A Study of Convergence of the PMARC Matrices applicable to WICS Calculations

by

Amitabha Ghosh
Department of Mechanical Engineering
Rochester Institute of Technology
Rochester, NY 14623

Final Report
NASA Cooperative Agreement No: NCC 2-937

Presented to

NASA Ames Research Center
Moffett Field, CA 94035

August 31, 1997

Table of Contents

Abstract

This report discusses some analytical procedures to enhance the real time solutions of PMARC matrices applicable to the Wall Interference Correction Scheme (WICS) currently being implemented at the 12 foot Pressure Tunnel. WICS calculations involve solving large linear systems in a reasonably speedy manner necessitating exploring further improvement in solution time. This paper therefore presents some of the associated theory of the solution of linear systems. Then it discusses a geometrical interpretation of the residual correction schemes. Finally some results of the current investigation are presented.

Introduction

WICS is a combined experimental and computational approach to correct for the wall interference effects in wind tunnel testing. The procedure involves replacing the test model by a combination of mathematical singularities. The use of the model force and moment measurements together with the wind tunnel wall pressure measurements facilitate calculation of the unknown strengths of sources and doublets representing the model and the support system The main benefit of the WICS procedure is in the pare-calculation of a database using the PMARC influence coefficient matrices. Often this process takes several days depending upon the number of singularities and panels in the model. Moreover the influence coefficient matrices arising from WICS calculations are neither sparse nor symmetric. Thus exploring a better linear solver would be a worthwhile effort. With this objective in mind, this investigation attempted to answer some of the related issues of the solution of linear systems.

This report re-visits some popular direct and iterative methods of solution of linear systems and systematically compares their performance for solution of large systems. For some time the convergence difficulties of PMARC matrices applicable to WICS calculation was believed to be ill-conditioning of the influence coefficient matrices. Thus the early part of this research was devoted to the study of ill-conditioning. As explained later, this issue was resolved by using double precision arithmetic, after it had also been established that the WICS matrices were indeed well-conditioned. There is no reason to suspect that such issues will arise again in WICS calculations. Thus the present focus is the speed of calculation.

As a preliminary set of geometrical ideas was developed, the interpretation of iterative solution procedure was attempted on small matrices. Preliminary results were not very promising when applied to typically large systems however. Computational efficiency was hampered in large matrices due to a large number of error-accumulating procedures. The tests finally suggested exploring the current linear solver in PMARC [1] in conjunction with the developed geometrical ideas, which resulted in improved performance.

The questions of solving large linear systems will always arise in our applications since internal flow models involving PMARC will typically grid the major portions of a whole wind tunnel. However, the issues arising from supercomputing and parallel processing will not be attempted here since they involve different strategies.

The basic layout of this report covers both theory and computational experiments. The theoretical ideas are substantiated by geometrical interpretations for small matrices. However large matrices involving very large solution data files are not presented as outputs. Instead, their performance is compared with respect to computational run time using double precision arithmetic. These are presented in a tabular fashion and discussed in the results section. The actual fortran programs that were developed under this project are available on the *sr71* and *ra-iris* systems at NASA-Ames and the author's home directory at Rochester Institute of Technology. Some sample fortran programs are attached herewith in the appendix.

Solution of Linear Systems

This field of study is very old and practically arises in any area of mathematical interest. The basic techniques of linear algebra are learnt in a variety of ways from high school mathematics through higher levels involving ideas from topology. The characteristic of all linear systems consists of a set of coefficients and unknowns related by a system of equations which never involve any powers of the unknowns more than one, neither do any of the equations involve any products of two unknown quantities. In a basic appearance we shall call such a set of equations an n-dimensional linear system, given by

$$a_{11}\,x_1 + a_{12}\,x_2 + a_{13}\,x_3 + \ldots\ldots\ldots + a_{1n}\,x_n = b_1 \qquad (1.1)$$
$$a_{21}\,x_1 + a_{22}\,x_2 + a_{23}\,x_3 + \ldots\ldots\ldots + a_{2n}\,x_n = b_2 \qquad (1.2)$$
$$a_{31}\,x_1 + a_{32}\,x_2 + a_{33}\,x_3 + \ldots\ldots\ldots + a_{3n}\,x_n = b_3 \qquad (1.3)$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$a_{n1}\,x_1 + a_{n2}\,x_2 + a_{n3}\,x_3 + \ldots\ldots\ldots + a_{nn}\,x_n = b_n \qquad (1.n)$$

In the above system, the right hand terms b_1 through b_n are all known, as well as, any terms involving the letter "a" with subscripts. The right hand known constants are expressible by a column vector **b** of size n, and the unknown variables x_1 through x_n may be expressed by another column vector **x**. Thus the linear system may be expressed in a more compact form by $A\,\mathbf{x} = \mathbf{b}$, such that A is a matrix of size (n x n) [i.e., n rows and n columns].

There are various questions of to answer about the existence of solutions and solvability which involve the matrix A of different numbers of rows and columns. However in our applications database no such cases will arise. Thus we shall always focus on a linear system that poses a unique solution and all the solutions arrived at by various means will involve real numbers. Also our system of

equations will be non-homogeneous, i.e., the vector **b** will never be a null vector. When a homogeneous system of n linear equations are solved, the determinant of A must be zero. Although, such systems involving eigenvalues and eigenvectors will be discussed later, the main thrust of the solution will involve numerical procedures adopted for non-homogeneous systems.

The focus of this investigation is in developing and understanding computational approaches. Thus traditional solutions of non-homogeneous linear systems given by Cramer's rule, cofactors and evaluation of determinants will be omitted here since these techniques are very expensive computationally. The traditional direct solvers include the Gaussian elimination technique, Gauss-Jordan technique and the L-U decomposition technique.

Direct Solvers

Gaussian Elimination:

In this approach the idea is to triangularize the matrix A through a set of algebraic reductions such that the resulting matrix A' is a strictly upper triangular form. There are different ways to achieve this. The most popular technique yields the diagonal elements of A' as one. The corresponding right hand column vector **b'** after the reductions are utilized in a back substitution process to solve for the unknown vector **x.** The reason Gaussian elimination is claimed as a direct solver is because, with sufficient accuracy, the calculations need to be performed only once. With single precision arithmetic however, on some limited memory computers, the calculations may need to be iterated using an error equation technique.

Although the process of Gaussian elimination is simple, it does not involve the least amount of calculational efforts. It may be shown that the calculations involved in such processes are of the order of n^3 operations. There are some associated computational questions to produce robustness in such calculations for a general matrix. These involve rearranging the equations so that the diagonal elements of A are always the largest. These are called the row pivoting and the column pivoting operations. The examples quoted for comparisons later in this report always involved a partial row pivoting strategy (for example, see the 9 x 9 truss problem later).

Gauss-Jordan Elimination:

This is a step further from the Gaussian elimination process. In this method the eliminations of the coefficients are carried out not only below the diagonal but above the diagonal also. The advantage of this method is that the reduced matrix A' is a diagonal one, eliminating the need for the back substitution process, which is a characteristic of the previous technique. The choice to use the Gaussian elimination or the Gauss-Jordan elimination is a personal one since the associated procedures involve the same amount of computational efforts. Thus Gauss-Jordan technique is not claimable as an improvement over the Gaussian elimination process. For example,

the subroutine *gss* in the current solver in PMARC is a Gauss Jordan one.

L-U Decompostion:

The ideas of Gaussian elimination is extended another step in the L-U decomposition process. There are different authors for such methods (for example Crout's method, Doolittle's method, etc).The idea is to obtain a simultaneous upper and lower triangularization of the original coefficient matrix A, such that A = L.U. There is again no direct advantage of adopting this procedure for computational efficiency over the Gaussian elimination technique. However such decomposition ideas are worthwhile to understand the direct and iterative numerical techniques presented later. Thus although no separate consideration of this technique will be presented in this report, note that the QR decomposition technique quoted later to check the ill-conditioning of the PMARC coefficient matrices involved such triangularization ideas.

In summary, the direct solvers available in literature today are all variations of the Gaussian elimination process. There are several different methodology to obtain solutions of linear systems by direct solvers. However since all direct solvers involve associated computations with the whole coefficient matrix A, there is no apparent computational benefit over the original Gaussian elimination, since all involve n^3 arithmetic operations.

Iterative Solvers

Iterative solvers became more popular with the advent of high speed computers. There are several benefits of choosing an iterative solver over a direct solver. Simple iterative solvers invariably reduce the programming efforts. However the more sophisticated ones may involve considerable amount of complex programming since they can act as a black box for the end user. The development of a robust calculation procedure has its roots in the linear algebraic techniques far beyond the reach of the end user. We however will develop some geometrical approaches here in an effort to understand the basic iterative techniques. There is a difficulty in presenting geometrical ideas beyond three dimensions. The visualization handicap will be supplemented by algebraic reasoning. The sections below present the basic and the more recent iterative solution procedures.

To give an example of an iterative process, let us assume an equation: $ax^2 + bx + c = 0$, where, a, b and c are constants and x is the unknown variable. We wish to determine the unknown variable x by repetitive calculations called iterations. Note that the equation's solution can be determined by the quadratic formula $x = \{-b \pm \sqrt{(b^2 - 4\,a\,c)}\}/(2a)$. However that will be considered a direct analytic solution. Instead an iterative procedure may be set up by casting the above equation into the following form $x(ax+b) = -c$, followed by, $x = -c/(ax+b)$. Thus a new value of x may be determined from a guessed value of x by the last equation. This is formalized by writing the

equation as $x^{(k+1)} = -c/(ax^{(k)} + b)$, where (k) represents the iteration number. In this process, the guessed solution will be changed and hopefully will be converged to the analytic solution of the equation. The theory behind solution of linear equations by iterative processes involves study of the procedures and errors associated with iterations and whether or, how quickly the calculated solution can be converged to the analytic solution. Note that contrary to this non-linear example, our system of equations is linear and associated unknown **x** is a vector with n components. Given below are the popular iterative solvers for linear systems.

Jacobi Method:

This technique is the most basic form of relaxation methods and is listed here for reference. The actual procedures adopted in this report did not employ this except to calculate part of the solution in the successive over-relaxation scheme. The point Jacobi technique involves decomposing A as follows.

The matrix A may be decomposed into a diagonal matrix D, an upper triangular U and a lower triangular matrix L. Then the resulting linear system may be written as

$$D\,\mathbf{x}^{(k+1)} = -(L + U)\,\mathbf{x}^{(k)} + \mathbf{b}\,,$$

where, (k) is the iteration number.

Thus the solution of the point Jacobi scheme may be sought using

$$\mathbf{x}^{(k+1)} = -\,D^{-1}\,(L + U)\,\mathbf{x}^{(k)} + D^{-1}\,\mathbf{b}$$

In the above equation, the matrix, $T_J = -\,D^{-1}\,(L + U)$, is called the associated point Jacobi iteration matrix of the linear system $A\,\mathbf{x} = \mathbf{b}$. Note that the second matrix $D^{-1}\,\mathbf{b}$ of the right hand side of the above equation is a known static vector, which will not change during the solution process. Whether the solution of the point Jacobi method converges to the analytical solution of the linear system depends on the norm of the iteration matrix being less than one. It may be shown that this condition on the norm is satisfied if the coefficient matrix A is strictly diagonally dominant.

The speed of calculation in a point Jacobi scheme is slow compared to the processes discussed below since the iteration vector $\mathbf{x}^{(k+1)}$ gets modified only once in each iteration.

Gauss-Seidel Method:

If the same decomposition that was used in the point Jacobi scheme is organized a little differently, the speed of calculations can be considerably improved. This method upgrades the vector **x** continuously within a single iteration using the most recent values of the components of **x** that are

available. The resulting scheme is called the point Gauss-Seidel scheme, where,

$$\mathbf{x}^{(k+1)} = - (D + L)^{-1} (U) \mathbf{x}^{(k)} + (D + L)^{-1} \mathbf{b}$$

where, the iteration matrix is $T_{GS} = - (D + L)^{-1} (U)$.

The point Gauss-Seidel scheme is convergent if and only if $\|T_{GS}\| \leq 1$. This condition may again be guaranteed if the matrix A is strictly diagonally dominant. The Stein-Rosenberg theorem relates the convergence and the matrix norms of the Jacobi and Gauss-Seidel iteration matrices [2].

Successive Over-relaxation Method:

The best speed of calculations utilizing the ideas of the point Jacobi and Gauss-Seidel schemes may be derived by a linear combination of those two solutions. The resulting scheme is called the point successive over-relaxation scheme (or, simply S.O.R scheme). Here the linear combination of the point Jacobi solution, $\mathbf{x}_J$ and the point Gauss-Seidel solution, $\mathbf{x}_{GS}$ is achieved to yield $\mathbf{x}_{SOR}$ as

$$\mathbf{x}_{SOR} = \omega \mathbf{x}_{GS} + (1 - \omega) \mathbf{x}_J$$

In the above equation, ω is called the relaxation parameter. It ranges in values typically between 0 and 2. However, higher values are possible to yield a convergent solution. When the value of ω is less than 1 the process is called under-relaxation and when ω is greater than one the process is called over-relaxation. Although ω is typically a constant, it may be chosen as a variable too. In each problem an optimum value of the relaxation parameter may be found analytically as well as by performing computational experiments. The iteration matrix of the point S.O.R scheme is given by

$$T_{SOR} = - (D + \omega L)^{-1} [\omega \mathbf{L} + (1 - \omega) \mathbf{U}] .$$

Using the substitutions of $P = D^{-1} L$ and $Q = D^{-1} U$, we may re-write the above as

$$T_{SOR} = - (I + \omega P)^{-1} [Q + (1 - \omega) \mathbf{I}]$$

where, I s the identity matrix of size n x n.

Two important theorems by Ostrowski and Reich relate the iteration matrices of Gauss-Seidel and S.O.R. processes [2]. The important condition which guarantees convergence of the above matrices in the complex space is that the component matrices L, U and D be Hermitian and positive definite. In the applications of our interest, a real, symmetric and diagonally dominant matrix A would meet all the conditions above if the acceleration parameter may be maintained in the range 0 to 2.

Conjugate Gradient Method:

All relaxation processes have the basic idea of reducing the residual vector $\mathbf{r}$ [= $\mathbf{b}$ - A $\mathbf{x}$] as the calculation proceeds from iteration to iteration. There are different procedures defined dependent on the search direction to modify the trial vector for $\mathbf{x}$. One of the most successful procedures is called the conjugate gradient method. In this process, the trial vector, $\mathbf{v}$ is modified as follows:

$$\mathbf{v}' = \mathbf{v} + t\,\mathbf{p}$$

where, t is a scalar, $\mathbf{v}'$ is the new trial vector and $\mathbf{p}$ is the direction vector in which the new solution vector, $\mathbf{v}'$ must be searched. The direction of $\mathbf{p}$ must not be chosen perpendicular to the error vector because then there will be no improvement in the solution vector $\mathbf{v}'$. If the direction of $\mathbf{p}$ is chosen same as the error vector the previous iteration step, we get the steepest descent method. If the direction of $\mathbf{p}$ is chosen as same as a weighted constant factor times the previous step's error vector the resulting procedure is called the simultaneous displacement or the Jacobi iteration procedure. The best selection of $\mathbf{p}$ comes from satisfying the relationship

$$A\,\mathbf{p}^{(k)} \cdot \mathbf{p}^{(k-1)} = \mathbf{p}^{(k)} \cdot A\,\mathbf{p}^{(k-1)} = 0$$

This is the basis of the so called conjugate gradient method. In the above relation, the dot indicates the inner product of two vectors. The vector $\mathbf{p}^{(k)}$ is taken as a linear combination of $\mathbf{r}^{(k-1)}$ and $\mathbf{p}^{(k-1)}$. In this procedure, the residual vectors and $\mathbf{p}$ with A$\mathbf{p}$ form two orthogonal systems, hence the name conjugate gradient method.

The conjugate gradient method is by far the best relaxation method among the traditional iterative procedures. The convergence in this process is quadratic and is guaranteed within n iterations where n x n is the size of the matrix A. The only drawback is that the matrix to analyze is required to be symmetric. There are variations of the conjugate gradient method possible for asymmetric matrices. However, as mentioned in the section below and substantiated by the results later, the best scheme tested in this investigation is a variant of the Lanczos method.

Recent Developments:

As we shall see in the graphical interpretations of the program gsol later, the search of the solution vector is facilitated by a choice of an orthonormal basis. QR factorization schemes and procedures associated with the Gram Schmidt orthogonalization yield such orthormal bases. However, if the basis can be established in an elegant way there are significant computational advantages.

In any iterative solution the Markov chain converges depending on the eigenvalues of the iteration matrix. If the largest eigenvalue is less than one convergence is assured. However the speed of convergence is also associated with the closeness of the eigenvalues. These ideas prompted a host of procedures [3, 4, 5, 6] related to the determination of eigenvalues. For large matrices the speed

of calculation is of prime importance. If there is a way to determine search directions while the matrix sizes are relatively small and self-correct the process while progressively larger matrices are handled, the process will surely be more desirable. The basis of the current solver in PMARC has this structure in the subroutine *lineq*. This procedure is based upon Davidson's method [7] of determination of a few of the smallest eigenvalues and eigenvectors of a large matrix.

Computational Efficiency

There are several ways computational efficiency is measured in solving linear systems. Most computational processes are dictated by the accuracy in calculations. This is very important in both direct and iterative solution methods. However accuracy plays different roles in these two methods. While direct solvers must be very accurate because calculations are performed only once, iterative solvers can handle more errors in early iterations if they can be annihilated rapidly. There is also the question of conditioning of matrices. If a matrix is well conditioned, any small perturbation in the coefficients will not produce large perturbations in the solution vector.
Assuming the matrices to explore are well conditioned, errors can only be of one kind - rounding errors. Unless any solution scheme models after application of some series, there is no concern about truncation errors. Rounding errors can be measured as local and global. Local errors that take place during each iteration become of global nature when all iterations cumulatively affect calculations. A classic example of sensitivity of a solution scheme on global errors is the Lanczos scheme. For quite some time, this elegant method was overlooked because the failures were not pegged down to cumulative errors. This important lesson that the author learnt is reflected in the design of *gsol*.

The speed of calculations is reflected in the number of arithmetic operations that each scheme is required to perform. The analytical estimate is typically related to the size of the matrix, n. For example, a scheme requiring n^3 operations would take approximately 8 times the computational effort if the size of the matrix is doubled. The other measure of speed as discussed before (for an iterative process) is going to be reflected in how rapidly the residuals are annihilated. This is typically measured by the ratio of the absolute values of the residual norms between two successive iteration steps. This can also be assessed by the Euclidean norm of the iteration matrix.

Another measure of computational efficiency is given by robustness. A robust calculation procedure will normally not fail if the conditions of matrix coefficients, the right hand side column vector, the starting guess solution, etc. change. This was the focus in the early part of this work.

Finally, modern linear algebra has been tremendously impacted by the architecture of computers. As the demand of speed increases, the concepts are modified to suit the need. Today the product of two matrices are done by block concepts much more than the conventional earlier methods. If a matrix is symmetric, half data storage is exploited. These and parallel architecture are keys of modern computing. This work exploited some storage optimization for 2004 x 2004 matrices primarily from the need to save memory on the *sr71* machine. Also some modularization was

adopted for programming ease. Other concepts of modern architecture were not exploited.

Graphical Interpretation of Residual Correction Schemes

In this section we shall explore the geometrical ideas associated with residual correction methods. The ideas will be developed from geometrical to algebraic reasoning for higher dimensions. Let A **x** = **b** be represented for a planar case first. Let $a_{11}\ x_1 + a_{12}\ x_2 = b_1$ and $a_{11}\ x_1 + a_{12}\ x_2 = b_2$ represent two straight lines in the x-y plane given by OP and OQ, with O as their point of intersection. The point O's coordinates are the so called solution of this system of equations which we wish to arrive at iteratively. Let the point S represent the coordinates of the starting guess solution. To reach the point O from S, let us first drop a perpendicular on OP from S. Let the point of intersection be A. Subsequently another perpendicular may be dropped on OQ from A. Let that point of intersection be B. With the knowledge of the points A and B we may drop one last perpendicular from B on OP to obtain C. Then the solution O of the system may be arrived at from A in one movement along AC by the amount AO, where, $AO = AB^2/AC$.

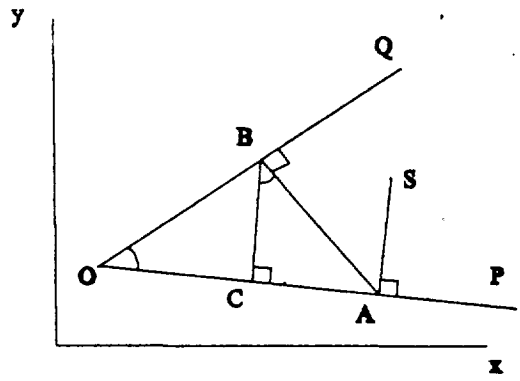

Figure 1.

This may be shown easily from the similar triangles OAB and ABC where the angle CAB is common. Instead of using the direct value from this formula for AO, suppose there is a multiplicating factor ω used with AO to reach O from A. This may be viewed upon as an accelerating factor in higher dimensions. For a set of n equations in n unknowns,

$res1 = b_1 - [a_{11}\ a_{12}\ a_{13} \ldots\ldots\ a_{1n}] \cdot x_S$, represents the residue from the first equation, which also represents the perpendicular distance of the point O from the hyperplane:

$a_{11}\ x_1 + a_{12}\ x_2 + a_{13}\ x_3 + \ldots\ldots\ldots\ldots + a_{1n}\ x_n = b_1$

Now this distance can be used to arrive at the point A if a component can be used in the direction toward A to obtain the position vector x_A. Thus,

$$x_A = x_S + \frac{res1}{|a_{11}\ a_{12}\ a_{13} \ldots\ a_{1n}|^2}\ [a_{11}\ a_{12}\ a_{13} \cdots\ a_{1n}]^T$$

In a similar manner, $\mathbf{x}_B$ may be written from $\mathbf{x}_A$ as

$$\mathbf{x}_B = \mathbf{x}_A + \frac{res2}{|a_{21}\ a_{22}\ a_{23}\ \cdots\ a_{2n}|^2}\ [a_{21}\ a_{22}\ a_{23}\ \cdots\ a_{2n}]^T$$

where, res2 $= b_2 - [a_{21}\ a_{22}\ a_{23} \ldots . a_{2n}] \cdot \mathbf{x}_A$. Similarly $\mathbf{x}_C$ may be obtained from $\mathbf{x}_B$ by projecting into the first hyperplane again. After C is obtained a single evaluation of O is made with the relaxation factor ω. Then the resulting point becomes a new point O and the process keeps repeating. In this manner, each new equation is selected from a set of n equations and the projections are made on the first hyperplane. This is the basis of the program *qmconj* given in the appendix.

An alternate to the above program will be instead of projecting each new residue onto the first hyperplane, each two new residues are formed from the sequence of equations (1.1,1.2), (1.2,1.3), (1.3,1.4), etc. This process works faster with larger values of ω and is the basis for program *simconj* given in the appendix. A proof of convergence for the above procedures is given below.

Let $\mathbf{x}_O^*$ be the new guess point arrived at after one iteration of the qmconj process. Since the vectors AB, BC and AC may written as

$\mathbf{ab} = \mathbf{x}_B - \mathbf{x}_A$, $\quad abl = \sqrt{(\mathbf{ab}.\mathbf{ab})}$
$\mathbf{bc} = \mathbf{x}_C - \mathbf{x}_B$, $\quad acl = \sqrt{(\mathbf{ac}.\mathbf{ac})}$
$\mathbf{ac} = \mathbf{x}_C - \mathbf{x}_A$, $\quad adl = \omega\ abl^2/acl$

Thus, $\mathbf{x}_O^* = \mathbf{x}_A + \omega\ abl^2\ (\mathbf{x}_C - \mathbf{x}_A) / acl^2 = \Omega\ \mathbf{x}_C + (1 - \Omega)\ \mathbf{x}_A$, where, $\Omega = \omega\ abl^2/acl^2$.

Note the structure of the acceleration in the last step above emerges exactly like the point successive over-relaxation scheme, where the position vectors for points C and A serve the replacement for the Gauss-Seidel and Jacobi iterations before. With the matrices substituted for a 4 x 4 system the three basic steps to the solution of the new guess vector would be given by

$$\mathbf{x}_A = \frac{b_1}{(a_{1i}\ a_{1i})^{1/2}} + M_1\ \mathbf{x}_O, \qquad \mathbf{x}_B = \frac{b_2}{(a_{2i}\ a_{2i})^{1/2}} + M_2\ \mathbf{x}_A, \qquad \mathbf{x}_C = \frac{b_1}{(a_{1i}\ a_{1i})^{1/2}} + M_1\ \mathbf{x}_B.$$

where, the repeated index i indicate summations over i =1,4 and the matrices M_1 and M_2 are

$$M_1 = \begin{bmatrix} (1-\frac{a_{11}^2}{a_{1t}a_{1t}}) & \frac{-a_{11}a_{12}}{a_{1t}a_{1t}} & \frac{-a_{11}a_{13}}{a_{1t}a_{1t}} & \frac{-a_{11}a_{14}}{a_{1t}a_{1t}} \\ \frac{-a_{11}a_{12}}{a_{1t}a_{1t}} & (1-\frac{a_{12}^2}{a_{1t}a_{1t}}) & \frac{-a_{12}a_{13}}{a_{1t}a_{1t}} & \frac{-a_{12}a_{14}}{a_{1t}a_{1t}} \\ \frac{-a_{11}a_{13}}{a_{1t}a_{1t}} & \frac{-a_{12}a_{13}}{a_{1t}a_{1t}} & (1-\frac{a_{13}^2}{a_{1t}a_{1t}}) & \frac{-a_{13}a_{14}}{a_{1t}a_{1t}} \\ \frac{-a_{11}a_{14}}{a_{1t}a_{1t}} & \frac{-a_{12}a_{14}}{a_{1t}a_{1t}} & \frac{-a_{13}a_{14}}{a_{1t}a_{1t}} & (1-\frac{a_{14}^2}{a_{1t}a_{1t}}) \end{bmatrix}$$

and,

$$M_2 = \begin{bmatrix} (1-\frac{a_{21}^2}{a_{2t}a_{2t}}) & \frac{-a_{21}a_{22}}{a_{2t}a_{2t}} & \frac{-a_{21}a_{23}}{a_{2t}a_{2t}} & \frac{-a_{21}a_{24}}{a_{2t}a_{2t}} \\ \frac{-a_{21}a_{22}}{a_{2t}a_{2t}} & (1-\frac{a_{22}^2}{a_{2t}a_{2t}}) & \frac{-a_{22}a_{23}}{a_{2t}a_{2t}} & \frac{-a_{22}a_{24}}{a_{2t}a_{2t}} \\ \frac{-a_{21}a_{23}}{a_{2t}a_{2t}} & \frac{-a_{22}a_{23}}{a_{2t}a_{2t}} & (1-\frac{a_{23}^2}{a_{2t}a_{2t}}) & \frac{-a_{23}a_{24}}{a_{2t}a_{2t}} \\ \frac{-a_{21}a_{24}}{a_{2t}a_{2t}} & \frac{-a_{22}a_{24}}{a_{2t}a_{2t}} & \frac{-a_{23}a_{24}}{a_{2t}a_{2t}} & (1-\frac{a_{24}^2}{a_{2t}a_{2t}}) \end{bmatrix}$$

Similarly, M_3 and M_4 matrices may be written following similar structures as M_1 and M_2. Thus the final expressions for the three steps $\mathbf{x_o}^*$, $\mathbf{x_o}^{**}$ **and** $\mathbf{x_o}^{***}$ before arriving at the upgraded initial guess may be derived. Finally the iteration matrix for the program *qmconj* may be obtained as $T_{qm} = (1 - \Omega)^{n-1} (M_1M_4M_1)(M_1M_3M_1)(M_1M_2M_1)$. In a similar manner the iteration matrix for the program *simconj* may be written as $T_{sm} = (1 - \Omega)^{n-1} (M_3M_4M_3)(M_2M_3M_2)(M_1M_2M_1)$.

Several computational experiments were performed to claim the convergence of the above iterative processes. It was shown that in each case the matrices were convergent. The norms of T_{qm} and T_{sm} were very small indicating rapid reduction of the residuals. Thus the procedures qmconj and simconj could be applied for small matrices fairly reliably, each time converging. Another advantage of these schemes over other previously discussed iterative methods was they could be applied for ill-conditioned matrices (see next section) as well as general solution procedures, where typically Gauss-Seidel type methods would fail due to the lack of diagonal dominance and conjugate gradient type

methods would fail due to the lack of symmetry. However, the procedures are not very efficient computationally specially for large matrices. Thus although the schemes were robust they were also computationally expensive. Nonetheless there was an important lesson to be learnt from these exercises, which reflected in basic geometrical interpretations of the convergence processes in a relaxation method.

Returning to the planar case of the above methods, we may obtain another arrangement by this process to reach O. Instead of obtaining the point A and then using it to obtain point B, let SA and SB be two perpendiculars from S onto OP and OQ. Then the arrangement would look like

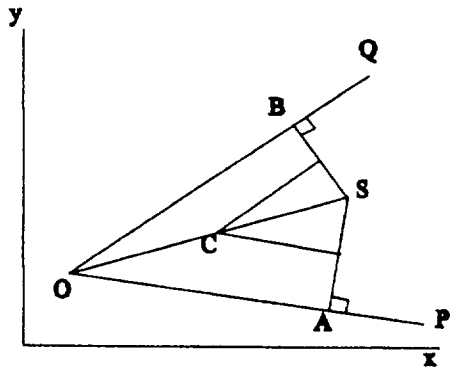

Figure 2.

Note that the distance SA represents the perpendicular distance of a point from a line. In higher dimensions, the corresponding distance represents the perpendicular distance of the point S from the n-dimensional hyperplane. If the figure 2 is interpreted another way, one would discover a nice geometrical feature imbedded in it. The perpendiculars SA and SO tend to suggest that OS resides on the diameter of a circle. In fact the points S, A, B and O are all on this circle with the diameter OS. If there was a way to quickly obtain the center C of this circle, obtaining the solution O is just a matter of doubling the distance from S to C. This also raises the question that although the above geometric reasoning holds for a circle, will it also hold for a sphere in n-dimensions. It turns out that the point C would become the circumcenter for the n-dimensional solid. The name circumcenter arises from the fact that, for a triangle, the circumcenter is the perpendicular bisector of all the three sides, and is at an equal distance from all the three vertices. The process to obtain C in n-dimensions was first implemented in the program *gsol2v* as a direct solver. There are two options to solve for the circumcenter. The first one is using the orthonormalization process as mentioned above. An alternate procedure is to calculate the Menger's determinant [8]. This process determines the circumradius first. However the solution of the resulting system must again be carried out using some speedy iterative procedure. Thus the direct solver was later modified as an iterative process in the program *gsol* to obtain the circumcenter using the current solver in PMARC.

The direct solution of the circumcenter C is found by going to the midpoints of each segment SA, SB,

etc. for the n perpendiculars to the hyperplanes for which linear system the solution is sought. This resulted in a Gram-Schmidt type orthogonalization procedure. The equations can be cast into a simple form starting from the point S. If the base points of perpendiculars to all the hyperplanes are obtained, each two new basepoints will make a triangle with the first basepoint of which the circumcenter is equidistant from the vertices. However for storage purposes of the large matrices, all basepoints are not calculated upfront. First the perpendiculars to the first two hyperplanes, A and B are obtained. Starting from A, the midpoint of segment AB is calculated. As each new base point C is introduced with the two original basepoints A and B, a scalar multiple λ is used to determine the proportional distance of the circumcenter on the normal to AB by the formula

$$\lambda = \frac{\frac{1}{2}[(x_c \cdot x_c) - (x_a \cdot x_a)] - (x_c - x_a) \cdot mc}{(x_c - x_a) \cdot nu}$$

where, **mc** and **nu** are the midpoint of segment **ab** and unit normal to segment **ab.** The process is repeated as new points are read in. The total amount of computational effort is minimized by the nested looping in the program.

Defective Matrices

As mentioned in the introduction, the difficulties of the PMARC convergence was believed to be the ill-conditioning of the coefficient matrices. It was proved later that the WICS calculations will always involve diagonally dominant coefficient matrices. Such matrices will not typically have small determinants. Thus the small perturbations in the coefficients will not produce large perturbations in the solution vector. However, since this research started before the stage when the well-conditioning could be claimed, the schemes developed under this research were tested with Hilbert matrices. A Hilbert matrix is very ill-conditioned and solutions are almost impossible for larger systems. The tests with Hilbert matrices were performed for a maximum of 20 x 20 size. It was believed that if a solution scheme successfully performed with a Hilbert matrix, it probably was very robust.

The Hilbert matrix has A's coefficients of the following form: $a_{ij} = 1/(i+j-1)$, $[1 \le i,j \le n]$ with the right hand vector **b** varied differently for different computational experiments. Three categories of the **b** vector was tested - i) larger terms toward the top of the column, ii) even size terms in the column and iii) larger terms toward the bottom of the column. It was found that the last category was the most difficult to solve in most of the cases. Both the routines *simconj* and *congrad* performed better than the Gauss Seidel technique when the matrix sizes increased. In this context it may also be mentioned that *simconj* or *qmconj* type approach has an additional advantage. These procedures will hold even if the linear system is over-determined, meaning that there are more number of equations than the number of unknowns. Then although standard reduction

processes would not work, these procedures will produce a "limit cycle" solution without much difficulty. Note that the conjugate gradient method is especially suitable for a symmetric matrix even though the matrix is ill-conditioned. In the case of an asymmetric matrix, the conjugate gradient method fails. The results are summarized in Table 2 in the appendix.

Results and Discussion

As mentioned previously, this work was started as the current solver in PMARC produced difficulty in convergence in certain configurations of WICS calculations. The nature of the current solver in PMARC was not known to the author. WICS calculations are based upon a procedure similar to the error equation approach mentioned earlier in the section discussing direct solvers. Thus it was important to test the convergence when the parameter *solres* was tightened. This action typically produced oscillatory behavior and no convergence for certain singularities. The first part of this investigation was therefore to determine the cause of the failure in the WICS computations. This was done by extracting a typical coefficient matrix that was utilized during the WICS calculations with doublets. This matrix was subsequently subjected to a direct Gaussian elimination technique and the QR decomposition technique outside the PMARC set of calculations. The prescription of the gridding and section definitions in PMARC were changed and a double precision arithmetic was used. From these actions the convergence difficulties were overcome. An independent look at the structure of the coefficient matrices showed that the diagonal dominance can always be guaranteed in such calculations. The difficulties experienced earlier can be ascribed to the loss of orthogonalization typical to a Lanczos process.

The appendix lists samples of the programs *conj, qconj, qmconj, simconj, gsol2v and gsol.* The results produced by these programs are compared with the Gauss-Seidel iterative technique and the conjugate gradient method (see *congrad* in the appendix) for small matrices. Also tests were performed for some sparse (9 x 9) planar truss problems, the ill-conditioned Hilbert matrices and some medium size [(79 x 79) and (84 x 84) respectively] external flow calculations over NACA 0012 and NACA 2412 airfoils. Finally some solutions on the PMARC calculations with a (2004 x 2004) matrix are presented. Then the comparisons are made of the Gauss-Seidel method, the S.O.R method, the conjugate gradient method, the direct solver *gsol2v*, the direct solver in Gauss-Jordan method, the iterative solver *gsol* and the current PMARC solver.

First consider the Table 1 in the appendix. The reason 4 x 4 matrices were chosen for most of the testing in this table is because a size 4 x 4 is general enough for the theory to hold even in higher dimensions and yet the calculations are not time consuming. Thus many comparisons could be made. Also note that the calculations presented in the tables are a subset of the case files available in the author's directory with more details. In all calculations in this section, the convergence criterion was chosen to be 0.5×10^{-5} in the 2-norm of the residual vector. Also smaller matrices were run with single precision arithmetic whereas, the large matrix in Table 3 results below were all run with a double precision arithmetic.

As one can see from Table 1, the number of iterations taken by the examples of 4 x 4 matrices *simconj* and *qmconj* performed better than the number of iterations taken by the Gauss-Seidel method in most of the cases. These routines also performed much better than simple relaxation method *conj* and an early variant of the geometrical approach, *qconj*. The rapidity with which these calculations were achieved could be compared with the conjugate gradient method if proper acceleration parameters were selected. Best acceleration parameter choices were dependent on the type of the matrices selected. In general, for the 4 x 4 cases the ω selection was around 1.2 - 1.3 for *qmconj* and around 1.6 - 1.7 for the program *simconj*. Exact values for each case is included in the casefiles.

The off-diagonally dominant matrices did not yield solutions by the conventional iterative methods such as Gauss-Seidel and S.O.R. methods, whereas these two routines consistently produced solutions. Also note that *qmconj* and/or *simconj* typically take fewer iterations to converge than Gauss-Seidel method even for the 4 x 4 diagonally dominant matrices. As the matrix became sparse though, the Gauss-Seidel method yielded better performance (see the 9 x 9 planar truss example). An interesting case applies to the diagonally dominant Hilbert type 20 x 20 matrix, where the original diagonal elements of the Hilbert matrix was modified to maintain diagonal dominance. In this case, the best performance was by the use of the conjugate gradient method confirming the superiority of this method for symmetric systems. Another interesting case was that of the under-determined system. In such situations, the matrix reductions produce a null row suggesting no unique solution of the system. Thus conventional methods would not work again. However these geometry based methods did not produce a division by zero, and quickly produced a solution satisfying the equations.

For the external panel method solution applied to the NACA 0012 airfoil, the best performance was from using the *qmconj* matrix among the geometry based methods. For some reason, believed to be rounding errors, the program *simconj* did not perform very well. Also note that for the well conditioned matrices the Gauss-Seidel method performed better. For the corresponding cases of the asymmetric NACA 2412 airfoil, *qmconj* arrived at the solution with fewer iterations in both 0° and 90° orientations. In summary, the routines *qmconj* or *simconj* achieved the objectives of robust calculations typically for the off-diagonally dominant cases when the conventional iterative methods had either no solutions or, had difficulty in arriving at them.

Table 2 quotes the results for some Hilbert matrices. As mentioned before, these matrices were run with three different choices of the right hand vector. The reason for this choice was to effect the ill-conditioning from mild to severe. In mild ill-conditioning the right hand column vector had larger elements toward the top rows and for the severe conditions the larger elements were toward the bottom rows. The tight convergence criterion was relaxed a decimal digit (i.e., $\epsilon = 0.5 \times 10^{-4}$) to allow faster convergence in the matrix sizes larger than 2 x 2. Also the larger matrices were run with mild ill-conditioning to speed up calculations. The calculations showed an advantage of using the *qconj* and *qmconj* routines. The routine *qconj* is clearly the best performer in all computations except the case 4 for a 3 x 3 matrix. This success may be attributed to the structure of the cycling in this program. All the geometry based routines suffer error accumulation and this routine helps reducing it through cycling. As the matrix sizes grew much bigger with ill-conditioning, these routine suffered

larger round-off and took too long to converge. This prompted the strategy change and the development of the second geometry based solver *gsol2v* (see Table 3).

gsol2v is called a direct solver because the determination of circumcenter and the circumradius are obtained by direct application of Gram-Schmidt type orthogonalization process. This method was sought as a new approach to obtain solution of linear systems. Although the picturization of the circumcenter concept is not possible for higher dimensions, the success of the approach shows that the low dimensional geometric reasoning can be extended algebraically to higher dimensions. Several issues of convexity must be discussed to obtain the algebraic proof of concepts for conventional iterative methods. The attempt here was to bypass these by the current geometrical approach. From this standpoint, this approach was successful. A new correlation between the geometry of spheres and solution of linear system was obtained. In planar dimensions, the nine-point circle establishes the link between the centroid, the orthocenter and the circumcenter. In higher dimensions, the distances and scalar products were needed instead of angles.

The use of *gsol2v* required much larger solution time compared to the direct solvers like Gauss-Jordan method. It may be mentioned here that among all direct solvers simultaneously involving the complete set of coefficients in the matrices, the Gaussian elimination or Gauss-Jordan methods are the most economical. Thus it was expected to be slower than those methods. However the argument in favor of *gsol2v* was that it could be applied without pivoting strategies. To improve the speed of this solver, the geometrical concepts in this method were combined with an excellent iterative solver. This is the idea behind the program *gsol*. Also note that although the direct solver gsol2v took more time than the program *impjord*, it was less time consuming than both point iterative methods such as Gauss-Seidel or the S.O.R. method (with the optimum acceleration parameter ≈ 1.6).

The algebraic iterative solver used in the *lineq* routine of PMARC has an excellent structure. It requires very little storage and is computationally very efficient. Whereas a traditional Gauss-Seidel method took over 15 hours for solving the 2004 x 2004 matrix, this solver obtained the solution in 4 minutes. The secret of this solver drew the author to study the eigenvalue based methods after Nesbet, Shavitt, Bender and Davidson [3 - 6]. There are a set of programs *nesbet, nespy, nes1, nes2,* etc. and the resulting output files available for these methods in the authors directories. These programs calculate the lowest eigenvalues and eigenvectors for large symmetric and asymmetric matrices. However the current solver fills the voids suffered in all those earlier programs. All of those [3 - 6] started the search for eigenvalues from the Raleigh quotient approach. However considerable difficulty was experienced trying to simultaneously change all components of the search vector. The current program can also handle asymmetric matrices. It starts from small matrices and applies corrections to the search direction as the matrix grows. It is also fully optimized for large matrix calculations. Note that although it uses the Gauss-Jordan method internally to invert matrices, the calculations take practically no time (see the *impjord* results in Table 3) since the main time consuming calculations are performed on much smaller matrices. The design of this procedure even offers advantages in restarting the unconverged solutions. Combining the geometrical approach in *gsol2v* therefore with this solver yielded the apparent benefits. The resulting process took just one more iteration to converge the 2004 x 2004 matrix. Also the calculation time to converge was

reduced from more than 3 hours for *gsol2v* down to 4 minutes in *gsol.*

Note the contrasted results by the conjugate gradient method in Table 3. Since the PMARC matrices are not symmetric for WICS calculations, conjugate gradient method could not be applied directly. The program *mixcon* therefore modified the solution approach by premultiplying the system $A\,\mathbf{x} = \mathbf{b}$ by A^T. This resulted in a symmetric system which could then be solved by the conjugate gradient method. This approach yielded the results of the matrix in 1 hour and 47 minutes and using 99 iterations. However if we subtract the time required to obtain the products A^TA and $A^T\mathbf{b}$, the basic conjugate gradient process took only 12 minutes. Thus calculations of such large matrices proved to be extremely sensitive to arithmetic operational counts. This also shows that besides the current solver, the next best approach in solving large systems will perhaps have to be based upon the conjugate gradient method. The program *gsol* offers the quickest alternate solution (in a single step) as long as the circumcenter could be obtained quickly. Further expansions are possible for this idea.

Concluding Remarks

The current investigation was prompted by a difficulty in convergence of PMARC matrices when applied for WICS calculations. The objective was to develop a robust solver that would work typically when other iterative methods failed to produce solutions. This objective was fulfilled by developing a geometry based solver in the approaches of *conj, qconj, qmconj* and *simconj.* There was an alternate geometrical approach developed parallel to the conventional algebraic approaches for iterative methods in the programs *gsol2v* and *gsol.* Throughout this investigation, an attempt was made to re-visit geometrical topics as a means to enhance physical and intuitive understanding of the convergence processes. The most recent and more complex methods are perhaps more sophisticated computationally. However a renewed exploration of geometrical concepts probably contributes to a much better understanding than abstract ideas offered in them.

Acknowledgements

This work was supported in part by the NASA cooperative agreement NCC 2-937. The author is grateful to Alan Boone for acting as the technical monitor and for introducing him to the WICS project. He is also thankful to Robert McMann and David Banducci for providing help in financial matters, Dale Ashby and Charles Bauschlicher for helpful discussions, Linda Thompson for providing excellent support of computer accounts and Charles Haines for providing the release time.

References

1. Ashby, D. L., Dudley, M. R., Iguchi, S. K., Browne, L. and Katz, J., "Potential Flow Theory and Operation Guide for the Panel Code PMARC," NASA TM 102851, NASA Ames Research Center, Moffett Field, California, January 1991.

2. Varga, R. S., "Matrix Iterative Analysis," Prentice Hall Inc., New Jersey, 1962.

3. Nesbet, R.K., "Algorithm for Diagonalization of Large Matrices," Journal of Chemical Physics, volume 43, pp. 311-312, 1965.

4. Shavitt, I.., "Modification of Nesbet's Algorithm for the Iterative Evaluation of Eigenvalues and Eigenvectors of Large Matrices," Journal of Computational Physics, volume 6, pp. 124-130, 1970.

5. Shavitt, I., Bender, C.F., Pipano, A. and Hosteny, R.P., "The Iterative Calculation of Several of the Lowest or Highest Eigenvalues and Corresponding Eigenvectors of Very Large Symmetric Matrices," Journal of Computational Physics, volume 11, pp. 98-108, 1973.

6. Hestenes, M.R. and Karush, W., "Method of Gradients for the Calculation of the Characteristic Roots and Vectors of a Real Symmetric Matrix," Journal of Research of the National Bureau of Standards, volume 47, no. 1, pp. 45-61, 1951

7. Davidson, E.R., "The Iterative Calculation of a Few of the Lowest Eigenvalues and Corresponding Eigenvectors of Large Real-Symmetric Matrices," Journal of Computational Physics, volume 17, pp. 87-94, 1975.

8. Berger, M., "Geometry I & II" (2 volumes), Springer Verlag Inc., New York, 1980.

Appendix

This appendix contains tables used for comparing results and samples of 7 fortran programs *conj, qconj, qmconj, simconj, gsol2v, congrad* and *gsol*. The solvers for Gaussian elimination, all of the eigenvalue based methods, *lineq*, QR Decomposition method, Gauss-Seidel and S.O.R. methods are not presented here for compactness and are available in the author's directories. Also all the details of the casefiles reported in the tables below may be found there.

Table 1
Comparison of Various Methods for Small Matrices

In the following results, the matrices are specified by their coefficients. The strategy was to compare the number of iterations to convergence and typically choose off-diagonally dominant matrices so that the robustness of calculations could be tested. Most cases started with the same starting solution: $x_1 = x_2 = \ldots = x_n = 1.0$ (unless specifically mentioned in the casefiles). The convergence criterion was 0.5×10^{-5} in the 2-norm of the residual. Also the acceleration parameters used in these were near optimal and mentioned in the casefiles.

Size	Coefficients of A and **b** (Comments)	No. of Iterations to Converge Method				
		conj	*qmconj*	*simconj*	G. S.	C.G.
4 x 4	2,1,-3,1;1,2,5,-1;-1,1,1,4;2,-3,2,-5 1,7,5,-4 (off-diagonal dominance)	n.a.	28	12	failed	failed
4 x 4	1,1,1,1;2,1,-1,1;-1,2,3,1;3,2,-2,-1 2,2,1,8 (similar to above)	208	38	70	failed	n.a.
4 x 4	-1,-4,2,1;2,-1,7,9;-1,1,3,1;1,-2,1,-4 -32,14,11,-4 (mild off-diag dom.)	42	15	9	failed	failed
4 x 4	1,1,1,0;-3,-17,1,2;4,0,8,-5;0,-5,-2,1 3,1,1,1 (mild off-diag. dominance)	97	27	33	failed	failed
4 x 4	2,-1,0,0;-1,3,-1,0;0,-1,3,-1;0,0,-1,2 -1,4,7,0 (diagonally dominant)	30	13	6	15	5
4 x 4	2,-1,0,0;-1,4,-1,0;0,-1,4,-1;0,0,-1,2 3,5,-15,7 (diagonally dominant)	30	13	6	10	5
4 x 4	2,-2,-1,3;-1,0,-1,1;3,-6,2,4;1,-4,3,1 5,0,7,2 (under-determined system)	22	8	5	failed	failed
4 x 4	1,1,1,1;1,1,-1,1;-1,2,3,1;3,2,-2,-1 3,2,-2,-1 (off-diagonal dominance)	52	24	33	failed	failed
9 x 9	Truss Problem (no re-arrangement)	148	117	145	failed	n.a.
9 x 9	Truss Problem (eqns. re-arranged)	139	127	53	16	n.a.
20 x 20	Re-arranged Hilbert Matrix	n.a.	large	large	n.a.	21

79 x 79	NACA 0012 Airfoil (Panel Method for External Flows)	568	72	400	44	n.c.
79 x 79	NACA 0012 Airfoil (Panel Method with $\alpha = 90°$)	906	83	938	53	n.c.
84 x 84	NACA 2412 Airfoil (Panel Method)	856	103	880	n.a.	n.c.

n.a. = not available, n.c. = non-convergent

Table 2
Comparison of the Methods applied to Hilbert Matrices of different sizes

Size (Comments)	Iterations to converge method to specified criterion				
	G.S.	*conj*	*qconj*	*qmconj*	*simconj*
2 x 2 type 1	37	624	13	5	18
2 x 2 type 2	34	559	17	7	16
2 x 2 type 3	36	697	21	7	20
3 x 3	199	n.c.	> 10,000	49	104
4 x 4	126	n.c.	14	203	109
6 x 6	427	n.c.	27	129	1048
10 x 10	261	n.c.	70	75	423
20 x 20	515	n.c.	52	512	1722

n. c. means no convergence was achieved in 10,000 iterations.

Table 3
Comparison of Various Methods for a PMARC Matrix of Size 2004 x 2004

Program Name	Method Used	Number of Iterations to convergence	Solution Time
gssd	Gauss-Seidel Method	6936	15 hours 11 min.
sor	S.O.R. (ω = 1.6)	5554	6 hours 22 min.
gsol2v	Finding Circumcenter	Direct Solver	3 hours 37 min.
mixcon	Conjugate Gradient	99	1 hour 47 min (*)
impjord	Gauss-Jordan Meth.	Direct Solver	1 hour 55 min.
gsol	Circumcenter Method	41	~ 4 minutes
lineq	Davidson's Approach	40	~ 4 minutes

* The time taken by the iteration portion only was 12 minutes.

Program Listings:

1) *conj.f*

```
      program conj
      parameter n = 20
      dimension a(n,n), b(n), x(n), y(n), res(n)
      OPEN(UNIT=13,FILE='a.dat',STATUS='OLD')
      OPEN(UNIT=14,FILE='b.dat',STATUS='OLD')
      read (13,*) ((a(i,j), j = 1,n), i = 1,n)
      read (14,*) (b(i), i = 1,n)
c     write(6,3)((a(i,j), j=1,n),i=1,n)
 3    format(1x, 9f8.3)
      write(6,3) (b(i), i=1,n)
c
c     The above line format must be changed
c
      write(6,10)
 10   format(//' Results using the Program conj.f :' //)
c     initial guess
      itr=0
      do 1 i=1,n
c 1   x(i)=0.0
 1    x(i)=1.0
 101  k=1
 100  continue
      sum=0.
      ysq=0.
      do 4 i=1,n
 4    y(i)=a(k,i)
      do 5 i = 1,n
```

```
        ysq=ysq+y(i)*y(i)
 5      sum=sum+y(i)*x(i)
        res(k)=b(k)-sum
        do 6 i=1,n
 6      x(i)=x(i)+res(k)*y(i)/ysq
        if(k.eq.n) go to 8
        k=k+1
        go to 100
c 8     if(itr.eq.1000) write(6,*) (x(i),i=1,n)
c 8     write(6,*) (res(i),i=1,n)
c       write(6,12) (x(i),i=1,n)
 8      resmx=0.
        do 11 i=1,n
        resmx=resmx+res(i)*res(i)
 11     continue
        resmx=sqrt(resmx)
        itr=itr+1
        if(itr.lt.10) write(6,*) itr, resmx
        if(resmx.gt.0.00005 .and. itr.lt.10000) go to 101
        write(6,12) (x(i),i=1,n)
 12     format(2x,'solution: ',5f12.5)
        write(6,13) itr
 13     format(' The above was obtained in ',i6,' iterations')
        stop
        end
```

2) *qconj.f*

```
        program qconj
        parameter n= 20
        dimension x0(n), xa(n),xb(n),xc(n),ab(n),bc(n),ac(n),a(n,n),b(n),
   1    y(n),res(n),err(n)
        OPEN(UNIT=13, FILE='a.dat',STATUS='OLD')
        OPEN(UNIT=14, FILE='b.dat',STATUS='OLD')
        read(13,*)((a(i,j),j=1,n),i=1,n)
        read(14,*)(b(i),i=1,n)
c       write(6,3)((a(i,j),j=1,n),i=1,n)
 3      format(1x,9f8.3)
c       write(6,3)(b(i),i=1,n)
c
        write(6,21)
 21     format(//5x,'Results with Program qconj.f'//)
c
c       a and b store the original Ax = B
cc      initial guess
        om=1.6
c       om=1.333
c       om=1.6
        itr=0
        do 1 i=1,n
 1      x0(i)=0.0
c 1     x0(i)=1.0
c
c       if n is even, proceed exactly. If n is odd, set last cycle steep descent
c
        ist = n/2
        iodd=n - 2*ist
        write(6,2) ist,iodd
 2      format(' ist, iodd=',2i5)
 100    continue
        do 200 ic=1,ist
        k=2*(ic-1)+1
        kp1=k+1
c       write(6,*) itr, k, kp1
        do 4 i=1,n
 4      y(i)=a(k,i)
        sum=0.
        ysq=0.
```

```
      do 5 i=1,n
      ysq=ysq+y(i)*y(i)
 5    sum=sum+y(i)*x0(i)
      res(k)=b(k)-sum
c     write(6,*)(x0(i),i=1,n)
      do 6 i=1,n
 6    xa(i)=x0(i)+res(k)*y(i)/ysq
      do 41 i=1,n
 41   y(i)=a(kp1,i)
      sum=0.
      ysq=0.
      do 51 i=1,n
      ysq=ysq+y(i)*y(i)
 51   sum=sum+y(i)*xa(i)
      res(kp1)=b(kp1)-sum
c     write(6,*)(xa(i),i=1,n)
      do 61 i=1,n
 61   xb(i)=xa(i)+res(kp1)*y(i)/ysq
      do 42 i=1,n
 42   y(i)=a(k,i)
      sum=0.
      ysq=0.
      do 52 i=1,n
      ysq=ysq+y(i)*y(i)
 52   sum=sum+y(i)*xb(i)
      res(k)=b(k)-sum
      do 62 i=1,n
 62   xc(i)=xb(i)+res(k)*y(i)/ysq
c     calculate ab, bc, and ac (vectors & lengths)
      do 43 i=1,n
      ab(i)=xb(i)-xa(i)
      bc(i)=xc(i)-xb(i)
 43   ac(i)=xc(i)-xa(i)
      sum=0.
      ysq=0.
      do 53 i=1,n
      ysq=ysq+ac(i)*ac(i)
 53   sum=sum+ab(i)*ab(i)
      abl=sqrt(sum)
      acl=sqrt(ysq)
      adl=om*abl*abl/acl
      do 63 i=1,n
 63   x0(i)=xa(i)+adl*ac(i)/acl
c 46  do 47 i=1,n
c 47  x0(i)=xb(i)
c
 200  continue
c
c     now store the odd-balls
c
      if(iodd.eq.1) then
      do 44 i=1,n
 44   y(i)=a(n,i)
      sum=0.
      ysq=0.
      do 54 i=1,n
      ysq=ysq+y(i)*y(i)
 54   sum=sum+y(i)*x0(i)
      res(n)=b(n)-sum
      do 64 i=1,n
 64   x0(i)=x0(i)+res(n)*y(i)/ysq
      end if
c
c     Error Calculation
c
      do 65 i=1,n
      sum=0.
      do 55 j=1,n
```

```
 55       sum=sum+a(i,j)*x0(j)
 65       err(i)=abs(b(i)-sum)
c         write(6,56)(err(i),i=1,n)
 56       format(' Errors:',5f10.5)
          errtot=0.
          do 66 i=1,n
 66       errtot=errtot+err(i)*err(i)
          errtot=sqrt(errtot)
          if(errtot.lt.0.00005 .or. itr.gt.100000) go to 67
c         write(6,*) itr, errtot
          itr=itr+1
          go to 100
 67       write(6,68)(x0(i),i=1,n)
 68       format(' Solution:',5f11.5)
          write(6,69) itr,om
 69       format(5x,'Obtained in ',i5,' iterations, with om=',f10.3)
          stop
          end
```

3) *qmconj.f*

```
          program qmconj
          parameter n= 20
          dimension x0(n), xa(n),xb(n),xc(n),ab(n),bc(n),ac(n),a(n,n),b(n),
    1     y(n),res(n),err(n)
c         real*8 sum,ysq,om,abl,acl,adl
          OPEN(UNIT=13, FILE='a.dat',STATUS='OLD')
          OPEN(UNIT=14, FILE='b.dat',STATUS='OLD')
c         OPEN(UNIT=15, FILE='xin.dat',FORM='FORMATTED',STATUS='OLD')
          read(13,*)((a(i,j),j=1,n),i=1,n)
          read(14,*)(b(i),i=1,n)
          write(6,3)((a(i,j),j=1,n),i=1,n)
 3        format(1x,9f8.3)
          write(6,3)(b(i),i=1,n)
c
          write(6,2)
 2        format(//5x,'Results using the Program qmconj.f'//)
c
c         a and b store the original Ax = B
cc        initial guess
case4     om=1.22
chilb4    om=1.97
case1     om=1.68
ccqm      om=1.29
          om=1.3
cpan      om=1.75
c         om=1.0
          itr=1
c         read(15,*)(x0(i),i=1,n)
          do 1 i=1,n
 1        x0(i)=0.
c 1       x0(i)=1.
c         write(6,3)(x0(i),i=1,n)
c
 100      continue
          do 200 k=2,n
c         write(6,*) itr, k
          do 4 i=1,n
 4        y(i)=a(1,i)
          sum=0.
          ysq=0.
          do 5 i=1,n
          ysq=ysq+y(i)*y(i)
 5        sum=sum+y(i)*x0(i)
          res(1)=b(1)-sum
c         write(6,*)(x0(i),i=1,n)
          do 6 i=1,n
```

```
 6      xa(i)=x0(i)+res(1)*y(i)/ysq
c
       do 41 i=1,n
 41    y(i)=a(k,i)
       sum=0.
       ysq=0.
       do 51 i=1,n
       ysq=ysq+y(i)*y(i)
 51    sum=sum+y(i)*xa(i)
       res(k)=b(k)-sum
c      write(6,*)(xa(i),i=1,n)
       do 61 i=1,n
 61    xb(i)=xa(i)+res(k)*y(i)/ysq
c      write(6,*)(xb(i),i=1,n)
       do 42 i=1,n
 42    y(i)=a(1,i)
       sum=0.
       ysq=0.
       do 52 i=1,n
       ysq=ysq+y(i)*y(i)
 52    sum=sum+y(i)*xb(i)
       res(1)=b(1)-sum
       do 62 i=1,n
 62    xc(i)=xb(i)+res(1)*y(i)/ysq
c      write(6,*)(xc(i),i=1,n)
c      calculate ab, bc, and ac (vectors & lengths)
       do 43 i=1,n
       ab(i)=xb(i)-xa(i)
       bc(i)=xc(i)-xb(i)
 43    ac(i)=xc(i)-xa(i)
       sum=0.
       ysq=0.
       do 53 i=1,n
       ysq=ysq+ac(i)*ac(i)
 53    sum=sum+ab(i)*ab(i)
       abl=sqrt(sum)
       acl=sqrt(ysq)
       adl=om*abl*abl/acl
       do 63 i=1,n
 63    x0(i)=xa(i)+adl*ac(i)/acl
c      write(6,*)(x0(i),i=1,n)
c
 200   continue
c
c      Error Calculation
c
       do 65 i=1,n
       sum=0.
       do 55 j=1,n
 55    sum=sum+a(i,j)*x0(j)
 65    err(i)=abs(b(i)-sum)
c      write(6,*)(x0(i),i=1,n)
c      write(6,56)(err(i),i=1,n)
 56    format(' Errors:',5f10.5)
       errtot=0.
       do 66 i=1,n
 66    errtot=errtot+err(i)*err(i)
       errtot=sqrt(errtot)
       if(itr.lt.10) write(6,*) itr, errtot
       if(errtot.lt.0.00005 .or. itr.gt.100000) go to 67
c      if(errtot.lt.0.000005 .or. itr.gt.100000) go to 67
       if(errtot.gt.10000.) go to 67
       itr=itr+1
        if(mod(itr,1000).eq.0) write(6,*) itr,errtot
       go to 100
 67    write(6,68)(x0(i),i=1,n)
 68    format(' Solution:',5f14.5)
       write(6,69) errtot,itr,om
```

```
 69   format(3x,'Err=',f10.6,' Obtained in',i7,' iterations, om=',f8.3)
      stop
      end
```

4) *simconj.f*

```
      program simconj
      parameter n= 20
      dimension x0(n), xa(n),xb(n),xc(n),ab(n),bc(n),ac(n),a(n,n),b(n),
     1  y(n),res(n),err(n)
c     real*8 sum,ysq,abl,acl,adl,bcl
      OPEN(UNIT=13, FILE='a.dat',STATUS='OLD')
      OPEN(UNIT=14, FILE='b.dat',STATUS='OLD')
c     OPEN(UNIT=15, FILE='xin.dat',FORM='FORMATTED',STATUS='OLD')
      read(13,*)((a(i,j),j=1,n),i=1,n)
      read(14,*)(b(i),i=1,n)
c     write(6,3)((a(i,j),j=1,n),i=1,n)
  3   format(1x,9f8.3)
c     write(6,3)(b(i),i=1,n)
c
      write(6,2)
  2   format(//5x,'Results using the Program simconj.f'//)
c
c     a and b store the original Ax = B
cc    initial guess
ccgm  om=1.2
      om=1.6
cb12  om=1.6
cpan  om=.05
c     om=1.2
c5    om=1.3
c     om=1.78
chilb4 om=1.95
chilb3 om=1.89
case4 om=2.9
c20tr om=2.368
      do 1 i=1,n
c 1   x0(i)=0.
  1   x0(i)=1.
c     read(15,*)(x0(i),i=1,n)
      itr=1
c
 100  continue
      do 200 k=1,n-1
      kp1=k+1
c     write(6,*) itr, k
      do 4 i=1,n
  4   y(i)=a(k,i)
      sum=0.
      ysq=0.
      do 5 i=1,n
      ysq=ysq+y(i)*y(i)
  5   sum=sum+y(i)*x0(i)
      res(k)=b(k)-sum
c     write(6,*)(x0(i),i=1,n)
      do 6 i=1,n
  6   xa(i)=x0(i)+res(k)*y(i)/ysq
c
      do 41 i=1,n
 41   y(i)=a(kp1,i)
      sum=0.
      ysq=0.
      do 51 i=1,n
      ysq=ysq+y(i)*y(i)
 51   sum=sum+y(i)*xa(i)
      res(kp1)=b(kp1)-sum
c     write(6,*)(xa(i),i=1,n)
```

```
      do 61 i=1,n
61    xb(i)=xa(i)+res(kp1)*y(i)/ysq
      do 42 i=1,n
42    y(i)=a(k,i)
      sum=0.
      ysq=0.
      do 52 i=1,n
      ysq=ysq+y(i)*y(i)
52    sum=sum+y(i)*xb(i)
      res(k)=b(k)-sum
      do 62 i=1,n
62    xc(i)=xb(i)+res(k)*y(i)/ysq
c     calculate ab, bc, and ac (vectors & lengths)
      do 43 i=1,n
      ab(i)=xb(i)-xa(i)
      bc(i)=xc(i)-xb(i)
43    ac(i)=xc(i)-xa(i)
      sum=0.
      ysq=0.
      do 53 i=1,n
      ysq=ysq+ac(i)*ac(i)
53    sum=sum+ab(i)*ab(i)
      abl=sqrt(sum)
      acl=sqrt(ysq)
      adl=om*abl*abl/acl
      do 63 i=1,n
63    x0(i)=xa(i)+adl*ac(i)/acl
c
200   continue
c
c     Error Calculation
c
      do 65 i=1,n
      sum=0.
      do 55 j=1,n
55    sum=sum+a(i,j)*x0(j)
65    err(i)=abs(b(i)-sum)
c     write(6,56)(err(i),i=1,n)
56    format(' Errors:',5f10.5)
      errtot=0.
      do 66 i=1,n
66    errtot=errtot+err(i)*err(i)
      errtot=sqrt(errtot)
      if(errtot.lt.0.00005 .or. itr.gt.1000000) go to 67
      if(errtot.gt.10000.) go to 67
      if(itr.lt.10) write(6,*) itr, errtot
      if(mod(itr,1000).eq.0) write(6,*) itr,errtot
      itr=itr+1
      go to 100
67    write(6,68)(x0(i),i=1,n)
68    format(' Solution:',5f11.5)
      write(6,69) errtot,itr,om
69    format(5x,'Err=',f10.6,' Obtained in ',i7,' iterations, with om=',f10.3)
      stop
      end
```

5) *congrad.f*

```
      program congrad
      parameter n = 20
      dimension a(n,n), b(n), e(n), r0(n),p1(n), pk(n), r1(n),x(n)
      dimension ap(n)
      open (unit = 3, file = 'a.dat', status = 'old')
      open (unit = 4, file = 'b.dat', status = 'old')
      read(3,*)((a(i,j), j=1,n),i=1,n)
      read(4,*)(b(i),i=1,n)
      write(6,111)
111   format(//'results obtained by congrad.f:'/)
```

```
      do 1 i =1,n
c     x(i) = 0.0
      x(i) = 1.0
 1    b(i) = - b(i)
c
      do 100 it=1,10000
      if(it.ge.2) then
      sumr0 = 0.
      sumr1 = 0.
      do 12 i = 1,n
      sumr0 = sumr0 + r0(i)*r0(i)
 12   sumr1 = sumr1 + r1(i)*r1(i)
      ekm1 = sumr1/sumr0
      do 13 i = 1,n
 13   pk(i) = -r1(i)+ekm1*pk(i)
      endif
c     write(6,*) ekm1
c     write(6,*) (pk(i),i=1,n)
c
      do 2 i=1,n
      sum = 0.
      do 3 j=1,n
 3    sum = sum+a(i,j)*x(j)
      r0(i) = sum + b(i)
      if(it.eq.1) pk(i) = - r0(i)
 2    p1(i) = - r0(i)
      do 4 i=1,n
      sum = 0.
      do 5 j =1,n
 5    sum = sum + a(i,j)*pk(j)
 4    ap(i) = sum
c     write(6,*) (ap(i),i=1,n)
      sumr = 0.
      sump = 0.
      do 6 i =1,n
      sumr = sumr + r0(i)*r0(i)
 6    sump = sump + ap(i)*pk(i)
c     write(6,*) sump
      qk = sumr/sump
      do 7 i = 1,n
      x(i) = x(i) + qk*pk(i)
 7    r1(i) = r0(i) + qk*ap(i)
c     write(6,*) qk
c     write(6,*) (x(i),i=1,n)
c     write(6,*) (r1(i),i=1,n)
      sumr = 0.
      do 8 i=1,n
 8    sumr = sumr + r1(i)*r1(i)
c     write(6,*) sumr
c     write(6,108)
 108  format(' ----------')
      sumr = sqrt(sumr)
      itr = it
      if(mod(itr,1000).eq.0) write(6,*) itr,sumr
      if(itr.le.10) write(6,*) itr,sumr
c     write(6,*) itr,sumr
      if(sumr.lt.0.000005) go to 101
 100  continue
      go to 103
 101  write(6,102) itr
 102  format(3x,'solution converged in',i4,' iterations')
      write(6,105) (x(i),i=1,n)
      go to 106
 103  write(6,104) itr
 104  format(3x,'Solution did not converge in',i6,' iterations')
 105  format(5f14.6)
 106  stop
      end
```

6) *gsol2v.f*

```
      program gsol2v
      parameter n = 2004
      dimension b(n), u(n),v(n,n),p(n),po(n)
      dimension x(n),a(n),tn(n),f1(n),f2(n),ab(n),bc(n)
c     x(i), p(i) are the 1 plus n base projection points
c
      open (unit = 3, file = 'a.dat', status = 'old')
      open (unit = 4, file = 'b.dat', status = 'old')
      open (unit = 7, file = 'gsol2v.out', status = 'new')
      read(4,*)(b(i),i=1,n)
      write(7,111)
 111  format(//'results obtained by gsol2v.f:'/)
      do 1 i =1,n
      x(i) = 1./sqrt(float(n+1))
 1    p(i) = x(i)
c
      do 9 i=1,n
c
      read(3,*)(a(j),j=1,n)
      do 3 j=1,n
 3    f2(j)=x(j)
      call dotp(a,f2,n,sun)
      call dotp(a,a,n,su)
      rs=b(i)-sun
c
c     create p(n) to store the n base points, one at a time
c
      do 2 j=1,n
 2    po(j)=p(j)
      do 7 j=1,n
 7    p(j)=rs*a(j)/su + x(j)
c     write(6,*)(p(j),j=1,n)
c
c     Start the nested Looping
c     Use the variable v(n,n) to store the orthonormal base
c
      if(i.eq.1) then
      do 8 j=1,n
 8    ab(j)=p(j)-x(j)
      call dotp(ab,ab,n,sum)
      abl=sqrt(sum)
      rl=abl/2.
      do 10 j=1,n
      u(j)=ab(j)/abl
 10   tn(j)=x(j)+rl*u(j)
c
c     calculate al and use later !
      do 19 j=1,n
      f1(j)=x(j)
 19   f2(j)=x(j)
      call dotp(f1,f2,n,al)
c
      do 11 j=1,n
 11   v(i,j)=u(j)
c
      else
c     j greater than 2 below
      do 12 j=1,n
 12   bc(j)=p(j)-po(j)
      call dotp(bc,bc,n,sum)
      do 13 j=1,n
 13   u(j)=bc(j)/sqrt(sum)
c
c     write(6,*)' u =',(u(j),j=1,n)
c     Gram-Schmidt Orthonormalization
```

```
c
      do 14 k=1,i-1
c
      do 15 j=1,n
      f1(j)=u(j)
 15   f2(j)=v(k,j)
      call dotp(f1,f2,n,sum)
      do 16 j=1,n
 16   v(i,j)=v(i,j)+sum*v(k,j)
 14   continue
      do 17 j=1,n
      v(i,j)=u(j)-v(i,j)
      f1(j)=v(i,j)
 17   f2(j)=v(i,j)
      call dotp(f1,f2,n,sum)
      do 18 j=1,n
 18   v(i,j)=v(i,j)/sqrt(sum)
c
c     write(6,*)' v=',(v(i,j),j=1,n)
c     Proceed with the new vector
c
      do 20 j=1,n
      f1(j)=p(j)
 20   f2(j)=p(j)
      call dotp(f1,f2,n,bet)
      do 21 j=1,n
      f1(j)=p(j)-x(j)
 21   f2(j)=tn(j)
c     write(6,*)' tn=',(tn(j),j=1,n)
      call dotp(f1,f2,n,gam)
      rnu=(bet-al)/2. - gam
      do 22 j=1,n
 22   f2(j)=v(i,j)
      call dotp(f1,f2,n,rde)
      rl=rnu/rde
      do 23 j=1,n
 23   tn(j)=tn(j)+rl*v(i,j)
c     write(6,*)' tn=',(tn(j),j=1,n)
      end if
 9    continue
c
      do 201 i=1,n
 201  x(i)=x(i)+2.*(tn(i)-x(i))
c
      write(7,*) (x(i),i=1,n)
c     write(6,207) (x(i),i=1,n)
 207  format(2x,5f15.5)
      stop
      end
      subroutine dotp(a,b,n,sum)
      dimension a(n),b(n)
      sum=0.
      do 1 i=1,n
 1    sum=sum+a(i)*b(i)
      return
      end
```

7) *gsol.f*

```
      Program gsol
      Parameter n = 2004
      Include 'param.dat'
      Include 'common.f'
      Dimension x(n),a(n),f2(n),p(n)
C     X(i), p(i) are the 1 plus n base projection points
C
      Open (unit = 3, file = 'a.dat', status = 'old')
```

```
      Open (unit = 4, file = 'b.dat', status = 'old')
      Read(3,*)((dubicww(i,j),j=1,n),i=1,n)
      Read(4,*)(rhsv(i),i=1,n)
      Write(6,111)
111   Format(//'results obtained by gsol22.f:'/)
      Do 1 I =1,n
1     X(i) = 0.
C
      Do 9 i=1,n
C
C     Read(3,*)(a(j),j=1,n)
      Do 12 j=1,n
12    A(j)=dubicww(i,j)
      Call dotp(a,x,n,sun)
      Call dotp(a,a,n,su)
      Rs=rhsv(i)-sun
C
C     Create p(n) to store the n base points, one at a time
C
      Do 7 j=1,n
7     P(j)=rs*a(j)/su
      Do 8 j=1,n
8     F2(j)=p(j)+x(j)
      Call dotp(f2,f2,n,sup)
      Call dotp(x,x,n,sux)
C     Write(7,*)(p(j),j=1,n),(sup-sux)/2.
      Rhsv(i)=(sup-sux)/2.
      Do 9 j=1,n
      Dubicww(i,j)=p(j)
9     Continue
C
      Call gsol22
C
      Do 10 i=1,n
10    X(i)=2.0*dub(i)-x(i)
      Write(78,11)
11    Format(5x,'with the above circumcenter, the calculated solution is:')
      Write(78,*) (x(i),i=1,n)
      Stop
      End
C
      Subroutine dotp(a,b,n,sum)
      Dimension a(n),b(n)
      Sum=0.
      Do 1 i=1,n
1     Sum=sum+a(i)*b(i)
      Return
      End

*deck doublet
   Subroutine gsol22
C
C-----------------------------------------------------------
C
   Include 'param.dat'
   Include 'common.f'
C
      Open(unit=16, file='data6',form='formatted',status='new')
      Open(unit=20, form='unformatted',status='unknown')
C
      Do 2 I = 1,nspdim
2     Diag(i) = dubicww(i,i)
C
C Rewind all scratch file 20 and assign unit number
C
   Imu = 20
   Rewind imu
C
```

```
C  Check to make sure that if inram is not 1, that it is set equal to nspdim
C
     If(inram.ne.1)then
      If(inram.ne.nspdim)then
        Write(16,601)
        Stop
      Endif
     Endif
C
C  Start the time step loop
C
     Tstime = 0.0
C
C  Write input data to output file
C
     Write(16,603)
C
  603 format(1x,10(/),31x,57('*')//
    +      54x,'program pmarc'/
    +      45x,'release version 12.21:  03/04/94'//
    +      51x,'matrix solver extracted from pmarc'/)
C
      Call solver
C
              Open(unit=78, file='gsol.out', status ='new')
              Write(78,*) (dub(i),i=1,nspdim)
C
C  Close and delete the scratch files
C
      Close(unit=20,status='delete')
C
     Return
  600 format(//1x,'time step',i4)
  601 format(//1x,'parameter inram not set to 1 or nspdim in pmarc'/
    +      1x,'source code. Reset this parameter, recompile code'/
    +      1x,'and try again.')
     End
C
*deck solver
     Subroutine solver
C
C-----------------------------------------------------------------------
c
     Include 'param.dat'
     Include 'common.f'
C
C  Update the pdub array so that it always holds the previous step's doublet
C  Solution
C
              Itstep=0
              Npan = nspdim
C
     Do 10 i=1,npan
      If(itstep.eq.0)then
       Pdub(i) = rhsv(i)
      Else
       Pdub(i) = dub(i)
              Endif
   10 continue
     Call lineq(it)
              Write(6,555) it
  555         Format(1x,'number of iterations    = 'i5)
     Write(16,600)it
C
     Return
  600 format(1x,'number of solver iterations = ',i4)
     End
     Subroutine lineq(it)
```

```
C
C   Program to solve linear equations (based on: j. Comp. Phisics,
C   "The iterative calculation of....", 17. Pp. 87-94, 1975)
C   For information: call charley bauschlicher (415) 694-6231
C
    Include 'param.dat'
    Include 'common.f'
C
    Dimension v(nspdim,20), w(nspdim,20),a(20,20),al(20),
   + buf(nspdim), gg(20), buf1(400), buf2(nspdim), buf3(nspdim)
C
    It = 0
            Npan = nspdim
            Solres = 0.000005
            Maxit = 200
    Thresh = solres
    Matdim = npan
    Ims = 0
c
C  Initial guess for starting solution vector
C
    If(itstep.gt.0)then
     Do 10 i=1,matdim
      Buf2(i) = pdub(i)
  10  Continue
     Go to 800
    Endif
    Do 20 i=1,matdim
     Ahnn = diag(i)
     If(abs(ahnn).lt.1.e-7)ahnn = 1.0e-7
     Buf2(i) = rhsv(i)/ahnn
  20 continue
  800 continue
    Write(16,600)
    Write(16,601)
            Write(6,899)
  899       Format(1x,'solution iteration history')
  810 continue
    Ims = ims + 1
    Call normal(buf2,matdim)
    It = it + 1
    Call frmab(buf2,buf3,matdim,imu)
    Do 30 i=1,matdim
     W(i,ims) = buf3(i)
     V(i,ims) = buf2(i)
  30 continue
    Do 40 i=1,ims
     Do 50 j=1,matdim
      Buf(j) = v(j,i)
      Buf2(j) = w(j,ims)
  50  Continue
     A(i,ims) = sdot(matdim,buf,1,buf2,1)
     If(i.eq.ims)go to 40
     Do 60 j=1,matdim
      Buf(j) = w(j,i)
      Buf2(j) = v(j,ims)
  60  Continue
     A(ims,i) = sdot(matdim,buf,1,buf2,1)
  40 continue
    Do 70 i=1,matdim
     Buf2(i) = v(i,ims)
  70 continue
    Gg(ims) = sdot(matdim,rhsv,1,buf2,1)
    Iq = 0
    Do 80 i=1,ims
     Do 90 j=1,ims
      Iq = iq + 1
      Buf1(iq) = a(j,i)
```

```
90  Continue
    Al(i) = gg(i)
80 continue
   Call gss(buf1,al,ims,ier)
   If(ier.eq.1)then
    Stop
   Endif
   Cony = abs(al(ims))
   Ifold = 0
   If(ims.eq.20)then
    Ifold = ims
    Call zero(buf2,matdim)
    Do 100 i=1,ims
     Do 110 j=1,matdim
      Buf(j) = w(j,i)
110    Continue
     Ali = al(i)
     Call saxpy(matdim,ali,buf,1,buf2,1)
100  Continue
    Do 120 j=1,matdim
     W(j,1) = buf2(j)
120  Continue
    X = 0.0
    Y = 0.0
    Do 130 i=1,ims
     Do 140 j=1,ims
      X = x + a(i,j) * al(i) * al(j)
140    Continue
     Y = y + gg(i) * al(i)
130  Continue
    A(1,1) = x
    Gg(1) = y
    Ims = 1
   Endif
   Call zero(buf2,matdim)
   Do 150 i=1,ims
    Do 160 j=1,matdim
     Buf(j) = w(j,i)
160  Continue
    Ali = al(i)
    If(ifold.ne.0)ali = 1.0
    Call saxpy(matdim,ali,buf,1,buf2,1)
150 continue
   Conx = 0.0
   Ipanel = 1
   Do 170 i=1,matdim
    If(abs(rhsv(i)).lt.1.e-7) go to 820
    Q = abs((buf2(i) - rhsv(i))/rhsv(i))
    If(conx.lt.q)then
     Ipanel = I
    Endif
    Conx = amax1(conx,q)
820  Continue
    Buf2(i) = buf2(i) - rhsv(i)
170 continue
   Write(16,602)it,cony,conx,ipanel
   Noconv = 0
   If(conx.lt.thresh.and.ifold.eq.0)go to 830
   Noconv = 1
   If(it.eq.maxit) go to 830
   Do 180 i=1,matdim
    Ahnn = diag(i)
    If(abs(ahnn).lt.1.0e-7)ahnn = 1.0e-7
    Buf2(i) = buf2(i)/ahnn
180 continue
   Call normal(buf2,matdim)
   Lp = max0(ifold,ims)
   Do 190 i=1,lp
```

```
      Do 200 j=1,matdim
       Buf(j) = v(j,i)
  200  Continue
      X = sdot(matdim,buf,1,buf2,1)
      Call saxpy(matdim,-x,buf,1,buf2,1)
      Call normal(buf2,matdim)
  190 continue
     If(ifold.eq.0)go to 810
     Call zero(buf3,matdim)
     Do 210 i=1,ifold
      Do 220 j=1,matdim
       Buf(j) = v(j,i)
  220  Continue
      Ali = al(i)
      Call saxpy(matdim,ali,buf,1,buf3,1)
  210 continue
     Do 230 i=1,matdim
      V(i,1) = buf3(i)
  230 continue
     Al(1) = 1.0
     Go to 810
  830 continue
     Call zero(buf,matdim)
     Do 240 i=1,ims
      Do 250 j=1,matdim
       Buf2(j) = v(j,i)
  250  Continue
      Alil = al(i)
      Call saxpy(matdim,alil,buf2,1,buf,1)
  240 continue
     Do 260 i=1,matdim
      Dub(i) = buf(i)
  260 continue
     If(noconv.eq.1) then
            Write(16,603)
            End if
  600 format(1h1)
  601 format(1x,'solution iteration history'/)
  602 format(' it=',i5,' al(i) ',f15.8,' hv-g ',f15.8,' panel = ',i5)
  603 format(//'----------no convergence----------')
     Return
     End
     Subroutine frmab(a,b,matdim,irawm)
C
     Include 'param.dat'
     Include 'common.f'
C
     Dimension a(matdim),b(matdim)
     Rewind irawm
     Npan = matdim
     Do 10 i=1,npan
      Ii = I
      B(i) = 0.0
      If(inram.eq.1)then
       Read(irawm)(dubicww(inram,j),j=1,npan)
       Ii = 1
      Endif
      Do 20 j=1,npan
       B(i) = b(i) + a(j) * dubicww(ii,j)
   20  Continue
   10 continue
     Return
     End
     Subroutine zero(a,len)
     Dimension a(len)
     Do 10 i=1,len
      A(i) = 0.0
   10 continue
```

```
      Return
      End
      Subroutine normal(a,len)
      Dimension a(len)
      X = 0.0
      Do 10 i=1,len
        X = x + a(i) * a(i)
   10 continue
      X = 1.0/sqrt(x)
      Do 20 i=1,len
        A(i) = a(i) * x
   20 continue
      Return
      End
      Subroutine gss(b,g,nmix,ier)
      Dimension b(nmix,nmix),g(nmix)
      Data zero1/1.0e-16/
      Ier = 0
      Do 10 i=1,nmix
       If(abs(b(i,i)).lt.zero1)go to 800
       Fx = 1./b(i,i)
       Go to 810
  800  Continue
       If(i.eq.nmix)go to 820
       I1 = I + 1
C  Pivot section
       Do 20 j=i1,nmix
        If(abs(b(j,i)).lt.zero1)go to 20
        Fx = 1./b(j,i)
        Do 30 l=i,nmix
          Temp = b(j,l)
          B(j,l) = b(i,l)
          B(i,l) = temp
   30   Continue
        Tmp = g(j)
        G(j) = g(i)
        G(i) = tmp
        Go to 810
   20  Continue
       Go to 820
  810  Continue
       G(i) = g(i) * fx
       Do 40 j=i,nmix
        B(i,j) = b(i,j) * fx
   40  Continue
       Do 50 j=1,nmix
        If(i.eq.j)go to 50
        Y = b(j,i)
        G(j) = g(j) - g(i) * y
        Do 60 k=i,nmix
          B(j,k) = b(j,k) - y * b(i,k)
   60   Continue
   50  Continue
   10 continue
      Return
  820 continue
      Write(16,600)
  600 format(' Abort invert  Singular matrix ')
      Ier = 1
      Return
      End
      Subroutine saxpy(n,sa,sx,incx,sy,incy)
C
C     Constant times a vector plus a vector.
C     Uses unrolled loops for increments equal to one.
C     Jack dongarra, linpack, 3/11/78.
C
      Dimension sx(n),sy(n)
```

```
C
      If(n.le.0)return
      If(sa.eq.0.0)return
      If(incx.eq.1.and.incy.eq.1)go to 20
C
C     Code for unequal increments or equal increments
C     Not equal to 1
C
      Ix=1
      Iy=1
      If(incx.lt.0)ix=(-n+1)*incx+1
      If(incy.lt.0)iy=(-n+1)*incy+1
      Do 10 i=1,n
       Sy(iy)=sy(iy)+sa*sx(ix)
       Ix=ix+incx
       Iy=iy+incy
   10 continue
      Return
C
C     Code for both increments equal to 1
C
C
C     Clean-up loop
C
   20 m=mod(n,4)
      If(m.eq.0)go to 40
      Do 30 i=1,m
       Sy(i)=sy(i)+sa*sx(i)
   30 continue
      If(n.lt.4)return
   40 mp1=m+1
      Do 50 i=mp1,n,4
       Sy(i)=sy(i)+sa*sx(i)
       Sy(i+1)=sy(i+1)+sa*sx(i+1)
       Sy(i+2)=sy(i+2)+sa*sx(i+2)
       Sy(i+3)=sy(i+3)+sa*sx(i+3)
   50 continue
      Return
      End
      Real function sdot(n,sx,incx,sy,incy)
C
C     Forms the dot product of two vectors.
C     Uses unrolled loops for increments equal to one.
C     Jack dongarra, linpack, 3/11/78.
C
      Dimension sx(n),sy(n)
C
      Stemp=0.0e0
      Sdot=0.0e0
      If(n.le.0)return
      If(incx.eq.1.and.incy.eq.1)go to 20
C
C        Code for unequal increments or equal increments
C        Not equal to one.
C
      Ix=1
      Iy=1
      If(incx.lt.0)ix=(-n+1)*incx+1
      If(incy.lt.0)iy=(-n+1)*incy+1
      Do 10 i=1,n
       Stemp=stemp+sx(ix)*sy(iy)
       Ix=ix+incx
       Iy=iy+incy
   10 continue
      sdot=stemp
      return
C
C     Code for both increments equal to 1
```