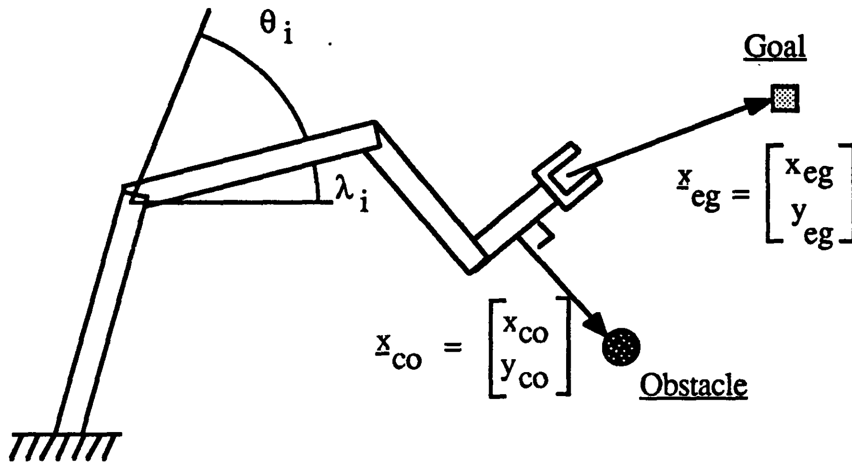


Fig A.1

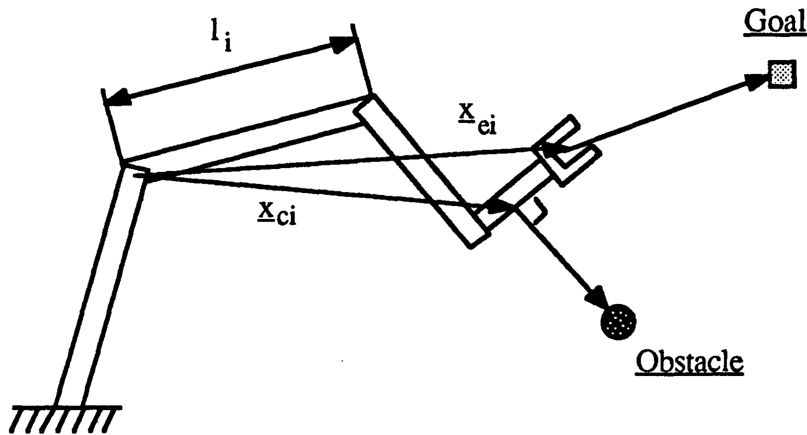


Fig A.2

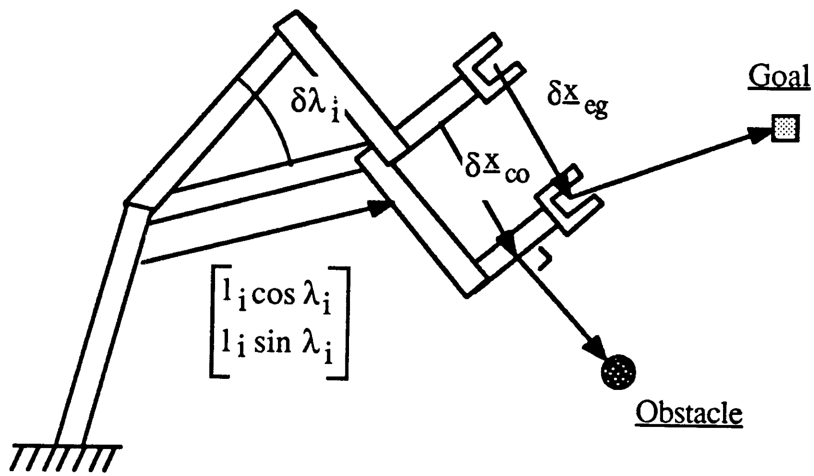


Fig A.3

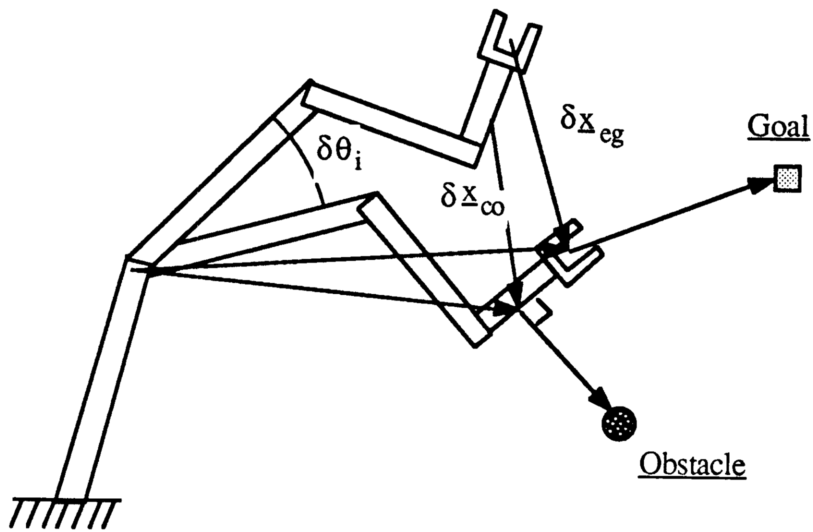


Fig A.4

A.2 Dynamics Equation

In this appendix, the dynamics used for the tracking control problem are developed. Derivations are based on the Lagrange's Equation

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = Q_i \tag{A.12}$$

$$L = K - P \tag{A.13}$$

where K and P are the kinematic and potential energies for the system, respectively, Q_i is the generalized force, and q_i is the generalized coordinate. There is always a symmetry matrix H (inertia tensor), such that,

$$K = \frac{1}{2}\dot{q}_i^T H \dot{q}_i = \sum_j H_{ij}\dot{q}_j\dot{q}_k \tag{A.14}$$

Also denote gravity force $\underline{g}$ as

$$g_i = \frac{\partial L}{\partial q_i} \tag{A.15}$$

By using the inertia tensor and gravity force vector, the right-hand side of above equation becomes

$$\begin{aligned}
\frac{d}{dt}\left(\frac{\partial K}{\partial \dot{q}_i}\right) - \frac{\partial K}{\partial q_i} + g_i &= \frac{d}{dt}\frac{\partial}{\partial \dot{q}_i}\left(\frac{1}{2}\sum_{j,k} H_{jk}\dot{q}_j\dot{q}_k\right) - \frac{\partial}{q_i}\left(\frac{1}{2}\sum_{j,k} H_{jk}\dot{q}_j\dot{q}_k\right) + g_i \\
&= \frac{d}{dt}\left\{\frac{1}{2}\left(\sum_k H_{ik}\dot{q}_k + \sum_j H_{ji}\dot{q}\right)\right\} - \sum_{j,k}\frac{1}{2}\frac{\partial H_{jk}}{\partial q_i}\dot{q}_j\dot{q}_k + g_i \\
&= \sum_j\left(H_{ij}\ddot{q}_j + \dot{H}_{ij}\dot{q}_j\right) - \sum_{j,k}\frac{1}{2}\frac{\partial H_{jk}}{\partial q_i}\dot{q}_j\dot{q}_k + g_i \\
&= \sum_j H_{ij}\ddot{q}_j + \sum_{j,k}\left(\frac{\partial H_{ij}}{\partial q_k} - \frac{1}{2}\frac{\partial H_{jk}}{\partial q_i}\right)\dot{q}_j\dot{q}_k + g_i
\end{aligned} \tag{A.16}$$

Define $C = [C_{ij}]$ as

$$C_{ij} = \frac{1}{2}\sum_k\left(\frac{\partial H_{ij}}{\partial q_k} + \frac{\partial H_{ik}}{\partial q_j} - \frac{\partial H_{jk}}{\partial q_i}\right)\dot{q}_k \tag{A.17}$$

Then

$$\sum C_{ij}\dot{q}_j = \frac{1}{2}\sum_{j,k}\left(\frac{\partial H_{ij}}{\partial q_k} + \frac{\partial H_{ik}}{\partial q_j} - \frac{\partial H_{jk}}{\partial q_i}\right)\dot{q}_j\dot{q}_k$$

$$= \sum_{j,k}\left(\frac{\partial H_{ij}}{\partial q_k} - \frac{1}{2}\frac{\partial H_{jk}}{\partial q_i}\right)\dot{q}_j\dot{q}_k \tag{A.18}$$

Therefore

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = \sum_j H_{ij}\ddot{q}_j + \sum_j C_{ij}\dot{q}_j + g_i \tag{A.19}$$

or,

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \underline{\dot{q}}}\right) - \frac{\partial L}{\partial \underline{q}} = H\underline{\ddot{q}} + C\underline{\dot{q}} + \underline{g} \tag{A.20}$$

The matrix C is chosen so that $\dot{H} - 2C$ becomes antisymmetric. In fact, each component of the matrix $(\dot{H} - 2C)$ is

$$\dot{H}_{ij} - 2C_{ij} = \sum_k \frac{\partial H_{ij}}{\partial q_k}\dot{q}_k - 2\cdot\frac{1}{2}\sum_k\left(\frac{\partial H_{ij}}{\partial q_k} + \frac{\partial H_{ik}}{\partial q_j} - \frac{\partial H_{jk}}{\partial q_i}\right)\dot{q}_k$$

$$= \sum_k\left(\frac{\partial H_{jk}}{\partial q_i} - \frac{\partial H_{ik}}{\partial q_j}\right)\dot{q}_k \tag{A.21}$$

$$\therefore\ (\dot{H} - 2C)^T = -(\dot{H} - 2C) \tag{A.22}$$

The resulting dynamics equation is of the form

$$H\underline{\ddot{q}} + C\underline{\dot{q}} + \underline{g} = \underline{\tau} \tag{A.23}$$

or, by substituting $C\underline{\dot{q}}$ by a vector $\underline{h}$,

$$H\underline{\ddot{q}} + \underline{h} + \underline{g} = \underline{\tau} \tag{A.24}$$

The vector $\underline{h}$ has following components.

$$h_i = \sum_{j,k} \left(\frac{\partial H_{ij}}{\partial q_k} - \frac{1}{2} \frac{\partial H_{jk}}{\partial q_i} \right) \dot{q}_j \dot{q}_k \tag{A.25}$$

As an example, the dynamics equations for a four-link planar manipulator arm are discussed below. Consider a manipulator arm with notation described in Fig A.5. The kinematic and potential energies are

$$\begin{aligned} K = \; & \frac{1}{2}(I_1 + (m_2 + m_3 + m_4) l_1^2)\dot{\lambda}_1^2 + (m_2 d_2 + (m_3 + m_4) l_2) l_1 c_{21} \dot{\lambda}_1 \dot{\lambda}_2 \\ & + (m_3 d_3 + m_4 l_3) l_1 c_{31} \dot{\lambda}_1 \dot{\lambda}_3 + m_4 d_4 l_1 c_{41} \dot{\lambda}_1 \dot{\lambda}_4 \\ & + \frac{1}{2}(I_2 + (m_3 + m_4) l_2^2)\dot{\lambda}_2^2 + (m_3 d_3 + m_4 l_3) l_2 c_{32} \dot{\lambda}_2 \dot{\lambda}_3 + m_4 d_4 l_2 \dot{\lambda}_2 \dot{\lambda}_4 \\ & + \frac{1}{2}(I_3 + m_4 l_3^2)\dot{\lambda}_3^2 + m_4 d_4 l_3)\dot{\lambda}_3 \dot{\lambda}_4 + \frac{1}{2} I_4 \dot{\lambda}_2^2 \end{aligned}$$

$$P = 0 \tag{A.26}$$

m_i = mass of link i

I_i = moment of inertia of link i around joint i

l_i = length of link i

d_i = distance between joint i and center-of-mass of link i

$c_{ij} = \cos(\lambda_i - \lambda_j)$

$s_{ij} = \sin(\lambda_i - \lambda_j)$

Here, absolute joint angle λ_i is used as a generalized coordinate. Generalized forces in this case are related to the joint torques as follows.

$$Q_1 = \tau_1 - \tau_2$$

$$Q_2 = \tau_2 - \tau_3$$

$$Q_3 = \tau_3 - \tau_4$$

$$Q_4 = \tau_4 \tag{A.27}$$

τ_j : actuator torque at joint j.

From equation (A.24), we get

$$H\ddot{\underline{\lambda}} + C\dot{\underline{\lambda}} = \underline{u} \tag{A.28}$$

$$H = \begin{bmatrix} p_{11}c_{11} & p_{12}c_{12} & p_{13}c_{13} & p_{14}c_{14} \\ p_{21}c_{21} & p_{22}c_{22} & p_{23}c_{23} & p_{24}c_{24} \\ p_{31}c_{31} & p_{32}c_{32} & p_{33}c_{33} & p_{34}c_{34} \\ p_{41}c_{41} & p_{42}c_{42} & p_{43}c_{43} & p_{44}c_{44} \end{bmatrix} \tag{A.29}$$

$$C = \begin{bmatrix} 0 & p_{12}s_{12}\dot{\lambda}_2 & p_{13}s_{13}\dot{\lambda}_3 & p_{14}s_{14}\dot{\lambda}_4 \\ p_{21}s_{21}\dot{\lambda}_1 & 0 & p_{23}s_{23}\dot{\lambda}_3 & p_{24}s_{24}\dot{\lambda}_4 \\ p_{31}s_{31}\dot{\lambda}_1 & p_{32}s_{32}\dot{\lambda}_2 & 0 & p_{34}s_{34}\dot{\lambda}_4 \\ p_{41}s_{41}\dot{\lambda}_1 & p_{42}s_{42}\dot{\lambda}_2 & p_{43}s_{43}\dot{\lambda}_3 & 0 \end{bmatrix} \tag{A.30}$$

(A.30)

$$p_{11} = I_1+(m_2+m_3+m_4)l_1^2$$

$$p_{12} = p_{21} = (m_2d_2+(m_3+m_4)l_2)l_1c_{12}$$

$$p_{13} = p_{31} = (m_3d_3+m_4l_3)l_1c_{13}$$

$$p_{14} = p_{41} = m_4d_4l_1c_{14}$$

$$p_{22} = I_2 + (m_3 + m_4)l_2^2$$

$$p_{23} = p_{32} = (m_3d_3 + m_4l_3)l_2c_{23}$$

$$p_{24} = p_{42} = m_4d_4l_2c_{24}$$

$$p_{33} = I_3 + m_4l_3^2$$

$$p_{34} = p_{43} = m_4d_4l_3c_{34}$$

$$p_{44} = I_4 \tag{A.31}$$

or

$$H\ddot{\underline{q}} + \underline{h} = \underline{u} \tag{A.32}$$

$$\underline{h} = \begin{bmatrix} p_{12}s_{12}\dot{\lambda}_2^2 + p_{13}s_{13}\dot{\lambda}_3^2 + p_{14}s_{14}\dot{\lambda}_4^2 \\ p_{21}s_{21}\dot{\lambda}_1^2 + p_{23}s_{23}\dot{\lambda}_3^2 + p_{24}s_{24}\dot{\lambda}_4^2 \\ p_{31}s_{31}\dot{\lambda}_1^2 + p_{32}s_{32}\dot{\lambda}_2^2 + p_{34}s_{34}\dot{\lambda}_4^2 \\ p_{41}s_{41}\dot{\lambda}_1^2 + p_{42}s_{42}\dot{\lambda}_2^2 + p_{43}s_{43}\dot{\lambda}_3^2 \end{bmatrix} \tag{A.33}$$

$\dot{H}$ - 2C is skew-symmetric matrix.

$$\dot{H} - 2C = \begin{bmatrix} 0 & -p_{12}s_{12}(\dot{\lambda}_1 + \dot{\lambda}_2) & -p_{13}s_{13}(\dot{\lambda}_1 + \dot{\lambda}_3) & -p_{14}s_{14}(\dot{\lambda}_1 + \dot{\lambda}_4) \\ p_{12}s_{12}(\dot{\lambda}_1 + \dot{\lambda}_2) & 0 & -p_{23}s_{23}(\dot{\lambda}_2 + \dot{\lambda}_3) & -p_{24}s_{24}(\dot{\lambda}_2 + \dot{\lambda}_4) \\ p_{13}s_{13}(\dot{\lambda}_1 + \dot{\lambda}_3) & p_{23}s_{23}(\dot{\lambda}_2 + \dot{\lambda}_3) & 0 & -p_{34}s_{34}(\dot{\lambda}_3 + \dot{\lambda}_4) \\ p_{14}s_{14}(\dot{\lambda}_1 + \dot{\lambda}_4) & p_{24}s_{24}(\dot{\lambda}_2 + \dot{\lambda}_4) & p_{34}s_{34}(\dot{\lambda}_3 + \dot{\lambda}_4) & 0 \end{bmatrix}$$

(A.34)

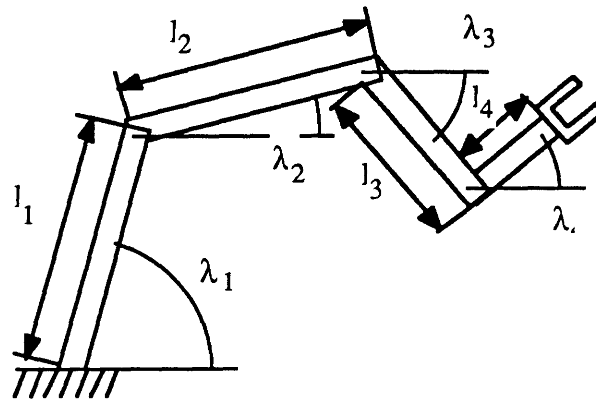


Fig A.5

A.3 Computer Listings

```
/******************************/
/* MANIPULATOR TRAJECTORY */
/* PLANNER USING               */
/* PSEUDO-INVERSE METHOD  */
/*   1988/5/20                  */
/******************************/
#include <stdio.h>
#include <math.h>

#define Nm 8
#define Nmp 9
#define Nob 4

main()
{
 FILE  *stream0,*stream1;
 char  c;
 float                                    t,      /* t      : time    */
        dt,                 /* dt         : sampling time              */
        l[Nm],              /* l          : length of the link         */
        q[Nm],              /* q          : joint angle                */
        dq[Nm],             /* dq         : joint angular velosity     */
        qd[Nm],             /* qd         : joint angle (desired)      */
        dqd[Nm],            /* dqd        : joint angular vel. (des.)  */
        ddqd[Nm],           /* ddqd       : joint angular acc. (des.)  */
        xob[Nob][2],        /* xob        : obstacle position          */
        xg[2],              /* xg         : goal position              */
        k1,k2,k3,k4,        /* ki         : planner gains              */
        dta,dug,dsoi,       /* d          : distance threshold         */
        deg,                /* deg        : goal distance              */
        dco,                /* dco        : obstacle clearance         */
        ds1,ds2,ds3,        /* dsi        : determinants for ith       */
                            /*              priority task              */
        cg[2][2],           /* cg         : goal position              */
        cob[Nob][2][2];     /* cob        : obstacle positions         */
 int   i,ii,j,Nstep,Ndatajamp;


 stream0 = fopen("data0.prn","w");
 stream1 = fopen("data1.prn","w");

/*
 initial condition and parameter setting
*/
 _clearscreen(_GCLEARSCREEN);

 GetInitCond(&Nstep,&dt,&Ndatajamp,l,qd,qd,dqd,dqd,ddqd,cg,cob,&k1,&k2,&k3,&k4,&dta,&dug,&dsoi);
 OutsideCondition(0.0,cg,cob,xg,xob);

/*
 headings for data files
*/
```

```
 fprintf(stream0,"*\ntime\t");
 for(i=0;i<Nm;++i){ fprintf(stream0,"qd[%d]\t",i); }
 for(i=0;i<Nm;++i){ fprintf(stream0,"dqd[%d]\t",i); }
 for(i=0;i<Nm;++i){ fprintf(stream0,"ddqd[%d]\t",i); }

 fprintf(stream1,"*\ntime\t");
 fprintf(stream1,"deg\tdco\tdet1\tdet2\tdet3\n");

/*
 main program
*/

 t=0.0;

 for(i=0;i<Nstep;++i){

 for(ii=0;ii<Ndatajamp;++ii){

  t = dt*(i*Ndatajamp + ii);

  OutsideCondition(t,cg,cob,xg,xob);

  DesiredTrajectory(l,xg,xob,k1,k2,k3,k4,xdta,dug,dsoi,dt,qd,dqd,ddqd,&deg,&dco,&ds1,&ds2,&ds3);

 }

  t = dt*(i+1)*Ndatajamp;

  fprintf(stream0,"%f\t",t);
  for(j=0;j<Nm;++j){ fprintf(stream0,"%f\t",qd[j]); }
  for(j=0;j<Nm;++j){ fprintf(stream0,"%f\t",dqd[j]); }
  for(j=0;j<Nm;++j){ fprintf(stream0,"%f\t",ddqd[j]); }
  fprintf(stream0,"\n");

  fprintf(stream1,"%f\t",t);
  fprintf(stream1,"%f\t%f\t%e\t%e\t%e\t%f\t%f\t",deg,dco,ds1,ds2,ds3);
  fprintf(stream1,"\n");

  _settextposition(1,1);
  printf("t=%f",t);
 }
  fclose(stream0);
  fclose(stream1);

  _clearscreen(_GCLEARSCREEN);

}


/****************************************************/
/* INITIAL CONDITIONS AND PARAMETER SETTINGS */
/****************************************************/

GetInitCond(Nstep,dt,Ndatajamp,l,q,qd,dq,dqd,ddqd,cg,cob,k1,k2,k3,k4,d1,d2,d3)
int  *Nstep,*Ndatajamp;
float *dt,l[Nm],q[Nm],qd[Nm],dq[Nm],dqd[Nm],ddqd[Nm],
    cg[2][2],cob[Nob][2][2],
```

```
    *k1,*k2,*k3,*k4,*d1,*d2,*d3;
{
 FILE *stream0;
 float dummy;
 int  i,j;

 stream0 = fopen("initial.dat","r");

 fscanf(stream0,"%f",dt);

 fscanf(stream0,"%d",Nstep);

 fscanf(stream0,"%d",Ndatajamp);

 for(i=0;i<Nm;++i){fscanf(stream0,"%f",&l[i]);
 }

 for(i=0;i<Nm;++i){fscanf(stream0,"%f",&q[i]);
 }
 for(i=0;i<Nm;++i){fscanf(stream0,"%f",&qd[i]);
 }
 for(i=0;i<Nm;++i){fscanf(stream0,"%f",&dq[i]);
 }
 for(i=0;i<Nm;++i){fscanf(stream0,"%f",&dqd[i]);
 }

 for(i=0;i<Nob;++i){
  fscanf(stream0,"%f",&cob[i][0][0]);
  fscanf(stream0,"%f",&cob[i][0][1]);
  fscanf(stream0,"%f",&cob[i][1][0]);
  fscanf(stream0,"%f",&cob[i][1][1]);
 }

 fscanf(stream0,"%f",&cg[0][0]);
 fscanf(stream0,"%f",&cg[0][1]);
 fscanf(stream0,"%f",&cg[1][0]);
 fscanf(stream0,"%f",&cg[1][1]);

 fscanf(stream0,"%f",&dummy);
 fscanf(stream0,"%f",&dummy);
 fscanf(stream0,"%f",&dummy);
 fscanf(stream0,"%f",&dummy);
 fscanf(stream0,"%f",&dummy);

 fscanf(stream0,"%f",k1);
 fscanf(stream0,"%f",k2);
 fscanf(stream0,"%f",k3);
 fscanf(stream0,"%f",k4);
 fscanf(stream0,"%f",d1);
 fscanf(stream0,"%f",d2);
 fscanf(stream0,"%f",d3);

 fclose(stream0);

 GetInitArmAccel(ddqd);

 printf("\n");
 printf("Nstep=%d\n",*Nstep);
```

```
  printf("Ndatajamp=%d\n",*Ndatajamp);
  printf("dt=%f\n",*dt);
  for(i=0;i<Nm;++i){ printf("%f\t",qd[i]); } printf("\n");
  for(i=0;i<Nm;++i){ printf("%f\t",dqd[i]); } printf("\n");
  for(i=0;i<Nm;++i){ printf("%f\t",ddqd[i]); } printf("\n");
  for(i=0;i<Nob;++i){ for(j=0;j<2;++j){
    printf("cob[%d][%d]=%f\t%f\n",i,j,cob[i][j][0],cob[i][j][1]);
  } }
  for(i=0;i<2;++i){ printf("cg[%d]=%f\t%f\n",i,cg[i][0],cg[i][1]); }

  printf("k1=%f\n",*k1);
  printf("k2=%f\t",*k2);
  printf("k3=%f\n",*k3);
  printf("k4=%f\t",*k4);
  printf("d1 ( <d2 <d3 ) =%f\n",*d1);
  printf("d2 ( <d3 >d1 ) =%f\n",*d2);
  printf("d3 ( >d2 >d1 ) =%f\n",*d3);

  return;
}


GetInitArmAccel(ddq)
float ddq[Nm];
{
  int i;

  for(i=0;i<Nm;++i){ ddq[i] = 0.0; }

  return;
}


/***************************/
/*  TRAJECTORY PLANNER */
/***************************/

DesiredTrajectory(l,xg,xob,k1,k2,k3,k4,dta,dug,dsoi,dt,qd,dqd,ddqd,deg,dco,ds1,ds2,ds3)
float   l[Nm],              /* l[i]         : length of link[i]          */
        xg[2],              /* xg           : goal position              */
        xob[Nob][2],        /* xob          : obstacle position          */
        k1,k2,k3,k4,        /* k            : gain                       */
        dta,dug,dsoi,       /* d            : distance threshold         */
        dt,                 /* dt           : sampling time              */
        qd[Nm],             /* qd           : desired joint coordinate   */
        dqd[Nm],            /* dqd          : desired joint coordinate   */
        ddqd[Nm],           /* ddqd         : desired joint coordinate   */
        *deg,               /* deg          : goal distance              */
        *dco,               /* dco          : obstacle clearance         */
        *ds1,*ds2,*ds3;     /* dsi          : determinants for ith       */
                            /*                priority task              */
{
float   sam[Nm],            /*  sam         : nearest point on link[i]   */
                            /*                from the obstacle          */
        dmin[Nm],           /*  dmin        : distance of link[i]        */
                            /*                from the obstacle          */
        xco[Nm][2],         /*  xco         : direction of the           */
                            /*                nearest point on obstacle  */
```

```
                        /*                    from the link[i]          */
        xdrc[2],        /* xdrc             : direction of the          */
                        /*                    nearest point on obstacle */
        xam[Nmp][2],    /* xam              : arm joint position        */
        xde[2],deo,xe[2],ve[2],
        dqdm[Nm],
        J[2][Nm],
        detCCT(),ddet,
        Id[Nm][Nm],o[Nm],
        A[Nm][2],BP[Nm][Nm],BM[Nm][Nm],C[2][Nm],
        v[2],v0[2],
        wp[Nm],wm[Nm];
  int   iam[Nm],        /* iam              : nearest link from the     */
                        /*                    obstacle                  */
         i,ii;


/*
 >> desired joint coordinate ( qd[i] )
*/
 for(i=0;i<Nm;++i){
  qd[i] = qd[i]+dqd[i]*dt+0.5*ddqd[i]*dt*dt;
 }

/*
 >> desired joint rate ( dqd[i] )
*/
 VecCoef(1.0,dqd,Nm,dqdm);

 /* nearest point */
 GetArmPos(l,qd,xam);
 *deg = sqrt((xam[Nmp-1][0]-xg[0])*(xam[Nmp-1][0]-xg[0])
        +(xam[Nmp-1][1]-xg[1])*(xam[Nmp-1][1]-xg[1]));
 xe[0] = xam[Nmp-1][0];
 xe[1] = xam[Nmp-1][1];
 NearestPoints(xam,xob,iam,sam,xco,dmin);
 DistBetweenEndAndObstacle(xam,xob,xde,&deo);
 IdentMat(Id,Nm);
 NullVec(o,Nm);

                                               /* dqd=f(ve,v0)          */
 GetJ(l,qd,Nm-1,1.0,J);                        /* J = f(l,qd)           */
 GetV0(xde,deo,1.0,dta,dug,dsoi,v0);           /* v0 = f(xde,deo)       */
 GetVe(xam,xg,v0,k1,k2,ve);                    /* ve = f(v0,xam,xg)     */
 PseudoInv(J,A);                               /* A  = J^+              */
 RemMat(Id,J,BP);                              /* BP = (I-(J^+)*J)      */
 MatMul(BP,Id,Nm,Nm,Nm,BM);                    /* BM = BP               */
 UpdateVec(o,A,ve,wp);                         /* wp = o + A*ve         */
 VecAdd(wp,o,Nm,wm);                           * wm = wp                */
 *ds1 = fabs(detCCT(J));                       /* ds1 = det(J*JT)       */

 GetJ(l,qd,iam[0],sam[iam[0]],J);              /* J = f(l,qd)           */
 xdrc[0] = xco[iam[0]][0];
 xdrc[1] = xco[iam[0]][1];
 GetV0(xdrc,dmin[iam[0]],k3,dta,dug,dsoi,v0);  /* v0 = f(xdrc,dmin)     */
 MatMul(J,BM,2,Nm,Nm,C);                       /* C  = J*BM             */
 PseudoInv(C,A);                               /* A  = C^+              */
 RemMat(BM,J,BP);                              /* BP = BM*(I-(J^+)*J)   */
```

```
 MatMul(BP,Id,Nm,Nm,Nm,BM);                /* BM = BP              */
 RemVec(v0,J,wm,v);                        /* v  = v0-J*wm         */
 UpdateVec(wm,A,v,wp);                     /* wp = wm + A*v        */
 VecAdd(wp,o,Nm,wm);                       /* wm = wp              */
 *ds2 = fabs(detCCT(C));                   /* ds2 = det(C*CT)      */

 GetJ(l,qd,iam[1],sam[iam[1]],J);          /* J = f(l,qd)          */
 xdrc[0] = xco[iam[1]][0];
 xdrc[1] = xco[iam[1]][1];
 GetV0(xdrc,dmin[iam[1]],k4,dta,dug,dsoi,v0); /* v0 = f(xdrc,dmin) */
 MatMul(J,BM,2,Nm,Nm,C);                   /* C  = J*BM            */
 PseudoInv(C,A);                           /* A  = C^+             */
 RemMat(BM,J,BP);                          /* BP = BM*(I-(J^+)*J)  */
 MatMul(BP,Id,Nm,Nm,Nm,BM);                /* BM = BP              */
 RemVec(v0,J,wm,v);                        /* v  = v0-J*wm         */
 UpdateVec(wm,A,v,wp);                     /* wp = wm + A*v        */
 VecAdd(wp,o,Nm,dqd);                      /* dqd = wp             */
 *ds3 = fabs(detCCT(C));                   /* ds3 = det(C*CT)      */


/*
 >> desired joint accleration ( ddqd[i] )
*/
 for(i=0;i<Nm;++i){
  ddqd[i] = (dqd[i]-dqdm[i])/dt;
 }

 *dco = dmin[iam[0]];

 return;
}



GetJ(l,q,k,s,J)
float  l[Nm],q[Nm],s,J[2][Nm];
int    k;
{
  int    i;

  for(i=0;i<k;i++){
   J[0][i] = -l[i]*sin(q[i]);
   J[1][i] =  l[i]*cos(q[i]);
  }
 J[0][k] = -s*l[k]*sin(q[k]);
 J[1][k] =  s*l[k]*cos(q[k]);
  for(i=k+1;i<Nm;i++){
   J[0][i] = 0.0;
   J[1][i] = 0.0;
  }

 return;
}



GetV0(x1,d,Vmax,dta,dug,dsoi,x2)
float x1[2],d,Vmax,dta,dug,dsoi,x2[2];
```

```
/*****************************/
/*  desired velocity at a point on   */
/*  a link which avoids obstacles  */
/*****************************/
{
 float alpha1;

 if(d<dta){alpha1 = Vmax;}
 else{
  if(d<dug){alpha1 = Vmax*(d-dug)*(d-dug)/(dug-dta)/(dug-dta);}
  if(dug<d){alpha1 = 0.0;}
 }

 x2[0] = -alpha1*x1[0]/(sqrt(x1[0]*x1[0]+x1[1]*x1[1]));
 x2[1] = -alpha1*x1[1]/(sqrt(x1[0]*x1[0]+x1[1]*x1[1]));

 return;
}



GetVe(xam,xg,vo,k1,k3,ve)
float  xam[Nmp][2],xg[2],vo[2],k1,k3,ve[2];
/************************************/
/*  desired end-effector velocity          */
/*  which avoids obstacle and              */
/*  approaches the destination             */
/************************************/
{
  float vg[2];
  float dcg;    /* ditance from the goal  */
  float kk;

  vg[0] = xg[0]-xam[Nmp-1][0];
  vg[1] = xg[1]-xam[Nmp-1][1];
  dcg   = sqrt((vg[0]*vg[0]+vg[1]*vg[1]));
  kk = k1/sqrt(dcg);
  ve[0] = kk*vg[0]+k3*vo[0];
  ve[1] = kk*vg[1]+k3*vo[1];

  return;
}


RemMat(A,B,C)
float  A[Nm][Nm],B[Nm][Nm],C[Nm][Nm];
{
 float F[Nm][Nm],G[Nm][Nm],H[Nm][Nm],I[Nm][Nm];

 IdentMat(I,Nm);
 PseudoInv(B,F);
  MatMul(F,B,Nm,2,Nm,G);
  MatSub(I,G,Nm,Nm,H);
  MatMul(A,H,Nm,Nm,Nm,C);

 return;
}
```

```
PseudoInv(J,JP)
float J[2][Nm],JP[Nm][2];
{
 float JT[Nm][2],A[2][2],B[2][2],det,Inv2();
 int i,j;
 char c;

 transp(J,2,Nm,JT);
 MatMul(J,JT,2,Nm,2,A);
 det = Inv2(A,B);
 if(det!=0.0){MatMul(JT,B,Nm,2,2,JP); }
 else{NullMat(JP,Nm,2);}

 return;
}



RemVec(v1,J,wm,v)
float v1[2],J[2][Nm],wm[Nm],v[2];
{
 float u[2];

 MatVec(J,wm,2,Nm,u);
 VecSub(v1,u,2,v);

 return;
}


UpdateVec(wm,A,v,wp)
float wm[Nm],A[Nm][2],v[2],wp[Nm];
{
 float  u[Nm];
 int    i;

 MatVec(A,v,Nm,2,u);
 VecAdd(wm,u,Nm,wp);

 return;
}



float detCCT(C)
float C[2][Nm];
{
 float CT[Nm][2],CCT[2][2],CCTI[2][2],Inv2();

 transp(C,2,Nm,CT);
 MatMul(C,CT,2,Nm,2,CCT);
 return(Inv2(CCT,CCTI));
}
```

```
/*********************************/
/* MANIPULATOR TRAJECTORY  */
/* PLANNER USING THE        */
/* STEEPEST DESCENT METHOD */
/*   1988/5/20             */
/*********************************/
#include <stdio.h>
#include <math.h>

#define Nm 8
#define Nmp 9
#define Nob 4

main()
{
 FILE  *stream0,*stream1;
 char  c;
 float  t,              /* t       : time                  */
        dt,             /* dt      : sampling time         */
        l[Nm],          /* l       : length of a link      */
        u[Nm],          /* u       : torque command        */
        dq[Nm],         /* dq      : joint angular velosity */
        qd[Nm],         /* qd      : joint angle (desired) */
        dqd[Nm],        /* dqd     : joint angular vel. (des.) */
        dqd1[Nm],       /* dqd1    : joint angular vel. (des.) */
        ddqd[Nm],       /* ddqd    : joint angular acc. (des.) */
        xob[Nob][2],    /* xob     : obstacle position     */
        xg[2],          /* xg      : goal position         */
        k1,k2,          /* k1,k2   : planner gain          */
        d1,d2,          /* d       : distance threshold    */
        a,              /* a       : filter gain           */
        dco,            /* dco     : distance from the     */
                        /*           obstacle              */
        deg,            /* deg     : distance to goal      */
        xe[2],          /* xe      : end-effector position */
        cg[2][2],       /* cg      : goal position         */
        cob[Nob][2][2]; /* cob     : obstacle position     */
 int   i,ii,j,Nstep,Ndatajamp;


 stream0 = fopen("data0.prn","w");
 stream1 = fopen("data1.prn","w");

/*
 initiald condition and parameter setting
*/
 _clearscreen(_GCLEARSCREEN);

 GetInitCond(&Nstep,&dt,&Ndatajamp,l,qd,qd,dqd,dqd1,ddqd,cg,cob,&k1,&k2,&d1,&d2,&a);
 OutsideCondition(0.0,cg,cob,xg,xob);

/*
 headings in the data file
*/
 fprintf(stream0,"*\ntime\t");
 for(i=0;i<Nm;++i){ fprintf(stream0,"qd[%d]\t",i); }
 for(i=0;i<Nm;++i){ fprintf(stream0,"dqd[%d]\t",i); }
```

```
 for(i=0;i<Nm;++i){ fprintf(stream0,"ddqd[%d]\t",i); }
 fprintf(stream0,"\n");

 fprintf(stream1,"*\ntime\t");
 fprintf(stream1,"deg\tdco\txe\tye\n");

/*
 main program
*/
 for(i=0;i<Nstep;++i){

 for(ii=0;ii<Ndatajamp;++ii){

  t = dt*(i*Ndatajamp+ii);

  OutsideCondition(t,cg,cob,xg,xob);

  DesiredTrajectory(l,xg,xob,k1,k2,d1,d2,a,dt,qd,dqd,dqd1,ddqd,&deg,&dco,xe);

  }

  t = dt*(i+1)*Ndatajamp;

  fprintf(stream0,"%f\t",t);
  for(j=0;j<Nm;++j){ fprintf(stream0,"%f\t",qd[j]); }
  for(j=0;j<Nm;++j){ fprintf(stream0,"%f\t",dqd[j]); }
  for(j=0;j<Nm;++j){ fprintf(stream0,"%f\t",ddqd[j]); }
  fprintf(stream0,"\n");

  fprintf(stream1,"%f\t",t);
  fprintf(stream1,"%f\t%f\t%f\t%f\t",deg,dco,xe[0],xe[1]);
  fprintf(stream1,"\n");

  _settextposition(1,1);
  printf("t=%f",t);
 }
  fclose(stream0);
  fclose(stream1);

  _clearscreen(_GCLEARSCREEN);
}


/***************************************************/
/* INITIAL CONDITIONS AND PARAMETER SETTINGS */
/***************************************************/

GetInitCond(Nstep,dt,Ndatajamp,l,q,qd,dq,dqd,ddqd,cg,cob,k1,k2,d1,d2,a)
int  *Nstep,*Ndatajamp;
float *dt,l[Nm],q[Nm],qd[Nm],dq[Nm],dqd[Nm],ddqd[Nm],
     cg[2][2],cob[Nob][2][2],
      *k1,*k2,*d1,*d2,*a;
{
 FILE *stream0;
 int  i;

 stream0 = fopen("initial.dat","r");
```

```
fscanf(stream0,"%f",dt);

fscanf(stream0,"%d",Nstep);

fscanf(stream0,"%d",Ndatajamp);

for(i=0;i<Nm;++i){fscanf(stream0,"%f",&l[i]);}

for(i=0;i<Nm;++i){ fscanf(stream0,"%f",&q[i]);}
for(i=0;i<Nm;++i){fscanf(stream0,"%f",&qd[i]);}
for(i=0;i<Nm;++i){fscanf(stream0,"%f",&dq[i]);}
for(i=0;i<Nm;++i){fscanf(stream0,"%f",&dqd[i]);}

for(i=0;i<Nob;++i){
  fscanf(stream0,"%f",&cob[i][0][0]);
  fscanf(stream0,"%f",&cob[i][0][1]);
  fscanf(stream0,"%f",&cob[i][1][0]);
  fscanf(stream0,"%f",&cob[i][1][1]);
}

fscanf(stream0,"%f",&cg[0][0]);
fscanf(stream0,"%f",&cg[0][1]);
fscanf(stream0,"%f",&cg[1][0]);
fscanf(stream0,"%f",&cg[1][1]);

fscanf(stream0,"%f",k1);
fscanf(stream0,"%f",k2);
fscanf(stream0,"%f",d1);
fscanf(stream0,"%f",d2);
fscanf(stream0,"%f",a);

fclose(stream0);

GetInitArmAccel(ddqd);

printf("\n");
printf("Nstep=%d\n",*Nstep);
printf("Ndatajamp=%d\n",*Ndatajamp);
printf("dt=%f\n",*dt);
for(i=0;i<Nm;++i){ printf("%f\t",qd[i]); } printf("\n");
for(i=0;i<Nm;++i){ printf("%f\t",dqd[i]); } printf("\n");
for(i=0;i<Nm;++i){ printf("%f\t",ddqd[i]); } printf("\n");
printf("cob[0][0]=%f\t%f\n",cob[0][0][0],cob[0][0][1]);
printf("cob[0][1]=%f\t%f\n",cob[0][1][0],cob[0][1][1]);
printf("cob[1][0]=%f\t%f\n",cob[1][0][0],cob[1][0][1]);
printf("cob[1][1]=%f\t%f\n",cob[1][1][0],cob[1][1][1]);
printf("cob[2][0]=%f\t%f\n",cob[2][0][0],cob[2][0][1]);
printf("cob[2][1]=%f\t%f\n",cob[2][1][0],cob[2][1][1]);
printf("cob[3][0]=%f\t%f\n",cob[3][0][0],cob[3][0][1]);
printf("cob[3][1]=%f\t%f\n",cob[3][1][0],cob[3][1][1]);
printf("cg[0]=%f\t%f\n\n",cg[0][0],cg[0][1]);
printf("cg[1]=%f\t%f\n\n",cg[1][0],cg[1][1]);

printf("k1=%f\n",*k1);
printf("k2=%f\n",*k2);
printf("d1 ( <d2 ) =%f\n",*d1);
printf("d2 ( >d1 ) =%f\n",*d2);
```

```
 printf("a ( filter gain ) =%f\n",*a);

 return;
}


GetInitArmAccel(ddq)
float ddq[Nm];
{
 int i;

 for(i=0;i<Nm;++i){ ddq[i] = 0.0; }

 return;
}


/***************************/
/*  TRAJECTORY PLANNER */
/***************************/

DesiredTrajectory(l,xg,xob,k1,k2,d1,d2,a,dt,qd,dqd,dqd1,ddqd,dg,dco,xe)
float   l[Nm],           /* l[i]      : length of link[i]             */
        xg[2],           /* xg        : goal position                 */
        xob[Nob][2],     /* xob       : obstacle position             */
        k1,k2,           /* k         : gain                          */
        d1,d2,           /* d         : thresholds                    */
        a,               /* a         : filter gain                   */
        dt,              /* dt        : sampling time                 */
        qd[Nm],          /* qd        : desired joint coordinate      */
        dqd[Nm],         /* dqd       : desired joint rate            */
        dqd1[Nm],        /* dqd1      : desired joint rate            */
        ddqd[Nm],        /* ddqd      : desired joint acc.            */
        *dg,             /* deg       : distance to the goal          */
        *dco,            /* dco       : obstacle clearance            */
        xe[2];           /* xe        : end-effector position         */
{
 float sam[Nm],          /*  sam      : location of the point         */
                         /*             nearest to the obstacle       */
        dmin[Nm],        /*  dmin     : distance from the obstacle */
        xdrc[Nm][2],     /*  xdrc     : direction of the              */
                         /*             nearest point on obstacle     */
        xam[Nmp][2],     /*  xam      : arm joint position            */
        xeg[2],deg,
        sdmin,
        xdrc1[2],
        dqdm[Nm],
        dqd2[Nm],        /* dqd2      : desired joint rate            */
        sat(),prox();
 int  iam,ii,i,j;
 char c;


  /*
   >> desired joint coordinate ( qd[i] )
  */
   for(i=0;i<Nm;++i){
   qd[i] = qd[i]+dqd[i]*dt+0.5*ddqd[i]*dt*dt;
```

```
    }

  /*
   >> desired joint rate ( dqd[i] )
  */
   VecCoef(1.0,dqd,Nm,dqdm);
   VecCoef(1.0,dqd1,Nm,dqd2);
   GetArmPos(l,qd,xam);
   xe[0] = xam[Nmp-1][0];
   xe[1] = xam[Nmp-1][1];
   xeg[0] = xe[0]-xg[0];
   xeg[1] = xe[1]-xg[1];
   deg = sqrt(xeg[0]*xeg[0]+xeg[1]*xeg[1]);

   NearestPoints(xam,xob,&iam,sam,xdrc,dmin);

   for(i=0;i<Nm;++i){
     dqd1[i] = -sat(deg/d2)*k1*l[i]
             *(xeg[1]*cos(qd[i])-xeg[0]*sin(qd[i]))/deg;
   }
   for(ii=0;ii<Nm;++ii){
     for(i=0;i<Nm;++i){
       if(i<ii){
       dqd1[i] = dqd1[i]
               -prox(deg,d1,d2)*k2*l[i]*(xdrc[ii][1]*cos(qd[i])
                -xdrc[ii][0]*sin(qd[i]))/dmin[ii]/dmin[ii];
        }
       if(i==ii){
               dqd1[i] = dqd1[i]
                       -prox(deg,d1,d2)*k2*sam[ii]*l[i]*(xdrc[ii][1]*cos(qd[i])
                        -xdrc[ii][0]*sin(qd[i]))/dmin[ii]/dmin[ii];
        }
      }
    }


  /*
   >> low-pass filter
  */
   for(i=0;i<Nm;++i){
    dqd[i] = (2-a*dt)/(2+a*dt)*dqdm[i]+ a*dt*(dqd1[i]+dqd2[i])/(2+a*dt);
   }

  /*
   >> desired joint accleration ( ddqd[i] )
  */
   for(i=0;i<Nm;++i){ddqd[i] = (dqd[i]-dqdm[i])/dt;}

   *dg = deg;
   *dco = dmin[iam];

 return;
}


/********************************/
```

```
/* MANIPULATOR TRAJECTORY  */
/* TRACKING CONTROL METHOD*/
/* FOR IBM-PC              */
/*   1988/5/20             */
/********************************/
#include <stdio.h>
#include <math.h>

#define Nm  4
#define Nmp 5
#define Nob 4

main()
{
 FILE   *stream0,*stream1,*stream2;
 char   c;
 float  t,              /* t          : time                        */
        dt,             /* dt         : sampling time               */
        l[Nm],          /* l[i]       : length of link[i]           */
        u[Nm],          /* u[i]       : torque command              */
        q[Nm],          /* q[i]       : joint angle                 */
        dq[Nm],         /* dq[i]      : joint angular velosity      */
        qd[Nm],         /* qd[i]      : joint angle (desired)       */
        dqd[Nm],        /* dqd[i]     : joint angular vel. (des.)   */
        ddqd[Nm],       /* ddqd[i]     joint angular acc. (des.)    */
        k1,k2,          /* k1,k2,     : planner gain                */
        d1,d2,          /* d          : distance threshold          */
        k[Nm],          /* k[i]       : feedback gain               */
        lamda,          /* lamda      : controller band width       */
        fai[Nm],        /* fai[i]     : controller precision        */
        s[Nm],          /* s[i]       : error                       */
        M0[Nm][Nm],     /*                                          */
        M1[Nm][Nm],     /*                                          */
        P0[Nm][Nm],     /*                                          */
        n[Nm],          /*                                          */
        UT[Nm][Nm],     /*              dynamics parameters         */
        M[Nm][Nm],      /*                                          */
        dM[Nm][Nm],     /*                                          */
        V,GetV(),       /*                                          */
        Ch[Nm][Nm],     /*                                          */
        h[Nm],          /*                                          */
        noise,u1[Nm];   /*                                          */
 int    i,ii,j,Nstep,Ndatajamp;
 int    result;
 time_t ltime;


 stream0 = fopen("data0.prn","r");
 stream1 = fopen("param.dat","r");
 stream2 = fopen("data2.prn","w");


 fscanf(stream1,"%f",&lamda);
 printf("lamda=%f\n",lamda);
 for(j=0;j<Nm;++j){ fscanf(stream1,"%f",&k[j]); printf("k[%d]=%f\n",j,k[j]);}
 for(j=0;j<Nm;++j){ fscanf(stream1,"%f",&fai[j]); printf("fai[%d]=%f\n",j,fai[j]);}
 fscanf(stream1,"%f",&k1); fscanf(stream1,"%f",&k2);
 fscanf(stream1,"%f",&noise);
```

```
printf("noise=%f\n",noise);

GetInitCond(&Nstep,&dt,&Ndatajamp,l,q,qd,dq,dqd,ddqd);

GetParameter(l,M0,M1,P0,n,UT);

t=0.0;

for(i=0;i<3;++i){
for(j=0;j<Nm;++j){
 do { fscanf(stream0,"%c",&c); printf("%c",c); }
 while(c!=']');
}
printf("\n");
}
fprintf(stream2,"*\ntime\t");
for(j=0;j<Nm;++j){ fprintf(stream2,"s[%d]\t",j); }
for(j=0;j<Nm;++j){ fprintf(stream2,"u[%d]\t",j); }
for(j=0;j<Nm;++j){ fprintf(stream2,"u1[%d]\t",j); }
fprintf(stream2,"V\n");

_clearscreen(_GCLEARSCREEN);

for(i=0;i<Nstep;++i){

for(ii=0;ii<Ndatajamp;++ii){

 fscanf(stream0,"%f",&t);
 for(j=0;j<Nm;++j){ fscanf(stream0,"%f",&qd[j]); }
 for(j=0;j<Nm;++j){ fscanf(stream0,"%f",&dqd[j]); }
 for(j=0;j<Nm;++j){ fscanf(stream0,"%f",&ddqd[j]); }

 UpdateError(dt,q,dq,qd,dqd,ddqd,lamda,s);

 for(j=0;j<Nm;++j){ fai[j] = fai[j] + (lamda*fabs(s[j])-lamda*fai[j])*dt; }

 DynamicsParameter(q,dq,M0,M1,P0,M,Ch,dM);

 TorqueOutput(k,lamda,fai,q,dq,qd,dqd,ddqd,s,M,Ch,UT,dM,u);

 u1[0] = u[0];
 u1[1] = u[1];
 u1[2] = u[2];
 u1[3] = u[3];
 ParamUncertainty(u1,noise);

 RungeKutta(dt,u1,M0,M1,P0,n,q,dq,ddqd);


 V = GetV(M,s);

}
 t = dt*(i*Ndatajamp+ii);

 fprintf(stream2,"%f\t",t);
fprintf(stream2,"%f\t%f\t%f\t%f\t",s[0],s[1],s[2],s[3]);
fprintf(stream2,"%f\t%f\t%f\t%f\t",u[0],u[1],u[2],u[3]);
fprintf(stream2,"%f\t%f\t%f\t%f\t",u1[0],u1[1],u1[2],u1[3]);
```

```
  fprintf(stream2,"%f\n",V);

  _settextposition(1,1);
  printf("t=%f\n",t);
  printf("s="); for(j=0;j<Nm;++j){ printf("%f\t",s[j]); } printf("\n");
  printf("u="); for(j=0;j<Nm;++j){ printf("%f\t",u[j]); } printf("\n");
  printf("u1="); for(j=0;j<Nm;++j){ printf("%f\t",u1[j]); } printf("\n");
  printf("V=%f\n",V);
 }
  fclose(stream0);
  fclose(stream1);
  fclose(stream2);

  _clearscreen(_GCLEARSCREEN);
}



/*****************************************************/
/* INITIAL CONDITIONS AND PARAMETER SETTINGS */
/*****************************************************/

GetInitCond(Nstep,dt,Ndatajamp,l,q,qd,dq,dqd,ddqd)
int  *Nstep,*Ndatajamp;
float *dt,l[Nm],q[Nm],qd[Nm],dq[Nm],dqd[Nm],ddqd[Nm];
{
 FILE *stream0;
 int i;

 stream0 = fopen("initial.dat","r");

 fscanf(stream0,"%f",dt);

 fscanf(stream0,"%d",Nstep);

 fscanf(stream0,"%d",Ndatajamp);

 for(i=0;i<Nm;++i){
  fscanf(stream0,"%f",&l[i]);
 }

 for(i=0;i<Nm;++i){ fscanf(stream0,"%f",&q[i]); }
 for(i=0;i<Nm;++i){ fscanf(stream0,"%f",&qd[i]); }
 for(i=0;i<Nm;++i){ fscanf(stream0,"%f",&dq[i]); }
 for(i=0;i<Nm;++i){ fscanf(stream0,"%f",&dqd[i]); }

 fclose(stream0);

 GetInitArmAccel(ddqd);

 printf("\n");
 printf("Nstep=%d\n",*Nstep);
 printf("Ndatajamp=%d\n",*Ndatajamp);
 printf("dt=%f\n",*dt);
 for(i=0;i<Nm;++i){ printf("%f\t",q[i]); } printf("\n");
 for(i=0;i<Nm;++i){ printf("%f\t",qd[i]); } printf("\n");
 for(i=0;i<Nm;++i){ printf("%f\t",dq[i]); } printf("\n");
 for(i=0;i<Nm;++i){ printf("%f\t",dqd[i]); } printf("\n");
```

```
 for(i=0;i<Nm;++i){ printf("%f\t",ddqd[i]); } printf("\n");

 return;
}


GetInitArmAccel(ddq)
float ddq[Nm];
{
  int i;

 for(i=0;i<Nm;++i){ ddq[i] = 0.0; }

 return;
}


/**************************/
/* DYNAMIC PARAMETERS */
/**************************/
GetParameter(l,M0,M1,P0,n,UT)
float l[Nm],M0[Nm][Nm],M1[Nm][Nm],P0[Nm][Nm],n[Nm],UT[Nm][Nm];
{
 float d[Nm],m[Nm],z[Nm],za[Nm],mq[Nm];

 GetCenterOfMass(d);

 GetMass(m);

 GetMomentOfInertia(z);

 GetMomentOfInertiaOfRotor(za);

 GetGearRatio(n);

 GetTransMatrix(n,UT);

 Getmq(z,m,l,mq);

 GetM0(mq,n,za,M0);

 GetM1(n,za,M1);

  GetP0(m,l,d,P0);

 return;
}


GetCenterOfMass(d)
float d[Nm];
{
 d[0] = 0.1135;  d[1] = 0.1135;  d[2] = 0.1255;  d[3] = 0.0762;
 return;
}


GetMass(m)
```

```
float m[Nm];
{
 m[0] = 1.410;  m[1] = 1.410;  m[2] = 1.135;  m[3] = 0.756;
 return;
}


GetMomentOfInertia(z)
float z[Nm];
{
 z[0] = .0297;  z[1] = .0297;  z[2] = .0297;  z[3] = .0067;
 return;
}


GetMomentOfInertiaOfRotor(za)
float za[Nm];
{
 za[0] = 5.02E-6;  za[1] = 5.02E-6;  za[2] = 5.02E-6;  za[3] = 1.89E-6;
 return;
}


GetGearRatio(n)
float n[Nm];
{
 n[0] = -146.;  n[1] = -146.;  n[2] = -98.5;  n[3] = -46.875;
 return;
}


GetTransMatrix(n,UT)
float n[Nm],UT[Nm][Nm];
{
 UT[0][0] = UT[0][1] = UT[0][2] = UT[0][3] = 1/n[0];
 UT[1][1] = UT[1][2] = UT[1][3] = 1/n[1];
 UT[2][2] = UT[2][3] = 1/n[2];
 UT[3][3] = 1/n[3];
 UT[1][0] = UT[2][0] = UT[2][1] = UT[3][0] = UT[3][1] = UT[3][2] = 0.0;

 return;
}


Getmq(z,m,l,mq)
float z[Nm],m[Nm],l[Nm],mq[Nm];
{
 mq[0] = z[0]+(m[1]+m[2]+m[3])*l[0]*l[0];
 mq[1] = z[1]+(m[2]+m[3])*l[1]*l[1];
 mq[2] = z[2]+m[3]*l[2]*l[2];
 mq[3] = z[3];
 return;
}
```

```
GetM0(mq,n,za,M0)
float mq[Nm],n[Nm],za[Nm],M0[Nm][Nm];
{
  int i,j;

  for(i=0;i<Nm;++i){
    for(j=0;j<Nm;++j){
      M0[i][j] = 0.0;
    }
  }
  M0[0][0] = mq[0]+(1-n[1])*za[1];
  M0[0][1] = n[1]*za[1];
  M0[1][1] = mq[1]+(1-n[2])*za[2];
  M0[1][2] = n[2]*za[2];
  M0[2][2] = mq[2]+(1-n[3])*za[3];
  M0[2][3] = n[3]*za[3];
  M0[3][3] = mq[3];

  return;
}


GetM1(n,za,M1)
float n[Nm],za[Nm],M1[Nm][Nm];
{
  int i,j;

  for(i=0;i<Nm;++i){
    for(j=0;j<Nm;++j){
      M1[i][j] = 0.0;
    }
  }
  for(i=0;i<Nm;++i){
    M1[i][i] = n[i]*za[i];
  }
  for(i=1;i<Nm;++i){
    M1[i][i-1] = (1-n[i])*za[i];
  }

  return;
}


GetP0(m,l,d,P0)
float m[Nm],l[Nm],d[Nm],P0[Nm][Nm];
{
  int i,j;

  for(i=0;i<Nm;++i){
    for(j=0;j<Nm;++j){
      P0[i][j] = 0.0;
    }
  }
  P0[0][1] = P0[1][0] = (m[1]*d[1]+(m[2]+m[3])*l[1])*l[0];
  P0[1][2] = P0[2][1] = (m[2]*d[2]+m[3]*l[2])*l[1];
  P0[2][3] = P0[3][2] = m[3]*l[2]*d[3];
  P0[0][2] = P0[2][0] = (m[2]*d[2]+m[3]*l[2])*l[0];
  P0[1][3] = P0[3][1] = m[3]*l[1]*d[3];
```

```
 P0[0][3] = P0[3][0] = m[3]*l[0]*d[3];

 return;
}


/*************************/
/* CONTROLLER OUTPUT */
/*************************/

UpdateError(dt,q,dq,qd,dqd,ddqd,lamda,s)
float dt,q[Nm],dq[Nm],
    qd[Nm],dqd[Nm],ddqd[Nm],
     lamda,s[Nm];
{
 int i;

 for(i=0;i<Nm;++i){s[i] = (dq[i]-dqd[i])+lamda*(q[i]-qd[i]);}

 return;

}


DynamicsParameter(q,dq,M0,M1,P0,M,Ch,dM)
float q[Nm],dq[Nm],M0[Nm][Nm],M1[Nm][Nm],P0[Nm][Nm],
    M[Nm][Nm],Ch[Nm][Nm],dM[Nm][Nm];
{
 int   i,j;

 for(i=0;i<Nm-1;++i){
   for(j=i+1;j<Nm;++j){
    M[i][j] = P0[i][j]*cos(q[i]-q[j]);
     M[j][i] = M[i][j];
   }
 }
 for(i=0;i<Nm;++i){
   M[i][i] = 0.0;
 }
 M[0][0] = M[0][0]+M0[0][0];
 M[0][1] = M[0][1]+M0[0][1];
 M[1][1] = M[1][1]+M0[1][1];
 M[1][2] = M[1][2]+M0[1][2];
 M[2][2] = M[2][2]+M0[2][2];
 M[2][3] = M[2][3]+M0[2][3];
 M[3][3] = M[3][3]+M0[3][3];

 for(i=0;i<Nm-1;++i){
   for(j=i+1;j<Nm;++j){
    dM[i][j] = -P0[i][j]*sin(q[i]-q[j])*(dq[i]-dq[j]);
    dM[j][i] = dM[i][j];
   }
 }
 for(i=0;i<Nm;++i){
   dM[i][i] = 0.0;
 }

 for(i=0;i<Nm;++i){
```

```
    for(j=0;j<Nm;++j){
     Ch[i][j] = P0[i][j]*sin(q[i]-q[j])*dq[j];
    }
  }
  for(i=0;i<Nm;++i){
    Ch[i][i] = 0.0;
  }

  return;

}



TorqueOutput(k,lamda,fai,q,dq,qd,dqd,ddqd,s,M,Ch,UT,dM,u)
float k[Nm],lamda,fai[Nm],q[Nm],dq[Nm],
     qd[Nm],dqd[Nm],ddqd[Nm],s[Nm],
      M[Nm][Nm],Ch[Nm][Nm],UT[Nm][Nm],dM[Nm][Nm],
       u[Nm];
{
  float M1[Nm][Nm],h1[Nm];
  float ddqr[Nm],dqr[Nm],u1[Nm],u2[Nm],u3[Nm],u4[Nm],u5[Nm],sat(),sgn();
  float ddq[Nm],uu[Nm];
  int   i;

  for(i=0;i<Nm;++i){
   ddqr[i] = ddqd[i] - lamda*(dq[i]-dqd[i]);
   dqr[i]  =  dqd[i] - lamda*( q[i]- qd[i]);
   u4[i]   =  -k[i]*sat(s[i]/fai[i]);
  }

  MatVec(M,ddqr,Nm,Nm,u1);
  MatVec(Ch,dqr,Nm,Nm,u2);
  VecAdd(u1,u2,Nm,u3);
  VecAdd(u4,u3,Nm,u5);
  MatVec(UT,u5,Nm,Nm,u);

  return;
}


float GetV(M,s)
float M[Nm][Nm],s[Nm];
{
    float s1[Nm],V;
    int   i;

    MatVec(M,s,Nm,Nm,s1);
    V=0.0;
    for(i=0;i<Nm;++i){ V=V+s[i]*s1[i]; }

    return(V);
}



ParamUncertainty(u,noise)
float u[Nm],noise;
```

```
{
 float r;
 int   i;

 for(i=0;i<Nm;++i){
  r = 2*noise*(0.5-rand()/32767);
  u[i] = (1+r)*u[i];
 }

 return;
}



/***********************/
/*  SYSTEM DYNAMICS */
/***********************/

RungeKutta(dt,u,M0,M1,P0,n,q,dq,ddqd)
float dt,u[Nm],M0[Nm][Nm],M1[Nm][Nm],
    P0[Nm][Nm],n[Nm],q[Nm],dq[Nm],ddqd[Nm];
{
 float qp[Nm],dqp[Nm],k1[Nm],k2[Nm],k3[Nm],k4[Nm];
 int j;


 for(j=0;j<Nm;++j){
  qp[j]  = q[j];
  dqp[j] = dq[j];
 }
 dyn(dt,qp,dqp,u,M1,n,M0,P0,ddqd,k1);

 for(j=0;j<Nm;++j){
  qp[j]  = q[j]+dt*dq[j]/2+dt*k1[j]/8;
  dqp[j] = dq[j]+k1[j]/2;
 }
 dyn(dt,qp,dqp,u,M1,n,M0,P0,ddqd,k2);

 for(j=0;j<Nm;++j){
  qp[j]  = q[j]+dt*dq[j]/2+dt*k2[j]/8;
  dqp[j] = dq[j]+k2[j]/2;
 }
 dyn(dt,qp,dqp,u,M1,n,M0,P0,ddqd,k3);

 for(j=0;j<Nm;++j){
  qp[j]  = q[j]+dt*dq[j]+dt*k3[j]/2;
  dqp[j] = dq[j]+k3[j];
 }
 dyn(dt,qp,dqp,u,M1,n,M0,P0,ddqd,k4);

 for(j=0;j<Nm;++j){
  q[j]  = q[j]+dt*(dq[j]+(k1[j]+k2[j]+k3[j]+k4[j])/6);
  dq[j] = dq[j]+(k1[j]+2*k2[j]+2*k3[j]+k4[j])/6;
 }

 return;
}
```

```
dyn(dt,q,dq,u,M1,n,M0,P0,ddqm,ddq)
float dt,q[Nm],dq[Nm],u[Nm],M1[Nm][Nm],n[Nm],
    M0[Nm][Nm],P0[Nm][Nm],ddqm[Nm],ddq[Nm];
{
 float M[Nm][Nm],A[Nm][Nm],h[Nm],qq[Nm],
g[Nm],tau[Nm],v1[Nm],v2[Nm],iM[Nm][Nm];
  int   i,j;

 g[0] = (u[0]-M1[0][0]*ddqm[0])*n[0];
 g[1] = (u[1]-M1[1][0]*ddqm[0]-M1[1][1]*ddqm[1])*n[1];
 g[2] = (u[2]-M1[2][1]*ddqm[1]-M1[2][2]*ddqm[2])*n[2];
 g[3] = (u[3]-M1[3][2]*ddqm[2]-M1[3][3]*ddqm[3])*n[3];

 tau[0] = g[0]-g[1];
 tau[1] = g[1]-g[2];
 tau[2] = g[2]-g[3];
 tau[3] = g[3];

 for(i=0;i<Nm-1;++i){
   for(j=i+1;j<Nm;++j){
    M[i][j] = P0[i][j]*cos(q[i]-q[j]);
    A[i][j] = P0[i][j]*sin(q[i]-q[j]);
    M[j][i] = M[i][j];
    A[j][i] = -A[i][j];
   }
 }
 for(i=0;i<Nm;++i){
   M[i][i] = 0.0;
   A[i][i] = 0.0;
  qq[i] = dq[i]*dq[i];
 }
 M[0][0] = M[0][0]+M0[0][0];
 M[0][1] = M[0][1]+M0[0][1];
 M[1][1] = M[1][1]+M0[1][1];
 M[1][2] = M[1][2]+M0[1][2];
 M[2][2] = M[2][2]+M0[2][2];
 M[2][3] = M[2][3]+M0[2][3];
 M[3][3] = M[3][3]+M0[3][3];

 MatVec(A,qq,Nm,Nm,h);
 VecSub(tau,h,Nm,v1);
 inv44(M,iM);
 MatVec(iM,v1,Nm,Nm,v2);
 VecCoef(dt,v2,Nm,ddq);

 return;
}


/*******************/
/* ENVIRIONMENT */
/*******************/
OutsideCondition(t,cg,cob,xg,xob)
float t,cg[2][2],cob[Nob][2][2],
    xg[2],xob[Nob][2];
{
```

```
 int i,j;

 for(i=0;i<2;++i){
  xg[i] = cg[i][0] + cg[i][1]*t;
 }

 for(i=0;i<Nob;++i){
  for(j=0;j<2;++j){
   xob[i][j] = cob[i][j][0] + cob[i][j][1]*t;
  }
 }

 return;
}


/*****************************************/
/* DISTANCE CALCULATION SUBROUTINES*/
/*****************************************/

GetArmPos(l,q,xam)
float l[Nm],
      q[Nm],
      xam[Nmp][2];
{
 float q1,xx,yy;
 int   i,j,k;

 xam[0][0] = 0.0;
 xam[0][1] = 0.0;

 for(i=1;i<Nmp;++i){
   xx = 0.0;
   yy = 0.0;
   for(j=0;j<i;++j){
    xx = xx + l[j]*cos(q[j]);
    yy = yy + l[j]*sin(q[j]);
   }

   xam[i][0] = xam[0][0]+xx;
   xam[i][1] = xam[0][0]+yy;
 }

 return;

}


DistBetweenEndAndObstacle(xam,xob,xde,deo)
float xob[Nob][2],xam[Nmp][2];
float xde[2],*deo;
{
 float xx,yy,d,k;
 int   i,j;

 *deo = fabs(xam[Nmp-1][0]-xob[0][0])
              +fabs(xam[Nmp-1][1]-xob[0][1]);
```

```
  for(j=0;j<Nob-1;j++){
   NearPt(xam[Nmp-1][0],xam[Nmp-1][1],xob[j][0],xob[j][1],xob[j+1][0],xob[j+1][1],&xx,&yy,&d,&k);
   if(d<*deo){
     *deo = d;
     xde[0] = xx;
     xde[1] = yy;
   }
  }


  return;
}



NearestPoints(xam,xob,iam,sam,xco,dmin)
float  xob[Nob][2],xam[Nmp][2],
       sam[Nm],xco[Nm][2],dmin[Nm];
int    iam[Nm];
{
  float xx,yy,d,k;
  int   i,j,ii;

  for(i=0;i<Nm;++i){
   dmin[i] = fabs(xam[i][0]-xob[0][0])+fabs(xam[i][1]-xob[0][1]);
   iam[i] = i;
   for(j=0;j<Nob;j++){
    NearPt(xob[j][0],xob[j][1],xam[i][0],xam[i][1],xam[i+1][0],xam[i+1][1],&xx,&yy,&d,&k);
    if(d<dmin[i]){
      dmin[i] = d;
      sam[i] = k;
      xco[i][0] = -xx;
      xco[i][1] = -yy;
    }
   }
  }

  for(i=1;i<Nm;++i){
   for(j=0;j<Nm-i;++j){
    if(dmin[iam[j]]>dmin[iam[j+1]]){
      ii = iam[j];
      iam[j] = iam[j+1];
      iam[j+1] = ii;
    }
   }
  }

  return;
}

NearPt(xx,xy,y1x,y1y,y2x,y2y,x,y,d,k)
float  xx,xy,y1x,y1y,y2x,y2y;
float  *x,*y,*d,*k;
/*****************************************/
/*    Distance between a line and a point           */
/*  (x,y)                      : position of the point    */
/*  (y1x,y1y),(y2x,y2y)  : end points of the line     */
/*  d                          : distance                  */
```

```
/*  k                       : position of the nearest    */
/*                            point in the line          */
/*  (x,y)                   : direction from the point   */
/*                            to the nearest point on    */
/*                            the line                   */
/******************************************************/
{
  float zx,zy,zd,zk;

  zk = ((y2x-y1x)*(xx-y1x) + (y2y-y1y)*(xy-y1y))
       /((y2x-y1x)*(y2x-y1x) + (y2y-y1y)*(y2y-y1y));
  if(zk<0){
    zk = 0;
  }
  if(zk>1){
    zk = 1;
  }

  zx = y1x-xx + zk*(y2x-y1x);
  zy = y1y-xy + zk*(y2y-y1y);
  zd = sqrt(zx*zx + zy*zy);

  *x = zx;
  *y = zy;
  *d = zd;
  *k = zk;

  return;
}



float sat(x)
float x;
{
  float y;

  if(x<-1){y=-1;}
  else if(x>1){y=1;}
  else{y=x;}

  return(y);
}



float prox(d,d1,d2)
float d,d1,d2;
{
  float y;

  if(d<d1){y = 0.0;}
  else{
    if(d<d2){y = -(d-d1)*(d-d1)*(2*d-3*d2+d1)/(d2-d1)/(d2-d1)/(d2-d1);}
    if(d2<=d){y = 1.0;}
  }

  return(y);
```

```
}


/**************************************************/
/*  LIBRARY OF MATRIX OPERATIONS SUBROUTINES */
/**************************************************/


/*
  >> MULTIPLIES COEFFICIENT TO A VECTOR <<
  vres[i] = a*v[i]  i=1,...,m
*/
VecCoef(a,v,m,vres)
int    m;
float a,*v,*vres;
{
  int i;
  for(i=0;i<m;++i){
    *vres++ = a*(*v++);
  }
}


/*
  >> ADDS TWO VECTORS <<
  vres[i] = v1[i] + v2[i] i=1,...,m
*/
VecAdd(v1,v2,m,vres)
int    m;
float *v1,*v2,*vres;
{
  int i;
  for(i=0;i<m;++i){
    *vres++ = *v1++ + *v2++;
  }
  return;
}


/*
  >> SUBTRACTS TWO VECTORS <<
  vres[i] = v1[i] - v2[i]  i=1,...,m
*/
VecSub(v1,v2,m,vres)
int    m;
float *v1,*v2,*vres;
{
  int i;
  for(i=0;i<m;++i){
    *vres++ = *v1++ - *v2++;
  }
  return;
}



/*
```

```
  >> GIVES IDENT MATRIX <<
  A = I  (m x m matrix)
 */
 IdentMat(A,m)
 int    m;
 float *A;
 {
  int i,j;
  for(i=0;i<m;++i){
    for(j=0;j<m;++j){
     if(i==j){*A++ = 1.0;}
     if(i!=j){*A++ = 0.0;}
    }
  }
  return;
 }


/*
  >> TRANSPOSES A MATRIX <<
  A - m x n (INPUT)
  B - n x m (OUTPUT)
*/
transp(A,m,n,AT)
int    m,n;
float *A,*AT;
{
  int    i,j;
  float  *pA,*pAT;
  pA = A;
  for(i=0;i<m;++i){
    for(j=0;j<n;++j){
      pAT = AT + j*m + i;
      *pAT = (*pA++);
    }
  }
  return;
}


/*
  >> ADDS TWO MATRICES <<
  C = A + B  A,B,C - m x n
*/
MatAdd(A,B,m,n,C)
int    m,n;
float  *A,*B,*C;
{
  int i,j;
  for(i=0;i<m;++i){
    for(j=0;j<n;++j){
      *C++ = *A++ + *B++;
    }
  }
  return;
}
```

```
/*
 >> SUBTRACTS TWO MATRICES <<
 C=A-B  A,B,C - m x n
*/
MatSub(A,B,m,n,C)
int    m,n;
float  *A,*B,*C;
{
 int i,j;
 for(i=0;i<m;++i){
  for(j=0;j<n;++j){
    *C++ = *A++ - *B++;
  }
 }
 return;
}


/*
 >> MULTIPLIES MATRIX WITH VECTOR <<
 arvec=ar*vec:  vec (n vector)
                arvec ( m vector)
                ar (m x n matrix)
*/
MatVec(ar,vec,m,n,arvec)
int    m,n;
float *ar,*vec,*arvec;
{
 int    i,j;
 float  prod,*pv;
 for(i=0;i<m;++i){
  prod = 0.0;
  pv = vec;
  for(j=0;j<n;++j){
   prod = prod + (*ar++)*(*pv++);
  }
  *arvec++ = prod;
 }
 return;
}


/*
   >> MULTIPLIES TWO MATRICES <<
   mat12=mat1*mat2:  mat1 - m x k
                     mat2 - k x n
                     mat12 - m x n
*/
MatMul(pmat1,pmat2,m,k,n,pmat12)
int    m,k,n;
float  *pmat1,*pmat2,*pmat12;
{
 int    row,col,i;
 float  *pm1,*pm2,*pm12;
 float  *pm2col,*pm12col,prod;

 pm12col = pmat12;
```

```
  pm2col = pmat2;
  for(col=0;col<n;++col){
    pm1 = pmat1;
    pm12 = pm12col++;
    for(row=0;row<m;++row){
      pm2 = pm2col;
      prod = 0.0;
      for(i=0;i<k;++i){
        prod += (*pm1++) * (*pm2);
        pm2 += n;
      }
      *pm12 = prod;
      pm12 += n;
    }
    pm2col++;
  }
  return;
}



/*
  >> NULIFIES MATRIX <<
  A = 0.0  (m x n matrix)
*/
NullMat(A,m,n)
int    m,n;
float  *A;
{
  int i,j;

  for(i=0;i<m;++i){
    for(j=0;j<n;++j){
      *A++ = 0.0;
    }
  }
  return;
}



/*
  >> NULIFIES VECTOR <<
  v[i]=0.0  i=1,...,m
*/
NullVec(v,m)
int    m;
float  *v;
{
  int i;

  for(i=0;i<m;++i){
    *v++ = 0.0;
  }
  return;
}
```

```
/*
  >>  INVERTS 2x2 MATRIX <<
*/
float Inv2(A,B)
float A[2][2],B[2][2];
{
 float d;

 d = A[0][0]*A[1][1] - A[0][1]*A[1][0];

 if(fabs(d)>1.0e-9){
  B[0][0] = A[1][1]/d;
  B[0][1] = -A[0][1]/d;
  B[1][0] = -A[1][0]/d;
  B[1][1] = A[0][0]/d;
   }

 else{
  B[0][0] = 0.00;
  B[0][1] = 0.00;
  B[1][0] = 0.00;
  B[1][1] = 0.00;
   }

 return(d);
}
```