ornl

ORNL/TM-12816

OAK RIDGE NATIONAL LABORATORY

MARTIN MARIETTA

Enhanced Code for the Full Space Parameterization Approach to Solving Underspecified Systems of Algebraic Equations

Version 1.0

Kristi A. Morgansen
François G. Pin

MANAGED BY
MARTIN MARIETTA ENERGY SYSTEMS, INC.
FOR THE UNITED STATES
DEPARTMENT OF ENERGY

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

ORNL/TM-12816

Robotics and Process Systems Division

ENHANCED CODE FOR THE FULL SPACE PARAMETERIZATION APPROACH TO SOLVING UNDERSPECIFIED SYSTEMS OF ALGEBRAIC EQUATIONS VERSION 1.0

Kristi A. Morgansen and François G. Pin

DATE PUBLISHED – March 1995

Prepared by the
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831
managed by
MARTIN MARIETTA ENERGY SYSTEMS, INC.
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-84OR21400

Contents

List of Figures

ABSTRACT

This paper describes an enhanced version of the code for the Full Space Parameterization (FSP) method that has recently been presented for determining optimized (and possibly constrained) solutions, $\mathbf{x}$, to underspecified systems of algebraic equations $\mathbf{b} = \mathbf{A}\mathbf{x}$. The enhanced code uses the conditions necessary for linear independence of the $m - n + 1$ vectors forming the solution as a basis for an efficient search pattern to quickly find the full set of solution vectors. A discussion is made of the complications which may be present due to the particular combination of the matrix $\mathbf{A}$ and the vector $\mathbf{b}$. The first part of the code implements the various methods needed to handle these particular cases before the solution vectors are calculated so that computation time may be decreased. The second portion of the code implements methods which can be used to calculate the necessary solution vectors. The respective expressions of the full solution space, $\mathbf{S}$, for the cases of the matrix $\mathbf{A}$ being full rank and rank deficient are given. Finally, examples of the resolution of particular cases are provided, and a sample application to the joint motion of a mobile manipulator for a given end-effector trajectory is presented.

1 INTRODUCTION

A common problem in mathematics and engineering which has been the focus of attention in the past few years is determining an inverse solution for

$$\mathbf{b} = \mathbf{A}\mathbf{x} \tag{1}$$

when $\mathbf{A}$ has fewer rows than columns. Since the system in Eq. 1 is under-specified, there will be a possible infinity of solutions. In order to choose among these solutions, an optimization approach is generally chosen. In many cases, this optimization must also take into account constraints on the system. If $\mathbf{A}$ has n rows and m columns, then typically $m - n$ constraints (and in some cases more) may be applied to the system without preventing the determination of a satisfactory solution.

The method most commonly used for solving Eq. 1 is the Moore-Penrose pseudo-inverse [12] which gives the solution for the least-norm of $\mathbf{x}$. An extension of this method is the Gradient Projection in which a single cost function, $\mathbf{Z}(\mathbf{x})$, is projected onto the null space of $\mathbf{A}$ according to

$$\mathbf{x} = \mathbf{A}^{\dagger}\mathbf{b} + \left[\mathbf{I} - \mathbf{A}^{\dagger}\mathbf{A}\right]\dot{\mathbf{Z}} \tag{2}$$

where $\mathbf{A}^{\dagger}$ is the pseudo-inverse $\mathbf{A}^{\dagger} = \mathbf{A}^{T}(\mathbf{A}\mathbf{A}^{T})^{-1}$. Some examples of applications for the gradient-projection method have been obstacle avoidance [11], manipulability [13], and maximization of criteria for a seven degree-of-freedom robot in [2]. Until now the only other unique approach to solving Eq. 1 for $\mathbf{x}$ has been to create a square matrix, $\mathbf{A}_s$, of size $m \times m$ which can then be used in

$$\mathbf{x} = \mathbf{A}_s^{-1}\mathbf{b} \tag{3}$$

to find the desired vector $\mathbf{x}$. Baillieul [1] suggests adding $m - n$ rows to $\mathbf{A}$ using vectors corresponding to the components of the gradient of a constraint equation. The derivatives are taken in the directions of a set of independent vectors spanning the null space of $\mathbf{A}$. A more general approach to this type of method is task space augmentation [9] in which a set of $m - n$ linearly independent functions of $\mathbf{x}$ are appended to $\mathbf{A}$. These functions can be chosen so that the system meets either physical constraints (e.g. see [5]) or follows user-specified time-dependent functions (e.g. see [10]). Obstacle avoidance using task space augmentation has been demonstrated by Sciavicco and Siciliano in [9]. One of the key drawbacks with the methods presented above occurs when more than one optimization criterion and/or constraint is to be applied: extending the particular solution optimization method with homogeneous solutions will not necessarily produce the optimal solution for the combination of the optimization criteria and constraints. A typical example of this "task prioritization" problem is that a least norm solution which is also to minimize torque will not actually produce the lowest torque with smallest change in joint angles. Part of the reason

for this difficulty is that the single-criterion/constraint methods were developed to simply produce a desired solution without examining the entire range of possibilities.

Recently a new method, Full Space Parameterization (FSP) [6, 7], has been developed that produces, in parameterized form, the entire space of possible solutions to the problem shown in Eq. 1. A more thorough treatment and proof of the results which will be shown in this article can be found in [6, 7]. The key aspect of FSP is that any vector $\mathbf{b}$ and matrix $\mathbf{A}$ which has n rows and m columns, where $n \leq m$, have a solution space which can be constructed as a hyperplane of a space spanned by a set of (typically $m-n+1$) linearly independent solution vectors $\mathbf{g}^k$, $k = 1 \ldots m-n+1$. These vectors are easily found solutions of submatrices of $\mathbf{A}$. The results of the proofs in [6, 7] will be used to find a pattern to aid in determining which columns of the matrix $\mathbf{A}$ should be blocked in order to quickly find the submatrices of $\mathbf{A}$ that lead to $m-n+1$ linearly independent solution vectors.

The FSP solution code and related algorithms presented in the remainder of the article will be demonstrated through application to the problem of redundancy resolution in robotic systems. The kinematic equations for a robotic system are typically described by the equation

$$\mathbf{X} = \mathbf{F}(\mathbf{q}) \tag{4}$$

in which $\mathbf{X}$ represents the position and orientation of some point in the system, and $\mathbf{q}$ represents the joint orientations for the system. Since joint displacements are necessary to control the system, we are more interested in the time derivative

$$\dot{\mathbf{X}} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}} \tag{5}$$

where $\mathbf{J}$ is the system Jacobian with components $\mathbf{J}_{ij} = \partial \mathbf{F}_i / \partial \mathbf{q}_j$. Usually, Eq. 5 is highly nonlinear, so a first-order linearized version is generally utilized for control of the system. The linear discretized form of Eq. 5 is

$$\frac{\Delta \mathbf{X}}{\Delta t} \approx \mathbf{J}_{\Delta t} \frac{\Delta \mathbf{q}}{\Delta t} \tag{6}$$

where $\mathbf{J}_{\Delta t}$ is the Jacobian assumed constant over the time step Δt. At each time step in the trajectory, Eq. 6 is solved by treating the system as a set of algebraic equations. When the Jacobian, $\mathbf{J}$, has more columns than rows ($n < m$), the system is said to be redundant and typically is solved using an optimization method. Of course, constraints may be included in the optimization method as appropriate.

The next section discusses how the proofs from [6, 7] are used to find a pattern among the $m-n+1$ solution vectors. The third section of the article contains a discussion of difficulties which may occur due to possible combinations of matrix $\mathbf{A}$ and vector $\mathbf{b}$ configurations. Section four presents an algorithm for simplifying the matrix $\mathbf{A}$ in order to decrease computation time. Section five presents the main algorithm for computing the vectors used to parameterize the entire space of solutions for a given $\mathbf{A}$ and $\mathbf{b}$. The sixth section gives some examples of the FSP method

applied to particular cases of matrix **A** and vector **b** configurations and to the joint motion planning problem for a mobile manipulator system. Concluding remarks will be presented in section seven, and a listing of the FSP code together with a brief user's guide are given in the appendices.

2 CONDITIONS FOR EXISTENCE AND INDEPENDENCE OF SOLUTION VECTORS

The first step in finding $m-n+1$ vectors spanning the solution space of $\mathbf{A}^{-1}\mathbf{b}$ is to find an initial square ($n \times n$ if $\mathbf{A}$ is of rank n) invertible submatrix $\mathbf{A}_1$ of $\mathbf{A}$. Once this submatrix has been found, $\mathbf{A}$ is reordered so that the n columns forming the submatrix are numbered $1 \ldots n$, and the remaining columns are numbered $n+1 \ldots m$. The first solution vector is found by inverting the submatrix formed from the first n columns, multiplying $\mathbf{b}$ by the result, and adding zeros to the components corresponding to columns $n+1 \ldots m$. An important point to note at this time is that although reordering the columns does not affect the invertibility of a submatrix (and is important for quickly finding the complete set of solution vectors), the positions of components in the solution vectors must correspond to the original locations of the columns of $\mathbf{A}$.

With the columns of $\mathbf{A}$ rearranged as specified, each of the $m-n$ columns not in the first invertible submatrix will be linearly dependent upon some combination of the first n columns. To find the second solution vector, a new submatrix of $\mathbf{A}$ is formed by replacing one of the n columns in the first submatrix by column $n+1$. Since the original $\mathbf{A}$ has at most rank n, column $n+1$ must be dependent on some combination of c_i, $i \in \{1 \ldots n\}$. Whichever column c_i in $\mathbf{A}_1$ is replaced by c_{n+1} must be part of the linear decomposition of c_{n+1}. We can then write

$$c_{n+1} = \sum \alpha_j c_j \qquad j \in \{1 \ldots, h, \ldots n\} \qquad \alpha_j \neq 0 \tag{7}$$

This second submatrix is guaranteed to be invertible as long as the column which is replaced by $n+1$ is in the linear decomposition of $n+1$ since the linear combination of columns will not be complete without both columns $n+1$ and the replaced column (see [6] or [7]). Because the replaced column is linearly dependent on c_{n+1} and possibly some columns which were not replaced, c_{n+2} is guaranteed to be linearly dependent on some combination of the columns in this second invertible submatrix. Following the method used to find the second submatrix, submatrices three through $m-n+1$ can be found by systematically substituting each of the remaining columns (3 through $m-n+1$) for one in the previous submatrix. The resulting pattern of blocked and unblocked columns will then resemble the following diagram.

$$\begin{array}{|cccccccccc|}
\hline
X & X & X & \cdots & X & \otimes & \otimes & \otimes & \cdots & \otimes \\
X & X & \otimes & \cdots & X & X & \otimes & \otimes & \cdots & \otimes \\
\otimes & X & \otimes & \cdots & X & X & X & \otimes & \cdots & \vdots \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
\otimes & X & \otimes & \cdots & \otimes & X & X & X & \cdots & \otimes \\
\otimes & X & \otimes & \cdots & \otimes & X & \otimes & X & \cdots & X \\
\hline
\end{array}$$

where each row corresponds to the blocking pattern for a single solution, X marks a column which is in the square submatrix of $\mathbf{A}$, and $\otimes$ marks a column which is not in the submatrix.

Now we would like to know the conditions for the linear independence of the solution vectors $\mathbf{g}^k$, $k = 1 \ldots m-n+1$. To determine conditions for $\mathbf{g}^1$ and $\mathbf{g}^2$ to be linearly independent, look at the equations

$$\mathbf{b} = \mathbf{g}_i^{1*} c_i \qquad i \in \{1 \ldots n\} \tag{8}$$

$$\mathbf{b} = \mathbf{g}_i^{2*} c_i + \mathbf{g}_{n+1}^{2*} c_{n+1} \qquad i \in \{1 \ldots n\} - \{h\} \tag{9}$$

where $\mathbf{g}_i^{1*}$ are the solution components of $\mathbf{A}_1^{-1}\mathbf{b}$, and $\mathbf{g}_i^{2*}$ are the solution components of $\mathbf{A}_2^{-1}\mathbf{b}$. $\mathbf{g}^l$ and $\mathbf{g}^{l*}$ differ in that $\mathbf{g}^l$ contains the zero components corresponding to the $m-n$ columns blocked in order to form $\mathbf{A}_l$. The index h corresponds to the column of $\mathbf{A}_1$ that was replaced by c_{n+1}. Now subtract Eq. 8 from Eq. 9:

$$\mathbf{0} = (\mathbf{g}_i^{1*} - \mathbf{g}_i^{2*}) c_i + \mathbf{g}_h^{1*} c_h - \mathbf{g}_{n+1}^{2*} c_{n+1} \qquad i \in \{1 \ldots n\} - \{h\} \tag{10}$$

Several cases exist for which this equality is satisfied. For simplification rename the quantity $(\mathbf{g}_i^{1*} - \mathbf{g}_i^{2*})$ as β_i. Then, consider the case where every β_i is 0. Eq. 10 reduces to

$$\mathbf{g}_h^{1*} c_h - \mathbf{g}_{n+1}^{2*} c_{n+1} = \mathbf{0} \tag{11}$$

If $\mathbf{g}_h^{1*} = 0$, then the only solution is $\mathbf{g}_{n+1}^{2*} = 0$. Since each β_i is zero, solutions $\mathbf{g}^1$ and $\mathbf{g}^2$ are identical. Such a pair of vectors is not acceptable, so another choice of submatrices must be made for which $\mathbf{g}_h^{1*} \neq 0$. Next, consider again that all β_i are 0, and $\mathbf{g}_h^{1*}$ in Eq. 11 is not zero. The only solution for Eq. 11 will be that the decomposition of c_{n+1} contains only the column c_h which implies $\mathbf{g}_{n+1}^2 \neq 0$. In this case $\mathbf{g}^1$ and $\mathbf{g}^2$ are independent. Finally, consider that at least one βi is nonzero. If $\mathbf{g}_h^1 = 0$, then the only solution to Eq. 10 is if c_{n+1} is dependent on a combination of the columns c_i not including c_h. This violates Eq. 7, hence $\mathbf{g}_h^1$ will not be zero. So to satisfy linear independence we are left with the case that at least one of the β_i is nonzero and $\mathbf{g}_h^1$ is nonzero. Let's rewrite Eq. 10 and include $\mathbf{g}_h^1 - 0$ in the β_i:

$$c_i \beta_i - \mathbf{g}_{n+1}^2 c_{n+1} = 0 \tag{12}$$

Since $\mathbf{A}_1$ is nonsingular, $c_i \beta_i \neq 0$, $i = \{1 \ldots n\}$ implying $\mathbf{g}_{n+1}^2 \neq 0$. In summary, if two vectors $\mathbf{g}^1$ and $\mathbf{g}^2$ are to be linearly independent, then the component of $\mathbf{g}^1$ corresponding to c_h (the column being replaced) must be nonzero, and consequently the component of $\mathbf{g}^2$ corresponding to c_{n+1} is guaranteed to be nonzero. Linear independence of the remaining vectors $\mathbf{g}^3$ through $\mathbf{g}^{m-n+1}$ follows similarly.

3 CLASSIFICATION OF REDUNDANCY

Under certain conditions, combinations of **A** and **b** will form a system that cannot be immediately solved with standard methods. A well-known example is when the matrix **A** has two or more dependent rows. A less-known example is the possible restriction of **x** components due to the specific combination of **A** and **b**. One of the important features of the FSP method is that it is capable of dealing with such cases quite easily. Before any difficulties can be handled, the key features of each set of **A** and **b** leading to complications in solution determination should be identified.

The types of cases that may be encountered will be grouped into two categories. In the first category, difficulties will be encountered due to the loss of row rank in **A**. The second category contains systems where the problems arise due to the specific combination of the matrix **A** and the vector **b**. In the latter case, the dimension of the solution space will be at most $m - n + 1$. Depending on the specifics of the combination of **A** and **b**, however, the exact calculation method used to find vectors forming the space will differ.

To recognize when rows of **A** are dependent, methods such as Singular Value Decomposition (SVD) can be used to identify which rows (if any) form the nullspace. If the nullspace of the rows is not empty (i.e. two or more rows are linearly dependent), then a constraint must be used when finding the final solution using the FSP method. Since invertible square submatrices of rank n cannot be found when **A** is rank deficient, one of the dependent rows must be eliminated from **A** during the calculation of the vectors forming the solution space of the system. Eliminating rows of **A** will, however, require that additional vectors be found to span the space. For example, if two rows of **A** are dependent and one is eliminated in order to calculate the solution space, a total of $m - n + 2$ rather than $m - n + 1$ independent vectors must be found since the reduced **A** matrix is $(n-1) \times m$ rather than $n \times m$. The change in dimension of the solution space due to the addition of vectors to the space will be resolved by the use of the mentioned constraints.

In order to find the solution space for a case in which the particular combination of **A** and **b** has caused a complication, the exact problem must be found. To reach this end, the idea of restricted vector components will be introduced. In the case of a standard well-conditioned **A** and **b** set, each component of **b** can be produced by contributions from at least two separate components in the corresponding row of **A**. Such a **b** component will be referred to as unrestricted. On the other hand, a restricted component occurs when a row of **A** has only one nonzero value. When a restricted component occurs, the element of $\mathbf{g}^k$ will be identical in each of the $k = 1 \ldots m-n+1$ solution vectors. For example, if $\mathbf{A}_{ij} \neq 0$ for only one j, then $\mathbf{g}_j^k = \mathbf{b}_i / \mathbf{A}_{ij}$. This row, i, is then removed from the matrix, the column j is back-substituted, removed from the matrix, and the search pattern is repeated for the remaining $(n-1) \times (m-1)$ matrix. In some cases fixing the value of $\mathbf{g}^k$ corresponding to one column may reveal another element of **b** as being restricted in the remaining $(n-1) \times (m-1)$ matrix.

The corresponding row must in turn be handled as previously explained. Hence, the matrix **A** and vector **b** must be examined carefully to determine if, and how many, restrictions exist.

A particular type of restriction which may occur is that all nonzero elements of **b** are restricted. Since the unrestricted **b** components are zero, all combinations of the columns of **A** which do not correspond to the restricted columns must form null space solutions. The algorithm which is used to deal with such a case will be discussed in the next section.

The last case to be identified is when either rows or columns of **A** are composed completely of elements that are substantially smaller than the other values in the matrix. Columns of small values imply that the corresponding component of the solution is almost inactive in the system (in robotic terms, that joint does not contribute to the motion of the end effector). The case of a row of small elements may be dealt with in the same manner as **A** having dependent rows, but it is much easier to simply eliminate that row and the corresponding element of **b** (assuming that it is also zero) and solve the resulting problem (which will require $m - n + 2$ vectors rather than $m - n + 1$).

One final check that should be made is that the vector **b** is actually in the range of the matrix **A**. Methods do exist which can be used to make this check before any attempt is made to calculate solution vectors. However, in typical robotics path planning applications (for which the code was originally written), **b** can be assumed to always be in the range of **A**. Based upon this assumption, the current version of the code does not check to see if **b** is actually in the range of **A**.

4 REDUCTION OF A

For any rectangular matrix **A** which has fewer rows than columns, the total number of unique square submatrices possible will be C_m^{m-n}. If the inverse solution to Eq. 1 is to be found using a computer, we would like to minimize the computation time as much as possible. One way to do this is to decrease the total number of submatrices of **A** which need to be searched. Whenever a set **b** and **A** have a restriction, we know that one element must be fixed in all of the solution vectors. Hence, including the restricted **b** element and **A** row in the solution calculation algorithm will be unnecessary. If, when a restriction is found, the element $\mathbf{b}_i$ and row $\mathbf{A}_i$ are removed from the system, then the total number of square submatrices which exist is reduced to C_{m-1}^{m-n}, and calculation time will decrease accordingly. For example, if **A** originally has six rows and ten columns a total number of 210 submatrices exist. However, if **A** has two restrictions, then it can be reduced to a matrix having four rows and eight columns and a total of only 70 submatrices. Preprocessing restricted elements thus eliminates the need to examine unnecessary submatrices. Once the vectors spanning the solution space have been found from a reduced matrix, the fixed elements of each $\mathbf{g}^k$ are included in to form the final, correct solution **x**.

The flow chart in Fig. 1 shows the algorithm that is currently being used to reduce the size of the matrix **A**. The main part of the reduction algorithm is dedicated to

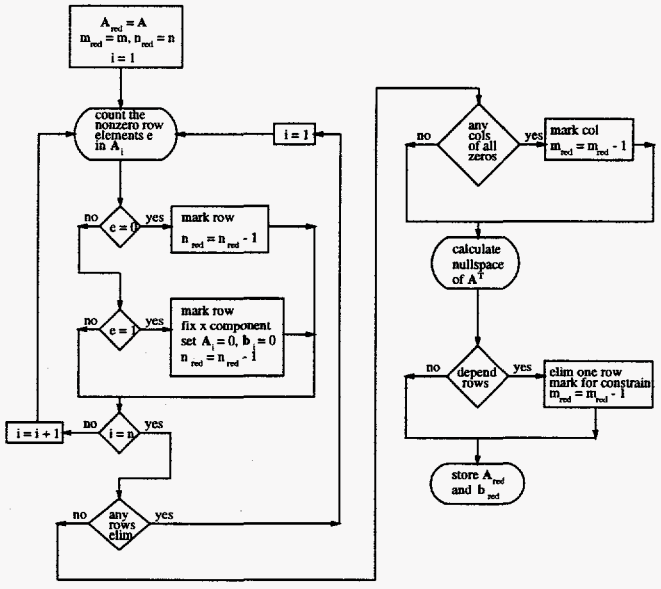


Figure 1: Flow chart for reduction of **A**.

eliminating the restricted elements of **A**. Starting with the first row, the number of nonzero elements are counted in each of the n rows of **A**. Any rows that have only zero elements are marked so that the row dimension of the system will be decreased in the final reduced matrix. If only one nonzero element is found, then the corresponding

element of $\mathbf{b}$ is restricted. The appropriate element of $\mathbf{x}$ is calculated, then the corresponding column is multiplied by this value and back-substituted to create a modified $\mathbf{b}$ vector. After back-substitution, all values in the column are set to zero (to enable the computer to recognize restricted elements in other rows), and a record is made to track which $\mathbf{x}$ values correspond to restricted $\mathbf{A}$ values. Since the number of nonzero elements in some rows will decrease when column elements are set to zero, the search must be repeated until an entire cycle from first through last rows produces no restricted elements.

Next, the system is searched for any columns of zeros that did not occur from eliminated restrictions. Because of machine round-off errors, a threshold must be set to determine how small a number must be to be considered zero. This number is determined in the procedure which reduces the $\mathbf{A}$ matrix before any other calculations are made. To set a value for the threshold value, a search is made to find the largest magnitude value in $\mathbf{A}$. This value is then divided by 1000 to set the zero threshold. Another problem which needs to be dealt with in the future, but which has not been addressed in this code, is the presence of columns of matrix elements which are all substantially larger than the other elements of the matrix $\mathbf{A}$. Columns which have values that are all much larger than other values in the matrix will not generate invertible square submatrices, since the maximum singular value of such square submatrices would be more than ten times as large as the smallest singular value causing the condition number larger than the acceptable maximum.

The final step in the reduction algorithm is to determine whether any rows of the matrix $\mathbf{A}$ are linearly dependent. The row dependencies can be checked by calculating the nullspace of the matrix $\mathbf{A}^T$. A simple method for finding nullspace vectors without needing to invert any matrices is the Singular Value Decomposition (SVD) method. When a matrix is decomposed using SVD, it is rewritten in the following form

$$\mathbf{A} = \mathbf{USV}^T \tag{13}$$

where $\mathbf{S}$ contains the singular values of the system, and $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices. The nonzero elements of the nullspace vectors of the columns of the matrix $\mathbf{A}$ correspond to those columns of the matrix $\mathbf{V}$ which produce zero singular values.

When using a computer to determine nullspace vectors with this type of method, two difficulties must be recognized and avoided. First, as mentioned in the previous paragraph, machine round-off errors require that a zero threshold value be set. For the examples to be given in section six, the singular value threshold value was set at 1.0×10^{-5}. Also, consideration must be given to the overall condition number of the matrix. The condition number is the ratio of the largest to smallest singular values. If this ratio is excessively large, then inclusion of the column corresponding to the smallest singular value causes the matrix to be nearly singular. Hence for matrices with large condition numbers, the smallest singular value should be regarded as zero, and the appropriate column of $\mathbf{V}^T$ should be considered a nullspace vector. In the current version of the code which calculates the nullspace vectors, the inverse of the

condition number (i.e. ratio of smallest to largest singular values) is used to determine whether or not the smallest singular value should be considered to be zero. In the examples, the maximum allowable condition number has been set to a value of 10.

5 SOLUTION SPACE CALCULATION

5.1 Solution space parameterization

The possible cases of combinations of **A** matrices and **b** vectors that will be encountered after matrix reduction can be placed in three categories: completely restricted, completely unrestricted, and loss of row rank. Cases in which **A** originally had both restricted and unrestricted components in combination with **b** will have been reduced to one of these three cases through the algorithm given in the previous section. We would now like to know how to create the necessary $m - n + 1$ vectors (where m and n now respectively refer to the number of columns and rows of the reduced system) to generate the solution space.

First, consider the case where the system is completely restricted. After eliminating all of the restrictions, we are left with a vector **b** which will be composed completely of zeros and a rectangular **A** matrix with decreased dimensions. In this situation, where **b** is zero, we do not need to try to find invertible square submatrices since any inverse multiplied by **b** would always be zero. As shown in the previous section, the SVD provides a direct method for finding nullspace vectors without needing to invert any matrices:

$$\mathbf{A} = \mathbf{USV}^T \tag{14}$$

So for a rectangular matrix of rank n which has n rows and m columns where $n < m$, we know that the SVD of the matrix **A** will produce a nullspace that must contain $m - n$ linearly independent vectors. The final vector needed to form a set of $m - n + 1$ solution vectors will be left as all zeros until the restricted elements are included. When the fixed **x** components are inserted into these $m - n + 1$ vectors, we will have $m - n + 1$ linearly independent vectors.

The next case, and the one which will occur most often, is where the reduced matrix **A** is completely unrestricted. The algorithm used to find the necessary solution vectors will be given later in this section. As long as the system has not lost row rank, the solution space of the system, **S**, is given by:

$$\mathbf{S} = \left\{ \quad \mathbf{x} \quad | \quad \mathbf{x} = \sum_{i=1}^{m-n+1} t_i \mathbf{g}^i, \quad \sum_{k=1}^{m-n+1} t_k = 1 \right\} \tag{15}$$

The final case to be dealt with is that in which the matrix **A** has dependent rows. In this situation, one of the dependent rows is eliminated from **A** before the solution vectors are calculated. Eliminating a row increases the total number of vectors to be found by one. However, the eliminated row constitutes a constraint that must be kept in the system and creates a solution space of the appropriate dimension. The $m - n + 2$ vectors will be found as described in the method below. With the additional constraint applied to restrict it, the solution space, **S**, is given by:

$$\mathbf{S} = \left\{ \quad \mathbf{x} \quad | \quad \mathbf{x} = \sum_{i=1}^{m-n+2} t_i \mathbf{g}^i, \quad \sum_{k=1}^{m-n+2} t_k = 1, \quad \sum_{i=1}^{m-n+2} \beta_i t_i = 1 \right\}. \tag{16}$$

To express this constraint for a system with dependent rows, first write the row that was eliminated in the solution-finding algorithm as

$$\mathbf{b}_j = \mathbf{A}_j \mathbf{x}. \tag{17}$$

The constraint for a dependent row is then

$$\sum_{i=1}^{m-n+2} \left(\frac{\mathbf{A}_j \mathbf{g}^i}{\mathbf{b}_j} \right) t_i = 1 \tag{18}$$

so that in Eq. 16

$$\beta_i = \frac{\mathbf{A}_j \mathbf{g}^i}{\mathbf{b}_j}. \tag{19}$$

Now, for the specific constraint necessary for a system where $\mathbf{A}$ has lost row rank, we must solve the appropriate equations given in [6, 7]. In these equations, $\mathbf{Z}_r = 0$ and $\mathbf{H} = 0$. Since we only have one constraint, the equations reduce to:

$$a = \mathbf{e}^T \mathbf{G}^{-1} \mathbf{e} \tag{20}$$

$$b_1 = \mathbf{e}^T \mathbf{G}^{-1} \beta^1 \tag{21}$$

$$c_1 = \beta^{1T} \mathbf{G}^{-1} \mathbf{e} \tag{22}$$

$$d_{11} = \beta^{1T} \mathbf{G}^{-1} \beta^1 \tag{23}$$

$$A_{11} = c_1 b_1 - a d_{11} \tag{24}$$

$$\nu = A_{11}^{-1} (a - c_1) \tag{25}$$

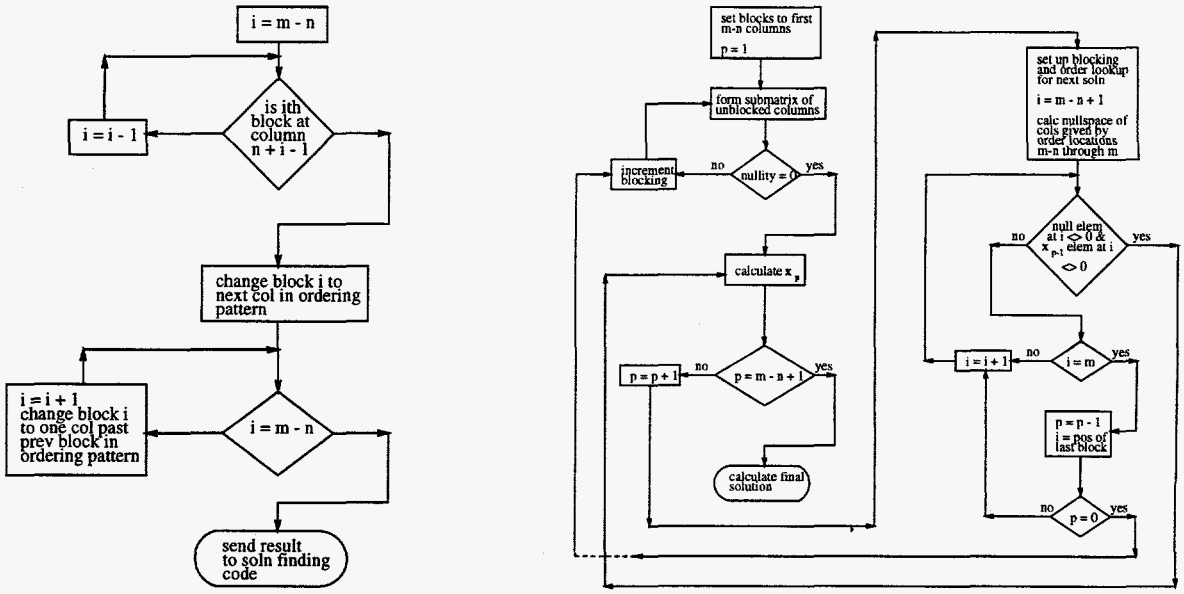
$$\mu = -\left(\frac{1 + \nu^T b_1}{a} \right) \tag{26}$$

$$\mathbf{t}^* = -\mu \mathbf{G}^{-1} \nu \beta \tag{27}$$

where $\mathbf{G}$ has components $G_{ij} = \mathbf{g}^{iT} \mathbf{g}^j$. a, b_1, c_1, d_{11}, A_{11}, ν, and μ are all scalars, and $\mathbf{t}^*$ is a vector of scalars.

5.2 Solution component vectors calculation

The flow chart for the algorithm to find a set of linearly independent solution vectors for an unrestricted matrix $\mathbf{A}$ is shown in Fig. 2. The algorithm is initialized by blocking the first $m - n$ columns of $\mathbf{A}$, and then iterating through choices of blocked columns until a first invertible submatrix is found. A blocked column (or block) is a column, c_i, which is not included in the $n \times n$ submatrix of $\mathbf{A}$ and whose corresponding solution component $\mathbf{g}_i$ is set to zero. The columns are next reordered (keeping a record of the order) so that the n columns forming the first submatrix are the first n columns in the matrix. This matrix is used to solve for the solution vector,



Incrementation algorithm Solution-finding algorithm

Figure 2: Flow chart for creation of $m - n + 1$ solution vectors.

$\mathbf{g}^1$. The remaining $m - n$ vectors are found using the pattern discussed in section 2. This pattern shows that each vector shares $m - n - 1$ of its blocked columns with the previous vector. The block which is not shared corresponds to whichever of the last $m - n$ columns will be used to create the next submatrix and solution vector. For example, if a solution vector has the following form:

$$\begin{array}{cccccccccccc} & & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & \\ \mathbf{g}^3 & = & [& X & X & \otimes & \otimes & X & X & X & X & \otimes & \otimes &]^T \end{array}$$

Then the following vector, $\mathbf{g}^{\triangleq}$, would have columns blocked as follows:

$$\begin{array}{cccccccccccc} & & & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & \\ \mathbf{g}^{\triangleq} & = & [& & & \otimes & \otimes & & & & & X & \otimes &]^T \end{array}$$

where column 9 is being used to form a new submatrix for $\mathbf{g}^{\triangleq}$ so that block is not shared with $\mathbf{g}^3$. The missing block will replace one of the unmarked columns which is in its linear decomposition and for which the corresponding component of the previous solution is nonzero. In the current version of the code, a square submatrix of $\mathbf{A}$ is formed from the n vectors in the previous invertible submatrix along with the next column to be added in. SVD is used to calculate the nullspace vector of these $n + 1$ columns, and this vector is used to determine which of the columns in the previous submatrix are in the linear decomposition of the new column and may be replaced. If none of the options for the final block in a given solution is satisfactory, then the blocking in the previous solution must be changed. In some cases it may be required

for the algorithm to return to the first solution and find a completely new blocking configuration for the entire set. However, as long as the matrix is of rank n, the total $n - m + 1$ vectors will be found to fit the given pattern.

6 EXAMPLES: THE MOBILE MANIPULATOR

This section gives examples of the different cases discussed in section 3 as applied to the system of a mobile manipulator which has a seven degree of freedom (d.o.f) arm and three d.o.f. base. For this system a total of

$$\begin{aligned} m-n+1 &= 10-6+1 \\ &= 5 \end{aligned} \tag{28}$$

solution vectors must be found.

6.1 Typical J and dx

Let's first consider a typical case found when calculating the joint displacements **dq** necessary for a Cartesian displacement, **dx**, of the end effector along a trajectory according to

$$\mathbf{dx} = \mathbf{J}\mathbf{dq} \tag{29}$$

The Jacobian, **J**, and **dx** vector are

$$\mathbf{J} = \begin{bmatrix} 0.371 & -0.635 & -0.015 & 0.001 & -0.000 & -0.006 & 0.000 & 1.0 & 0.0 & -0.245 \\ -0.860 & -0.011 & 0.880 & 0.000 & 0.000 & 0.343 & 0.000 & 0.0 & 1.0 & -0.528 \\ 0.000 & -0.867 & 0.000 & -0.851 & -0.343 & 0.000 & 0.000 & 0.0 & 0.0 & 0.000 \\ 0.000 & 0.000 & -0.021 & 0.000 & 0.000 & 0.000 & -1.000 & 0.0 & 0.0 & 0.000 \\ 0.000 & -1.000 & -0.000 & -1.000 & -1.000 & 0.000 & -0.001 & 0.0 & 0.0 & 0.000 \\ 1.000 & 0.000 & -1.000 & -0.000 & -0.000 & -1.000 & -0.000 & 0.0 & 0.0 & 1.000 \end{bmatrix} \tag{30}$$

$$\mathbf{dx} = \begin{bmatrix} 0.009 & 0.000 & 0.001 & -0.000 & 0.000 & 0.018 \end{bmatrix}^T \tag{31}$$

None of the elements of **dx** are restricted, no rows of **J** are dependent, and **J** has no rows or columns of zeros. The first solution is found after only three incrementations of the final block. The blocking pattern for the entire set of solutions is shown below

⊗	⊗	⊗	⊗	X	X	X	X	X	X
⊗	⊗	⊗	X	X	X	X	⊗	X	X
⊗	⊗	X	⊗	X	X	X	⊗	X	X
⊗	X	⊗	⊗	X	X	X	⊗	X	X
X	⊗	⊗	⊗	X	X	X	⊗	X	X

In this case, the final four solutions were found immediately from the first correct solution.

$$\begin{bmatrix} \mathbf{g}^1 \\ \mathbf{g}^2 \\ \mathbf{g}^3 \\ \mathbf{g}^4 \\ \mathbf{g}^5 \end{bmatrix} = \begin{bmatrix} 0.000 & 0.000 & 0.000 & -0.002 & -0.002 & 0.000 & 0.000 & 0.013 & 0.010 & 0.018 \\ 0.000 & 0.000 & 0.000 & -0.002 & 0.002 & -0.053 & -0.000 & 0.000 & -0.000 & -0.034 \\ 0.000 & 0.000 & -0.052 & -0.002 & 0.002 & 0.000 & 0.001 & 0.000 & 0.028 & -0.034 \\ 0.000 & -0.021 & 0.000 & 0.020 & 0.001 & 0.000 & 0.000 & 0.000 & 0.009 & 0.018 \\ 0.022 & 0.000 & 0.000 & -0.002 & 0.002 & 0.000 & 0.000 & 0.000 & 0.017 & -0.004 \end{bmatrix} \tag{32}$$

From the vectors $\mathbf{g}^k$ a least norm solution can be found (see [6, 7]) as

$$\mathbf{dq} = \begin{bmatrix} 0.004 & -0.003 & -0.002 & 0.001 & 0.002 & -0.007 & 0.000 & 0.007 & 0.010 & 0.004 \end{bmatrix}^T \tag{33}$$

6.2 J with restrictions

Next consider the case where some elements of **J** are restricted:

$$\mathbf{J} = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 \end{bmatrix} \tag{34}$$

$$\mathbf{dx} = \begin{bmatrix} 1.0 & 2.0 & 3.0 & 0.0 & 5.0 & 6.0 \end{bmatrix}^T \tag{35}$$

After eliminating the restrictions and the row of zeros, the system has reduced to:

$$\mathbf{J} = \begin{bmatrix} 1.0000 & 2.0000 & 3.0000 & 4.0000 & 5.0000 \end{bmatrix} \tag{36}$$

$$\mathbf{dx} = 6.0000 \tag{37}$$

from which the blocking pattern is immediately determined to be

⊗	⊗	⊗	⊗	X
⊗	⊗	⊗	X	⊗
⊗	⊗	X	⊗	⊗
⊗	X	⊗	⊗	⊗
X	⊗	⊗	⊗	⊗

The solution vectors without the restricted components

$$\begin{bmatrix} \mathbf{g}^i_1 & \mathbf{g}^i_2 & \mathbf{g}^i_3 & \mathbf{g}^i_4 & \mathbf{g}^i_5 \end{bmatrix}^T = \begin{bmatrix} 2.0 & -1.0 & 3.0 & 0.0 & -1.0 \end{bmatrix}^T \qquad i = \{1 \ldots 5\} \tag{38}$$

are

$$\begin{bmatrix} \mathbf{g}^{1*} \\ \mathbf{g}^{2*} \\ \mathbf{g}^{3*} \\ \mathbf{g}^{4*} \\ \mathbf{g}^{5*} \end{bmatrix} = \begin{bmatrix} 6.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.2 \\ 0.0 & 0.0 & 0.0 & 1.5 & 0.0 \\ 0.0 & 0.0 & 2.0 & 0.0 & 0.0 \\ 0.0 & 3.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \tag{39}$$

The solution space, **S**, is again given by Eq. 14 and the least-norm solution in this case (see [6, 7]) is:

$$\mathbf{dq} = \begin{bmatrix} 2.0 & -1.0 & 3.0 & 0.0 & -1.0 & 0.109 & 0.218 & 0.327 & 0.436 & 0.546 \end{bmatrix}^T \tag{40}$$

6.3 Completely restricted J

The next system involves a **J** and **dx** pair that are completely restricted:

$$\mathbf{J} = \begin{bmatrix} 1.00 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.00 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & -0.5 & 0.0 & 0.0 & -0.6 \\ 0.00 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 1.3 & 0.0 & 0.0 \\ 0.01 & 0.1 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.00 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 4.0 & -3.0 \\ 0.00 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 2.0 & 3.0 & 0.4 \end{bmatrix} \tag{41}$$

$$\mathbf{dx} = \begin{bmatrix} 0.00 & 0.00 & 0.00 & 0.01 & 0.00 & 0.00 \end{bmatrix}^T \tag{42}$$

After eliminating restrictions, the system is

$$\mathbf{J} = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 & -0.5 & 0.0 & 0.0 & -0.6 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 1.3 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 4.0 & -3.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 2.0 & 3.0 & 0.4 \end{bmatrix} \tag{43}$$

$$\mathbf{dx} = \begin{bmatrix} 0.00 & 0.00 & 0.00 & 0.00 \end{bmatrix} \tag{44}$$

Since the system is completely restricted, the solution vectors must be found from the nullspace. The nullspace only generates $m-n$ vectors, so the final vector is composed of all zeros except for the restricted components. For this system, the solution vectors without the restricted components

$$\begin{bmatrix} \mathbf{g}^{\mathbf{i}}{}_1 & \mathbf{g}^{\mathbf{i}}{}_2 \end{bmatrix} = \begin{bmatrix} 0.0 & 0.1 \end{bmatrix} \qquad i = \{1 \ldots 5\} \tag{45}$$

are

$$\begin{bmatrix} \mathbf{g}^{1*} \\ \mathbf{g}^{2*} \\ \mathbf{g}^{3*} \\ \mathbf{g}^{4*} \\ \mathbf{g}^{5*} \end{bmatrix} = \begin{bmatrix} 0.0000 & -0.0241 & 0.6047 & -0.5454 & -0.4355 & 0.0185 & 0.1210 & 0.3629 \\ 0.0000 & 0.2132 & -0.7183 & -0.5454 & -0.1516 & -0.1640 & 0.2743 & 0.1263 \\ 0.0000 & -0.6331 & -0.1001 & -0.4593 & 0.2671 & 0.4870 & -0.1419 & -0.2226 \\ -0.5087 & -0.1942 & -0.1843 & 0.2010 & -0.7285 & 0.1494 & -0.1345 & -0.2408 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \end{bmatrix} \tag{46}$$

Again using Eq. 14 and the least norm optimization to determine the final solution:

$$\mathbf{dq} = \begin{bmatrix} 0.000 & 0.100 & -0.127 & -0.160 & -0.100 & -0.337 & -0.262 & 0.123 & 0.030 & 0.007 \end{bmatrix}^T \tag{47}$$

6.4 J with loss of row rank

This example shows the results of finding a solution vector **x** for a matrix **A** which has dependent rows.

$$\mathbf{J} = \begin{bmatrix} 1.00 & 2.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & -0.50 & 0.00 & 0.00 & -1.00 \\ 0.00 & 1.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.00 \\ 1.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 4.00 & -3.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 2.00 & 3.00 & 0.00 \end{bmatrix}^T \tag{48}$$

$$\mathbf{dx} = \begin{bmatrix} 1.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.00 \end{bmatrix}^T \tag{49}$$

After eliminating the first row, the $m - n + 2 = 6$ solution vectors are found to be:

$$\begin{bmatrix} \mathbf{g}^1 \\ \mathbf{g}^2 \\ \mathbf{g}^3 \\ \mathbf{g}^4 \\ \mathbf{g}^5 \\ \mathbf{g}^6 \end{bmatrix} = \begin{bmatrix} 0.00 & 1.00 & 0.00 & -1.00 & 0.00 & 0.00 & 0.00 & -0.00 & 0.00 & 0.00 \\ 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & -1.78 & -1.00 & 0.66 & 0.88 \\ 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 1.99 & 0.00 & -1.00 & 0.00 & -0.00 \\ 0.00 & 1.00 & 0.00 & 0.00 & -2.67 & 0.00 & 0.00 & -1.00 & 0.67 & 0.00 \\ 0.00 & 1.00 & 0.89 & 0.00 & 0.00 & 0.00 & 0.00 & -1.00 & 0.67 & 0.89 \\ 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \end{bmatrix} \tag{50}$$

For this example, a single-constraint optimization must be used. The β values are calculated from Eq. 18 to be:

$$\beta = \begin{bmatrix} 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 \end{bmatrix}^T \tag{51}$$

The solution space is given by Eq. 15, and the final least norm solution found (see [6, 7]) is then:

$$\mathbf{x} = \begin{bmatrix} 0.6273 & 0.3727 & 0.0497 & -0.2547 & -0.0373 & 0.0683 & -0.0248 & -0.1180 & 0.0559 & 0.0621 \end{bmatrix}^T \tag{52}$$

6.5 Trajectory

The final example involves a complete trajectory that was formed using a least norm optimization to find a single solution out of the entire space. Figs. 3 and 4 show the user-specified starting and target end effector position/orientation. The final joint angles are simply used to position the end-effector and are not intended to be reached in the final configuration of the platform and manipulator. Figs. 5-8 show the motion of the system as it moves from the initial to final configuration for the end effector. The end effector moves smoothly along its specified path to reach the desired configuration. The diagrams were made with an XWindows graphics modeling program created by Derek Carlson.

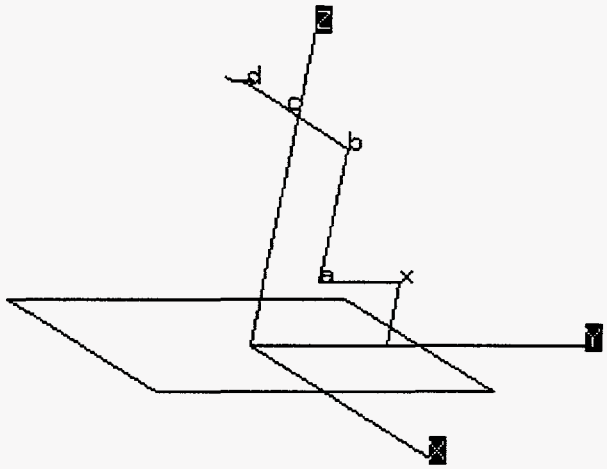


Figure 3: Starting end effector position

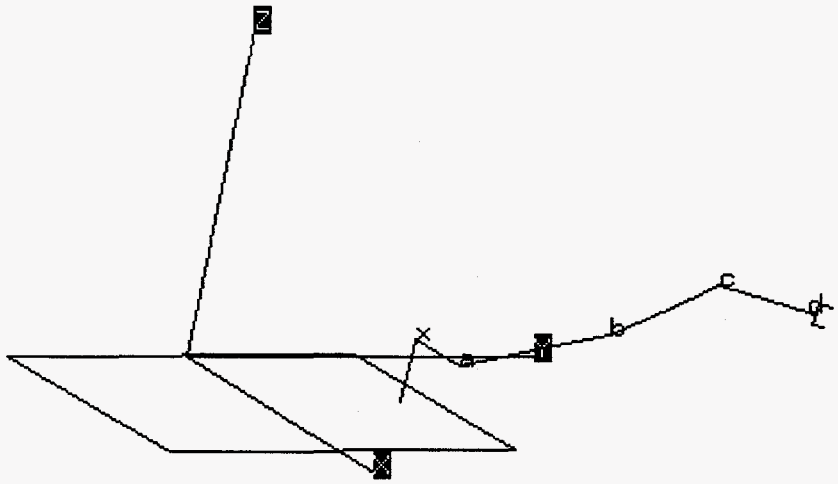


Figure 4: Final end effector position

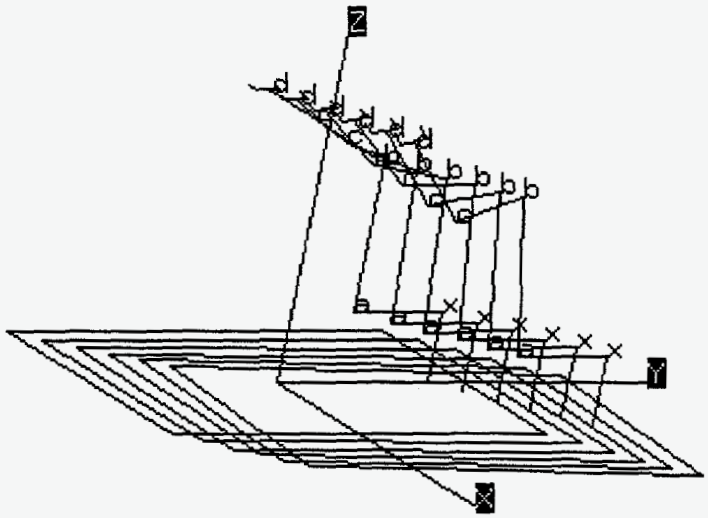


Figure 5: First intermediate platform motion

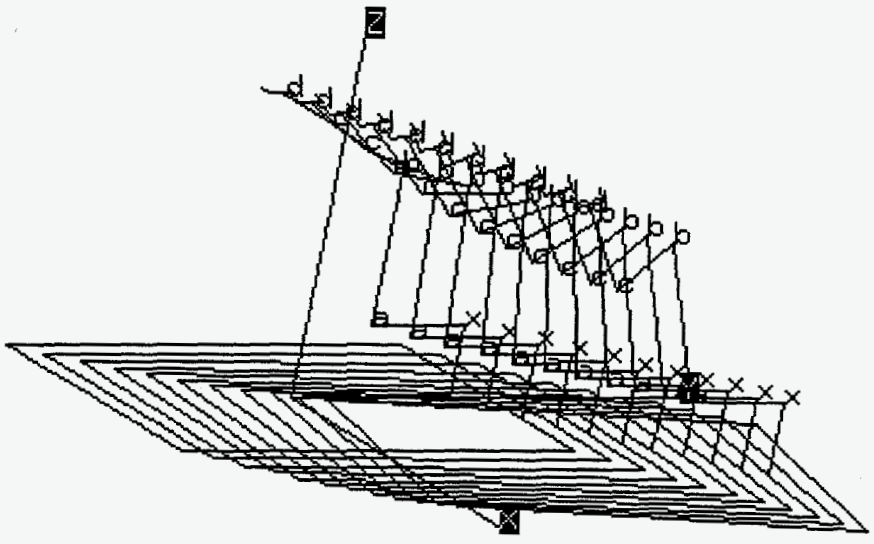


Figure 6: Second intermediate platform motion

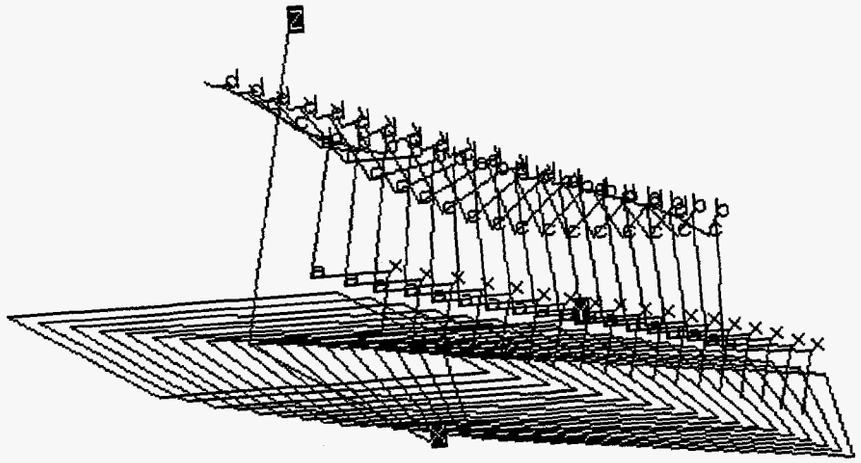


Figure 7: Third intermediate platform motion

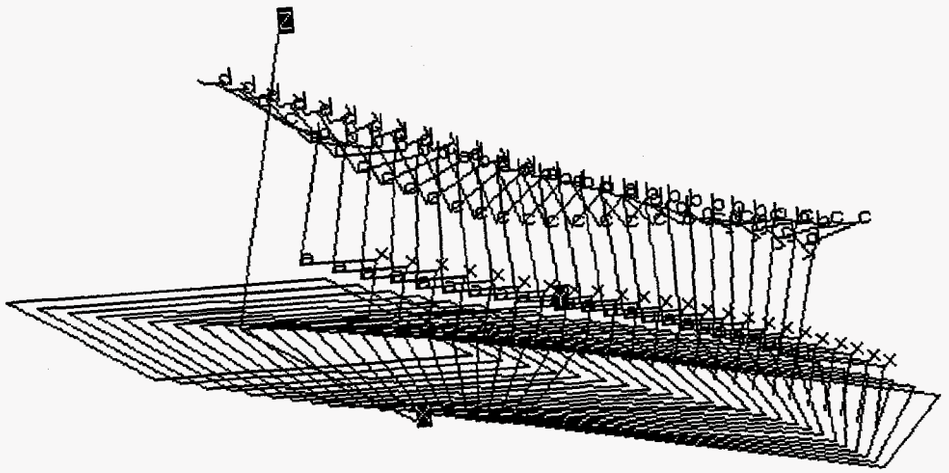


Figure 8: Complete platform motion

7 CONCLUSION

This article has discussed the enhanced code for the recently developed Full Space Parameterization (FSP) method. The proof in [6] for existence of the $m - n + 1$ vectors necessary to form the solution space $\mathbf{S}$ (for system $\mathbf{A}$ and $\mathbf{b}$ where $\mathbf{A}$ has m columns and n rows) was augmented with a discussion of the conditions necessary to ensure that the $m - n + 1$ vectors will be linearly independent. Also, a pattern was shown to exist among the columns that are chosen to form the submatrices. This pattern has been used to speed up the search for the complete set of solution vectors. The possible complications that could arise between a matrix $\mathbf{A}$ and a vector $\mathbf{b}$ were discussed and placed into two categories: those in which $\mathbf{A}$ has lost row rank, and those in which the specific combination of $\mathbf{A}$ and $\mathbf{b}$ causes restricted elements. In the first case an additional constraint must be included in the solution space $\mathbf{S}$. Specific cases of restrictions were also discussed. An algorithm was presented and discussed to eliminate from the system the dependent rows and restricted elements of $\mathbf{A}$ and $\mathbf{b}$ in order to decrease computation time. One of the main features of the presented code is the algorithm given to speed up the search for the set of $m - n + 1$ solution vectors for any matrix $\mathbf{A}$ and vector $\mathbf{b}$. The respective parameterizations of the solution space, $\mathbf{S}$, for the cases of the matrix $\mathbf{A}$ being full rank and rank deficient were given. Examples of the different combinations of $\mathbf{A}$ and $\mathbf{b}$ (from a mobile manipulator) were presented with final solution vector sets and least norm solutions $\mathbf{x}$. Also a complete trajectory was shown with solutions found using the FSP method with least norm optimization. The code from which these examples were made as well as directions for using the code as either a stand-alone program or as an included procedure have been included in the Appendices.

8 ACKNOWLEDGMENT

This research was supported in part by the U.S. Air Force Air Combat Command, Munition Material Handling Equipment Focal Point, under Interagency Agreement 2146-H055-A1 between the U.S. Air Force San Antonio Air Logistic Center and the U.S. Department of Energy.

References

[1] Baillieul, J. and D. P. Martin "Resolution of Kinematic Redundancy," in *Robotics: Proceeding of Symposia in Applied Mathematics.* Providence, RI: American Mathematical Society, 1990, pp. 49-90.

[2] Dubey, R.V., J. A. Euler, and S. M. Babcock "An Efficient Gradient Projection Optimization Scheme for a Seven-Degree- of-Freedom Redundant Robot with Spherical Wrist," *Proc. 1988 International Conf. Robot. Automat.,* IEEE Computer Society Press, Washington, pp. 28-36.

[3] Maciejewski, A. A. and C. A. Klein "The Singular Value Decomposition: Computation and Applications to Robotics", *International Journal of Robotics Research*, Vol. 8, No. 6, pp. 63-79, 1989.

[4] Press, W. H., S. A. Teukolsky, W. T. Velterling, and B. P. Flannery *Numerical Recipes in C*, Cambridge University Press, second edition, 1992.

[5] Oh, S.Y., D. Orin, and M. Bach "An Inverse Kinematic Solution for Kinematically Redundant Robot Manipulators," *Journal of Robotic Systems,* Vol. 1, No. 3, pp. 235-249, 1984.

[6] Pin, F. G. et al. "A New Solution Method for the Inverse Kinematic Joint Velocity Calculations of Redundant Manipulators". *Proc. IEEE Int. Conf. on Robotics and Automation*, pp. 96-102, 1994.

[7] Pin., F. G. "The Full Space Parameterization (FSP) Method for Solution of Underspecified Systems of Algebraic Equations: Theory and Application to Redundant Manipulator Control with Changing Criteria and Constraints". in review for publication at *Oak Ridge national Laboratory Technical Report No. ORNL/TM-12510*, 1993.

[8] Pin, F. G., J. C. Culioli, and D. B. Reister "Using Minimax Approaches to Plan Optimal Task Commutation Configurations for Combined Mobile Platform-Manipulator Systems". *IEEE Trans. Robotics and Automation*, Vol. 10, No. 1, pp. 44-54, 1994.

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

[10] Seraji, H. "Configuration Control of Redundant Manipulators: Theory and Implementation," *IEEE Transactions on Robotics and Automation*, vol. 5, No. 4, pp. 472-490, 1989.

[11] Vukobratovic, M. and Kircanski, M. "A Dynamic Approach Nominal Trajectory Synthesis for Redundant Manipulators," *IEEE Trans. Syst., Man, Cybern.,* SMC-14, no. 4, pp. 580-586, 1984.

[12] Whitney, D.E. "Resolved Motion Rate Control of Manipulators and Human Prostheses," *IEEE Trans. Man-Machine Systems,* MMS-10, pp. 47-53, 1969.

[13] Yoshikawa, T. "Manipulability of Robotic Mechanisms". *International Journal of Robotics Research*, Vol. 4, No. 2, pp. 3-9, 1985.

A USER'S GUIDE

The code which will be given in Appendix B can be used either as a stand-alone program or as a procedure called by another program. In the first case the matrix **A** and the vector **b** are read from a user-specified file while in the second case they are passed in as arguments. For **A** and **b** to be read correctly from the file, **A** must be listed first row by row followed by **b**. In either case, all user-defined parameters are specified in the program header file. Procedures LeastNorm and RankLostSoln have been included as examples of solution techniques for unconstrained least norm optimization and single-constraint least norm optimization as discussed in the article. The version of the program which has been provided utilizes routines from *Numerical Recipes in C*. The user will either need to provide the files nrutil.c and svdcmp.c (from [4]) or provide suitable alternatives and alter the code accordingly. All other necessary procedures are given in Appendix B. As a note, the given files have not been optimized in any way, and ample room is available for improvement.

The code, as given in Appendix B, is set up to be used as an independent program. No modifications need to be made to the file FSP.c. In the file FSP.h, the user must choose values for K2_BND. K2_BND determines the maximum allowable condition number (ratio of largest to smallest singular values of a matrix) which the algorithm will consider acceptable. The file from which the matrix **A** and vector **b** are to be read must be set on the line where the variable infile is defined. The size of infile should also be set appropriately. For example if the file where **A** and **b** are stored is name "TestData", the line should read:

```
char infile[8] = "TestData";
```

The user must specify the diagnostic output filename in the same way as the input filename and size were given. Also, the size of the matrix must be set with the variables M and N.

To modify the code so that it run as a procedure inside another program, the 'char infile ...' line must be removed from the FSP.h file. As with the stand-alone version, the values for M, N, K2_BND, and outfile must be set as the user desires. Both the variable declarations and the memory freeing lines for Aorig and borig must be removed from the code. Also the procedure call to read in the matrix **A** and vector **b** must be removed. The procedure itself can also be removed from the end of the code, however, this step is not absolutely necessary. The line main () must be changed to

```
void FSP (struct MATRIX *Aorig, struct MATRIX *borig)
```

where FSP can be renamed to whatever procedure name the user desires. Aorig and borig are the matrix **A** and the vector **b** which will be used to calculate the solution

x and must be defined in the program which calls FSP.c using the variable declaration

```
struct MATRIX *Aorig, *borig;
```

and the commands

```
Aorig = mat_malloc(N,M)
borig = mat_malloc(N,1)
```

in order to allocate memory space of the appropriate dimension. N and M are respectively the number of rows and columns of the matrix A. When the variables Aorig and borig are no longer needed by the program the memory must be freed using the commands

```
mat_free(Aorig)
mat_free(borig)
```

The files matrix.h and matrix.c must be present in the same directory as the files FSP.h and FSP.c in order for the necessary matrix functions such as the nullspace calculation to be found. The file in which FSP.c is called as a procedure must also have

```
#include "matrix.h"
```

at the beginning in order for the function calls mat_malloc and mat_free to be found.

B CODE LISTINGS

```
/* This File contains the size of the A, where A          */
/* is located, where to write the output data, and.       */
/* user-defined threshold values                          */

int             N = 6;          /* number of rows         */
int             M = 10;         /* number of cols         */

char            infile[10]  = "testfile6";      /* user-defined file      */
char            outfile[10] = "data6";          /* user-defined file      */

#define K2_BND      10                          /* user-defined value     */

#define TRUE  1
#define FALSE 0


int CheckB (struct MATRIX *b, int m, float SMALL);
```

```
/*************************************************************************/
/*                                                                       */
/*  This program is the independent version of the code which uses the   */
/*  FSP method to find a set of vectors spanning the solution space of   */
/*  the problem:                                                         */
/*                x = A^(-1) * b                                         */
/*  The final solution, x, is found from the user-specified optimization */
/*  method.                                                              */
/*                                                                       */
/*                                                                       */
/*  Kristi A. Morgansen                                                  */
/*  Robotics and Process Systems Division                                */
/*  Oak Ridge National Laboratory                                        */
/*  P.O. Box 2008                                                        */
/*  Oak Ridge, TN  37831-6304                                            */
/*                                                                       */
/*  August 26, 1994                                                      */
/*                                                                       */
/*************************************************************************/
```

```
#include <stdio.h>
#include <math.h>
#include "matrix.h"
#include "FSP.h"

main ()                                                          main
{
FILE *check;

struct MATRIX *Aorig,  /* original A                                  */
              *Ared,   /* reduced A                                   */
              *Asub,   /* submatrix from square submatrix is found    */
              *Asqr,   /* square submatrix                            */
              *g,      /* array of solution vectors                   */
              *borig,  /* desired movement in work space coordinates  */
              *bred,   /* reduced b                                   */
              *block,  /* locations of blocked columns for each soln  */
              *n,      /* null space vectors                          */
              *n_vec,  /* mtx of nullspace vectors for all soln vectors */
              *n_temp, /* used to reorder the nullspace vectors       */
              *Xelim,  /* comps of final X that were elim. in reduction */
              *X;      /* final solution for joint space movement     */
float    K2,           /* matrix condition number:  sv_max/sv_min     */
         K2_0,         /* condition number of first solution          */
         SMALL,        /* threshold for zero                          */
         N_BND;        /* min acceptable value for nonzero nspace comp */
int      i, j, k, I,   /* loop counters                               */
         bcheck,       /* check for completely zero bred              */
         ChangeBlock,  /* which of four possible blocks is being moved */
         LoopCount,    /* how many loop cycles have been executed     */
         NextToFind,   /* vector number being searched for            */
```

```
        ColElim[M],      /* marks columns eliminated from original A      */
        RowElim[N],      /* which rows are to be eliminated from the A    */
        Ordering[M],     /* columns are indexed for effic, soln finding   */
        index,           /* used with the Ordering[] variable             */
        nullity,         /* dimension of null space                       */
        SystemComplete, /* marks when all necessary vecs have been found      */
        binrange,        /* marks if b is in the range of the A           */
        Nred,            /* number of rows in reduced A                   */
        Mred;            /* number of columns in reduced A                */


Aorig   = mat_malloc(N,M);  /* allocate memory space for variables */
Ared    = mat_malloc(N,M);
borig   = mat_malloc(N,1);
bred    = mat_malloc(N,1);
Xelim   = mat_malloc(M,1);
X       = mat_malloc(M,1);

SystemComplete = FALSE;
LoopCount = 0;

GetData(Aorig, borig);          /* receive the A and b vector from a file      */
check = fopen(outfile, "w");    /* file where rslts of this prog are stored    */
            /* Eliminate all the nonredundancies from the A and b */
ReduceA(Aorig, Ared, borig, bred, Xelim, N, M, &Nred, &Mred,
                                                ColElim, RowElim, &SMALL);

g = mat_malloc(Mred-Nred+1,M);
   for (i=0; i<(Mred-Nred+1); i++)
      for (j=0; j<M; j++)
         g->p[i][j] = 0.0e0;
n = mat_malloc(Mred,Nred);
n_temp = mat_malloc(Mred,1);

fprintf(check, "      ORIGINAL SYSTEM\n\n");
PrintA(check, Aorig, borig, N, M);
fprintf(check, "      REDUCED SYSTEM\n\n");
PrintA(check, Ared, bred, Nred, Mred);
fflush(check);

bcheck=CheckB(bred, Nred, SMALL);
if (bcheck == 0)
   {
   X=Xelim;
   fprintf(check, "GIVEN MATRIX IS COMPLETELY RESTRICTED\n\n");

   Asub = mat_malloc(Nred,Mred);
   for (i=0; i<Nred; i++)
      for (j=0; j<Mred; j++)
         Asub->p[i][j] = Ared->p[i][j];

   mat_null(Asub, &nullity, n, &K2);
```

```
                        /* set the solution vectors for a completely      */
                        /* restricted system                              */
    for (i=0; i<Mred; i++)
        for (j=0; j<(Mred-Nred); j++)                                          110
            g->p[j][i] = n->p[i][j];
    for (i=0; i<Mred; i++)
        g->p[Mred-Nred][i] = 0.0;

    LoopCount = 1;
    SystemComplete = TRUE;
    }
else
    {
    if (Nred != N)                                                             120
        {
        fprintf(check, "GIVEN MATRIX IS RESTRICTED FOR %d ", N-Nred);
        fprintf(check, "OUT OF %d VECTOR COMPONENTS\n\n", N);
        }

    block = mat_malloc(Mred-Nred+1,Mred-Nred);
    Asub = mat_malloc(Nred,Nred+1);
    Asqr = mat_malloc(Nred,Nred);
    n_vec = mat_malloc(Mred-Nred+1,Mred);
                                                                               130
    ChangeBlock = Mred-Nred-1;
    NextToFind = 0;
                        /* the smallest acceptable value for a nullspace  */
                        /* component is set to be slightly less than      */
                        /* 1/(# of components in nullspace vector)        */
    N_BND = 0.9/(Nred+1.0);

            /* initialize the blocking positions for the first solution vector */
    for (i=0; i<(Mred-Nred); i++)
        block->p[0][i] = i;                                                    140
    for (i=0; i<Mred; i++)
        Ordering[i]=i;
    index=block->p[0][Mred-Nred-1];

    k=0; I=0;
    for (i=0; i<Mred; i++)
        if (block->p[0][k] != i)
            {
            for (j=0; j<Nred; j++)
                Asqr->p[j][I] = Ared->p[j][i];                                 150
            I++;
            }
        else
            k++;

    mat_null(Asqr, &nullity, n, &K2);
    LoopCount = 1;

                    /* loop until an acceptably well-conditioned first    */
```

```
                /* solution is found                                    */
while (((nullity != 0) || (fabs(K2) > K2_BND)) && (block->p[0][0]<Nred))
   {
   LoopCount++;

   while (block->p[0][ChangeBlock]>(Nred+ChangeBlock-1))
      ChangeBlock--;

   if (block->p[0][0]==Nred)
      ChangeBlock=0;

   block->p[0][ChangeBlock]++;        /* change column being blocked     */
                                      /* set all following blocks        */
   for (i=ChangeBlock+1; i<(Mred-Nred); i++)
   block->p[0][i] = block->p[0][ChangeBlock]+i-ChangeBlock;

   for (i=0; i<Mred; i++)
      Ordering[i]=i;

   index = block->p[0][Mred-Nred-1];
   ChangeBlock = Mred-Nred-1;

   k=0; I=0;
   for (i=0; i<Mred; i++)
      if (block->p[0][k] != i)
         {
         for (j=0; j<Nred; j++)
            Asqr->p[j][I] = Ared->p[j][i];
         I++;
         }
      else
         k++;

   if (block->p[0][0] <= Nred)
      mat_null(Asqr, &nullity, n, &K2);

   }

K2_0 = K2;


   /*************************************************************/
   /* Find the M-N+1 solution vectors according to the pattern  */
   /* given in the article associated with this algorithm.      */
   /************************************************ ************/

   /* look for the soln vectors until either all combinations of */
   /* blocked columns have been tried or the complete set is found */
while ((block->p[0][0] <= Nred) && (!SystemComplete) && (K2_0 < K2_BND))
   {
   BLOCK_COL_FIND_X(block->p[NextToFind],g->p[NextToFind],
                                          bred,Ared,Mred,Nred,check);
   fprintf(check, "gvector %2d : ", NextToFind);
```

```
for (i=0; i<Mred; i++)
  fprintf(check, "%8.6f ", g->p[NextToFind][i]);
fprintf(check, "\n\n");


if (NextToFind < (Mred-Nred))  /* check if not all solns found */
   {
     NextToFind++;                                                    220
                                /* set initial blocking for next soln */
     for (i=0; i<(Mred-Nred-NextToFind); i++)
       block->p[NextToFind][i] = block->p[NextToFind-1][i];
     block->p[NextToFind][Mred-Nred-1] =
               block->p[NextToFind-1][Mred-Nred-NextToFind];
     for (i=(Mred-Nred-NextToFind); i<(Mred-Nred-1); i++)
       block->p[NextToFind][i] = block->p[NextToFind-1][i+1];

                                /* set column ordering for next soln */
   for (i=0; i<(Mred-Nred); i++)                                      230
      Ordering[i] = block->p[NextToFind][i];

   Set_Ordering(Ordering, Mred, Nred);

   index = Mred-Nred-1;

                                /* find column dependencies for next soln*/
   for (i=0; i<Nred+1; i++)
      for (j=0; j<Nred; j++)
         Asub->p[j][i] = Ared->p[j][Ordering[i+(Mred-Nred-1)]];       240

   mat_null(Asub, &nullity, n, &K2);



   LoopCount++;

   for (i=0; i<(Mred-Nred-1); i++)
      n_temp->p[i][0] = 0;
   for (i=(Mred-Nred-1); i<Mred; i++)                                 250
      n_temp->p[i][0] = n->p[i-(Mred-Nred-1)][0];

   for (i=0; i<Mred; i++)
      n_vec->p[NextToFind][Ordering[i]] = n_temp->p[i][0];
   } /* if NTF < Mred-Nred */
else                                    /* all solns have been found */
   {
   SystemComplete = TRUE;
   fprintf(check, "BLOCKING\n");
   for (i=0; i<(Mred-Nred+1); i++)                                    260
      {
      for (j=0; j<(Mred-Nred); j++)
         fprintf(check, "%7.4f ", block->p[i][j]);
      fprintf(check, "\n");
      }
```

```
        }


    if (!SystemComplete)
        {                                                                    270
        ChangeBlock=Mred-Nred-1;
        if (NextToFind != 0)
            index++;
        }

                        /* check that the blocking configuration        */
                        /* meets the criterion for the next solution    */
                        /* to be independent from the previous solns    */
                        /* if it is not, cycle until an acceptable      */
                        /* configuration is found                       */  280
                        /* previous solns may have to be rejected       */
    while
(
  (
    (
    (NextToFind==0) && (block->p[0][0] <= Nred) &&
                    ( (nullity != 0) || (fabs(K2) > K2_BND) )
    )
  )
    ||                                                                       290
    (
    (NextToFind!=0) && ( (index>= Mred) || ( (fabs(n_vec->p[NextToFind][Ordering[index]]) < N_BND) ||
                                             (fabs(  g->p[NextToFind-1][Ordering[index]]) < (SMALL/10))
    )
  )
)

{

        if (NextToFind == 0)     /* find a new first solution */             300
            {
            while (block->p[0][ChangeBlock]>(Nred+ChangeBlock-1))
                ChangeBlock--;

            if (block->p[0][0]==Nred)
                ChangeBlock=0;

            block->p[0][ChangeBlock]++;  /* change column being blocked  */
                                         /* set all following blocks     */
            for (i=ChangeBlock+1; i<(Mred-Nred); i++)                        310
            block->p[0][i] = block->p[0][ChangeBlock]+i-ChangeBlock;

            for (i=0; i<Mred; i++)
                Ordering[i]=i;

            index = block->p[0][Mred-Nred-1];
            ChangeBlock = Mred-Nred-1;
```

```
        k=0; I=0;
        for (i=0; i<Mred; i++)                                          320
            if (block->p[0][k] != i)
                {
                for (j=0; j<Nred; j++)
                    Asqr->p[j][I] = Ared->p[j][i];
                I++;
                }
            else
                k++;

        if (block->p[0][0] <= Nred)                                     330
            {
            mat_null(Asqr, &nullity, n, &K2);
            K2_0 = K2;
            LoopCount++;
            }
        } /* if (MTF == 0) */
    else                        /* find a new soln other than the first */
        {
        index++;
                                                                        340
      while (index >= Mred)
            {
            NextToFind--;
            if (NextToFind != 0)
                {
                for (i=0; i<(Mred-Nred); i++)
                    Ordering[i]=block->p[NextToFind-1][i];

                Set_Ordering(Ordering, Mred, Nred);
                                                                        350
                index=Mred-Nred;
                while (block->p[NextToFind][Mred-Nred-1] != Ordering[index])
                    index++;
                index++;
                } /* if (MTF !=0) */
            else
                {
                for (i=0; i<Mred; i++)
                    Ordering[i]=i;
                                                                        360
                index = block->p[0][Mred-Nred-1];
                } /* else if (MTF != 0) */
            } /* if (index == Mred) */
        } /* else if (MTF == 0) */
    } /* while (nullity != 0) ... */

                            /* set the final block for a new solution */
    block->p[NextToFind][Mred-Nred-1] = Ordering[index];

} /* end while !SystemComplete */                                      370
```

```
mat_free(block);
mat_free(Asqr);
mat_free(n_vec);
} /* else if Nred == 0 */



                                /* print diagnostic info */
                                                                        380

fprintf(check, "\nXelim\n");
for (i=0; i<M; i++)
   fprintf(check, "%7.4f ", Xelim->p[i][0]);
fprintf(check, "\n");

     /* check if the eliminated b elements in bred can be          */
     /* produced from the g vectors and the reduced A matrix       */
     /* ie check that b is in the range of A                       */
if (!SystemComplete)                                                    390
{
     fprintf(stderr,"System did not complete!\n");
     fprintf(check,"System did not complete!\n");
}

     /* check that the original b was not all zeros (special case) */
bcheck = CheckB(borig, M, SMALL);

     /* check whether the original A had dependent rows  */
if (SystemComplete && (bcheck != 0))                                    400
   {
   i = 0;
   j = 1;
   while ((i<N) && (j != 0))
      {
      if (RowElim[i] != 2)
         i++;
      else
         {                                    /* user-defined procedure */
         RankLostSoln(g, X, Aorig, Mred, Nred, Xelim, ColElim, RowElim,  410
                      borig, check);
         j = 0;

         fprintf(check, "\nFINAL SOLUTION:\n");
         for (i=0; i<M; i++)
            fprintf(check, "%7.4f ", X->p[i][0]);
         fprintf(check, "\n");

         binrange = CheckRange(borig, Aorig, X, RowElim, Mred, check);
         if (binrange != 1)                                             420
            {
            fprintf(check, "INACCURATE FINAL SOLUTION\n");
            fprintf(check, "    B NOT IN RANGE OF A\n\n");
            }
```

```
            fprintf(check, "LoopCount for final solution: %4d\n", LoopCount);
            }
         }
      if (j == 1)
         {                                                              430
                                          /* user-defined procedure */
         LeastNorm(X, g, Mred, Nred, Xelim, ColElim);

         fprintf(check, "\nFINAL SOLUTION:\n");
         for (i=0; i<M; i++)
            fprintf(check, "%7.4f ", X->p[i][0]);
         fprintf(check, "\n");
         fprintf(check, "LoopCount for final solution: %4d\n", LoopCount);
         }
      }                                                                 440
   else
      if (K2 > K2_BND)
         {
         fprintf(check, "\n\nNO WELL-CONDITIONED SUBMATRICES FOUND\n");
         fprintf(check, "If result seems incorrect, the maximum allowable\n");
         fprintf(check, "condition number can be increased by changing the\n");
         fprintf(check, "value of K2_BND in the file FSP.h.\n");
         }
      else if (bcheck != 0)
         fprintf(check, "\n\nB NOT IN RANGE OF A\n");                    450
      else
         {
         fprintf(check, "\n\nSPECIAL CASE\nB IS ZERO VECTOR\n");
         fprintf(check, "CONSTRAINED OPTIMIZATION MUST BE USED\n");
             /* insert appropriate equations here */
         }

mat_free(Aorig);        /* free up all variable memory space */
mat_free(Ared);
mat_free(Asub);                                                         460
mat_free(borig);
mat_free(bred);
mat_free(Xelim);
mat_free(X);
mat_free(n);
mat_free(n_temp);
mat_free(g);

fclose(check);
}                                                                       470


   /*********************************************************************/
   /* calculate a g vector based upon a A and given column blocking     */
   /*********************************************************************/
BLOCK_COL_FIND_X    (float *ColToBlock, float *g,
                     struct MATRIX *b, struct MATRIX *A,
```

```
                    int Mred, int Nred, FILE *check)
{
int Acol, Tcol, r, I, i, j;                                          480
float blocktemp[Mred-Nred], temp;
struct MATRIX *Atemp, *gtemp, *btemp;

Atemp = mat_malloc(Nred,Nred);
gtemp= mat_malloc(Nred,1);
btemp = mat_malloc(Nred,1);

for (i=0; i<Nred; i++)
   btemp->p[i][0]=b->p[i][0];
                                                                     490
   /* the columns blocked need to be listed from smallest to largest */
for (i=0; i<(Mred-Nred); i++)
   blocktemp[i]=ColToBlock[i];
for (i=(Mred-Nred-1); i>0; i--)
   for (j=i-1; j>=0; j--)
      if (blocktemp[i]<blocktemp[j])
         {
         temp=blocktemp[i];
         blocktemp[i]=blocktemp[j];
         blocktemp[j]=temp;                                          500
         }

j=0;
Acol=0;
for (Tcol=0; Tcol<=(Nred-1); Tcol++)
   {
   for (r=0; r<=(Nred-1); r++)
      if (Acol != blocktemp[j])
         Atemp->p[r][Tcol]=A->p[r][Acol];
      else                                                           510
         {
         Acol++;
         r-=1;
         j++;
         }
      Acol++;
   }

mat_pseudoinv(Atemp);
                                                                     520
gtemp = mat_mul2(Atemp, btemp);

/*Mow add a zero to where column was blocked */

j=0;
I=0;
for (i=0; i<Mred; i++)
   if (i==blocktemp[j])
      {
      g[i]=0.0e0;                                                    530
```

```
        j++;
        I++;
        }
    else
        g[i]=gtemp->p[i-I][0];

mat_free(Atemp);
mat_free(gtemp);
mat_free(btemp);
}

        /*************************************/
        /* dot product of two float vectors  */
        /*************************************/
double ROW_DOT_PRODUCT (float *a, float *b, int n)
{
int i;
double result=0;
for (i=0; i<n; i++)
   result+=a[i]*b[i];
return(result);
}



     /*********************************************************************/
     /* Restricted work space motions can be identified by rows of the    */
     /* A which only have one nonzero element.  Since the                 */
     /* corresponding column must be present in all final joint space     */
     /* solutions, the appropriate joint space motion will be calculated  */
     /* before any redundancy resolution is performed, and the            */
     /* appropriate motions and joints will be removed from the           */
     /* work space and A respectively                                     */
     /*********************************************************************/
ReduceA (struct MATRIX *Aorig, struct MATRIX *Ared, struct MATRIX *borig,
        struct MATRIX *bred, struct MATRIX *Xelim, int N, int M, int *Nred,
        int *Mred, int *ColElim, int *RowElim, float *SMALL)
{
int i, j, k, m,          /* loop counters                                  */
    StillChecking,       /* flag to mark when all nonredundancies are gone */
    LastNred,
    nullity,
    Restriction,         /* num of joints which contrib to a work space d.o.f. */
    NonZeroCol;          /* column which has nonzero element for give row  */
double btemp[N];         /* holds the b vector as it is modified by backsub */
float N_BND,             /* zero threshold for nullspace vector componenets */
      K2;                /* matrix condition number:  sv_max/sv_min        */
struct MATRIX *n,
              *Atemp,
              *Atrans;

Atemp = mat_malloc(N,M);
Atemp = mat_cp2(Aorig);
```

```
Atrans = mat_malloc(M,N);
n = mat_malloc(M,N);

*Nred = N;     /* number of rows in the reduced A        */
*Mred = M;     /* number of columns in the reduced A     */
N_BND = 0.9/(N+1);
StillChecking = 1;                                                    590

     /* initialize variables */
for (i=0; i<N; i++)
   {
   RowElim[i]=0;
   btemp[i]=borig->p[i][0];
   }
for (i=0; i<M; i++)
   {
   Xelim->p[i][0]=0;                                                  600
   ColElim[i]=0;
   }

     /* determine the value for SMALL based on the largest element of A */
*SMALL = 0;
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    if (fabs(Atemp->p[i][j]) > *SMALL)
      *SMALL = fabs(Atemp->p[i][j]);
*SMALL = *SMALL/1000;                                                 610

     /* Check each row for the number of nonzero elements.  If only one   */
     /* element is nonzero, solve for the joint space motion, and          */
     /* modify the b vector so that the appropriate row and column         */
     /* can be eliminated from the A.  After a row is eliminated           */
     /* the remaining A must be rechecked for any new restrictions         */
while (StillChecking)
   {
   LastNred = *Nred;
   for (i=0; i<N; i++)                                                620
      {
      j=-1;
      Restriction=0;

           /* check for nonzero row elements */
      while ((j<(M-1)) && (Restriction < 2))
         {
         j++;
         if (fabs(Atemp->p[i][j]) > *SMALL)
            {                                                          630
            Restriction++;
            NonZeroCol = j;
            }
         }

           /* if a row only has one nonzero element, eliminate it from */
```

```
            /* the A                                                    */
        if ((Restriction == 1) && (RowElim[i] != 1))
            {
            Xelim->p[NonZeroCol][0]=btemp[i]/Atemp->p[i][NonZeroCol];                    640
            for (k=0; k<N; k++)
                btemp[k]=
                        btemp[k]-Atemp->p[k][NonZeroCol]*Xelim->p[NonZeroCol][0];
            for (k=0; k<N; k++)
                Atemp->p[k][NonZeroCol]=0.0;

            ColElim[NonZeroCol]=1;
            RowElim[i]=1;
            *Nred=*Nred-1;
            *Mred=*Mred-1;                                                               650
            }
        else            /* row of all zeros, also eliminated */
            if ((Restriction == 0) && (RowElim[i] != 1))
                {
                RowElim[i]=1;
                *Nred=*Nred-1;
                }
        } /* for (i=0... */
        if (*Nred == LastNred)
            StillChecking = 0;                                                           660
    } /* while (StillChecking) */


        /* check for columns of zeros */
for (i=0; i<M; i++)
    {
    j=0;
    while ((j<N) && (fabs(Aorig->p[j][i]) < *SMALL))
        j++;
                                                                                         670
    if (j==N)
        {
        *Mred=*Mred-1;
        ColElim[i]=1;
        }
    }

      /* check for dependent rows */
for (i=0; i<Aorig->rows; i++)
    for (j=0; j<Aorig->cols; j++)                                                        680
        Atrans->p[j][i] = Aorig->p[i][j];
mat_null(Atrans, &nullity, n, &K2);
for (i=0; i<nullity; i++)
    {
    j=0;
    while (fabs(n->p[j][i]) < N_BND)
        j++;
    if (RowElim[j] != 1)
        {
```

```
      RowElim[j] = 2;                                                    690
      *Nred=*Nred-1;
      }
   }

      /* store the reduced A in the variable Ared, and the          */
      /* modified b in bred                                         */
j=-1;
for (i=0; i<N; i++)
  if (RowElim[i] == 0)
      {                                                                  700
      j++;
      bred->p[j][0]=btemp[i];
      m=-1;
      for (k=0; k<M; k++)
         if (ColElim[k] == 0)
            {
            m++;
            Ared->p[j][m] = Aorig->p[i][k];
            }
      }                                                                  710

mat_free(Atemp);
mat_free(Atrans);
}



      /*****************************************************************/
      /* Check that after reducing the A, not all requested            */  720
      /* workspace motions are zero (trivial motion).                  */
      /*****************************************************************/
int CheckB      (struct MATRIX *b, int m, float SMALL)
{
int i;

i=0;
      /* stop checking b when a nonzero element is found */
while ((i<m) && (fabs(b->p[i][0]) < SMALL))
   i++;                                                                  730

if (i==m)
   return(0);
else
   return(m);
}


      /*****************************************/
      /* Print the A and b vector to file      */                        740
      /*****************************************/
PrintA  (FILE *check, struct MATRIX *A, struct MATRIX *b, int m, int n)      PrintA
```

```
{
int i, j;

fprintf(check, "       A\n\n");

for (i=0; i<m; i++)
   {
   for (j=0; j<n; j++)                                                    750
      fprintf(check, "%7.4f ", A->p[i][j]);
   fprintf(check, "\n");
   }

fprintf(check, "\n\n       b\n\n");

for (i=0; i<m; i++)
   fprintf(check, "%7.4f ", b->p[i][0]);

fprintf(check, "\n\n\n\n");                                               760
}


     /*********************************************************************/
     /* Create the look-up table which holds the order in which the       */
     /* columns of the A should be blocked to form square sub-As          */
     /*********************************************************************/
Set_Ordering      (int *Ordering, int Mred, int Nred)                  Set_Ordering
{
int temp[Mred-Nred],                                                      770
    i, j, k,
    hold;

for (i=0; i<(Mred-Nred); i++)
   temp[i]=Ordering[i];

for (i=(Mred-Nred); i>0; i--)
   for (j=0; j<(i-1); j++)
      if (temp[j]>temp[j+1])
         {                                                                780
         hold = temp[j+1];
         temp[j+1] = temp[j];
         temp[j]=hold;
         }

k=Mred-Nred-1;
for (i=0; i<temp[0]; i++)
   {
   k++;
   Ordering[k]=i;                                                         790
   }

for (i=0; i<(Mred-Nred-1); i++)
   for (j=temp[i]+1; j<temp[i+1]; j++)
      {
```

```
        k++;
        Ordering[k]=j;
        }

for (i=(temp[Mred-Nred-1]+1); i<Mred; i++)                              800
    {
    k++;
    Ordering[k]=i;
    }
}


        /*************************************************************/
        /* check that the original b is in the range of the A        */
        /*************************************************************/  810
int CheckRange  (struct MATRIX *b, struct MATRIX *Aorig, struct MATRIX *X,
                 int *RowElim, int Mred)
{
int i, j, k;
float CheckValue,
        temp[Aorig->cols];

for (i=0; i<(Aorig->rows); i++)
    if (RowElim[i] == 2)
        {                                                               820
        for (j=0; j<Aorig->cols; j++)
            temp[j]=X->p[j][0];
        CheckValue = ROW_DOT_PRODUCT(Aorig->p[i], temp, Aorig->cols);
        if (fabs(b->p[i][0]-CheckValue) > fabs(b->p[i][0]/10))
            return(0);
        }

return(1);
}
                                                                        830

      /*************************************************************/
      /* This procedure calculates the final solution based on     */
      /* the least norm                                            */
      /*************************************************************/
LeastNorm       (struct MATRIX *X, struct MATRIX *g, int Mred, int Nred,
                 struct MATRIX *Xelim, int *ColElim)
{
struct MATRIX *G,       /* Grammian formed of solution vectors       */
              *t,       /* weighting factors, one for each vector    */  840
              *x,       /* dummy variable for calculations           */
              *e,       /* vertical vector of ones                   */
              *eT;      /* horizontal vector of ones                 */
float   Xtemp[M],       /* used to replace eliminated X components   */
        denominator;    /* used in calculation of X                  */
int i, j, k, M;

    M = g->cols;
```

```
    t = mat_malloc(Mred-Nred+1,1);                                          850
    x = mat_malloc(Mred-Nred+1,1);
    e = mat_malloc(Mred-Nred+1,1);
    eT= mat_malloc(1,Mred-Nred+1);
    G = mat_malloc(Mred-Nred+1,Mred-Nred+1);

    for (i=0; i<(Mred-Nred+1); i++)     /* initialize the vectors of ones */
        {
        e->p[i][0]=1.0e0;
        eT->p[0][i]=1.0e0;
        }                                                                   860

    for (i=0; i<(Mred-Nred+1); i++)
        for (j=0; j<(Mred-Nred+1); j++)
            G->p[i][j]=ROW_DOT_PRODUCT(g->p[i], g->p[j], Mred);

    mat_pseudoinv(G);
    t=mat_mul2(G,e);
    x=mat_mul2(eT,G);
    x=mat_mul2(x,e);
    denominator=(1/x->p[0][0]);                                             870
    mat_sca(t,denominator);

    for (i=0; i<M; i++)
        Xtemp[i] = 0.0e0;
    for (i=0; i<Mred; i++)
        for (j=0; j<(Mred-Nred+1); j++)
            Xtemp[i] += (t->p[j][0])*(g->p[j][i]);

    k = 0;
    for (j=0; j<M; j++)                                                     880
        if (ColElim[j] == 1)
            X->p[j][0]=Xelim->p[j][0];
        else
            {
            X->p[j][0]=Xtemp[k];
            k++;
            }

    mat_free(eT);
    mat_free(e);                                                            890
    mat_free(G);
    mat_free(t);
    mat_free(x);
}


    /*********************************************************************/
    /* Least norm solution for case in which A has lost row rank         */
    /*********************************************************************/
RankLostSoln    (struct MATRIX *g, struct MATRIX *X, struct MATRIX *A,      900
                 int Mred, int Nred, struct MATRIX *Xelim,
```

```
                int *ColElim, int *RowElim, struct MATRIX *b, FILE *check)
{
struct MATRIX *G,
              *e,
              *eT,
              *x,
              *beta,
              *betaT,
              *t;                                                    910
double a, bscalar, c, d, Ascalar, num, sum;
double nhu, mu, Xtemp[M];
int i, j, k, M;

   M = g->cols;

   t = mat_malloc(Mred-Nred+1,1);
   x = mat_malloc(Mred-Nred+1,1);
   e = mat_malloc(Mred-Nred+1,1);
   eT= mat_malloc(1,Mred-Nred+1);                                     920
   G = mat_malloc(Mred-Nred+1,Mred-Nred+1);
   beta = mat_malloc(Mred-Nred+1,1);
   betaT = mat_malloc(1,Mred-Nred+1);

   for (i=0; i<(Mred-Nred+1); i++)     /* initialize the vectors of ones */
      {
      e->p[i][0]=1.0e0;
      eT->p[0][i]=1.0e0;
      }
                                                                      930
                                       /* construct Grammian and inverse */
   for (i=0; i<(Mred-Nred+1); i++)
      for (j=0; j<(Mred-Nred+1); j++)
         G->p[i][j]=ROW_DOT_PRODUCT(g->p[i], g->p[j], Mred);
   mat_pseudoinv(G);
   fprintf(check, "\n\nGrammian inverse:\n");
   for (i=0; i<(Mred-Nred+1); i++)
      {
      for (j=0; j<(Mred-Nred+1); j++)
         fprintf(check, "%10.4f ", G->p[i][j]);                       940
      fprintf(check, "\n");
      }


                                       /* calculate beta values */
   j=0;
   while (RowElim[j] != 2)
      j++;
   for (i=0; i<(Mred-Nred+1); i++)
      {                                                               950
      num = ROW_DOT_PRODUCT(A->p[j], g->p[i], M);
      beta->p[i][0] = num/b->p[j][0];
      betaT->p[0][i] = beta->p[i][0];
      }
```

```
fprintf(check, "\nbeta values:\n");
for (i=0; i<(Mred-Nred+1); i++)
    fprintf(check, "%7.4f ", beta->p[i][0]);
fprintf(check, "\n");

x = mat_mul2(eT,G);
x = mat_mul2(x,e);
a = x->p[0][0];
x = mat_mul2(eT,G);
x = mat_mul2(x,beta);
bscalar = x->p[0][0];
x = mat_mul2(G,e);
x = mat_mul2(betaT,x);
c = x->p[0][0];
x = mat_mul2(G,beta);
x = mat_mul2(betaT,x);
d = x->p[0][0];

Ascalar = c*bscalar - a*d;

nhu = (a-c)/Ascalar;

mu = - (1+nhu*bscalar)/a;

fprintf(check, "\nscalar values:\n");
fprintf(check, "a: %7.4f  b: %7.4f  c: %7.4f  d: %7.4f  \n", a, bscalar, c, d);
fprintf(check, "A: %7.4f  mu: %7.4f  nhu: %7.4f\n", Ascalar, mu, nhu);

t = mat_mul2(G,e);
mat_sca(t,mu);
x = mat_mul2(G,beta);
mat_sca(x,nhu);
for (i=0; i<(Mred-Nred+1); i++)
    t->p[i][0] = -t->p[i][0] - x->p[i][0];

fprintf(check, "\nt values: \n");
for (i=0; i<(Mred-Nred+1); i++)
    fprintf(check, "%7.4f ", t->p[i][0]);
fprintf(check, "\n");
sum = t->p[0][0]+t->p[1][0]+t->p[2][0]+t->p[3][0]+t->p[4][0]+t->p[5][0];
fprintf(check, "sum:  %7.4f\n", sum);

sum = 0;
for (i=0; i<(Mred-Nred+1); i++)
    sum += beta->p[i][0]*t->p[i][0];

fprintf(check, "\nsum betai*ti:  %7.4f \n", sum);

for (i=0; i<M; i++)
    Xtemp[i] = 0.0e0;
for (i=0; i<Mred; i++)
    for (j=0; j<(Mred-Nred+1); j++)
```

```
            Xtemp[i] += (t->p[j][0])*(g->p[j][i]);

    k = 0;                                                          1010
    for (j=0; j<M; j++)
        if (ColElim[j] == 1)
            X->p[j][0]=Xelim->p[j][0];
        else
            {
            X->p[j][0]=Xtemp[k];
            k++;
            }

    mat_free(eT);                                                   1020
    mat_free(e);
    mat_free(G);
    mat_free(t);
    mat_free(x);
    mat_free(beta);
    mat_free(betaT);
}


                                                                    1030
    /*************************************************************/
    /* read in A and b vector to be used for testing the code    */
    /*************************************************************/
GetData (struct MATRIX *A, struct MATRIX *b)                        GetData
{
FILE *fpin;
int i, j;

fpin = fopen(infile, "r");
                                                                    1040
for (i=0; i<N; i++)
   for (j=0; j<M; j++)
       fscanf(fpin, "%f", &A->p[i][j]);

for (i=0; i<N; i++)
   fscanf(fpin, "%f", &b->p[i][0]);

fclose(fpin);
}
                                                                    1050
```

```
/*****************************************************************/
/* This is the header file for the matrix procedures that        */
/* are necessary for FSP.c.  The procedure files and header      */
/* files from Numerical Recipes in C are necessary in order      */
/* for the code to find everything that it needs.  The           */
/* following procedures were taken from a file created by        */
/* Ole H. Dorum at Oak Ridge National Laboratory:                */
/*      mat_malloc                                               */
/*      mat_mul2                                                 */
/*      mat_free                                                 */
/*      mat_sca                                                  */
/*      mat_pseudoinv                                            */
/*      mat_cp2                                                  */
/*      mat_pr                                                   */
/*****************************************************************/


struct  MATRIX *mat_malloc();  /* Matrix allocation                */
struct  MATRIX *mat_mul2();    /* mult two matrices and return result */
struct  MATRIX *mat_cp2();     /* copy one matrix to another       */

void    svdcmp();              /* Singular Value Decomposition     */
void    mat_free();            /* Free matrix pointer              */
void    mat_sca();             /* Scale a matrix by a factor       */
void    mat_pseudoinv();       /* Inverse or pseudoinverse using SVD */
void    mat_null();            /* Find the matrix null space       */
void    mat_pr();              /* print a matrix                   */

struct MATRIX
{
  float **p;     /* Pointer to array of pointers to matrix rows.  */
  int   rows;    /* Row dimension of the matrix.                  */
  int   cols;    /* Column dimension of the matrix.               */
};
```

```
/*********************************************************
*
* This file contains the matrix procedures necessary
* to run the program FSP.c.  Please see the header file
* for more information.
*
*********************************************************/

#include <stdio.h>
#include <stdlib.h>                                                    10
#include <math.h>

#include "matrix.h"

#define SVD_THRESHOLD 1.0e-6
#define SV_SMALL 1.0e-4

#define SWAP(a,b) {float temp = (a); (a) = (b); (b) = temp;}
#define MAX(a,b) ((a) > (b) ? (a) : (b))
#define PrintIfBiggerThan 0                                            20
#define SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -fabs(a))

static float at,bt,ct;
#define PYTHAG(a,b) ((at=fabs(a)) > (bt=fabs(b)) ? \
(ct=bt/at,at*sqrt(1.0+ct*ct)) : (bt ? (ct=at/bt,bt*sqrt(1.0+ct*ct)): 0.0))


/*************************************************************************/

                                                                       30

struct MATRIX *mat_malloc(rows, cols)
int rows, cols;
{
  int     i;
  float   *mat, **row;
  struct MATRIX *matrix;

  /* Allocate space for structure, elements and pointers.
   *                                                                   40
   * Note, that the allocated number of row pointers is MAX(row, cols)
   * because it facilitates transposing rectangular matrices.
   */
  mat = (float * ) malloc(rows * cols * sizeof(float));
  row = (float **) malloc((MAX(rows, cols)) * sizeof(float *));
  matrix = (struct MATRIX *) malloc( sizeof( struct MATRIX ) );

  if(!mat || !row || !matrix)
    {
      fprintf(stderr, "Matrix allocation failed\n");                   50
      return(NULL);
    }
```

```
  matrix->p = row;
  matrix->rows = rows;
  matrix->cols = cols;

  /* The Nth element of the array row points to the 1st element
   * on the Nth row. Thus, **m = *m[0] = m[0][0]
   *                                                              60
   * Calculate the addresses of the pointers pointing to the
   * rows of the matrix
   */

  for(i = 0; i < rows; i++)
  {
    row[i] = mat;
    mat += cols;
  }
                                                                  70
  return(matrix);
}


  /*************************************************************************/
  /* The 'c' matrix must already be declared float of size: arows x bcols. */
  /*************************************************************************/
void mat_mul(a, b, c)
struct MATRIX *a, *b, *c;
{                                                                 80
  int i, j, k;

  for (k = 0; k < b->cols; k++)
    for (i = 0; i < a->rows; i++)
    {
      c->p[i][k] = 0;
      for (j = 0; j < a->cols; j++)
        c->p[i][k] += a->p[i][j]*b->p[j][k];
    }
}                                                                 90

  /*************************************************************************/
  /* The function will automatically create the result matrix of correct */
  /* dimension. The pointer to this matrix structure is returned.        */
  /*************************************************************************/
struct MATRIX *mat_mul2(a, b)
struct MATRIX *a, *b;
{
  struct MATRIX *c;
                                                                  100
  c = mat_malloc(a->rows, b->cols);

  mat_mul(a, b, c);

  return(c);
}
```

```
  /*********************************************************************/
  /* The pseudoinverted matrix will still reside in a. In the case of a */
  /* square matrix, the result will actually be the inverted matrix. If */
  /* the matrix is rectangular, the pseudoinverse will have the correct */
  /* dimension.                                                         */
  /*********************************************************************/
void mat_pseudoinv(a)
struct MATRIX *a;
{
  struct MATRIX *q2, *z, *tmp;
  struct MATRIX *q1t, *q2t, *zt, *qq, *qq2;
  float  **A, **Q2, *Z;
  float  z_min, z_max;
  int    M, N, i, j;
  float  **conv_2_nric_ptr();

  M = a->rows; N = a->cols;

  /* Allocate space for q2 matrix, vector Z[1..N], z and tmp
   */
  q2 = mat_malloc(N, N);
  z  = mat_malloc(N, N);
  Z  = vector(1, N);

  /* Create pointers to a and q2 conforming with Num-Rec-in-C format.
   */
  A  = conv_2_nric_ptr(a);
  Q2 = conv_2_nric_ptr(q2);

  /* Compute A[1..M][1..N]'s singular value decomposition (SVD): A = Q1*Z*Q2_tra
   *
   * Q1  will replace A, and the diagonal value of singular values Z is output
   * as a vector Z[1..N]. The matrix Q2 (not the transpose Q2_tra) is output
   * as Q2[1..N][1..N]. M must be greater than or equal to N; If it is smaller,
   * then A should be filled up to square with zero rows.
   */
  svdcmp(A, M, N, Z, Q2);

  /* (Singular values = squareroot of the eigenvalues), find maximum.
   */
  z_max = 0.0;
  for (i = 1; i <= N; i++) if (Z[i] > z_max) z_max = Z[i];

  /* Set threshold value of the minimum singular value allowed
   * to be nonzero.
   */
  z_min = z_max*SVD_THRESHOLD;

  /* Invert while copying from the Z vector into the z+ matrix, and weed out
   * the too small singular values.
   */
```

```
  for (i = 0; i < N; i++)                                                    160
    for (j = 0; j < N; j++)
      z->p[i][j] = ((i == j) && Z[i+1] > z_min) ? 1.0/Z[i+1] : 0.0;
     /* z->p[i][j] = 1.0/Z[i+1];*/


  /*
   * A_pseudoinv = Q2 * Z_pseudoinv * Q1_tra
   *
   * Returned matrix A from svdcmp() is actually Q1, therefore:
   */                                                                        170
 /* printf("Testing the inverse:\n");*/
  q1t = mat_tra2(a);
  qq = mat_mul2(a,q1t);
  /*printf("Q1t*q1 = \n");
  mat_pr(qq);*/


  mat_tra(a);                /* Transpose Q1 to Q1_tra                    */
  tmp = mat_mul2(z, a);       /* tmp = Z_pseudoinv * Q1_tra                 */
  mat_mul(q2, tmp, a);       /* A_pseudoinv = Q2 * tmp = Q2 * Z_pseudo * Q1_tra */   180

  q2t = mat_tra2(q2);
  qq2 = mat_mul2(q2,q2t);
/*  printf("Q2t*Q2 = \n");
  mat_pr(qq2);*/



  mat_free(q2);
  mat_free(z);                                                               190
  mat_free(tmp);
  free_vector(Z, 1, N);
  free( (char *) A);
  free( (char *) Q2);

}


  /************************************************/
  /* Return the null space of a matrix using svd */                          200
  /************************************************/
void mat_null(a, n_rank, n, K2)
struct MATRIX *a, *n;
int *n_rank;
float *K2;
{
  struct MATRIX *a_sqr,
                *v,
                *S_temp;
                                                                             210
  float **U, *S, **V,
        *vector(),
```

```
        **conv_2_nric_ptr();

float s_min, s_max, temp;

void free_vector(),
     free_ivector();

int i, j, R, C, *order;                                          220
int *ivector();

R = a->rows;
C = a->cols;

v = mat_malloc(C,C);
V = conv_2_nric_ptr(v);
S = vector(1,C);
S_temp = mat_malloc(C+1,1);
order = ivector(1,C);                                            230

if (R < C)
  {
  a_sqr = mat_malloc(C,C);
  for (i=0; i<R; i++)
    for (j=0; j<C; j++)
      a_sqr->p[i][j] = a->p[i][j];
  for (i=R; i<C; i++)
    for (j=0; j<C; j++)
      a_sqr->p[i][j] = 0.0;                                      240
  }
else
  {
  a_sqr = mat_malloc(R,C);
  a_sqr = mat_cp2(a);
  }

U = conv_2_nric_ptr(a_sqr);

svdcmp(U,C,C,S,V);                                               250

for (i=1; i<(C+1); i++)
  {
  S_temp->p[i][0] = S[i];
  order[i] = i;
  }
for (i=C; i>1; i--)
  for (j=i-1; j>=1; j--)
    if (fabs(S_temp->p[i][0])>fabs(S_temp->p[j][0]))
      {                                                          260
      temp = S_temp->p[i][0];
      S_temp->p[i][0] = S_temp->p[j][0];
      S_temp->p[j][0] = temp;
      temp = order[i];
      order[i] = order[j];
```

```
        order[j] = temp;
        }
  if (R<=C)
    s_min = fabs(S_temp->p[R][0]);
  else                                                          270
    s_min = fabs(S_temp->p[C][0]);
  s_max = fabs(S_temp->p[1][0]);

  *n_rank = 0;
  while (fabs(S_temp->p[C-*n_rank][0]) < SV_SMALL)
    {
    for (j=1; j<=C; j++)
      n->p[j-1][*n_rank] = V[j][order[C-*n_rank]];
    *n_rank = *n_rank + 1;
    }                                                           280

  if ((fabs(s_min)>=SV_SMALL) && (fabs(s_min/s_max) < SV_SMALL))
    {
    for (j=1; j<=C; j++)
      n->p[j-1][*n_rank] = V[order[j]][C-*n_rank];
    *n_rank = *n_rank+1;
    }

  *K2 = s_max/s_min;
                                                              290
  mat_free(a_sqr);
  free_vector(S,1,C);
  mat_free(S_temp);
  mat_free(v);
  free((char *) V);
  free((char *) U);
}


  /*********************************************/                300
  /* The matrix 'a' is scaled by the factor 'c' */
  /*********************************************/
void mat_sca(a, c)
struct MATRIX *a;
float  c;
{
  float *p;
  int   i;

  p = *a->p;                                                    310

  for (i = a->rows*a->cols; i--;)
    *p++ *= c;
}


  /*************************************************************/
  /* The copied matrix 'cpy' must already be declared float of */
```

```
  /* same size as 'a'.                                          */
  /************************************************************/
void mat_cp(a, cpy)
struct MATRIX *a, *cpy;
{
  float *p, *q;
  int   i;

  p = *a->p; q = *cpy->p;

  for (i = a->rows*a->cols; i--;)
    *q++ = *p++;
}


  /*************************************************************************/
  /* The function will automatically create the result matrix of correct   */
  /* dimension which will be the transpose of A. The pointer to this matrix */
  /* structure is returned.                                                */
  /*************************************************************************/
struct MATRIX *mat_tra2(A)
struct MATRIX *A;
{
  int   i, j;
  struct MATRIX *At;

  At = mat_malloc(A->cols, A->rows);

  for (i = A->rows-1; i--;)
    for (j = A->cols-1; j > i; j--)
      At->p[i][j] = A->p[j][i];

  return(At);
}


  /*********************************************************************/
  /* The transposed matrix still resides in 'a' after transposition */
  /*********************************************************************/
void mat_tra(a)
struct MATRIX *a;
{
  int   i, j, temp;
  float *p;

  if (a->rows == a->cols)              /* Square matrix */
  {
    for (i = a->rows-1; i--;)
      for (j = a->cols-1; j > i; j--)
        SWAP(a->p[i][j], a->p[j][i]);
  }
  else                                 /* Rectangular matrix */
  {
```

```
    temp = a->rows;
    a->rows = a->cols;
    a->cols = temp;

    /* Recompute pointers to the new rows of the matrix
     */
    p = *a->p;
    for(i = 0; i < a->rows; i++)
    {                                                          380
      a->p[i] = p;
      p += a->cols;
    }

    /* No need to swap elements if the matrix has only one row or column.
     */
    if (!((a->rows == 1) || (a->cols == 1)))
    {
      struct MATRIX *m;
                                                               390
      /*...otherwise put elements in temporary matrix and
       * copy from it into the columns of the transposed matrix.
       */
      m = mat_malloc(a->rows, a->cols);
      mat_cp(a,m);
      p = *m->p;

      for (j = 0; j < a->cols; j++)
        for (i = 0; i < a->rows; i++)
          a->p[i][j] = *p++;                                   400

      mat_free(m);
    }
  }
}


  /*************************************************************************/
  /* The function will automatically create the result matrix of correct */
  /* dimension . The pointer to this matrix structure is returned.      */  410
  /*************************************************************************/
struct MATRIX *mat_cp2(a)
struct MATRIX *a;
{
  int   i;
  float *p, *q;
  struct MATRIX *cpy;

  cpy = mat_malloc(a->rows, a->cols);
                                                               420
  mat_cp(a, cpy);

  return(cpy);
}
```

```
  /*************************************************************************/
  /* Returns an pointer which points to an array of pointers to matrix row */
  /* elements in a way so that the matrix a->p[0..n][0..m] can be accessed */
  /* from [1..n-1][1..m-1] which is required by the Numerical Rec. i C.    */
  /*************************************************************************/
float **conv_2_nric_ptr(a)
struct MATRIX *a;
{
  float **new_ptr_2_row;
  int    i;

  /* Allocate space for an array of OFFSET pointers.
   */
  new_ptr_2_row = (float **) malloc((a->rows + 1) * sizeof(float *));

  /* The array of pointers to row must be offset by -1
   */
  for (i = 0; i < a->rows; i++)
    new_ptr_2_row[i+1] = a->p[i] - 1;

  /* To not have element[0][] dangling, let it point to a->p[0][0].
   */
  new_ptr_2_row[0] = a->p[0];

  return( new_ptr_2_row );
}


void mat_free(m)
struct MATRIX *m;
{
  free( (char *) *m->p);
  free( (char *)  m->p);
  free( (char *)  m);
}


void mat_prf(a)
struct MATRIX *a;
{
  int i, j;

  printf("\n");
  for (i = 0; i < a->rows; i++)
  {
    for (j = 0; j < a->cols; j++)
    {
      printf("\t% f", a->p[i][j]);
      }
    printf("\n");
```

```
  }
  printf("\n");
}

void mat_pr(a)
struct MATRIX *a;
{
  int i, j;

  printf("\n");
  for (i = 0; i < a->rows; i++)
  {
    for (j = 0; j < a->cols; j++)
      {
        if (abs(a->p[i][j]) >= 0.005)
          printf("\t% .3f", a->p[i][j]);
        else
          printf("\t  - ");
      }
    printf("\n");
  }
  printf("\n");
}
```

INTERNAL DISTRIBUTION

1. E. C. Bradley
2. B. L. Burks
3. J. B. Chesser
4. W. E. Dixon
5. J. V. Draper
6. V. B. Graves
7. D. C. Haley
8. W. R. Hamel
9. J. N. Herndon
10. J. F. Jansen
11. R. L. Kress
12. C. T. Kring
13. D. S. Kwon
14. E. D. Miller
15. M. W. Noakes
16. C. E. Oliver

17-21. F. G. Pin

22. K. E. Plummer
23. A. H. Primm
24. B. S. Richardson
25. S. L. Schrock
26. S. Shekhar
27. D. M. Speaks
28. K. U. Vandergriff
29. V. K. Varma
30. B. S. Weil
31. H. R. Yook

32-33. Laboratory Records

34. Laboratory Recored-ORNL RC
35. RPSD Publications Office
36. ORNL Patent Section
37. Central Research Library
38. Document Reference Section

EXTERNAL DISTRIBUTION

39. Dr. John Baillieul, Boston University, Department of Aerospace and Mechanical Engineering, 110 Cummington Street, Boston, Massachusetts 02215.

40. Dr. Matthew Berkemeir, Boston University, Department of Aerospace and Mechanical Engineering, 110 Cummington Street, Boston, Massachusetts 02215.

41. Dr. Roger Brockett, Harvard University, Harvard Robotics Laboratory, Pierce Hall, 29 Oxford Street, Cambridge, Massachusetts 02138.

42. Capt. Brian Cassiday, Robotics and Automation Center of Excellence, San Antonio Air Logistics Center, Kelly Air Force Base, Texas 78241.

43. Dr. David Castanon, Boston University, Department of Electrical, Computer, and Systems Engineering, 44 Cummington Street, Boston, Massachusetts 02215.

44. Dr. Pierre Dupont, Boston University, Department of Aerospace and Mechanical Engineering, 110 Cummington Street, Boston, Massachusetts 02215.

45. Dr. Michael Gevelber, Boston University, Department of Manufacturing Engineering, 44 Cummington Street, Boston, Massachusetts 02215.

46. Dr. Michael Leahy, 1715 Hamlet Court, Montgomery, Alabama 36117-1757.

47. S. R. Martin, Jr., Acting Program Manager, Fusion and Nuclear Technology Branch, Energy Programs Division, Department of Energy, X-10 Site, Post Office Box 2008, Oak Ridge, Tennessee 37831-6269.

48-52. Kristi Morgansen, Harvard Robotics Laboratory, Pierce hall, 29 Oxford Street, Harvard University, Cambridge, Massachusetts 02138.

53. Dr. Ann Stokes, Boston University, Department of Aerospace and Mechanical Engineering, 110 Cummington Street, Boston, Massachusetts 02215.

54. SMSG Tom Turner, AFSEO/SKZ, 207 W. D. Avenue, Suite 303, Eglin Air Force Base, Florida 32542.

55. Office of Assistant Manager for Energy Research and Development, Oak Ridge Operations Office, Department of Energy, Post Office Box 2008, Oak Ridge, Tennessee 37831-6269.

56. Office of Scientific and Technical Information, Post Office Box 62, Oak Ridge, Tennessee 37831.