1994 NASA/ASEE SUMMER FACULTY FELLOWSHIP PROGRAM

JOHN F. KENNEDY SPACE CENTER
UNIVERSITY OF CENTRAL FLORIDA

AUTOMATED PATH PLANNING OF THE PAYLOAD INSPECTION AND PROCESSING SYSTEM

PREPARED BY:	Dr. Robert M. Byers
ACADEMIC RANK:	Assistant Professor
UNIVERSITY AND DEPARTMENT:	University of Central Florida Department of Mechanical and Aerospace Engineering
NASA/KSC	
DIVISION:	Advanced Systems
BRANCH:	Automation Group
NASA COLLEAGUE:	Eduardo Lopez Gabor Tamasi
DATE:	August 12, 1994
CONTRACT NUMBER:	University of Central Florida NASA-NGT-60002 Supplement: 17

Automated Path Planning of the Payload Inspection and Processing System

R.M. Byers
University of Central Florida, Orlando, FL

Abstract

The Payload Changeout Room Inspection and Processing System (PIPS) is a highly redundant manipulator intended for performing tasks in the crowded and sensitive environment of the Space Shuttle Orbiter payload bay. Its dexterity will be exploited to maneuver the end effector in a workspace populated with obstacles. A method is described by which the end effector of a highly redundant manipulator is directed toward a target via a Lyapunov stability function. A cost function is constructed which represents the distance from the manipulator links to obstacles . Obstacles are avoided by causing the the vector of joint parameters to move orthogonally to the gradient of the workspace cost function. A *C* language program implements the algorithm to generate a joint history. The resulting motion is graphically displayed using the Interactive Graphical Robot Instruction Program (IGRIP) produced by Deneb Robotics. The graphical simulation has the potential to be a useful tool in path planning for the PIPS in the Shuttle Payload Bay environment.

Contents

List of Figures

1 Introduction

The range of motion achievable by a robot manipulator's end effector is a function of the number and type of joints or degrees of freedom it possesses. Any degrees of freedom in excess of the minimum number required to reach an arbitrary end effector position and orientation within the reachable workspace are considered "redundant". Commercial manipulators typically possess six or fewer DOF for primarily "anthropomorphic" tasks such as industrial assembly and are therefore not redundant.

There are some tasks for which such standard manipulators are not well suited, such as those requiring an extended reach in a confined workspace. For that reason, so-called "serpentine" manipulators have attracted interest. Their designation and appearance suggest the long reach and dexterity associated with snakes or tentacles. They achieve this snake-like ability by possessing a high degree of redundancy. This redundancy allows them, theoretically, to "wriggle" an end effector into a confined or difficult to reach point while allowing the robot arm to be configured in such a way as to not contact the surrounding environment.

The Payload Processing and Inspection System seeks to exploit the dexterity of the serpentine truss to service space shuttle orbiter payloads in the Payload Changeout Room (PCR). Because of the dimensions of the PCR and the sensitivity of shuttle payloads, there are specific tasks which are difficult , costly or hazardous to perform by humans due to lack of access. These tasks include:

- photographic inspections.
- visual inspections
- spot cleaning
- cover installation and removal
- line replaceable unit (LRU) installation and removal
- connector installation and removal.

References [2] and [1] discuss the requirements for inspection and processing of space related payloads and the feasibility for employing a manipulator to perform such tasks.

Several approaches for achieving collision avoidance with redundant manipulators have been suggested. Maciejewski and Klein [3], Nakamura [4], and Wegerif, et al [5] make use of the Moore–Penrose pseudo–inverse [6] to generate the joint rates to move the end effector and null motion to avoid obstacles. The pseudo–inverse solution is hampered by the existence of singularities for which the pseudo–inverse is undefined. Under these circumstances, no motion in the specified direction is possible. Sciavicco, and Siciliano [7] make use of a Lyapunov stability function to track a prescribed trajectory and augment the configuration space to a accommodate obstacle avoidance constraints. An alternative approach is used by Pasch [2] and Asano [8]. They prescribe an obstacle free end effector path and cause each joint to adhere to that path in a "follow the leader" mode. All of these methods require that at least the end effector's trajectory and velocity be prescribed. This presumes that a suitable velocity function for the end effector is readily determined. Only Wegerif [5], who makes use of sensors to detect obstacle proximity, allows for the end effector to deviate from the prescribed path as an emergency measure.

There are several limitations inherent in these approaches. The pseudo inverse kinematic solution may result in singular configurations for which some small motions of the end effector require excessive and physically unrealizable joint speeds. Although redundant degrees of freedom seem to offer some potential for singularity avoidance, Baker and Wampler [9] show that singularity free trajectories cannot be guaranteed. The requirement to specify the end effector trajectory and velocity

presumes that a suitable trajectory is easily determined. Such a trajectory must not only itself be obstacle and singularity free, but must allow for the permissible motion of the entrained links. Null motion may not be sufficient to cause the entrained links to avoid obstacles because such motion is constrained by the end effector trajectory requirements. Furthermore, as discussed by Doty, et al, [10] the pseudoinverse solution to robot manipulator kinematics can lead to inconsistent results (i.e. results that are not invariant with respect to changes in the reference frame and/or changes in the dimensional units used to express the problem).

In Ref. [11] the principal investigator presented an alternative method for determining an acceptable robot trajectory which allows the end effector's path, as well as the entrained link's to be free to move around obstacles. The control algorithm uses a Lyapunov stability approach to generate a family of joint rates which will move the end effector toward a desired target. The relative motion of the joints can be weighted to meet operational requirements such as rate or deflection limits. Because the end effector path is not specified, there are no requirements for inverse solutions, and singular joint configurations are only encountered at the reachable workspace boundaries.

Obstacles are avoided by determining the distance from each link to the surface of each obstacle in the workspace. An obstacle gradient vector, indicates the direction, in the joints space toward the obstacle array. By selecting only joint motion which is orthogonal to this direction, collisions with obstacles are avoided.

In the current work , the collision avoidance algorithm is applied to a notional PIPS based on the Foster–Miller serpentine truss [2] with sixteen degrees of freedom. Both the end effector's desired final position and orientation may be specified. The algorithm is coded in the C programming language and graphically displayed using the IGRIP software.

2 Manipulator Kinematics

Typically, robot motion is sufficiently slow so that it is adequately controlled by commanding joint velocities in response to the robot kinematics. Serpentine motion and the requirements for collision avoidance are especially complex. It is sufficient to describe the motion in terms of the end effector position and velocity.

The end effector position is a function of the vector of generalized joint displacements $\underline{q}$.

$$\underline{r} = \underline{r}(q(t), t) \in \Re^m \tag{1}$$

Figure 1 illustrates that the location of a point in space given by the 3×1vector $\underline{r}$ can be expressed in terms of an inertial frame by its position in an intermediate frame, $\underline{r}_B$, the location of the origin of the intermediate frame, $\underline{r}_A$and the orientation of that frame with respect to the inertial frame, given by the 3×3 direction cosine matrix R_O^A.

$$\underline{r} = R_O^A \underline{r}_B + \underline{r}_A \tag{2}$$

It is appealing to express the transformation in the form

$$\underline{r} = T_O^A \underline{r}_B \tag{3}$$

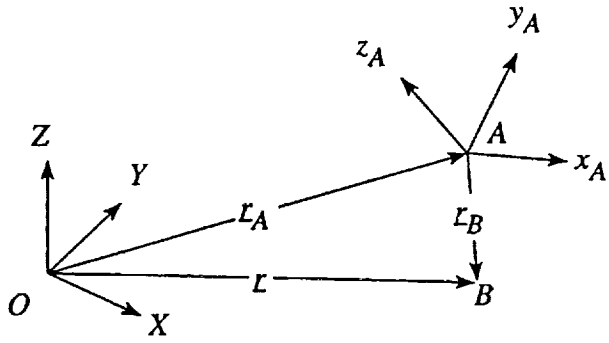

Figure 1: General Transform of a Vector

This is accomplished by defining the 4 × 4 transformation matrix relationship

$$\begin{bmatrix} \underline{r} \\ \hline 1 \end{bmatrix} = \left[\begin{array}{ccc|c} & R_O^A & & \underline{r}_A \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix} \underline{r}_B \\ \hline 1 \end{bmatrix} \tag{4}$$

The well known Denavitt–Hartenberg convention [13], is a convenient convention for describing the transformation between link coordinate frames and is shown in Fig. 2. The length a_i is the

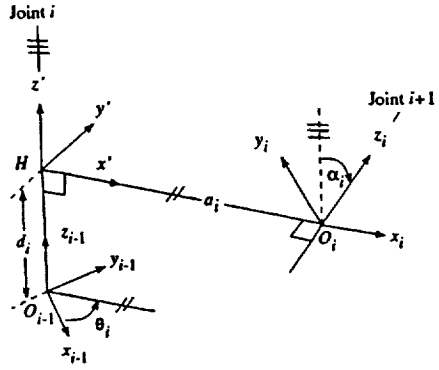

Figure 2: Denavitt–Hartenberg Coordinate Transform Convention

length of the common normal between the frames. For a revolute joint this is link length. The length d_i is the distance between the origin $O_{i-1}$and the point H_i. In a prismatic joint, this is the variable component.The angle α_i is the rotation of the joint axis i and the z_i axis. about the common normal; the "twist" of the link. The angle θ_i is the rotation angle between the x_{i-1} axis and the common normal H_iO_i measured about the z_{i-1} axis in the right–hand sense. In a revolute joint, this is the variable parameter. In the D–H convention, the 4 × 4 transformation between link

frames is given by

$$T_{i-1}^{i} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

For a manipulator consisting of n links, the position and orientation of the end effector (frame n) with respect to an inertial frame (frame 0), is expressed in terms of the link transformations

$$\begin{aligned} T_0^n &= \prod_{i=1}^{n} T_{i-1}^{i} \\ &= \left[\begin{array}{ccc|c} & R_O^{target} & & \underline{r}_{target} \\ -- & -- & -- & -- \\ 0 & 0 & 0 & 1 \end{array}\right] \\ &= \begin{bmatrix} \underline{r}_{n_1} & \underline{r}_{n_2} & \underline{r}_{n_3} & \underline{r}_{n_4} \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{6}$$

The vectors $\underline{r}_{n_1}$, $\underline{r}_{n_2}$, and $\underline{r}_{n_3}$ are unit column vectors of the direction cosine matrix which relates the end effector's orientation to the inertial frame. The vector $\underline{r}_{n_4}$ gives the end effectors location with respect to the origin of the base frame.

The velocity of the end effector is given by

$$\frac{d\underline{r}}{dt} = \frac{\partial \underline{r}}{\partial \underline{q}} \frac{d\underline{q}}{dt} = J\dot{\underline{q}} \tag{7}$$

where J is the *Jacobian* matrix. For an end effector trajectory specified by $\dot{\underline{r}}$ the required joint rates are given by

$$\dot{\underline{q}}_r = J^* \dot{\underline{r}} \tag{8}$$

where

$$J^* = J^T \left(JJ^T\right)^{-1} \tag{9}$$

is the pseudo-inverse for $n > 3$. Equation (8) gives the minimum norm joint rates which satisfy the end effector trajectory $\dot{\underline{r}}$. When $|JJ^T| = 0$ the pseudo inverse is undefined and infinite joint rates are required to satisfy the specified end effector velocity. Obviously, even when the manipulator is in a singular configuration, it is still possible to move the end effector in directions other than the singular direction.

There are several limitations to the pseudo inverse velocity kinematics solution of robot motion. As with all pseudo inverse kinematic solutions, the end effector's trajectory must be specified and takes priority over obstacle avoidance. Choosing an acceptable end effector path can be a difficult task in a complex workspace and it sometimes occurs that the specified path precludes obstacle avoidance. To further complicate matters, null motion for obstacle avoidance may be incompatible with the task of singularity avoidance. Finally, Doty, et al [10] notes that noninvariant results may be obtained from the pseudo-inverse solution.

3 Lyapunov Stability Approach to Manipulator Control

As an alternative to the operator prescribing the end effector path, the end effector may be driven to its target by use of a Lyapunov stability function. The desired end effector position may be represented by a target transformation

$$T_O^{target} = \left[\begin{array}{ccc|c} & R_O^{target} & & \underline{r}_{target} \\ -- & -- & -- & -- \\ 0 & 0 & 0 & 1 \end{array}\right] = \left[\begin{array}{cccc} \underline{r}_{T_1} & \underline{r}_{T_2} & \underline{r}_{T_3} & \underline{r}_{T_4} \\ 0 & 0 & 0 & 1 \end{array}\right] \tag{10}$$

where $\underline{r}_{T_i}$, $i = 1,2,3$, are the unit column vectors of R_O^{target} and $\underline{r}_{T_4} = \underline{r}_{target}$. The difference between the manipulator's actual configuration and the desired configuration is given by the array of vectors

$$\underline{\epsilon}_i = \underline{r}_{T_i} - \underline{r}_{n_i} \tag{11}$$

The scalar Lyapunov function is chosen

$$V = \sum_{i=1}^{4} \frac{1}{2} \underline{\epsilon}_i{}^T \underline{\epsilon}_i \tag{12}$$

V may be viewed as the "energy" of the system and is always positive. To drive V to zero, and hence the end effector to the target position and orientation , it is sufficient that $\dot{V} < 0$ for every subinterval of time on $t_0 \leq t \leq t_f$.

Taking the time derivative of V gives

$$\begin{aligned} \dot{V} &= \sum_{i=1}^{4} \underline{\epsilon}_i^T \dot{\underline{\epsilon}}_i \\ &= -\sum_{i-1}^{4} \underline{\epsilon}_i^T \dot{\underline{r}}_{n_i} \\ &= -\sum_{i=1}^{4} \underline{\epsilon}_i^T \left(J_i \dot{\underline{q}}_r \right) \end{aligned} \tag{13}$$

where

$$J_i = \frac{\partial r_{n_i}}{\partial \underline{q}} \qquad i = 1, \ldots 4 \tag{14}$$

It is obvious that $\dot{V} < 0$ is guaranteed by choosing the joint rate vector

$$\dot{\underline{q}}_r = M \sum_{i=1}^{4} \left(\frac{J_i^T \underline{\epsilon}_i}{\| J_i^T \underline{\epsilon}_i \|} \right) = M \hat{\underline{\epsilon}}_i \tag{15}$$

where M is a $n \times n$ positive definite scaling matrix, Eq. (13) becomes

$$\dot{V} = -\sum_{i=1}^{4} \frac{\underline{\epsilon}_i^T J_i M J_i^T \underline{\epsilon}_i}{\| J_i^T \underline{\epsilon}_i \|} \tag{16}$$

which is always negative. Substituting Eq. (15) into Eq. (7) gives

$$\dot{\underline{r}} = \sum_{i=1}^{4} J_i M \hat{\underline{\xi}}_i \tag{17}$$

No matrix inversion is required, and therefore the control is not sensitive to singularities. In contrast to Eq. (8) which gives joint rate to satisfy a desired trajectory, Eq. (17) moves the end effector in response to a family of joint rates which depend on the relative priority of joint motion caused by the matrix M. In addition, this matrix enforces appropriate unit transformations.

4 Joint Motion Weighting

Generally, the boundary conditions and obstacle avoidance requirements can be satisfied by an infinite number of joint trajectories by modification of the M matrix. The composition of the M matrix is determined by the various requirements on the hardware or end effector task.

In addition to avoiding obstacles, manipulator arms are frequently limited by the manipulator architecture in the magnitude of the joint deflections and joint rates which can be achieved. The M matrix may be selected to enforce joint rate and joint displacement limits.

It is useful to think of the M matrix as the non-linear stiffness matrix. The deflection of the *ith* joint is bounded by $q_{i_{min}} \leq q_i \leq q_{i_{max}}$. Defining

$$\begin{aligned} \Delta_i &= q_{i_{max}} - q_{i_{min}} \\ \Gamma_i &= q_{i_{max}} + q_{i_{min}} \\ f_i &= \frac{2q_i - \Gamma_i}{\Delta_i} \\ \eta_i &= \text{sign}\,(e_i) \\ k_i &\leq \frac{\dot{q}_{i_{max}}}{2} \end{aligned}$$

The elements of M are given by

$$m_{ij} = \begin{cases} k_i(1 - \eta_i f_i) & i = j \\ 0 & i \neq j \end{cases} \tag{18}$$

Equation (18) causes the *ith* joint rate toward the joint limit to approach zero near the limit and the rate to be near the maximum if away from the limit.

5 Obstacle Avoidance

With the end effector motion no longer prescribed, much greater latitude is allowed in obstacle avoidance. Joint motion which moves the manipulator away from obstructions is no longer subordinated the end effector path.

Obstacle avoidance requires that the distance to obstacles v *vis-a-vis* the manipulator links be known. In a realistic environment, devices in the workspace may be numerous and complexly shaped. CAD models of high complexity, such as exist in the Payload Changeout Room may be imported to IGRIP. The MIN_DISTANCE function in the GSL language returns the minimum designated

links and devices in the CAD environment. IGRIP can be set to disregard any devices outside of a selected radius.

The cost function C_{ij} is the minimum distance between the ith link and the jth obstacle (Fig. 3). Contact of the ith link with the jth obstacle is indicated by $C_{ij} = 0.2$.

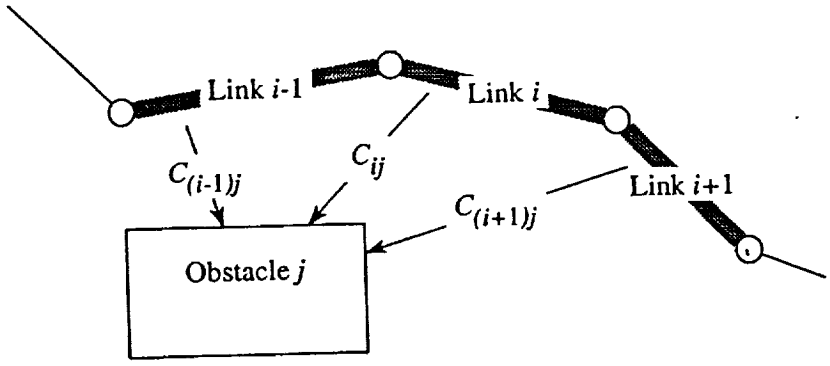

Figure 3: Obstacle Cost Functions

The potential function

$$P = \sum_{i=1}^{nl} \sum_{j=1}^{no} \frac{1}{C_{ij}} \tag{19}$$

where nl is the number of links and no is the number of obstacles. $P \to \infty$ upon contact with an obstacle. The gradient of the potential function with respect to the joint space vector is given by

$$\underline{\mu} = \frac{\partial P}{\partial \underline{q}} \tag{20}$$

The time rate of change of P can thus be expressed

$$\frac{dP}{dt} = \underline{\mu}^T \dot{\underline{q}} \tag{21}$$

Assuming that a trajectory exists which allows the end effector to reach the target without penetrating any obstacles, then if $\dot{P} \leq 0$ throughout the maneuver, the collision avoidance points will not encounter the obstacle surfaces. The component of $\dot{\underline{q}}_r$ which is orthogonal to $\underline{\mu}$ is found via the Gram–Schmidt orthogonalization method.

$$\dot{\underline{q}} = \dot{\underline{q}}_r - \left(\dot{\underline{q}}_r^T \hat{\underline{\mu}}\right) \hat{\underline{\mu}} \tag{22}$$

where $\hat{\underline{\mu}}$ is a unit vector in the direction $\underline{\mu}$. Equation (22) may be written

$$\begin{aligned} \dot{\underline{q}} &= \left[E_3 - \left(\hat{\underline{\mu}}\hat{\underline{\mu}}^T\right)\right] \dot{\underline{q}}_r \\ \Rightarrow \dot{\underline{q}} &= \tilde{M}\dot{\underline{\epsilon}} \end{aligned} \tag{23}$$

where

$$\tilde{M} = \left(E_n - \hat{\underline{\mu}}\hat{\underline{\mu}}^T\right) M \tag{24}$$

$\bar{M}$ is the obstacle avoidance metric. This matrix is positive semi-definite. The fact that this matrix possess a zero eigenvalue becomes evident when $\underline{\mu}$ is parallel to $\dot{\underline{q}}_r$. In this circumstance, it is impossible for the end effector to move closer to the target. The most obvious case occurs when the target is unreachable or the manipulator has entered a dead-end path. The operator may take some steps to avoid the manipulator from entering a dead-end by designating intermediate targets, or waypoints.

As an alternative to measuring distances from the links to devices in the workspace, obstacles may be modelled as primitive solids. In this approach, the centroid of the jth object is located at $\underline{o}_j = [x_j \quad y_j \quad z_j]^T$ and has the dimensions $2a_j$, $2b_j$, $2c_j$, along its principal axes. The orientation of the solid with respect to the inertial frame is given by a direction cosine matrix. Obstacle avoidance points p_i, $i = 1, \ldots, n_p$ are designated along the manipulator arm. In the simulation model, these points are the joints and the link midpoints.The location of the jth obstacle *vis-a-vis* the ith obstacle avoidance point is approximated by the super-ellipsoid function

$$C_j(p_i) = \left(\frac{x_i - x_j}{a_j}\right)^{k_j^x} + \left(\frac{y_i - y_j}{b_j}\right)^{k_j^y} + \left(\frac{z_i - z_j}{c_j}\right)^{k_j^z} \tag{25}$$

The the desired shape of the jth obstacle is approximated by selecting appropriate values for k_j^x, k_j^y, and k_j^z greater than or equal to one. For $k_j^x, = k_j^y = k_j^z = 1$ the surface is an octahedron. Setting $k_j^x, = k_j^y = k_j^z = 8$ approximates a rectangular parallelopiped. Contact with the surface of the jth obstacle by the ith collision avoidance point is approximated by $C_j(p_i) = 1$. The workspace potential function is defined by

$$P = \sum_j^{n_o} \sum_i^{n_p} [C_j(p_i) - 1]^{-1} \tag{26}$$

where np is the number of collision avoidance points. The gradient vector $\underline{\mu}$ is generated by a finite difference method as described above.

6 Graphical Representation with the IGRIP software

The Interactive Graphical Robot Instructional Program is a product of Deneb Robotics Inc. [12] It is a computer graphics based package for workcell layout, simulation and offline programming which permits the graphical simulation of virtually any robotic device. Devices used in the workcells may be added by modelling them with any of several CAD systems. A device has both geometric and non-geomentric information stored with it. Non-geometric information including kinematics, dynamics, velocities, etc., can be entered through interactive menus.

IGRIP allows robot programming via the Graphical Simulation Language (GSL), which in turn, can communicate with programs written in *C* programming language. This capability will eventually be exploited to imbed the robot control algorithm into the IGRIP simulation.

7 The Payload Inspection and Processing System

The Payload Inspection and Processing System (PIPS) is conceived as a highly redundant manipulator with a serpentine truss configuration. It is based on the Foster-Miller Serpentine Truss currently under development at the Kennedy Space Center. The truss shown in Fig. 4 can accomodate up to twelve degrees of freedom.

For the purposes of examing the efficacy of control algorithms, a notional PIPS, shown in Fig 5 has been designed. The illustration was generated in IGRIP. The Foster Miller Truss, with twelve

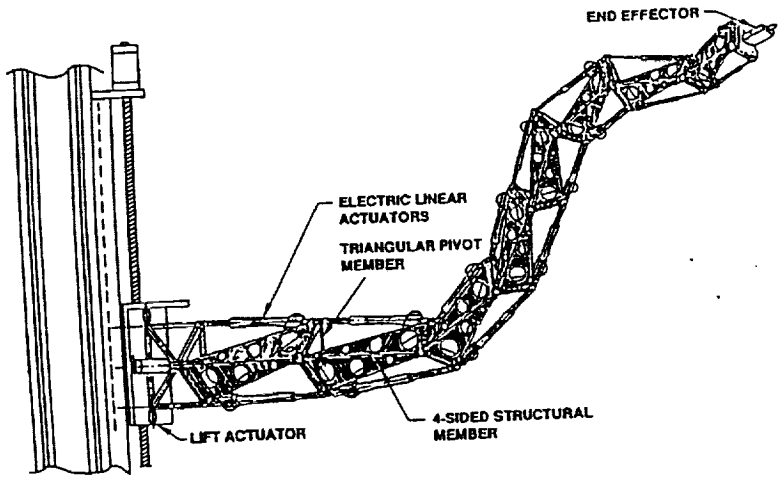

Figure 4: Foster Miller Serpentine Truss

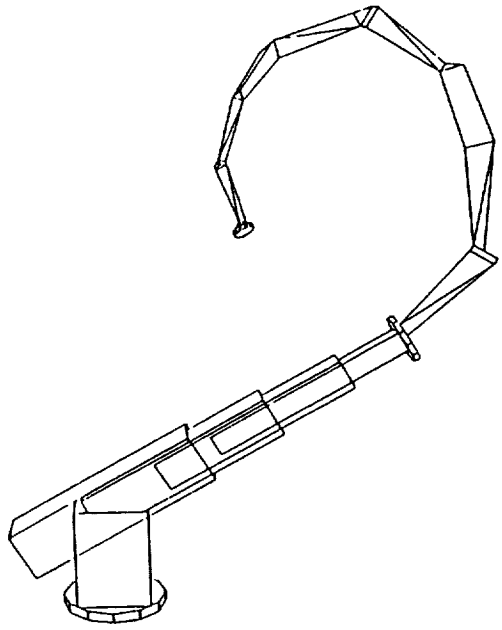

Figure 5; Notional PIPS

degrees of freedom, is mounted to a pedastal with two revolute joints and three telescoping prismatic joints. At the end of the truss, a revolute wrist is mounted, giving the complete system eighteen joints and sixteen independent degrees of freedom. The table of the Denavitt–Hartenberg parameters for the nominal "home" position, is shown in Table 1.

Table 1: Denevitt–Hartenberg Parameters for Notional PIPS

i	α_i(deg)	θ_i(deg)	a_i (in)	d_i (in)	joint type
1	90.0	90.0	0.0	22.0	revolute
2	90.0	0.0	0.0	0.0	revolute
3	0.0	0.0	-2.593	20.0	prismatic
4	0.0	-45.0	0.0	1.0	prismatic
5	-90.0	90.0	0.0	0.0	prismatic
6	90.0	0.0	20.0	0.0	revolute
7	-90.0	0.0	1.25	0.0	revolute
8	90.0	0.0	20.0	0.0	revolute
9	-90.0	0.0	16.003	0.0	revolute
10	90.0	0.0	16.004	0.0	revolute
11	-90.0	0.0	1.188	0.0	revolute
12	90.0	0.0	16.004	0.0	revolute
13	-90.0	0.0	12.006	0.0	revolute
14	90.0	0.0	12.004	0.0	revolute
15	-90.0	0.0	.813	0.0	revolute
16	90.0	0.0	12.004	0.0	revolute
17	-90.0	90.0	.833	-.833	revolute
18	0.0	45.0	0.0	1.793	revolute

8 Algorithm Implementation

The algorithm described above is executed in the *C* language program, Collision Avoidance Path Planner (CAPP.c). The program flow is shown in Fig.6(a). In order to make use of utilities imbedded in IGRIP, CAPP will itself become a library utility which can be accessed by a program written in a GSL program which directly controls the graphical simulation and shown in Fig. 6(b).

The CAPP program has demonstrated the ability to generate obstacle free trajectories for the PIPS model. In Fig7 the PIPS is shown manevering in a simple representation of the PCR/Shuttle Payload Bay environment. In the simulation, it is desired to view a point, to the aft of the large cylindrical payload from a distance of six inches.

9 Conclusions and Recommendation

An algorithm has been presented which will move the end effector of a redundant manipulator toward a target state while avoiding collisions of the arm with obstacles in the workspace. Allowing the end effector path to be free avoids the problem of singularities found in the pseudo–inverse solution of the robot kinematics. In addtion, it simplifies the operator's workload and allows greather latitude for obstacle avoidance. The algorithm is straightforward and requires only modest computing power.

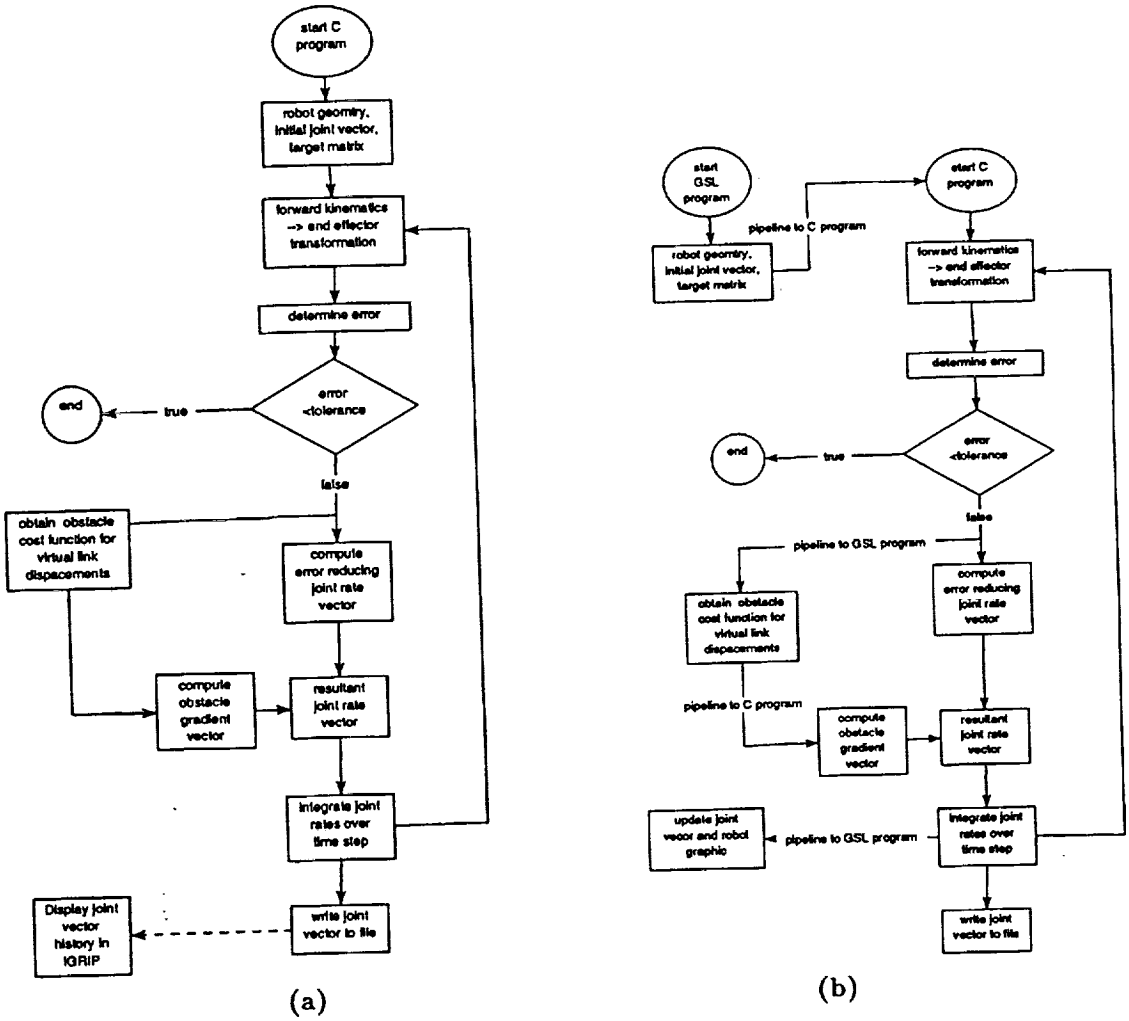

Figure 6: CAPP Program Flow
(a) Current Flow, (b) IGRIP Imbedded Flow

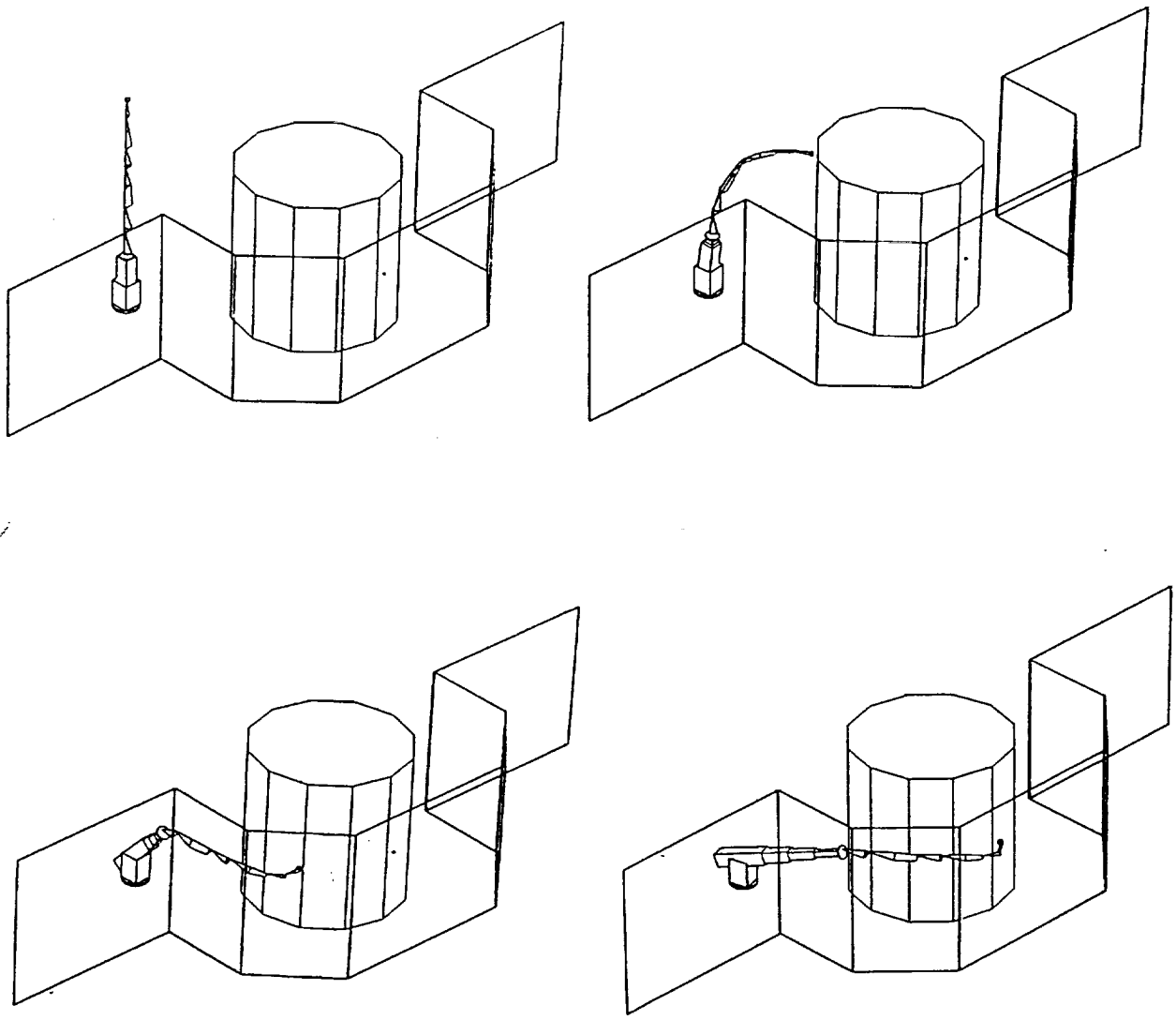

Figure 7: Simulated PIPS Maneuver

Although it is applied here to a highly redundant manipulator, redundancy is not explicitly required for its implementation.

Serpentine manipulators such as the PIPs are envisioned for employment in complex and costly environments. This method provides a tool for path planning by which specific maneuvers may be simulated without risk. A nominal joint history may be generated which is subsequently used as an open loop trajectory to be tracked by a robot with distributed control.

Offline processing of the robot trajectory, while sufficient to demonstrate the efficacy of the CAPP algorithm, is cumbersome and has severe shortcomings. Equation (25) has only limited utility to model a complex environment, such as exists in the shuttle payload bay. Processing time increases dramatically as the number and complexity of obstacles increases beyond a few simple shapes.

For that reason, it is recommended that future research be directed at various methods of interactively linking the GSL and C languages in IGRIP. This will allow the algorithm to interrogate IGRIP for distance information given by the MIN_DISTANCE utility. This should allow very complex and realisitic CAD models to be exploited and greatly reduced execution time.

There are several unresolved problems with automated path planning. In its current incarnation, the manipulator path is influenced to a great degree by its initial configuration with respect to the workspace. Heretofore, the "home" configuration has been chosen arbitrarily. It would be useful to the operator to have specific rules by which to chose an optimal configuration. The weighting of the joint motion is also somewhat arbitrary, currently only inforcing joint rate limits. The scaling between revolute and prismatic joints requires a more rigorous basis.

Currently, the operator may designate way-points which assist the algorithm in finding a collision free path. However, heuristics should be developed which help the manipulator avoid dead-ends and to choose between multiple paths around an obstacle.

A Appendix: Collision Avoidance Path Planner Source Code Listing

```
/*********************************************************************
*        COLLISION AVOIDANCE PATH PLANNER
*          Dr. Robert M. Byers, Unversity of Central Florida
*          8/4/94
*         Robot end effector directed to a point in space with
*        a desired orientation .
*         Obstacles are modelled by  hyperellipsoids
*         and may be oriented via 1-2-3 euler angles
*         Robot parameters contained in 'input.dat'
*      joint angles written to 'joints.dat'
*********************************************************************/
#include <stdio.h>
#include <math.h>
/************************************
*     function prototypes
************************************/
void  matrix_mult(float**matrix1,float**matrix2);
void end_effector(int n, float*q_p, float target_p[3][4],float error_p[3][4],float err_mag[4]);
void integrate(int n, float *var1, float *var2, float err, float err_dot, float step_s);
void  joints_print(int n,float *var, FILE *file);
void   **obstacle_transformation(int n, float **obst, float  vector_n[3]);
 float **IdentityMatrix(int n);
float ***JacobianMatrix(int n_dof, int n_obs,float *q_p, float  **ob);
float **transformation_matrix(int n, float *var);
float    *joint_rates(int  n,  float  *metric_p,  float   error_p[3][4],   float  target_p[3][4],
float***jb, float step_s);
float  *mem_alloc_1(int n);
float **mem_alloc_2(int nrows, int ncols);

 int *flag;
float *alpha,*theta,*a,*d;
/*****************************MAIN***********************************/

main()
{
/**********************
*   local variables     *
**********************/
int             i, j,k, num_dof, num_obstacles, num_waypoints, waypoint_counter;
float   ***jacobian;
float           *rate,*q;
float          *metric;
float   **obstacle;
float          **waypoint;
float   target[3][4], error[3][4];
float   rate_mag,  tolerance, move_dist;
float   error_mag[4],step_size, error_mag_old,error_prod, error_prod_old,error_prod_dist;
float   error_step=-1.0;
FILE *data;
FILE *joints;
joints=fopen("joints.dat","w");

        if((data=fopen("input.dat", "r"))==NULL)
        {
        printf("input file could not be opened\n");
        exit(-1);
        }

/* *************************
*               input data
```

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

fscanf(data,"%d", &num_dof);

alpha=mem_alloc_1(num_dof);
theta=mem_alloc_1(num_dof);
a=mem_alloc_1(num_dof);
d=mem_alloc_1(num_dof);
metric=mem_alloc_1(num_dof);
q=mem_alloc_1(num_dof);

if((flag=(int*)malloc(sizeof(int)*num_dof)) == (int *) NULL)
        {
        fprintf(stderr, "Error mallocing flag\n");
        exit(-1);
        }


for (i=0;i<num_dof;i++)
        {
        fscanf(data,"%f   %f   %f   %f      %f      %d\n",   &alpha[i],   &theta[i],   &a[i],   &d[i],
&metric[i],&flag[i]);

        if(flag[i]==1)
                q[i]=theta[i];
                else
                q[i]=d[i];
        }
/**********************************
*    read in target information
*    and way points
**********************************/
waypoint_counter=0;
fscanf(data,"%f   %f  %d", &step_size, &tolerance, & num_waypoints);
        waypoint=mem_alloc_2(3,num_waypoints);


for(i=0;i<num_waypoints;i++)
        for(j=0;j<3;j++)
                fscanf(data, "%f ", &waypoint[j][i]);
for(i=0;i<3;i++)
        target[i][3]=waypoint[i][0];

for(i=0;i<3;i++)
        for (j=0;j<3;j++)
                fscanf(data,"%f  ",&target[i][j]);

/**********************************
*   read obstacle array
**********************************/
fscanf(data, "%d", &num_obstacles);

obstacle=mem_alloc_2(12,num_obstacles);

for(i=0;i<num_obstacles;i++)
        for(j=0;j<12;j++)
                fscanf(data, "%f", &obstacle[j][i]);

fclose(data);
```

```
joints_print(num_dof,q,joints);

/* ************************************************
*         for loop until all waypoints  passed
************************************************/

while (waypoint_counter<num_waypoints)
{

/*****************************************
*     determine end effector position
*****************************************/
end_effector( num_dof, q,  target,error,error_mag);

for(j=0;j<4;j++)
       printf("%f  ",error_mag[j]);
       printf("\n");
error_mag_old=error_mag[3];
move_dist=error_mag_old;
error_prod=sqrt(error_mag[0]*error_mag[0]+error_mag[1]*error_mag[1]+error_mag[2]*error_mag[2]);
error_prod_old=error_prod;
/* ************************************************
*         while loop until error within tolerance
************************************************/
while (error_mag[3]>tolerance||error_prod>.4)
       {

       /**********************************
       *   form the jacobian matrix
       **********************************/
       jacobian=JacobianMatrix(num_dof,num_obstacles, q, obstacle);

       /*****************************************************
       *    determine joint rate vector toward target
       *****************************************************/
       rate=joint_rates(num_dof,metric, error, target,jacobian,step_size);

       /*****************************************************
       *   integrate joint rates to update joint parameters
       *****************************************************/
       integrate(num_dof, rate, q,error_mag[3],error_step, step_size);

       /*****************************************************
       *      recompute position vector and error vector
       *****************************************************/
       end_effector( num_dof, q,  target,error,error_mag);
       for(j=0;j<4;j++)
              printf("%f  ",error_mag[j]);
       printf("\n");
       error_step=error_mag[3]-error_mag_old;
       error_mag_old=error_mag[3];
error_prod=sqrt(error_mag[0]*error_mag[0]+error_mag[1]*error_mag[1]+error_mag[2]*error_mag[2]);
       error_prod_old=error_prod;

       /*****************************************
       *     print joint angles to "joints.dat"
       *****************************************/
       if(fabs(move_dist-error_mag[3])>1.0 ||fabs(error_prod_dist-error_prod)>.1)
              {
              move_dist=error_mag[3];
              error_prod_dist=error_prod;
              joints_print(num_dof,q,joints);
```

```
            }

      } /* end   error tolerance  while*/
/*****************************************************************************/
if(waypoint_counter==num_waypoints-1)
      printf("Target point reached\n");
else
      printf("\n Waypoint %d reached\n ", waypoint_counter);

waypoint_counter+=1;

for(i=0;i<3;i++)
      target[i][3]=waypoint[i][waypoint_counter];

}  /*close  waypoint counter  while loop */

/*******************************************
*     print  final  joint angles to "joints.dat"
*******************************************/
joints_print(num_dof,q,joints);

fclose(joints);

free(alpha);
free(theta);
free(a);
free(d);
free(metric);
free(flag);
free(q);
free(rate);

for(i=0;i<12;i++)
      free(obstacle[i]);
      free(obstacle);
for(i=0;i<3;i++)
      free(waypoint[i]);
      free(waypoint);

for(i=0;i<3;i++)
      {
      for(j=0;j<num_dof;j++)
            free(jacobian[i][j]);
      free(jacobian[i]);
      }
free(jacobian);

}
/*********************************
*    end of main program
*********************************/
/*****************TRANSFORMATION MATRIX*****************/
float **transformation_matrix(int n, float *var)
{
static float **transform;

int i;

if(!(transform)){
transform=(float **)malloc(sizeof(float*)*4);
for(i=0;i<4;i++)
```

```
        if((transform[i]=(float *)malloc(sizeof(float)*4)) == (float *) NULL)
                fprintf(stderr, "Error mallocing transform\n");
}


/*rotation matrix*/
if(flag[n]==1)
        {
        transform[0][0]=cos(var[n]);
        transform[0][1]=-cos(alpha[n])*sin(var[n]);
        transform[0][2]=sin(alpha[n])*sin(var[n]);
        transform[0][3]=a[n]*cos(var[n]);
        transform[1][0]=sin(var[n]);
        transform[1][1]=cos(alpha[n])*cos(var[n]);
        transform[1][2]=-sin(alpha[n])*cos(var[n]);
        transform[1][3]=a[n]*sin(var[n]);
        transform[2][0]=0;
        transform[2][1]=sin(alpha[n]);
        transform[2][2]=cos(alpha[n]);
        transform[2][3]=d[n];
        transform[3][0]=0;
        transform[3][1]=0;
        transform[3][2]=0;
        transform[3][3]=1;
        }
        else
        {
        transform[0][0]=cos(theta[n]);
        transform[0][1]=-cos(alpha[n])*sin(theta[n]);
        transform[0][2]=sin(alpha[n])*sin(theta[n]);
        transform[0][3]=a[n]*cos(theta[n]);
        transform[1][0]=sin(theta[n]);
        transform[1][1]=cos(alpha[n])*cos(theta[n]);
        transform[1][2]=-sin(alpha[n])*cos(theta[n]);
        transform[1][3]=a[n]*sin(theta[n]);
        transform[2][0]=0;
        transform[2][1]=sin(alpha[n]);
        transform[2][2]=cos(alpha[n]);
        transform[2][3]=var[n];
        transform[3][0]=0;
        transform[3][1]=0;
        transform[3][2]=0;
        transform[3][3]=1;
        }

        return(transform);
}

/****************MATRIX MULTIPLICATION*********************/
void  matrix_mult(float**matrix1,float**matrix2)
{
float **matrix3=mem_alloc_2(4,4);
int i,j,k;

 for(i=0;i<4;i++)
        for (j=0;j<4;j++){
                                matrix3[i][j]=0;
                for (k=0;k<4;k++)
        matrix3[i][j]+=matrix1[i][k]*matrix2[k][j];}

for(i=0;i<4;i++)
{
```

```
        free(matrix1[i]);
        matrix1[i]=matrix3[i];
        }
}

/*******************IDENTITY MATRIX*******************/
float **IdentityMatrix(int n)
{
float **matrix=mem_alloc_2(n,n);
int i,j;


for(i=0;i<n;i++)
        for(j=0;j<n;j++)

                if(i==j)
                        matrix[i][j]=1;
                else
                        matrix[i][j]=0;

return(matrix);


}
/*******************JACOBIAN MATRIX*****************************/
float ***JacobianMatrix(int n_dof, int n_obs, float *q_p,float  **ob)
{
float ***jacobian;
float ***temp;
float  **result_plus, **result_minus;
float     *q_plus=mem_alloc_1(n_dof);
float     *q_minus=mem_alloc_1(n_dof);
float          *gradient=mem_alloc_1(n_dof);
float   cost_minus;
float   cost_plus;
float          **mu_matrix=mem_alloc_2(n_dof,n_dof);
float              potential_plus, potential_minus;
float              end_point_plus[3],end_point_minus[3];
float             mid_point_plus[3],mid_point_minus[3];
float              gradient_mag=0.0;
int i, j, k,kk;

jacobian=(float***)malloc(sizeof(float**)*3);
for (i=0;i<3;i++)
        {
        jacobian[i]=(float**)malloc(sizeof(float*)*n_dof);
        for(j=0;j<n_dof;j++)
        jacobian[i][j]=(float*)malloc(sizeof(float)*4);
        }
        temp=(float***)malloc(sizeof(float**)*3);
for (i=0;i<3;i++)
        {
        temp[i]=(float**)malloc(sizeof(float*)*n_dof);
        for(j=0;j<n_dof;j++)
                temp[i][j]=(float*)malloc(sizeof(float)*4);
        }

/*****************************************
*  virtual joint displacement loop
*****************************************/
```

```
for(j=0;j<n_dof;j++)    /* outer loop start*/
        {
        for (i=0;i<n_dof;i++)
                if(i==j)
                {
                q_plus[i]=q_p[i]+.005;
                q_minus[i]=q_p[i]-.005;
                }
                else
                {
                q_plus[i]=q_p[i];
                q_minus[i]=q_p[i];
                };

result_plus=IdentityMatrix(4);
result_minus=IdentityMatrix(4);
potential_plus=0.0;
potential_minus=0.0;
for(i=0;i<3;i++)
        {
        end_point_plus[i]=0.0;
        end_point_minus[i]=0.0;
        }

/*find change in r for a  plus/minus permutation of q*/

        for(i=0;i<n_dof;i++)
                {
/*******************************************************
*   cost function for joint locations
*******************************************************/
                matrix_mult(result_plus,transformation_matrix(i, q_plus));
                matrix_mult(result_minus,transformation_matrix(i,q_minus));

                        for(kk=0;kk<3;kk++)
                                {
                        mid_point_plus[kk]=(result_plus[kk][3]+end_point_plus[kk])/2.0;
                        mid_point_minus[kk]=(result_minus[kk][3]+end_point_minus[kk])/2.0;
                                end_point_plus[kk]=result_plus[kk][3];
                                end_point_minus[kk]=result_minus[kk][3];
                                }
  /*****************************************************
  *    link endpoint collsion avoidance cost function
  *****************************************************/
                        for(k=0;k<n_obs;k++)
                                {
                                obstacle_transformation(k, ob,end_point_plus);
                                obstacle_transformation(k,ob,end_point_minus);
                                obstacle_transformation(k, ob,mid_point_plus);
                                obstacle_transformation(k,ob,mid_point_minus);

                                cost_plus=  -1.0;
                                cost_minus= -1.0;
                                for(kk=0;kk<3;kk++)
                                        {

                        cost_plus+=pow((end_point_plus[kk]-ob[kk][k])/(ob[kk+3][k]+6.0),ob[kk+6][k]);

cost_minus+=pow((end_point_minus[kk]-ob[kk][k])/(ob[kk+3][k]+6.0),ob[kk+6][k]);
                                        }
                                potential_plus+=1.0/cost_plus;
                                potential_minus+=1.0/cost_minus;
```

```
                        /*****************************************************
                        *    link midpoint collision avoidance cost function
                        *****************************************************/
                        cost_plus= -1.0;
                        cost_minus= -1.0;
                        for(kk=0;kk<3;kk++)
                                {
                    cost_plus+=pow((mid_point_plus[kk]-ob[kk][k])/(ob[kk+3][k]+3.0),ob[kk+6][k]);

cost_minus+=pow((mid_point_minus[kk]-ob[kk][k])/(ob[kk+3][k]+3.0),ob[kk+6][k]);
                                }
                        potential_plus+=1.0/cost_plus;
                        potential_minus+=1.0/cost_minus;
                        }
                }

        /*************************************************
        *     obstacle gradient vector
        *************************************************/
        gradient[j]=(potential_plus-potential_minus)/.01;
        gradient_mag+=gradient[j]*gradient[j];

        /*************************************************
        *     rate only  jacobian
        *************************************************/
        for(i=0;i<3;i++)
                for(k=0;k<4;k++)
                temp[i][j][k]=(result_plus[i][k]-result_minus[i][k])/.01;

} /* end virtual displacement loop */

/*************************************************
*     normalize  gradient vector
*************************************************/

gradient_mag=sqrt(gradient_mag);
for(i=0;i<n_dof;i++)
        gradient[i]=gradient[i]/gradient_mag;

for(i=0;i<n_dof;i++)
        for(j=0;j<n_dof;j++)
                {
                if(i==j)
                        mu_matrix[i][j]=1.0-gradient[i]*gradient[j];
                else
                        mu_matrix[i][j]=-gradient[i]*gradient[j];
                }

/*************************************************
*     obstacle avoidance  jacobian
*************************************************/

for(k=0;k<4;k++)
        {
        for(i=0;i<3;i++)
        for(j=0;j<n_dof;j++)
                {
                jacobian[i][j][k]=0.0;
                for(kk=0;kk<n_dof;kk++)
                jacobian[i][j][k]+=mu_matrix[j][kk]*temp[i][kk][k];
                }
        }
```

```
for(i=0;i<4;i++)
        {
        free (result_plus[i]);
        free (result_minus[i]);
        }
free (result_plus);
free (result_minus);

for(i=0;i<3;i++)
        {
        for(j=0;j<n_dof;j++)
        free(temp[i][j]);
                free(temp[i]);
        }
        free(temp);
free (q_plus);
free (q_minus);
free (gradient);

return(jacobian);
}
/***************************END EFFECTOR POSITION AND ERROR*****************/
void end_effector(int n, float*q_p, float target_p[3][4],float error_p[3][4], float err_mag[4])
{
int i,j;
float **result;

result=IdentityMatrix(4);                    /*initialize transformation matrix*/
        for(i=0;i<n;i++)                     /*carry out sequential matrix multiplication*/
        matrix_mult(result,transformation_matrix(i, q_p));
        printf("%.2f    %.2f     %.2f\n", result[0][3], result[1][3], result[2][3]);
/***************************************
*     determine end effector error
***************************************/

for(j=0;j<4;j++)
        {
        for(i=0;i<3;i++)
        error_p[i][j]=target_p[i][j]-result[i][j];

err_mag[j]=sqrt(error_p[0][j]*error_p[0][j]+error_p[1][j]*error_p[1][j]+error_p[2][j]*error_p[2][j]);
        }
for(i=0;i<4;i++)
        free(result[i]);
free(result);

}
/************************JOINT RATES*************************************/
float    *joint_rates(int  n,  float  *metric_p,  float    error_p[3][4],    float  target_p[3][4],
float***jb, float step_s)
{
float rate_mag=0;
float *rate_p,error_mag;

int i,j,k;
rate_p=mem_alloc_1(n);


error_mag=sqrt(error_p[0][3]*error_p[0][3]+error_p[1][3]*error_p[1][3]+error_p[2][3]*error_p[2][3]);

/***************************************************
*     target position approch rates
```

```
*************************************************/
for(i=0;i<n;i++)
        {
        rate_p[i]=0;
        for(j=0;j<3;j++)
                rate_p[i]+=metric_p[i]*jb[j][i][3]*error_p[j][3]/error_mag;
                rate_mag+=rate_p[i]*rate_p[i];
        }

/*************************************************
*     target  orientation rates
*************************************************/

for(i=0;i<n;i++)
        {
        for(j=0;j<3;j++)
                for(k=0;k<3;k++)
                rate_p[i]+=10*jb[j][i][k]*target_p[j][k];

        rate_mag+=rate_p[i]*rate_p[i];
        }

rate_mag=sqrt(rate_mag);

/*************************************************
*     rate limit  10 degrees /sec
*************************************************/

for(i=0;i<n;i++)
        {
        rate_p[i]=rate_p[i]/rate_mag;
        if (flag[i]==1 && fabs(rate_p[i])>.175*step_s)
        rate_p[i]=.175*step_s*rate_p[i]/sqrt(rate_p[i]*rate_p[i]);
        }
return(rate_p);
}

/*********************NUMERICAL INTEGRATION*****************/
void integrate(int n, float *var1, float *var2, float err, float err_dot, float max_step)
{
int i;
float step;
float step_s;

/* first order euler's method integration */
        step_s=max_step;
if(err>10*step_s)
        {
        if (err>fabs(err_dot))
                step=step_s;
        else
                step=fabs(err/err_dot)*step_s;
        }
 else
        step=.25*step_s;

 printf("%.2f \n",step);

 for(i=0;i<n; i++)
        var2[i]+=var1[i]*step;
 }
```

```
/************************PRINT OUTPUT************************************/
void joints_print(int n, float *var,FILE *file)
{
int i;
for(i=0;i<n/2;i++)
fprintf(file, "%f  ", var[i]);
fprintf(file, "\n");

for(i=n/2;i<n;i++)
fprintf(file, "%f  ", var[i]);
fprintf(file, "\n");
}
/*************************MEMORY ALLOCATION 1*************************************/

float  *mem_alloc_1(int n)
{
float *var;

if((var = (float *)malloc(sizeof(float)*n)) == (float *) NULL)
{
       fprintf(stderr, "mallocing  error\n");
       exit(-1);
}
return(var);
}
/*************************MEMORY ALLOCATION 2*************************************/

float  **mem_alloc_2(int nrows,int ncols)
{
float **var;
int i;
if((var=(float **)malloc(sizeof(float*)*nrows)) == (float **) NULL)
       {
       fprintf(stderr, " mallocing error\n");
       exit(-1);
       }
for(i=0;i<nrows;i++)
       {
       if((var[i]=(float*)malloc(sizeof(float)*ncols)) == (float *) NULL)
              {
              fprintf(stderr, " mallocing error\n");
              exit(-1);
              }
       }
return(var);
}
/*****************************OBSTACLE ORIENTATION TRANSFORMATION*********/
void   **obstacle_transformation(int n, float **obst,  float  vector_n[3])
{

float  pry[3][3];
float  vector_r[3];
int i,j;

pry[0][0]=cos(obst[11][n])*cos(obst[10][n]);
pry[0][1]=cos(obst[11][n])*sin(obst[10][n])*sin(obst[9][n])-sin(obst[11][n])*cos(obst[9][n]);
pry[0][2]=cos(obst[11][n])*sin(obst[10][n])*cos(obst[9][n])+sin(obst[11][n])*sin(obst[9][n]);

pry[1][0]=sin(obst[11][n])*cos(obst[10][n]);
pry[1][1]=sin(obst[11][n])*sin(obst[10][n])*sin(obst[9][n])+cos(obst[11][n])*cos(obst[9][n]);
pry[1][2]=sin(obst[11][n])*sin(obst[10][n])*cos(obst[9][n])-cos(obst[11][n])*sin(obst[9][n]);
```

```
pry[2][0]=-sin(obst[10][n]);
pry[2][1]=cos(obst[10][n])*sin(obst[9][n]);
pry[2][2]=cos(obst[10][n])*cos(obst[9][n]);

for(i=0;i<3;i++)
        {
        vector_r[i]=0;
        for(j=0;j<3;j++)
        vector_r[i]+=pry[i][j]*vector_n[j];
        }
for(i=0;i<3;i++)
        vector_n[i]=vector_r[i];
}
```

References

[1] Richardson, B., Sklar, M., and Fresa, M., "PCR Inspection and Processing Robot Study, Final Report", McDonnell Douglas Space Systems - Kennedy Space Division, Nov 5 1993

[2] Pasch, K., "Self-Contained Deployable Serpentine Truss for Prelaunch Access of the Space Shuttle Orbiter Payloads", NAS -2659-FM-9106-387, Final Report, Contract No. NAS 10-11659, NASA, Kennedy Space Center, FL Aug. 1990.

[3] Maciejewski, A., and Klein, C., "Obstacle Avoidance for Kinematically Redundant Manipulators in Dynamically Varying Environments", *The International Journal for Robotics Research*, Vol. 4, No. 3, Fall 1985, pp. 109-117.

[4] Nakamura, Y., *Advanced Robotics, Redundancy and Optimization*, Addison-Wesley Publishing Co., Inc., Redwood City, C, 1991.

[5] Wegerif, D., Rosinski, D., and Parton, W., "Results of Proximity Sensing Research for Real-Time Collision Avoidance of Articulated Robots Working Near the Space Shuttle", *Proceedings of the 6th Annual Conference on Recent Advances in Robotics*, University of Florida, Gainesville, FL, 19-20 April 1993.

[6] Penrose, R., "On the Best Approximate Solutions of Linear Matrix Equations", *Proceedings, Cambridge Philosophy Society*, 52:17-19.

[7] Sciavicco, L., Siciliano, B., "A Solution Algorithm to the Inverse Kinematic Problem for a Redundant Manipulator", *IEEE Journal of Robotics and Automation*, Vol. 4., No. 4., Aug. 1988, pp. 403-410.

[8] Asano, K., et al, "Multi-Joint Inspection Robot", *IEEE Transactions on Industrial Electronics*, Vol. IE-30, No. 3, August 1983, pp. 277-281.

[9] Baker, D.R., and Wampler, C.W., "On the Inverse Kinematics of Redundant Manipulators", *The International Journal of Robotics Research*, Vol. 7., No. 2., March/April 1988.

[10] Doty K., Melchiorri, C., and Bonivento, C., "A Theory of Generalized Inverses Applied to Robotics", *The International Journal of Robotics Research*, Vol. 12, No. 1, Feb. 1993, pp 1-19.

[11] Byers, R., " Control of a Serpentine Manipulator with Collision Avoidance" Final Report, 1993 NASA/ASEE Summer Faculty Fellowship Program, NASA CR-194678, Contract NGT 60002 Suppl. #11, Kennedy Space Center, FL.

[12] "IGRIP Users Reference Manual", Deneb Robotics, Inc. 1990.

[13] Denavit, J. and Hartenberg, R.S., "A Kinematic Notation for Lower–Pair Mechanixms Based on Matrices," *Journal of Applied Mechanics*, pp. 215–221, June 1955.

[14] Khatib, O., and Le Maitre, J.-F., "Dynamic Control of Manipulators Operating in a Complex Environment", *3rd Symp. Theory and Practice of Robot Manipulators*, Elsevier, pp. 267-282.

REFERENCES

[15] Luenberger, D., *Optimization by Vector Space Methods*, John Wiley & Sons, Inc., New York, 1969.

[16] Nakamura, Y., and Hanafusa, H., "Optimal Redundancy Control of Robot Manipulators", *International Journal of Robotics Research*, vol. 6, No. 1., Spring 1987. pp 32-42.