ITERATIVE METHODS FOR SOLVING LARGE LINEAR SYSTEMS IN THE MOMENT METHOD ANALYSIS OF ELECTROMAGNETIC SCATTERING

By

JAMES MICHAEL STURM

Bachelor of Science

Oklahoma State University

1991

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1993

OKLAHOMA STATE UNIVERSITY

ITERATIVE METHODS FOR SOLVING LARGE LINEAR SYSTEMS IN THE MOMENT METHOD ANALYSIS OF ELECTROMAGNETIC SCATTERING

Thesis Approved:

James C. West
Thesis Advisor

[illegible]

J Chandler

Thomas C. Collins
Dean of the Graduate College

ACKNOWLEDGMENTS

I extend my sincere thanks and appreciation to my thesis advisor Dr. James C. West in the Electrical Engineering department of Oklahoma State University for the guidance and patience he has shown me in this endeavor. Along with Dr. West's help, I received considerable support from his doctoral candidate, Ruimin Chen. Not only did Ruimin supply the moment method source code for testing the iterative algorithms, he copied his notes on the development of the magnetic field integral equation. Dr. John Chandler of the Computer Science department at OSU assisted me in improving the efficiency of my source code and in suggesting algorithms to test. This work was supported by the Office of Naval Research under grant N00014-92-J-1206.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Since its development in World War II, radar has provided accurate detection and ranging of aircraft and meteorological disturbances. Recent efforts have been to understand radar backscatter from ocean waves at near grazing incidence for naval applications. For radars aboard ships or low-flying aircraft, the ocean surface presents an unpredictably complex shadowed surface [38]. Experimental characterization of the scattering process is unreliable because the sea-surface statistics are not well known, and the instantaneous surface profile at the time of the radar measurement cannot be determined. Instead, most theoretical analyses of the scattering process have centered on numerically modeling the interactions between the radar waves and the sea surface. Most promising are the applications of the moment method (MM) [15] and the geometrical theory of diffraction (GTD) [18] to this problem. Unfortunately, these techniques are computationally extensive, limiting the size of the surface that can be modeled, and therefore limiting their application to unrealistically small surfaces.

When adapting the MM to electrically large scatterers like the ocean, large systems of linear equations must be solved [15]. Direct methods such as LU (Lower / Upper) Decomposition or Gaussian elimination have been used extensively to solve such systems. These algorithms are popular because they provide exact solutions (neglecting computer round-off errors) after a finite number of multiplications. Unfortunately, direct methods are inefficient in both computer storage requirements as well as number of floating point operations. As the system size increases, these requirements can overwhelm even the largest supercomputers.

Iterative numerical solutions to the matrix equation overcome many of the inefficiencies of direct methods, producing approximate results with many fewer multiplications and reduced memory storage. Iterative methods repeatedly approximate the solution to the linear system until a desired precision is achieved [41]. Unfortunately, most iterative methods are not guaranteed to converge to the solution unless certain matrix properties are met.

Iterative methods can be implemented in ways that require much less storage than direct methods. Both Gaussian elimination and LU Decomposition modify the matrix in the process of finding the solution. Such methods, therefore, require the storage of non-zero matrix elements in memory or on disks and cannot easily handle matrices that exceed the storage capacities of the computer. Iterative methods, on the other hand, do not alter the matrix elements, allowing for configurations where the matrix is not stored at all. Instead, matrix elements are regenerated as needed.

Iterative methods often require much less execution time than direct methods. Since they treat all matrices uniformly, direct methods will produce the correct result every time, but only at great expense computationally. Given a matrix equation of dimension N, direct methods require O(N^3) multiplications to reach an answer [20]. Iterative methods, however, use such properties as diagonal dominance and sparseness to reduce the amount of computations significantly. Most iterative methods converge successfully using O(N^2) multiplications [32].

Since the computer uses binary representation for decimal and real numbers, errors can result from the computer's inability to store numbers of infinite precision. Both direct and indirect methods are prone to these round-off errors. Since most direct methods modify the matrix coefficients, every subsequent operation carries with it the errors of the previous operation. Fortunately, these errors are negligible for well-conditioned or moderately ill-conditioned problems [22]. Iterative methods have the advantage of using more of the original data, since the coefficient matrix is not modified [16].

R. F. Harrington [15] provided the foundation for the moment method as applied to electromagnetic problems. In 1980, Glisson and Wilton [10] outlined techniques of applying the moment method to electromagnetic scattering from conducting strips, bent rectangular plates, and bodies of revolution. In their treatment, the electric field integral equation (EFIE) was employed to model the radiation. Rao et. al. [28] extended the technique to surfaces of arbitrary shape. In their application of the EFIE and the moment method, triangular surface patches were used to represent the current distribution. This investigation successfully modeled scattering from surfaces such as square plates, bent plates, circular disks, and spheres [28].

Sarkar et. al. [32] surveyed numerical methods for use with scattering and radiation problems. They discussed both direct and iterative methods for solving the matrix equations. Iterative methods mentioned included the Jacobi, Gauss-Seidel, successive overrelaxation, steepest descent, and conjugate-gradient (CG) algorithms. This work included a discussion of convergence and round-off errors inherent in each technique. Recently, most scattering works that use iterative solution methods have concentrated on the CG technique, primarily because it is guaranteed to converge after a finite number of iterations. For example, Sarkar and Rao [30] used this technique in their analysis of scattering from arbitrarily oriented wire antennas. Besides confirming the attractiveness of the CG method, they advocated not storing an interaction matrix. Instead, they solved the scattering equation by incorporating it directly into the CG method instead of first generating a matrix equation using the moment method. Sultan and Mittra [34] analyzed the field distribution inside inhomogeneous lossy dielectric objects using the CG method. They also proposed not storing an interaction matrix but regenerating matrix elements as needed by the iterative method. Peterson and Mittra [25] [26] applied CG to individual and periodic structures as well as to large electromagnetic scatterers.

While CG has been most popular recently, there are several other iterative techniques, both newly developed and previously well known, that may be useful for the calculation of rough surface scattering. For example, standard techniques such as Jacobi [7], Gauss-Seidel [7], and successive overrelaxation [41] usually require more iterations to converge, but require fewer calculations per iteration and therefore may ultimately be more efficient. Although these methods are not guaranteed to converge in the general case, careful selection of the mathematical description of the scattering process yields systems that always converge.

Recently, several new iterative techniques have been introduced. Hageman and Young [14] recommended symmetric successive overrelaxation (SSOR) with Chebychev or conjugate gradient acceleration. In 1976, Fletcher [6] presented a new variation of the conjugate gradient method for indefinite systems called the method of biconjugate gradients (BCG). Although BCG does not require the matrix to be positive definite, convergence is not guaranteed for asymmetric matrices. Shortly thereafter, Young and Jea [40] introduced conjugate-gradient acceleration of nonsymmetrizable iterative methods, giving birth to such methods as ORTHODIR [40], ORTHOMIN [40], and ORTHORES [40]. These algorithms speed convergence of basic iterative techniques for sparse matrices. In 1986, Saad and Schultz [29] outlined the generalized minimal residuals (GMRES) method for solving nonsymmetric linear systems. This method is theoretically equivalent to ORTHODIR and the generalized conjugate residual (GCR) method developed by Elman [5]. Saad and Schultz claim that GMRES requires less multiplications and storage space than either of these methods. More recently, Sonneveld [33] introduced a variation of CG named the conjugate gradient squared (CGS) method that, according to Sonneveld, requires less work per digit of solution than many other algorithms. Finally, H. A. Van Der Vorst [35] proposed a variant of the BCG method, termed BICGSTAB, that converges in a more stable manner.

The goal of this thesis is to investigate the suitability of several iterative techniques for solving the linear systems of equations generated when applying the moment method to rough-surface-scattering problems. Chapter 2 investigates the electromagnetic analysis and development of the moment method code required for the solution of the ocean scattering problem. Chapter 3 discusses the theory of iterative methods that solve linear systems of equations, while chapter 4 details the computer implementation of these algorithms. Experimental results are presented in Chapter 5, and conclusions are given in Chapter 6.

CHAPTER 2

ELECTROMAGNETIC ANALYSIS

Introduction

This chapter discusses the electromagnetic problem and the formulation of the moment method code that was used for testing the iterative methods. The purpose of the investigation is to better model radar scattering from the ocean surface at near grazing angles. The radar scattering can be found first by numerically applying the moment method to the magnetic field integral equation (MFIE) [2] to determine the induced surface current, and then by finding the reradiation of this current using Maxwell's equations. Since the MFIE achieves well-conditioned matrix equations best suited for iterative solution, it has been favored over another integral equation, the electric field integral equation (EFIE) [27].

3-D Magnetic Field Integral Equation

The scattering from a perfectly conducting, 1λ by 5λ, where λ is the radar wavelength, surface is modeled in this investigation. The surface has uniform sinusoidal roughness in the x-direction, and is constant in the y-direction, as shown in Figure 1.

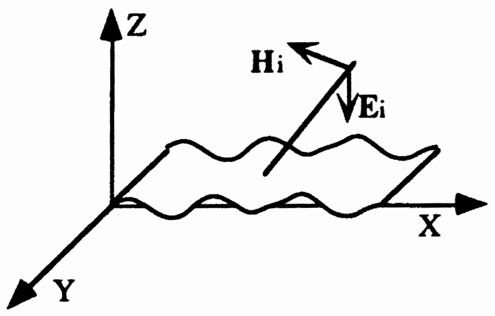

Figure 1. Geometry for Sinusoidal Wave Surface.

The electromagnetic plane of incidence is the $x-z$ plane. For this geometry, unit vectors for the rectangular coordinate system in the x, y, and z directions are $\hat{\mathbf{a}}_x$, $\hat{\mathbf{a}}_y$, and $\hat{\mathbf{a}}_z$, respectively. The incident electric and magnetic fields are given by $\mathbf{E}_i$ and $\mathbf{H}_i$, respectively.

This surface profile is unrealistic. The actual ocean surface is described by a two-dimensional roughness spectrum. Also, the small dimensions of the surface will introduce edge-diffraction effects into the calculated backscatter. Finally, the ocean surface is not actually a perfect electric conductor (PEC) at microwave frequencies. However, this geometry is adequate for determining the suitability of each iterative technique for general rough-surface scattering calculations.

The surface current density, $\mathbf{J}_S$, induced upon a PEC is proportional to the magnitude of the magnetic field tangent to the surface, and is given by [2]

$$\mathbf{J}_S = \hat{\mathbf{a}}_n \times (\mathbf{H}_s + \mathbf{H}_i) = \hat{\mathbf{a}}_n \times \mathbf{H}, \tag{2.1}$$

where $\mathbf{H}_s$ is the scattered magnetic field, $\mathbf{H}$ is the total magnetic field, and $\hat{\mathbf{a}}_n$ is the unit vector normal to the surface. The magnetic field integral equation (MFIE) describing the current induced on a perfectly conducting surface by an incident field is [27] [2]

$$\mathbf{J}_S = 2\hat{\mathbf{a}}_n \times \mathbf{H}_i + \frac{1}{2\pi}\hat{\mathbf{a}}_n \times \fint_S \mathbf{J}_S \times \nabla' G(R)\, ds', \tag{2.2}$$

where $\fint$ is the principle value integral around the singularity at $R = 0$ [27], the integration is over the entire surface of the scatterer, and $G(R)$ is the three dimensional Green's function given by [2]

$$G(R) = \frac{e^{-jkR}}{R}. \tag{2.3}$$

In (2.3), the propagation constant is

$$k = \frac{2\pi}{\lambda}, \tag{2.4}$$

and the scalar distance between the source and observation points is

$$R = |\mathbf{R}| = |\mathbf{r} - \mathbf{r}'| = \sqrt{(x-x')^2 + (y-y')^2 + (z-z')^2}, \tag{2.5}$$

where the vector from the origin to the observation point is

$$\mathbf{r} = \hat{\mathbf{a}}_x x + \hat{\mathbf{a}}_y y + \hat{\mathbf{a}}_z z, \tag{2.6}$$

and the vector from the origin to the source point is

$$\mathbf{r}' = \hat{\mathbf{a}}_x x' + \hat{\mathbf{a}}_y y' + \hat{\mathbf{a}}_z z'. \tag{2.7}$$

In these equations, the primed coordinate system denotes the source, and the unprimed coordinate system denotes the observation. The MFIE forces the magnetic field boundary condition of (2.1) to be met at the scattering surface. Once the surface current density is found by solving (2.2), the reradiated fields are determined using Maxwell's equations.

The three-dimensional vector MFIE can be rearranged into a set of scalar equations suitable for solving using the moment method [4]. First, the current density for the surface of Figure 1 is broken into tangential and normal components as

$$\mathbf{J}_S = \hat{\mathbf{a}}_y J_y + \hat{\mathbf{a}}_t J_t, \tag{2.8}$$

where J_y is the component in the y direction, J_t is the tangential component in the x-z plane, and $\hat{\mathbf{a}}_t$ is the unit vector tangent to the surface and perpendicular to $\hat{\mathbf{a}}_y$. The unit vectors are mutually orthogonal satisfying

$$\begin{aligned} \hat{\mathbf{a}}_y \times \hat{\mathbf{a}}_t &= \hat{\mathbf{a}}_n \\ \hat{\mathbf{a}}_n \times \hat{\mathbf{a}}_y &= \hat{\mathbf{a}}_t \; . \\ \hat{\mathbf{a}}_t \times \hat{\mathbf{a}}_n &= \hat{\mathbf{a}}_y \end{aligned} \tag{2.9}$$

Taking the dot product of (2.2) and $\hat{\mathbf{a}}_y$ gives

$$\hat{\mathbf{a}}_y \cdot \mathbf{J}_S = \hat{\mathbf{a}}_y \cdot 2\hat{\mathbf{a}}_n \times \mathbf{H}_i + \frac{1}{2\pi}\hat{\mathbf{a}}_y \cdot \hat{\mathbf{a}}_n \times \oint_S \mathbf{J}_S \times \nabla' G(R)\, ds'. \tag{2.10}$$

Next, applying the identity

$$\mathbf{A}\cdot\mathbf{B}\times\mathbf{C} = \mathbf{A}\times\mathbf{B}\cdot\mathbf{C} \tag{2.11}$$

to (2.10) yields

$$\hat{\mathbf{a}}_y \cdot \mathbf{J}_S = \hat{\mathbf{a}}_y \times \hat{\mathbf{a}}_n \cdot 2\mathbf{H}_i + \frac{1}{2\pi}\hat{\mathbf{a}}_y \times \hat{\mathbf{a}}_n \cdot \oint_S \mathbf{J}_S \times \nabla' G(R)\, ds'. \tag{2.12}$$

Evaluating the unit vector cross products and rearranging gives

$$\hat{\mathbf{a}}_y \cdot \mathbf{J}_S + \frac{1}{2\pi}\oint_S \hat{\mathbf{a}}_t \cdot \mathbf{J}_S \times \nabla' G(R)\, ds' = -2\hat{\mathbf{a}}_t \cdot \mathbf{H}_i. \tag{2.13}$$

Substituting for the surface current density in (2.8) and applying (2.11) again yields

$$J_y + \frac{1}{2\pi}\oint_S \hat{\mathbf{a}}_t \times \hat{\mathbf{a}}_y J_y \cdot \nabla' G(R)\, ds' = -2H_{it}, \tag{2.14}$$

where H_{it} is the component of the incident magnetic field tangent to the surface.

Next, the tangential unit vector is broken into its orthogonal components as

$$\hat{\mathbf{a}}_t = l_x\hat{\mathbf{a}}_x + l_z\hat{\mathbf{a}}_z, \tag{2.15}$$

where

$$l_x = \frac{dx}{\sqrt{dx^2 + dz^2}}, \tag{2.16}$$

and

$$l_z = \frac{dz}{\sqrt{dx^2 + dz^2}}. \tag{2.17}$$

Substituting (2.15) into (2.14) yields

$$J_y + \frac{1}{2\pi}\int_S (l_x\hat{\mathbf{a}}_x + l_z\hat{\mathbf{a}}_z)\times\hat{\mathbf{a}}_y J_y \cdot \nabla' G(R)\, ds' = -2H_{it}, \tag{2.18}$$

$$J_y + \frac{1}{2\pi}\int_S J_y (l_x\hat{\mathbf{a}}_z - l_z\hat{\mathbf{a}}_x)\cdot \nabla' G(R)\, ds' = -2H_{it}. \tag{2.19}$$

Taking the gradient of the Green's function,

$$\nabla' G(R) = \frac{1+jkR}{R^2} e^{-jkR}\frac{\mathbf{R}}{R}, \tag{2.20}$$

and substituting into (2.19) gives

$$J_y + \frac{1}{2\pi}\int_S J_y (l_x\hat{\mathbf{a}}_z - l_z\hat{\mathbf{a}}_x)\cdot \mathbf{R}\frac{1+jkR}{R^3} e^{-jkR}\, ds' = -2H_{it}. \tag{2.21}$$

Evaluating the dot product produces the scalar MFIE for the y component of the current,

$$\frac{1}{2}J_y + \frac{1}{4\pi}\int_S [(z-z')l_x - (x-x')l_z] J_y \frac{1+jkR}{R^3} e^{-jkR}\, ds' = -H_{it}. \tag{2.22}$$

Taking the dot product of (2.2) with $\hat{\mathbf{a}}_t$ and following a similar procedure yields the scalar MFIE for the tangential component of the current:

$$\frac{1}{2}J_t + \frac{1}{4\pi}\int_S [(z-z')l_x - (x-x')l_z] J_t \frac{1+jkR}{R^3} e^{-jkR}\, ds' = H_{iy}, \tag{2.23}$$

where H_{iy} is the y component of the incident magnetic field. Since (2.22) and (2.23) contain the unknown both outside and inside the integral, these are integral equations of the second type [27], which are well suited for iterative solution since they lead to moment method coefficient matrices that are diagonally dominant.

Moment Method

The moment method (MM) is used to convert integro-differential equations to a form easily solvable by computers [15]. The discretization of the equation produces a linear matrix equation that can be solved easily by either direct or indirect methods. The inhomogeneous equation to be solved is represented by

$$L(f) = g, \tag{2.24}$$

where L is a linear operator, f is the unknown function to be determined, and g is the known driving function.

The moment method uses a finite set of independent basis functions to model the unknown function f as

$$f \equiv \sum_{n=1}^{N} \alpha_n f_n, \tag{2.25}$$

where f_n are the basis functions and α_n are unknown coefficients. Substituting (2.25) into (2.24) gives

$$\sum_{n=1}^{N} \alpha_n L(f_n) = \tilde{g}. \tag{2.26}$$

The N unknown coefficients are determined by forming an inner product between N appropriately selected weighting functions and both sides of (2.26). These inner products are of the form

$$\langle w, a \rangle = \iint_S (w^* \cdot a)\, ds, \tag{2.27}$$

where w is the weighting function and a represents either side of (2.26). The effect of (2.27) is to evaluate (2.26) N times using independent properties of the functions. The set of weighting functions should be linearly independent to ensure the equations are linearly independent. Additional constraints on the weighting functions are given in [2].

The residual is defined as the difference between the actual driving function and the approximate driving function obtained from the approximate solution, given by

$$R = g - \tilde{g} = g - \sum_n \alpha_n L(f_n). \tag{2.28}$$

Since only a finite number of basis functions are used to approximate the driving function, the residual in (2.28) cannot be set equal to zero everywhere on the structure. Instead, the moment method reduces a weighted average of the residual to zero over the entire structure using the weighting functions as defined in (2.27).

Applying the weighting functions, w_m, to (2.28) gives N weighted residuals given by

$$R_m = \langle w_m, g \rangle - \sum_{n=1}^{N} \alpha_n \langle w_m, L(f_n) \rangle. \tag{2.29}$$

Setting the weighted residuals equal to zero yields the set of N linear equations and N unknowns given by

$$\sum_{n=1}^{N} \alpha_n \langle w_m, L(f_n) \rangle = \langle w_m, g \rangle; \quad m = 1, \ldots, N. \tag{2.30}$$

The matrix form of (2.30) is

$$[l_{mn}][\alpha_n] = [g_m], \tag{2.31}$$

where the matrix is

$$[l_{\mathrm{mn}}] = \begin{bmatrix} \langle w_1, L(f_1)\rangle & \langle w_1, L(f_2)\rangle & \cdots \\ \langle w_2, L(f_1)\rangle & \langle w_2, L(f_2)\rangle & \cdots \\ \cdots & \cdots & \cdots \end{bmatrix}, \tag{2.32}$$

the driving vector is

$$[g_{\mathrm{m}}] = \begin{bmatrix} \langle w_1, g\rangle \\ \langle w_2, g\rangle \\ \cdots \end{bmatrix}, \tag{2.33}$$

and the unknown coefficient vector is

$$[\alpha_{\mathrm{n}}] = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \cdots \end{bmatrix}. \tag{2.34}$$

The solution of (2.31),

$$[\alpha_{\mathrm{n}}] = [l_{\mathrm{mn}}]^{-1}[g_{\mathrm{m}}], \tag{2.35}$$

by direct or iterative methods gives the unknown coefficients, α_{n}, which complete the approximate solution of the unknown function in (2.25).

Choosing appropriate basis and weighting functions can greatly simplify the integrals in (2.32) and (2.33) and thereby decrease computational requirements. To simplify the evaluation of the coefficient matrix integral, subdomain basis functions may be used. These basis functions exist only over a finite portion of the scattering surface and include pulse functions, triangular functions, piecewise linear functions, or sinusoidal functions [15]. Dirac delta functions often are chosen for the weighting functions. This technique, called point matching or collocation, simplifies the inner product in (2.27) to an evaluation of a at discrete points [15]. Therefore, the inner product in (2.32) reduces to evaluation of $L(f_{\mathrm{n}})$ at discrete points on the surface, and (2.33) reduces to an evaluation of the driving function g at discrete points on the surface. A further simplification of (2.33) can be made by using delta functions for both the basis and weighting functions [27].

Application to Scattering Problem

The matrix coefficients for the moment method treatment of the scattering problem are given by

$$\begin{aligned} l_{\mathrm{mn}} &= \langle w_{\mathrm{m}}, L(f_{\mathrm{n}}) \rangle \\ &= \int_{S} w_{\mathrm{m}} \left\{ \frac{1}{2} f_{\mathrm{n}} + \frac{1}{4\pi} \int_{S} [(z-z')l_x - (x-x')l_z] f_{\mathrm{n}} \frac{1+jkR}{R^3} e^{-jkR}\, ds' \right\} ds \end{aligned} \tag{2.36}$$

In the moment method implementation used in this study, Dirac delta functions are used as both the basis functions and weighting functions:

$$f_{\mathrm{n}} = w_{\mathrm{n}} = \delta(x - x_{\mathrm{n}}, y - y_{\mathrm{n}}). \tag{2.37}$$

Substituting (2.37) into (2.36) yields

$$l_{\mathrm{mn}} = \begin{cases} \frac{1}{2}, & \mathrm{m} = \mathrm{n} \\ \frac{1}{4\pi}[(z_{\mathrm{m}} - z_{\mathrm{n}})l_x - (x_{\mathrm{m}} - x_{\mathrm{n}})l_z] \frac{1+jkR_{\mathrm{mn}}}{R_{\mathrm{mn}}^3} e^{-jkR_{\mathrm{mn}}}, & \mathrm{m} \neq \mathrm{n} \end{cases}, \tag{2.38}$$

where

$$R_{\mathrm{mn}} = \sqrt{(x_{\mathrm{m}} - x_{\mathrm{n}})^2 + (y_{\mathrm{m}} - y_{\mathrm{n}})^2 + (z_{\mathrm{m}} - z_{\mathrm{n}})^2}. \tag{2.39}$$

The driving vector is determined from

$$g_{\mathrm{m}} = \langle w_{\mathrm{m}}, -H_{\mathrm{it}} \rangle = -H_{it}\big|_{(x_{\mathrm{m}}, y_{\mathrm{m}}, z_{\mathrm{m}})}. \tag{2.40}$$

Both (2.38) and (2.40) assume square surface patches as discussed below.

The surface of interest of the simulated ocean surface in Figure 1 is a rectangular strip one wavelength wide by five wavelengths long. A two-dimensional representation of this configuration is shown in Figure 2, as seen from above the surface.

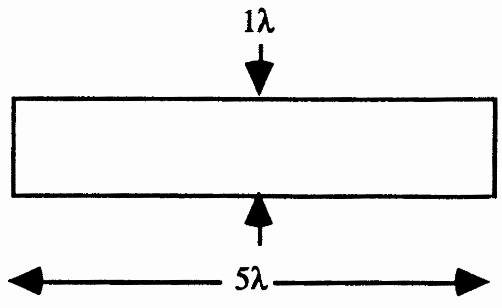

Figure 2. Sinusoidal Surface Region of Interest.

This region is then divided into a grid of square surface patches as shown in Figure 3 for a grid of degree 2.

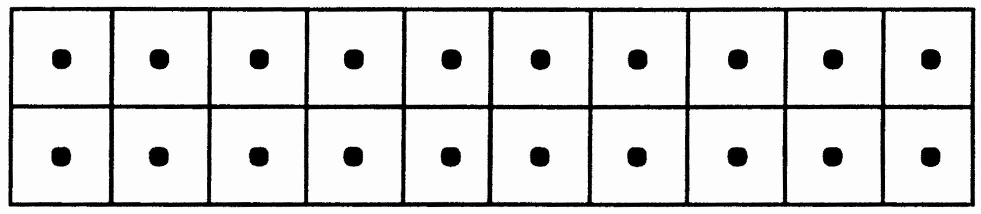

Figure 3. Moment Method Grid.

The delta basis (and weighting) functions are centered in each grid cell. The degree of coarseness of the grid, D, allows the user to improve the approximation of the current distribution on the surface and, therefore, the scattering from the surface. The value of N, the number of surface patches and therefore the matrix equation dimension, is determined according to

$$N = 5D^2. \tag{2.41}$$

Note that when using point matching, the electromagnetic boundary conditions are satisfied only at the center of each patch. The solution of the matrix equation gives the current density from which the radar cross section of the surface is calculated. A complete listing of the MM source code appears for reference in Appendix H.

CHAPTER 3

ITERATIVE METHODS

Introduction

This chapter discusses the theory of iterative methods used to solve large systems of linear equations. As discussed in Chapter 2, the moment method implementation of the MFIE produces a general, non-Hermitian, complex matrix equation of the form

$$\overline{A}\overline{x} = \overline{b}, \tag{3.1}$$

where $\overline{A}$ is the $N \times N$ coefficient matrix, $\overline{x}$ is the $N \times 1$ unknown solution, and $\overline{b}$ is the $N \times 1$ known source vector. In this discussion, square $N \times N$ matrices and $N \times 1$ column-vectors are denoted by upper- and lower-case italicized letters, respectively, with overscores. Scalar quantities are denoted by italicized letters without overscores. $\overline{A}^{\mathrm{T}}$ is the transpose of $\overline{A}$, and $\overline{A}^{*}$ is the conjugate-transpose of $\overline{A}$. Similarly, $\overline{r}^{\mathrm{T}}$ is the $1 \times N$ row-vector transpose of $\overline{r}$, and $\overline{r}^{*}$ is the $1 \times N$ row-vector conjugate-transpose of $\overline{r}$.

Algorithm Theory

Iterative methods for solving linear systems successively approximate the solution of (3.1) until a desired precision is achieved. A useful measure of the accuracy of the approximate solution is the residual vector, $\overline{r}_k$, given by

$$\overline{r}_k = \overline{b} - \overline{A}\overline{x}_k, \tag{3.2}$$

where $\overline{x}_k$ represents the approximate solution vector after k iterations of the algorithm. As the magnitude of the residual vector decreases, the approximate solution vector approaches the true solution vector. The scalar ratio of the residual magnitude to the approximate solution magnitude gives an optimistic approximation for the number of significant digits in the solution vector and is given by [34]

$$\gamma = \frac{\sqrt{\langle \bar{r}_k^*, \bar{r}_k \rangle}}{\sqrt{\langle \bar{x}_k^*, \bar{x}_k \rangle}}, \tag{3.3}$$

where $\langle \bar{x}, \bar{y} \rangle$ represents the inner product of two $N \times 1$ vectors defined by

$$\langle \bar{x}, \bar{y} \rangle = \bar{x}^T \bar{y} = \bar{y}^T \bar{x} = x_1 y_1 + \ldots + x_n y_n . \tag{3.4}$$

Instead of explicitly recalculating the residual in (3.2), most iterative methods update the previous residual successively each iteration. The advantage of using this technique is to eliminate one matrix access and the N^2 complex multiplications required per iteration for the residual calculation. Unfortunately, since errors in the residual are passed to the next iteration, propagated residuals only approximate the true residual in (3.2). As a result, both the stopping criterion in (3.3) and the number of significant digits in the solution vector magnitude differ slightly from the actual values. The effect of the propagated residual errors on convergence behavior is not included in this investigation.

Since many iterative algorithms for solving systems of linear equations exist, several characteristics were used to identify candidates for evaluation. First, the iterative method must be able to solve the general matrix equation in (3.1) with minimal real execution time. Additionally, the iterative method must not require extensive storage space for matrices and vectors and must limit the number of matrix accesses. When the iterative algorithm is configured to regenerate matrix elements as needed, frequent matrix accesses can decrease the overall efficiency of the technique. Finally, the iterative method must demonstrate stable convergence for arbitrary MFIE problems. Included in this investigation are the conjugate gradient, biconjugate gradient, conjugate gradient squared, BICGSTAB, Gauss-Seidel, Jacobi, and successive overrelaxation iterative methods.

Notation

Iterative methods based upon the conjugate gradient method are specific forms of a general method for solving (3.1) that uses a scaled search direction to form the next approximation to the solution vector. The goal of this method is to minimize a quadratic function

$$f(\bar{x}) = \langle \bar{x}, \bar{A}\bar{x} \rangle - 2\langle \bar{x}, \bar{b} \rangle, \tag{3.5}$$

which is equivalent to solving the system in (3.1) for symmetric and positive definite matrices [16] [19]. The next approximation to the solution vector is obtained by selecting a search direction, multiplying by a scalar, and adding it to the old approximation. Applying the scaled search direction to (3.5) gives [19]

$$f(\bar{x} + \alpha\bar{p}) = \langle \bar{x} + \alpha\bar{p}, \bar{A}(\bar{x} + \alpha\bar{p}) \rangle - 2\langle \bar{x} + \alpha\bar{p}, \bar{b} \rangle, \tag{3.6}$$

where $\bar{p}$ is the search direction and α is the scalar weight of the direction vector. The result in (3.6) reduces to the general form for a quadratic equation [19],

$$f(\bar{x} + \alpha\bar{p}) = f(\bar{x}) + 2\alpha\langle \bar{p}, \bar{A}\bar{x} - \bar{b} \rangle + \alpha^2 \langle \bar{p}, \bar{A}\bar{p} \rangle. \tag{3.7}$$

The solution for the scalar α that minimizes (3.7) is given by [16] [19]

$$\alpha = \frac{\langle \bar{p}, \bar{b} - \bar{A}\bar{x} \rangle}{\langle \bar{p}, \bar{A}\bar{p} \rangle}. \tag{3.8}$$

By successively choosing search directions and applying (3.8), better approximations to the solution are found. This idea forms the basis of a general iterative method in the form

$$\bar{x}_{i+1} = \bar{x}_i + \alpha_i \bar{p}_i, \tag{3.9}$$

where $\bar{x}_{i+1}$ is the new approximation to the solution, $\bar{x}_i$ is the current solution, $\bar{p}_i$ is an appropriately chosen search direction, and α_i is the scalar weight for the direction vector given by (3.8). The technique for choosing direction vectors is the distinguishing feature among iterative methods using this general form. Most CG methods are numerically unstable and unsuitable for ill-conditioned problems [22].

Conjugate Gradient

The conjugate gradient (CG) method is based upon the method of conjugate directions (CD), which uses a set of N $\bar{A}$-orthonormal vectors for the search directions. $\bar{A}$-orthonormal vectors, or conjugate vectors, satisfy

$$\langle \bar{p}_i, \bar{A}\bar{p}_j \rangle = 0 \text{ for } (i \neq \text{j}) \tag{3.10}$$

and

$$\langle \bar{p}_i, \bar{A}\bar{p}_i \rangle = 1, \tag{3.11}$$

for symmetric, positive definite matrices. The orthonormal set of search directions in CD are found using the Gram-Schmidt technique [19]. No other restrictions are placed upon the direction vectors. It can be shown [16] [19] that conjugate gradient iteration converges to the exact solution (ignoring round-off errors) after N iterations.

Hestenes and Stiefel [16] presented CG as a special case of the CD method. The set of generated direction vectors are mutually conjugate in CG just as in the CD method. CG, however, places the additional restriction that the set of generated residual vectors are mutually orthogonal as given by

$$\langle \bar{r}_i, \bar{r}_j \rangle = 0,\ i \neq j. \tag{3.12}$$

Although CG was first considered to be a direct method similar to Gaussian elimination and LU decomposition, today it is considered to be an iterative process that is guaranteed to converge in a finite number of steps [19].

Conjugate Gradient Method [16]

```
$\bar{r}_0 = \bar{b} - \bar{A}\bar{x}_0$
$\bar{p}_0 = \bar{A}^*\bar{r}_0$
loop
    $\alpha_i = \langle (\bar{A}^*\bar{r}_i)^*, \bar{A}^*\bar{r}_i \rangle / \langle (\bar{A}\bar{p}_i)^*, \bar{A}\bar{p}_i \rangle$
    $\bar{x}_{i+1} = \bar{x}_i + \alpha_i \bar{p}_i$
    $\bar{r}_{i+1} = \bar{r}_i - \alpha_i \bar{A}\bar{p}_i$
    $\beta_i = \langle (\bar{A}^*\bar{r}_{i+1})^*, \bar{A}^*\bar{r}_{i+1} \rangle / \langle (\bar{A}^*\bar{r}_i)^*, \bar{A}^*\bar{r}_i \rangle$
    $\bar{p}_{i+1} = \bar{A}^*\bar{r}_{i+1} + \beta_i \bar{p}_i$
end loop
```

The CG method of interest is the version for general nonsingular matrices [16, equation 10:2]. In this case, the system of (3.1) is replaced with the equivalent system of the form

$$\bar{A}^*\bar{A}\bar{x} = \bar{A}^*\bar{b}, \tag{3.13}$$

where $\bar{A}^*\bar{A}$ is symmetric and positive definite. Although the matrix-matrix product $\bar{A}^*\bar{A}$ is not explicitly performed in the algorithm, the conversion to (3.13) introduces one additional matrix-vector product per iteration not required in the symmetric, positive definite case. The adaptation to the complex case is made by replacing the matrix transpose with the conjugate transpose.

New residual vectors are calculated as the difference between the old residual and a scalar multiple of $\bar{A}\bar{p}_i$. This calculation produces a propagated residual that is subject to round-off errors. New direction vectors are found by adding the residual vector to a scalar multiple of the previous direction vector. For the generalized algorithm, the direction vectors are formed by adding $\bar{A}^{*}\bar{r}_i$ to a scalar multiple of the previous direction vector. In either case, it can be proved [16] that the direction vector $\bar{p}_i$ represents the gradient of the quadratic function (3.5) at $\bar{x}_i$. Therefore, the new solution vector is modified in the direction of the gradient of the quadratic function. This feature gives the conjugate gradient technique its name.

Biconjugate Gradient

Fletcher [6] introduced the method of biconjugate gradients (BCG), based upon Lanzcos' algorithm for determining the eigenvalues of a nonsymmetric matrix [21].

```
Biconjugate Gradient Method [6] [35]

$\bar{x}_0$ is initial guess
$\bar{r}_0 = \bar{\bar{r}}_0 = \bar{b} - \bar{A}\bar{x}_0$
$\rho_0 = 1$
$\bar{p}_0 = \bar{\bar{p}}_0 = 0$
for $i$ = 1, 2, 3, ...
    $\rho_i = \langle \bar{\bar{r}}_{i-1}, \bar{r}_{i-1} \rangle$
    $\beta_i = (\rho_i / \rho_{i-1})$
    $\bar{p}_i = \bar{r}_{i-1} + \beta_i \bar{p}_{i-1}$
    $\bar{\bar{p}}_i = \bar{\bar{r}}_{i-1} + \beta_i \bar{\bar{p}}_{i-1}$
    $\bar{v}_i = \bar{A}\bar{p}_i$
    $\alpha_i = \rho_i / \langle \bar{\bar{p}}_i, \bar{v}_i \rangle$
    $\bar{x}_i = \bar{x}_{i-1} + \alpha_i \bar{p}_i$
    if $\bar{x}_i$ accurate enough, quit
    $\bar{r}_i = \bar{r}_{i-1} - \alpha_i \bar{v}_i$
    $\bar{\bar{r}}_i = \bar{\bar{r}}_{i-1} - \alpha_i \bar{A}^T \bar{\bar{p}}$
end
```

A generalization of the symmetric CG algorithm, BCG was developed to solve symmetric indefinite systems. For these systems, the symmetric CG method might break down due to division by zero in the calculations of α_i and β_i. Van Der Vorst [35] included an algorithm for BCG that is investigated here. BCG algorithm is equivalent to the symmetric CG method except that inner products of the form $\langle \bar{a}, \bar{b} \rangle$ are replaced with $\langle \bar{a}^*, \bar{A}\bar{b} \rangle$ [6].

Like the CG method, BCG moves the solution a scalar distance along the direction vector. The difference between the symmetric CG and BCG lies in the method of determining these distances and directions. Whereas CG uses inner products involving two vectors $\bar{r}_i$ and $\bar{p}_i$ to find the new parameters, BCG uses four including a residual $\bar{r}_i$, pseudo-residual $\bar{\bar{r}}_i$, direction $\bar{p}_i$, and pseudo-direction $\bar{\bar{p}}_i$.

The residual vectors are the difference between the previous residual and a scalar multiple of $\bar{A}\bar{p}_i$. Similarly, the pseudo-residual vectors are the difference between the previous pseudo-residual vector and a scalar multiple of $\bar{A}^*\bar{\bar{p}}_i$. Since this calculation involves the coefficient matrix transpose, the performance of BCG may suffer due to inefficient matrix element access. The direction and pseudo-directions are calculated from the residuals and pseudo-residuals, respectively.

BCG gets its name from both the biorthogonality and biconjugacy conditions that exist between the pairs of vectors in the algorithm [6]. The residual vectors are biorthogonal, satisfying

$$\langle \bar{r}_{k+1}, \bar{\bar{r}}_k \rangle = \langle \bar{\bar{r}}_{k+1}, \bar{r}_k \rangle = 0, \tag{3.14}$$

and the direction vectors are biconjugate, satisfying

$$\langle \bar{\bar{p}}_{k+1}, \bar{A}\bar{p}_k \rangle = \langle \bar{p}_{k+1}, \bar{A}\bar{\bar{p}}_k \rangle = 0. \tag{3.15}$$

In (3.14), each new residual vector is orthogonal to the set of previous pseudo-residual vectors, and each new pseudo-residual vector is orthogonal to the set of previous residual vectors [35]. Similarly in (3.15), each new pseudo-direction vector is $\bar{A}$-orthogonal or conjugate to the set of previous direction vectors, and each new direction vector is conjugate to the set of previous pseudo-direction vectors. Although both of the residuals converge to zero in a finite number of steps, only the residual modifies the solution [35].

Conjugate Gradient Squared

In 1989 Sonneveld [33] published the conjugate gradient squared (CGS) method, a variation of the CG algorithm.

```
Conjugate Gradients Squared Method [33]

$\bar{r}_0 = \bar{b} - \bar{A}\bar{x}_0$; $\bar{\bar{r}}_0$ is chosen
$\bar{q}_0 = \bar{p}_{-1} = 0; \rho_{-1} = 1$
while $residual > tolerance$ loop
    $\rho_n = \langle \bar{\bar{r}}_0, \bar{r}_n \rangle$; $\beta_n = \rho_n / \rho_{n-1}$
    $\bar{u}_n = \bar{r}_n + \beta_n \bar{q}_n$
    $\bar{p}_n = \bar{u}_n + \beta_n(\bar{q}_n + \beta_n \bar{p}_{n-1})$
    $\bar{v}_n = \bar{A}\bar{p}_n$
    $\sigma_n = \langle \bar{\bar{r}}_0, \bar{v}_n \rangle$; $\alpha_n = \rho_n / \sigma_n$
    $\bar{q}_{n+1} = \bar{u}_n - \alpha_n \bar{v}_n$
    $\bar{r}_{n+1} = \bar{r}_n - \alpha_n \bar{A}(\bar{u}_n + \bar{q}_{n+1})$
    $\bar{x}_{n+1} = \bar{x}_n + \alpha_n(\bar{u}_n + \bar{q}_{n+1})$
    $n = n + 1$
end loop
```

A close relative of the BCG method, the CGS method uses a squared polynomial relationship to avoid the explicit computation of the pseudo-residuals using the transposed coefficient matrix [35] [33]. The development of the squared polynomial relationship is beyond the scope of this paper.

The squared polynomial relationship not only simplifies the computational requirements of CGS and does not require access to the transposed matrix, but in some cases speeds the convergence of the residual to zero over the BCG method [33]. Like BCG and CG, CGS is a finite iterative method in the absence of round-off errors [33]. In spite of Sonneveld's claim that CGS theoretically requires half of the computational effort as BCG, Van Der Vorst [35] claims that BCG might outperform CGS in certain cases.

Additionally, Van Der Vorst [35] notes situations where CGS exhibits nonuniform convergence behavior that could produce erroneous results. When started near the solution, the propagated residual in CGS may not approximate the true residual as given by (3.2), thereby causing premature convergence [35]. When the true residual (3.2) was calculated in place of the propagated residual, some of Van Der Vorst's experiments showed that CGS did not converge or required many additional iterations to produce a sufficiently accurate solution [35]. The exact reason for the erratic convergence behavior of CGS is currently a topic of research [35].

BICGSTAB

Van Der Vorst's BICGSTAB [35] method promises to be more stable than CGS.

```
BICGSTAB Method [31]

$\bar{x}_0$ is initial guess
$\bar{r}_0 = \bar{b} - \bar{A}\bar{x}_0$
$\rho_0 = \alpha = \omega_0 = 1$
$\bar{v}_0 = \bar{p}_0 = 0$
for $i = 1,2,3...$
    $\rho_i = \langle \bar{r}_0, \bar{r}_{i-1} \rangle$
    $\beta = (\rho_i / \rho_{i-1})(\alpha / \omega_{i-1})$
    $\bar{p}_i = \bar{r}_{i-1} + \beta(\bar{p}_{i-1} - \omega_{i-1}\bar{v}_{i-1})$
    $\bar{v}_i = \bar{A}\bar{p}_i$
    $\alpha = \rho_i / \langle \bar{r}_0, \bar{v}_i \rangle$
    $\bar{s} = \bar{r}_{i-1} - \alpha\bar{v}_i$
    $\bar{t} = \bar{A}\bar{s}$
    $\omega_i = \langle \bar{t}, \bar{s} \rangle / \langle \bar{t}, \bar{t} \rangle$
    $\bar{x}_i = \bar{x}_{i-1} + \alpha\bar{p}_i + \omega_i\bar{s}$
    if $\bar{x}_i$ is accurate enough then quit
    $\bar{r}_i = \bar{s} - \omega_i\bar{t}$
end
```

This relative of the BCG method retains rapid convergence while overcoming some of the convergence instabilities found in CGS when initializing the algorithm near the actual solution. To accomplish this improvement, BICGSTAB improves the efficiency and stability of the residual vector calculation in the CGS algorithm using a stable recurrence relationship. BICGSTAB shares with CG, BCG, and CGS the property of convergence in a finite number of iterations in the absence of round-off errors. Van Der Vorst also presents evidence that BICGSTAB outperforms CGS and BCG in some cases in convergence stability and efficiency [35].

Jacobi and Gauss-Seidel

The Jacobi and Gauss-Seidel methods [7] [19] are examples of simple iterative solution techniques. In each Jacobi iteration, the jth equation is solved for the jth unknown using the other $N-1$ components of the previous solution vector. At the end of the iteration, the solution vector is updated with new approximate values. The Jacobi method is guaranteed to converge for diagonally dominant systems.

```
Jacobi Method [19]

$\bar{x}_0$ is initial guess
for i = 1,2,3...
    $\bar{r}_i = \bar{u}_i = \bar{b} - \bar{A}\bar{x}_{i-1}$
    $\bar{u}_i = \bar{u}_i - \mathrm{diag}(\bar{A})\bar{x}_{i-1}$
    $u_i(j) = u_i(j) / A(j,j)$ for $j = 1,\ldots,n$
    $\bar{x}_i = \bar{u}_i$
    if $\bar{x}_i$ is accurate enough, quit
end loop
```

The Gauss-Seidel method [7] [19] is a slight modification of the Jacobi method. Gauss-Seidel uses the individual updated solution terms immediately, while the Jacobi method updates all terms before using the updated solution vector. While this is less efficient computationally than the Gauss-Seidel method, the Jacobi method is more easily adapted to parallel or vector computer architectures [19]. In these environments, many of the linear equations can be solved simultaneously on different processors. Unfortunately, since Gauss-Seidel relies on recent information, it must be performed in a serial fashion and cannot be adapted to parallel or vector processors [19]. Like the Jacobi method, Gauss-Seidel is guaranteed to be stable only for diagonally dominant systems.

```
Gauss - Seidel Method [19]

$\bar{x}_0$ is initial guess
for i = 1,2,3...
    $\bar{r}_i = \bar{b} - \bar{A}\bar{x}_{i-1}$
    $\bar{x}_i = \bar{b} - (\bar{A}\bar{x}_{i-1} - diag(\bar{A})\bar{x}_{i-1})$
    $x_i(j) = x_i(j) / A(j,j)$ for $j = 1,\dots,n$
    if $\bar{x}_i$ is accurate enough, quit
end loop
```

Successive Overrelaxation

The successive overrelaxation method (SOR) [41] [19] is closely related to the GS method, using an additional parameter called the relaxation factor.

```
Successive Overrelaxation Method [41] [19]

$\bar{x}_0$ is initial guess
for i = 1,2,3...
    $\bar{r}_i = \bar{b} - \bar{A}\bar{x}_{i-1}$
    $\overline{temp}_i = \bar{b} - (\bar{A}\bar{x}_{i-1} - diag(\bar{A})\bar{x}_{i-1})$
    $x_i(j) = \omega(temp_i(j) / A(j,j)) + (1-\omega)x_{i-1}(j)$ for $j = 1,\dots,n$
    if $\bar{x}_i$ is accurate enough, quit
end loop
```

The range of this factor is from 0 to 2, with a factor of 1 reducing SOR to Gauss-Seidel. Relaxation factors less than 1 undercorrect the solution and relaxation factors greater than 1 overcorrect the solution. Choosing an optimal relaxation factor can dramatically speed convergence of the SOR method over either the Jacobi or the GS methods [41]. Unfortunately, finding the proper relaxation factor can be a difficult task. Since it uses updated values for the solution vector, SOR cannot be implemented efficiently on a parallel or vector processors [19].

CHAPTER 4

COMPUTER IMPLEMENTATION

Introduction

This chapter discusses the implementation of the iterative algorithms using the FORTRAN programming language. Three independent computer systems were used to code the algorithms. The primary system used for code development was an IBM RS/6000 320H workstation with a AIX 3.2 FORTRAN compiler. The RS/6000 uses a reduced instruction set computer (RISC) microprocessor to achieve higher floating-point operation performance levels than possible with complex instruction set computer (CISC) processors. A DEC 5000-240 RISC workstation and an IBM 3090-200S vector processing mainframe computer were used to assure code portability and to allow testing on different architectures. Several features are common in the implementation of each of the iterative methods.

The implemented algorithms share the following common programming structure.

```
subroutine(arguments)
variable declarations;

initialize variables;

do iteration while precision is low
   refine solution;
   find new precision;
end do loop

return solution;
end
```

Not only does this structure facilitate the implementation of each algorithm, it ensures that algorithms are compared according to their iterative refinement characteristics and not according to structural biases. The main features of the structure include the variable declaration and initialization, the iterative loop, and the return of the solution.

Each iterative method requires an initial guess for the solution vector. To find this guess, the physical optics approximation [2] for the surface current,

$$\mathbf{J}_S^{PO} \approx 2\hat{\mathbf{a}}_n \times \mathbf{H}_i \tag{4.1}$$

was considered. Unfortunately, given a guess near the final solution, iterative methods behave differently, with some converging more rapidly and others becoming unstable. Specifically, Van Der Vorst notes that CGS can suffer from instability in certain cases where the initial solution guess is near the final solution [35]. Given this problem, a guess of zero provides a more equitable way of testing the algorithms.

The stopping criterion for the iterative algorithms involves the ratio between the magnitude of the residual vector and the magnitude of the solution vector given in (3.3) [34]. If the magnitude of the residual is less than 0.1% of the magnitude of the solution vector, the iterative algorithm is stopped. This technique and criterion allows the surface currents to be determined to within three significant figures, which is adequate for radar cross-section calculations. The use of this stopping criterion introduces small variations in the accuracy of the final solutions, depending upon the behavior with which the algorithms converge. For instance, if an algorithm converges very quickly, its final ratio might fall far below the threshold. A slowly converging algorithm, however, stops with a ratio just under the threshold, and the relative error in the solution may greatly exceed the relative convergence tolerance. Even though the stopping thresholds of the two algorithms are identical, the first algorithm will produce more accurate results. The effects of nonuniform convergence are not considered in the analysis of algorithm performance.

FORTRAN stores two dimensional arrays in column-major format. In this format, all the matrix elements of the first column are stored sequentially in memory, followed by the elements in the next columns. Unfortunately, multiplying a matrix by a column vector on the right requires the algorithm to step through the matrix rows, resulting in inefficient access when matrix elements have to be retrieved from the disk. To prevent this problem, the matrix transpose was stored for all of the algorithms.

Algorithm Implementation

Presented below are the FORTRAN implementations of the iterative algorithms, taken directly from the theoretical algorithms given in Chapter 3 unless noted. The algorithms use the matrix transpose to take advantage of FORTRAN's column major storage format. However, in BCG and CG both the matrix and its transpose must be accessed, so there is no benefit in storing the transpose. Double precision variables are used for all variables to minimize round-off errors.

Conjugate Gradient

Many implied modifications were implemented in the CG algorithm.

```
Implemented Conjugate Gradient

$\bar{x} = 0$
$\bar{r} = \bar{b} - \bar{A}\bar{x}$
$\bar{p} = \bar{A}^*\bar{r};\ ms = \langle \bar{p}^*, \bar{p} \rangle$
loop while $gam > eps$
    $\bar{a}p = \bar{A}\bar{p}$
    $den = \langle \bar{a}p^*, \bar{a}p \rangle$
    $alpha = ms / den$
    $\bar{x} = \bar{x} + (alpha)\bar{p}$
    $\bar{r} = \bar{r} - (alpha)\bar{a}p$
    $oldms = ms$
    $\bar{a}ctr = \bar{A}^*\bar{r}$
    $ms = \langle \bar{a}ctr^*, \bar{a}ctr \rangle$
    $beta = ms / oldms$
    $\bar{p} = \bar{a}ctr + (beta)\bar{p}$
    $mr = \sqrt{\langle \bar{r}^*, \bar{r} \rangle};\ mx = \sqrt{\langle \bar{x}^*, \bar{x} \rangle}$
    $gam = mr / mx$
end loop
```

Specifically, the original algorithm contains five references to the coefficient matrix per iteration, although careful inspection reveals that only two of these are unique, the calculation of $\overline{A}\overline{p}$ and $\overline{A}^{*}\overline{r}$. In the implemented algorithm, matrix-vector products are saved temporarily in vectors resulting in far fewer complex multiplications. For instance, the vector $\overline{ap}$ was used to store the matrix-vector product $\overline{A}\overline{p}$, and the $\overline{actr}$ vector stores the product $\overline{A}^{*}\overline{r}$. In addition, only the transposed coefficient matrix is stored so that the matrix rows can be accessed sequentially from memory for the $\overline{A}\overline{p}$ calculation. Therefore, the $\overline{A}^{*}\overline{r}$ calculations suffer due to inefficient access of the column elements in page swapping configurations. The FORTRAN code appears in Appendix C.

Biconjugate Gradient

The implemented biconjugate gradient algorithm closely follows the Van Der Vorst [35] theoretical algorithm discussed in Chapter 3.

Implemented Biconjugate Gradient Method

```
$\overline{x} = 0$
$\overline{r} = (\overline{r}q) = \overline{b} - \overline{A}\overline{x}$
$\overline{p} = (\overline{p}q) = 0$
rho = 1
loop while $gam < eps$
    $oldrho = rho$
    $rho = \langle \overline{r}q, \overline{r} \rangle$
    $beta = rho / oldrho$
    $\overline{p} = \overline{r} + (beta)\overline{p}$
    $\overline{p}q = \overline{r}q + (beta)\overline{p}q$
    $(\overline{n}u) = \overline{A}\overline{p}$
    $sigma = \langle \overline{p}q, \overline{n}u \rangle$
    $alpha = rho / sigma$
    $\overline{x} = \overline{x} + (alpha)\overline{p}$
    $\overline{r} = \overline{r} - (alpha)\overline{n}u$
    $\overline{r}q = \overline{r}q - (alpha)\overline{A}^{T}\overline{p}q$
    $mr = \sqrt{\langle \overline{r}^{*}, \overline{r} \rangle}; mx = \sqrt{\langle \overline{x}^{*}, \overline{x} \rangle}$
    $gam = mr / mx$
end loop
```

Note that in this algorithm, a complex variable *sigma* has been introduced as a temporary holder for the inner product $\langle \bar{p}, \bar{v} \rangle$. BCG accesses the matrix once and the matrix transpose once in each iteration. Since one type of matrix access is not favored over the other and only one matrix is stored, the overall efficiency of the implemented algorithm suffers for page swapping configurations. The FORTRAN code is given in Appendix A.

Conjugate Gradient Squared

The conjugate gradient squared method was implemented using the Sonneveld theoretical algorithm discussed in Chapter 3. Very few changes were made in this implementation.

```
Implemented Conjugate Gradient Squared Method

$\bar{x} = 0$
$\bar{r} = \bar{r}0 = \bar{b} - \bar{A}\bar{x}$
$\bar{p} = \bar{q} = 0$
$rho = 1$
loop while $gam > eps$
    $oldrho = rho$
    $rho = \langle \bar{r}0^*, \bar{r} \rangle$
    $beta = rho / oldrho$
    $\bar{u} = \bar{r} + (beta)\bar{q}$
    $\bar{p} = \bar{u} + (beta)(\bar{q} + (beta)\bar{p})$
    $\bar{v} = \bar{A}\bar{p}$
    $sigma = \langle \bar{r}0^*, \bar{v} \rangle$
    $alpha = rho / sigma$
    $\bar{q} = \bar{u} - (alpha)\bar{v}$
    $\bar{u}pq = \bar{u} + \bar{q}$
    $\bar{r} = \bar{r} - (alpha)\bar{A}(\bar{u}pq)$
    $\bar{x} = \bar{x} + (alpha)(\bar{u}pq)$
    $mr = \sqrt{\langle \bar{r}^*, \bar{r} \rangle}; mx = \sqrt{\langle \bar{x}^*, \bar{x} \rangle}$
    $gam = mr / mx$
end loop
```

First, the vector $\bar{u}pq$ is introduced to represent the summation of the vectors $\bar{u}$ and $\bar{q}$. Although relatively insignificant, this vector saves one repetition of this addition, as shown above. Second, the $\bar{r}0$ in the inner products for the calculation of *rho* and *sigma* are changed to the conjugate form. The FORTRAN code appears in Appendix D.

BICGSTAB

The most important change made to the BICGSTAB algorithm implementation was the use of conjugate multiplications in the inner products to allow the method to be applied to complex systems of equations. The FORTRAN code is given in Appendix B.

```
Implemented BICGSTAB Method

$\bar{x} = 0$
$\bar{r} = \bar{r}0 = \bar{b} - \bar{A}\bar{x}$
$rho = alpha = omega = 1$
$\bar{s} = \bar{t} = \bar{p} = \bar{n}u = 0$
loop while $gam > eps$
    $oldrho = rho$
    $rho = \langle r0^{*}, r \rangle$
    $beta = (rho)(alpha) / (oldrho)(omega)$
    $\bar{p} = \bar{r} + beta(\bar{p} - (omega)\bar{n}u)$
    $\bar{n}u = \bar{A}\bar{p}$
    $alpha = rho / \langle \bar{r}0^{*}, \bar{n}u \rangle$
    $\bar{s} = \bar{r} - (alpha)\bar{n}u$
    $\bar{t} = \bar{A}\bar{s}$
    $num = \langle \bar{t}^{*}, \bar{s} \rangle ; den = \langle \bar{t}^{*}, \bar{t} \rangle$
    $omega = num / den$
    $\bar{x} = \bar{x} + (alpha)\bar{p} + (omega)\bar{s}$
    $\bar{r} = \bar{s} - (omega)\bar{t}$
    $mr = \sqrt{\langle \bar{r}^{*}, \bar{r} \rangle} ; mx = \sqrt{\langle \bar{x}^{*}, \bar{x} \rangle}$
    $gam = mr / mx$
end loop
```

Jacobi and Gauss-Seidel

Implementation of the Jacobi method and the Gauss-Seidel methods are based entirely upon the algorithms discussed in Chapter 3 with one notable exception.

Instead of using a separate matrix access to find the residual as defined by (3.2), the residual is determined in the same step as the solution vector update. Although this saves considerable computation in the residual calculation, the generated residuals do not reflect all improvements made to the solution in the current iteration, and therefore are somewhat larger than the final residuals calculated after the iteration is completed.

```
Implemented Jacobi Method

$\bar{x} = 0$
loop while $gam > eps$
    $\bar{r} = \bar{b} - \bar{A}\bar{x}$
    $\bar{u} = \bar{b} - (\bar{A}\bar{x} - diag(\bar{A})\bar{x})$
    $u(j) = u(j) / A(j,j)$ for $j = 1,\ldots,n$
    $\bar{x} = \bar{u}$
    $mr = \sqrt{\langle \bar{r}^*, \bar{r} \rangle}; mx = \sqrt{\langle \bar{x}^*, \bar{x} \rangle}$
    $gam = mr / mx$
end loop
```

```
Implemented Gauss - Seidel Method

$\bar{x} = 0$
loop while $gam > eps$
    $\bar{r} = \bar{b} - \bar{A}\bar{x}$
    $\bar{x} = \bar{b} - (\bar{A}\bar{x} - diag(\bar{A})\bar{x})$
    $x(j) = x(j) / A(j,j)$ for $j = 1,\ldots,n$

    $mr = \sqrt{\langle \bar{r}^*, \bar{r} \rangle}; mx = \sqrt{\langle \bar{x}^*, \bar{x} \rangle}$
    $gam = mr / mx$
end loop
```

In the Jacobi method, the updates to the solution vector are made at the end of the iteration. Therefore, since the residual is calculated at the beginning of the iteration, it represents the actual residual after the previous iteration.

```
Jacobi Residual Calculation

do $i = 1,n$
    $rsum = sum = 0$
    do $j = 1,n$
        $rsum = rsum + a(j,i)x(j)$
    end do
    $sum = rsum - a(i,i)x(i)$
    $u(i) = (b(i) - sum) / a(i,i)$
    $r(i) = b(i) - rsum$
end do
do $i = 1,n$
    $x(i) = u(i)$
end do
```

The net effect of this process is to increase by one the number of iterations required for convergence to a desired precision. This additional work is insignificant compared with recalculating the residual separately, which requires both an additional matrix access and matrix-vector product per iteration. The pseudo-code representation shows the residual calculation using the coefficient matrix transpose.

Unlike in the Jacobi method, the Gauss-Seidel updates to the solution vector are made continually during the iteration. Therefore, as the residual elements are calculated, better and better values of the solution are used in the calculation. The net effect of this process is that the last elements of the residual vector are calculated from more updated solutions. This can lead to an additional iteration being required. As with the Jacobi method, these extra iterations do not significantly affect the execution time when compared to the cost of another matrix access for every iteration. A pseudo-code representation of this technique is shown below for the transpose coefficient matrix case.

```
Gauss - Seidel Residual Calculation

$do\ i = 1,n$
    $rsum = sum = 0$
    $do\ j = 1,n$
        $rsum = rsum + a(j,i)x(j)$
    $end\ do$
    $sum = rsum - a(i,i)x(i)$
    $x(i) = (b(i) - sum) / a(i,i)$
    $r(i) = b(i) - rsum$
$end\ do$
```

The source code listings for both the Gauss-Seidel and Jacobi methods are found in Appendices E and F, respectively.

Successive Overrelaxation

Since the successive overrelaxation method (SOR) is a generalized version of the Gauss-Seidel method, the same arguments given above apply to it as well. The implementation is shown below. The only change to the GS algorithm is the relaxation factor. The FORTRAN source code for SOR can be found in Appendix G.

```
Implemented Successive Overrelaxation Method

$\bar{x} = 0$
loop while $gam > eps$
    $\bar{r} = \bar{b} - \bar{A}\bar{x}$
    $\bar{temp} = \bar{b} - (\bar{A}\bar{x} - diag(\bar{A})\bar{x})$
    $x(j) = \omega(\bar{temp}(j) / A(j,j)) + (1-\omega)x(j)$ for $j = 1,\ldots,n$
    $mr = \sqrt{\langle \bar{r}^*, \bar{r} \rangle}; mx = \sqrt{\langle \bar{x}^*, \bar{x} \rangle}$
    $gam = mr / mx$
end loop
```

Storage and Computational Requirements

Table I summarizes the storage requirements of the algorithms as given in the literature and tabulated in the actual implementation for nonsparse systems. In some cases, vectors were added to the implementation to save redundant multiplications. Note that for this table, *N* represents the matrix dimension. Only double precision complex variables are indicated.

Table I. Algorithm Storage Requirements.

Algorithm	Minimum Algorithm Storage (nonsparse systems)	Actual Storage (nonsparse systems)
Biconjugate Gradient	N^2+7N+4	N^2+7N+6
BICGSTAB	N^2+8N+5	N^2+8N+8
Conjugate Gradient	N^2+4N+2	N^2+6N+3
Conjugate Gradient Squared	N^2+8N+5	N^2+9N+7
Gauss-Seidel	N^2+3N	N^2+3N+3
Jacobi	N^2+4N	N^2+4N+3
Successive Overrelaxation	N^2+3N	N^2+3N+3

Note that in every implementation double precision variables are used to minimize the effects of round-off errors in the solution and residual vectors. Real and complex variables use eight and sixteen bytes of storage, respectively.

To avoid redundant multiplications, some matrix-vector products are stored in temporary vectors for use later in the execution of the algorithm. Table II shows the computational requirements per iteration of the actual implementations of the algorithms. Only complex multiplications are indicated in this table.

Table II. Algorithm Computational Requirements per Iteration.

Algorithm Name	Actual Multiplications
Biconjugate Gradient	$2N^2+9N+2$
BICGSTAB	$2N^2+12N+5$
Conjugate Gradient	$2N^2+7N+2$
Conjugate Gradient Squared	$2N^2+10N+2$
Gauss-Seidel	N^2+3N+1
Jacobi	N^2+3N+1
Successive Overrelaxation	N^2+3N+1

CHAPTER 5

EXPERIMENTAL RESULTS

Introduction

The MFIE rough-surface-scattering code discussed in Chapter 2 serves as the basis for three experiments designed to investigate the performance and convergence properties of the iterative algorithms. The tests consider the effects of matrix element storage (MES), matrix element recalculation (MER), parameter changes (angle of incidence and coarseness of the grid used in the moment method representation of the surface), and machine dependence on the execution times, number of iterations, and percentage CPU utilization. Table III below summarizes the experiments performed.

Table III. Summary of Experiments.

Algorithm	MES	MER	Angle	Grid Ratio	DEC 5000	IBM 3090
BCG	X	X	X	X	X	X
BICGSTAB	X	X	X	X	X	X
CG	X	X	X	X	X	X
CGS	X	X	X	X	X	X
GS	X	X	X	X	X	X
Jacobi	X	X	X	X	X	X
SOR	X			X		

The first test measured how matrix storage configurations affect the execution times for the iterative methods. In the "matrix element storage" (MES) configuration, the computer physically stores the matrix elements in random access memory. For very large matrices, the physical RAM capacity of the computer is exceeded, and the computer uses page swapping to store the matrix elements. In the "matrix element recalculation" (MER) configuration, the matrix elements are recalculated as they are needed by the algorithm. Although this method is very inefficient computationally, very little physical memory is required and the need for page swapping is eliminated.

Another experiment investigated the effects of changing the user parameters for the moment method code. These parameters include the wavelength, the angle of incidence, and the sinusoidal surface height. The goal of this experiment was to determine which algorithms perform the best under varying conditions.

Although less influential on the convergence properties than matrix storage techniques and parameter changes, the choice of computer system can affect the efficiency of an algorithm. Scalar machines offer high speed execution in a sequential manner. On the other hand, vector-processing machines are able to perform matrix and vector operations more efficiently than scalar machines by taking advantage of simultaneous operations. Iterative techniques that require sequential vector updates are less able to efficiently use this feature.

To obtain results from both scalar and vector computer architectures, three computer systems were used for the testing. The scalar IBM RS/6000 320H workstations served as the primary testing facilities. Each workstation features 32 megabytes (MB) of random access memory (RAM) and 100 MB of disk space reserved for paging. A second UNIX workstation, the DEC 5000-240, served as a secondary scalar system for testing. Although the DEC has 128 MB of RAM and 300 MB of paging space, the F77 2.1 FORTRAN compiler does not allow programs to use these memories to full capacity. Finally, a few experiments were executed using the TSO operating system on the IBM 3090-200S vector computer with 128 MB of RAM and 128 MB of expanded memory. Unfortunately, programs are limited to 16 MB of disk and memory usage, so only small matrices were tested.

Objective Testing Techniques

An automated program was developed to obtain CPU execution times and real execution times. Reliability was obtained through redundant testing and averaging of the experimental results. The FORTRAN source code for the experiments is located in Appendix H for reference.

The CPU execution time describes the amount of time, in seconds, that the computer devoted to execution of the algorithm. Although it does not include any time that the computer waited for paging data to and from the disk, it does include execution of the additional overhead involved in paging. The CPU times were found using system calls for the two UNIX machines and using the job control language (JCL) output on the IBM 3090. Unlike the CPU times, the real execution time represents the amount of actual time spent executing a program and includes any time the computer waits for peripheral devices. The real execution times were found by polling the system clock. Appendix I contains source code for the polling routines.

Results

Figures 4 through 14 represent the experimental results of the investigation. Unless noted otherwise, the incident wavelength was 1.0 m, the angle of incidence was 0.0 degrees (perpendicular to the surface), and the sinusoidal surface amplitude was 0.1 m. in all tests. Additionally, all tests were implemented on the IBM RS/6000 unless otherwise noted.

Convergence Behavior

In an effort to correct programming errors and identify unsuitable algorithms before extensive testing, the convergence behaviors of the representative iterative methods were investigated, and the results are plotted in Figure 4. This graph shows the ratio of the magnitudes of the residual to the approximate solution after each iteration for matrices of dimension 720. Represented by (3.3), this value describes the number of significant digits in the approximate solution.

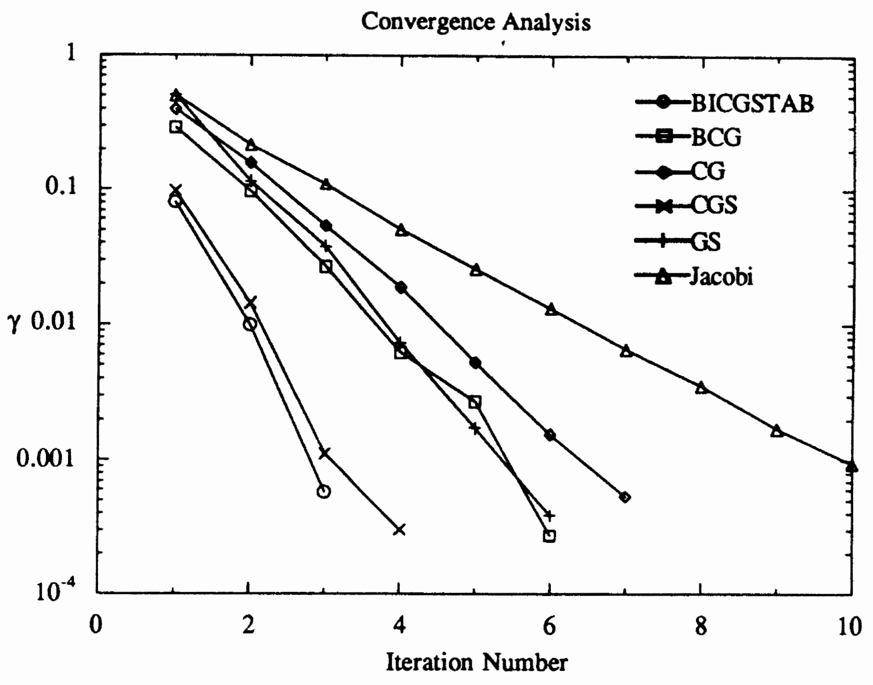

Figure 4. Convergence Behavior for N=720.

All six algorithms uniformly converged to a relative convergence $\gamma = 10^{-3}$, with BICGSTAB requiring only three iterations and Jacobi needing ten.

Note that the number of iterations required for convergence does not necessarily predict the amount of required execution time. For instance, the CGS and BICGSTAB algorithms required about half of the number of iterations as Gauss-Seidel, but experiments show that Gauss-Seidel required less execution time. Each CGS and BICGSTAB iteration required two matrix references and therefore are computationally equivalent to about two Gauss-Seidel iterations.

Matrix Element Storage

The results of the MES configuration, where all matrix coefficients are stored in memory or disk through paging, are plotted in Figures 5,6, and 7 and compared with the "exact" LU Decomposition solution.

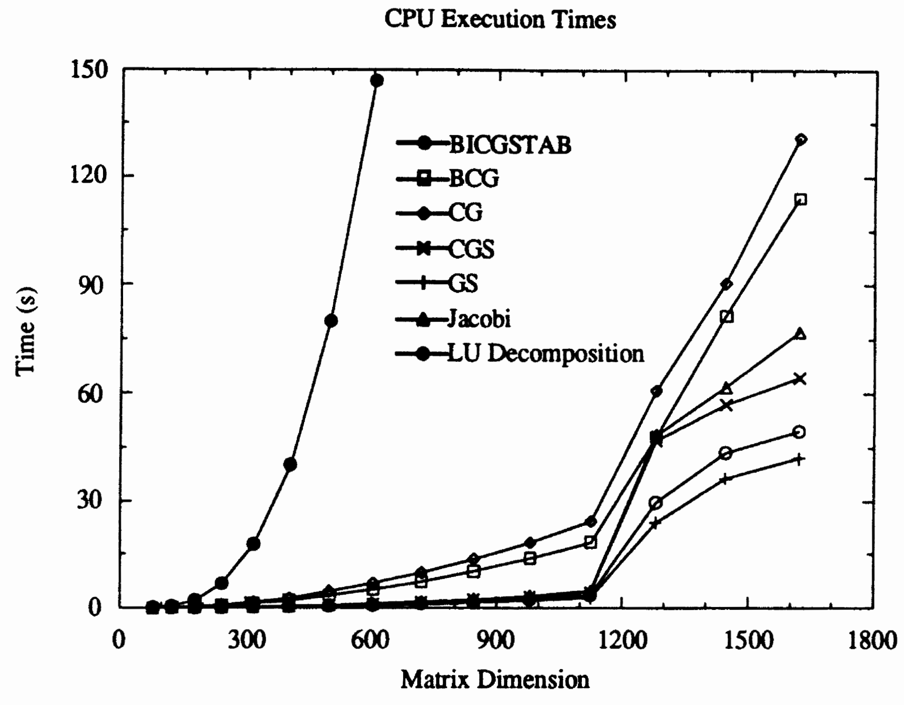

Figure 5. Matrix Element Storage CPU Execution Times.

As seen in Figure 5 , all the algorithms were able to solve a matrix equation of dimension 1620 in less than 130 CPU seconds. For this matrix dimension, the Gauss-Seidel and BICGSTAB methods required less than 50 seconds of execution time. Next were the CGS and Jacobi algorithms, needing 64 and 77 CPU seconds to converge, respectively. Since they require twice the number of iterations as CGS and BICGSTAB, the BCG and Conjugate Gradient methods needed almost two minutes to converge. The CPU time for LU decomposition is about 23 times longer than the worst iterative method and over 72 times longer than the best. Of course, the solutions given by LU decomposition are much more accurate than those given by the iterative methods.

The most notable feature of Figure 5, however, is the dramatic increase in execution time for matrix equations with 1125 or more unknowns. As the matrix equation increased beyond this dimension, the entire matrix could not be stored in the computer's random access memory and page swapping occurred. The excessive computational overhead needed to keep track of page swapping shows up in the figure as increased execution times.

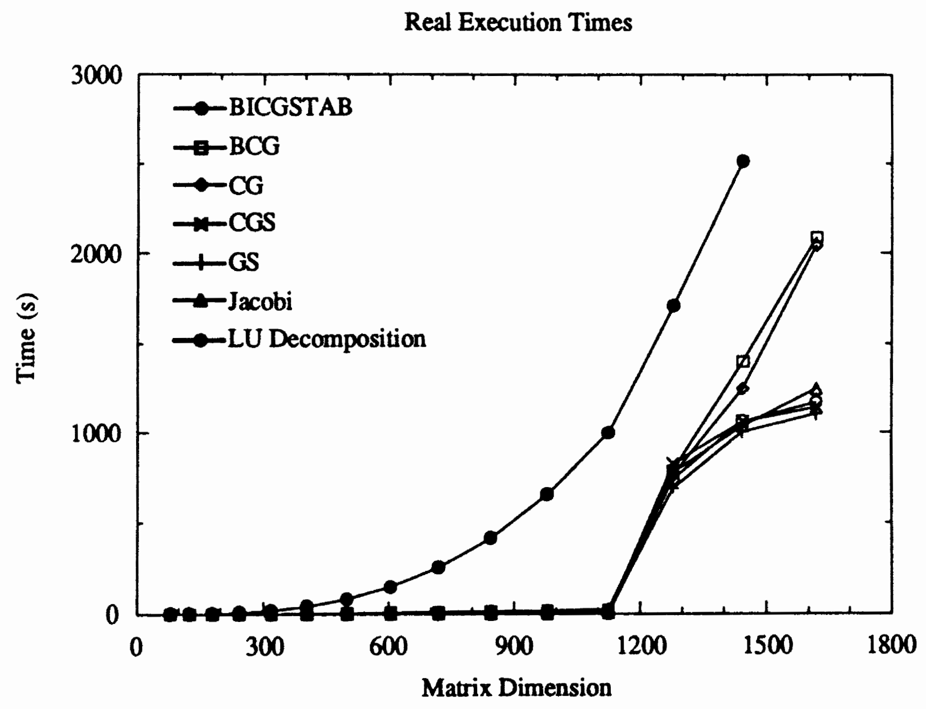

Figure 6. Matrix Element Storage Real Execution Times.

Figure 6 shows that real execution times were much greater than the CPU times for large matrix sizes. As before, the Gauss-Seidel, CGS, BICGSTAB, and Jacobi algorithms required the least execution time. These four methods were able to solve the largest matrix equation in less than twenty-five minutes. The other algorithms performed as expected based on Figure 5. LU Decomposition required nearly one hour of real execution time, about 3.3 times the Gauss-Seidel execution time and 1.8 times the conjugate gradient execution time. Obviously for large matrices, page swapping greatly reduced the efficiency of solution. The marked increase in the execution times for matrices of dimension 1125 or more can be attributed to exceeding the 32 MB of RAM in the RS/6000 320H machines. For smaller matrices, the differences between the CPU times and real times are insignificant.

The effects of page swapping on execution times can be seen in Figure 7. This diagram shows the percentage CPU utilization, which is the percentage of time the CPU spends executing the program's instructions.

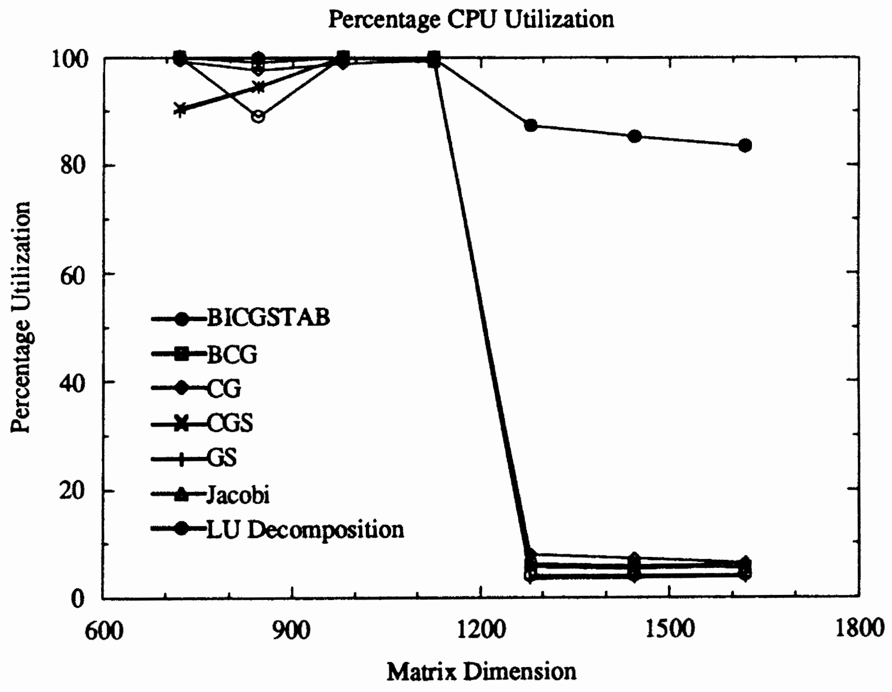

Figure 7. Matrix Element Storage Percentage CPU Utilization.

By taking a ratio of CPU execution time to real execution time for the algorithms, the percentage utilization was found. All the iterative techniques showed the steep decrease in percentage CPU utilization that would be expected from page swapping. CPU utilization approached only a few percent as the computer had to wait for data to be retrieved from disk. Interestingly, LU Decomposition did not suffer as severe a drop in CPU utilization as the iterative methods. This discrepancy is attributed to the difference in matrix access in direct solution methods. Since LU Decomposition modifies the matrix elements in the solution process, a direct comparison of its utilization to that of iterative methods cannot be made.

<u>Matrix Element Recalculation</u>

The MER configuration, where the matrix coefficients are recalculated when they are needed by the iterative algorithm and not stored, demonstrates that computationally inefficient techniques that avoid page swapping may outperform MES for large matrices.

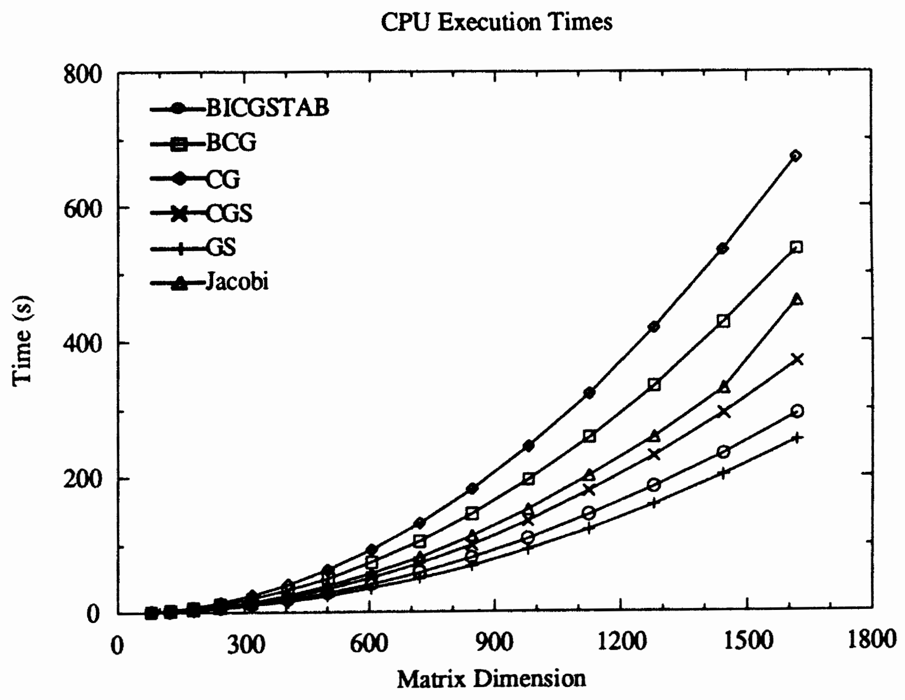

Figure 8. Matrix Element Recalculation CPU Execution Times.

Figures 8,9, and 10 show the results of these experiments. As seen in Figure 8, the CPU execution times of MER were much greater than the respective MES CPU times. Whereas all the MES algorithms solved the largest matrix equation in under two and one-half CPU minutes, the quickest MER algorithm needed almost twice that time. As before, Gauss-Seidel completed execution first, followed by BICGSTAB, CGS, and Jacobi. The other algorithms were slower with the BCG and Conjugate Gradient methods taking between nine and eleven minutes to complete. Each of these algorithms required more iterations and therefore more matrix elements to be recalculated than the other iterative schemes.

Even though the MER algorithms required much more CPU execution time, great savings are achieved in real execution times for large matrices.

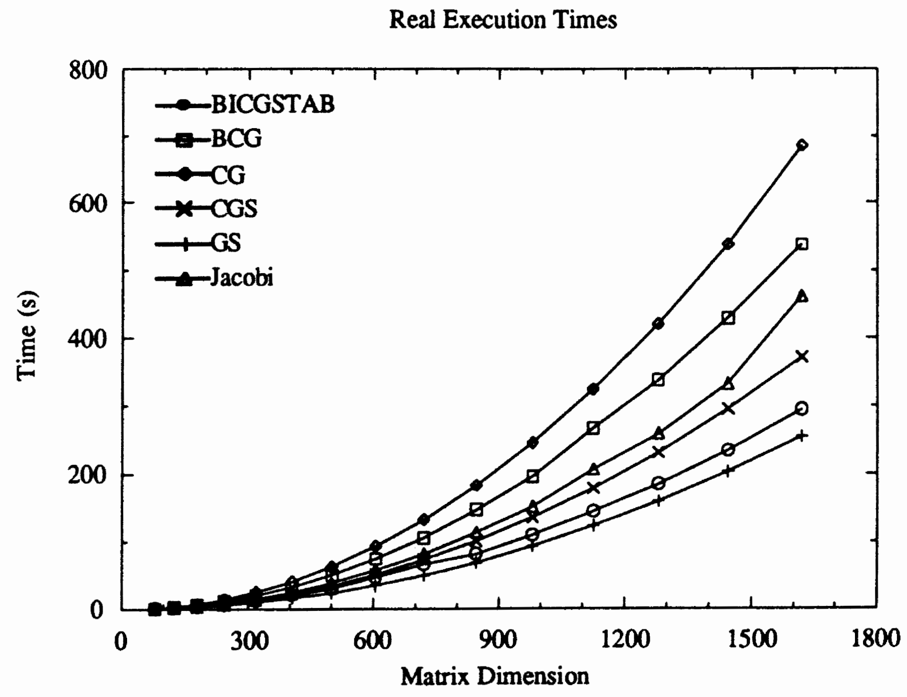

Figure 9. Matrix Element Recalculation Real Execution Times.

In Figure 9, the real execution times for all iterative techniques were much less than the corresponding MES real execution times. The MES configurations required more than three times the amount of real execution time to reach completion. Figure 9 also shows that the real execution times for the MER algorithms differed only slightly from the MER CPU execution times. Since the real and CPU execution times are very close, the percentage CPU utilization for the algorithms approach 100 percent, as seen in Figure 10. Figures 8 through 10 show the improvement that can be gained by not storing the matrix elements in RAM.

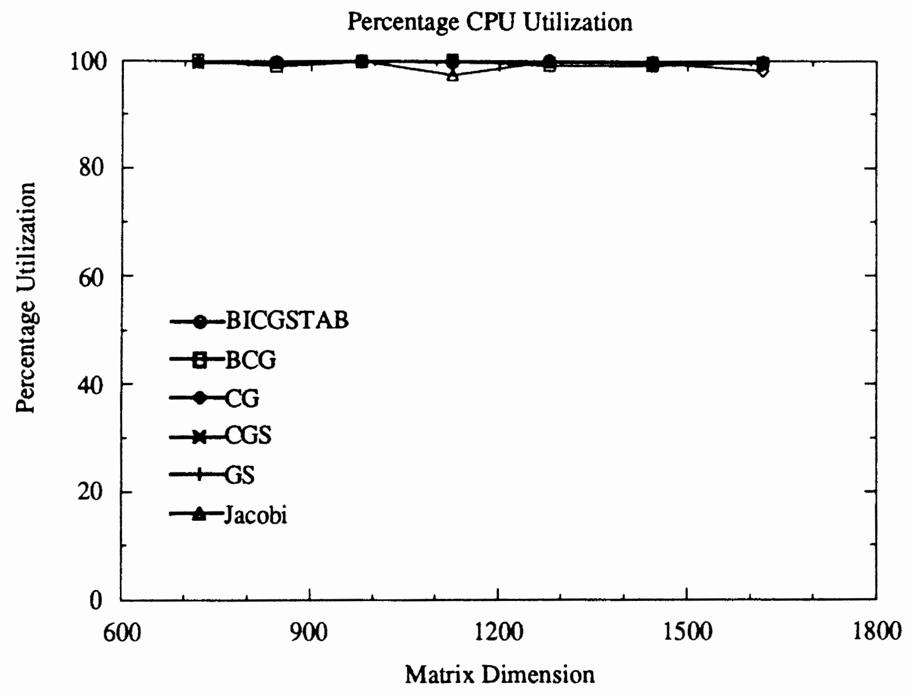

Figure 10. Matrix Element Recalculation Percentage CPU Utilization.

Electromagnetic Parameter Changes

The experimental results given above for the matrix element storage configurations are meaningless unless the algorithms perform well under different circumstances. Since it is possible that the electromagnetic parameters tested for the MES and MER configurations might be special cases, the effect of these changes was investigated next. The two primary parameters that the user can specify are the angle of incidence and the height of the sinusoidal surface.

Due to limitations in the MFIE representation of the scattering process, the accuracy of the solution to the scattering matrix equation decreases as the incident angle approaches grazing. It is in these areas where edge effects are more prominent, and therefore the system is more difficult to model. As seen in Table IV, the angle of incidence does not significantly affect the number of iterations the algorithms required to converge. In the table, bold numerals indicate a transition in the number of required iterations.

Table IV. Effect of Illumination Angle on Number of Iterations for N=320.

Incident Angle	BICGSTAB	BCG	CG	CGS	GS	Jacobi
0-9	3	6	7	4	6	10
10	**4**	**7**	7	4	**5**	**9**
11-12	4	7	**8**	4	5	9
13	4	**8**	8	4	5	9
14	4	8	8	**5**	5	9
15	4	8	8	5	5	**8**
16	4	8	8	**4**	5	8
17-18	**3**	**7**	8	4	5	8
19	3	7	8	4	5	**7**
20-21	3	7	**7**	**3**	5	7
22-29	3	**6**	7	3	5	7
30-33	3	**5**	**6**	3	5	7
34	3	**6**	6	3	5	7
35-40	3	6	**7**	3	5	7
41-42	3	**7**	7	**4**	5	7
43-47	**4**	7	7	4	**6**	**8**
48-58	4	**8**	7	4	6	**9**
59-60	4	**7**	7	4	6	9
61-68	4	7	7	4	6	**10**
69-74	4	**6**	7	4	6	10
75-81	**3**	6	7	4	6	10
82-88	3	**7**	7	**3**	6	10
89-90	3	6	7	3	6	10

Most iterative algorithms require fewer iterations to solve the matrix equations when the incident angle is between 30 and 33 degrees. When the incident angle is zero as in the other experiments, the Jacobi method requires more iterations (10) than at any other angle. This indicates that in general Jacobi might perform better than these tests show.

The ratio of the sinusoidal surface amplitude to the incident wavelength also affects the convergence behavior of the iterative methods. As this ratio increases, the degree of roughness increases and the moment method grid is less able to accurately model the surface radiation, yielding a MM interaction matrix that is less well-conditioned. This hypothesis is confirmed from the experimental results shown in Figure 11, which shows the effect of change in this ratio on the number of iterations required for convergence.

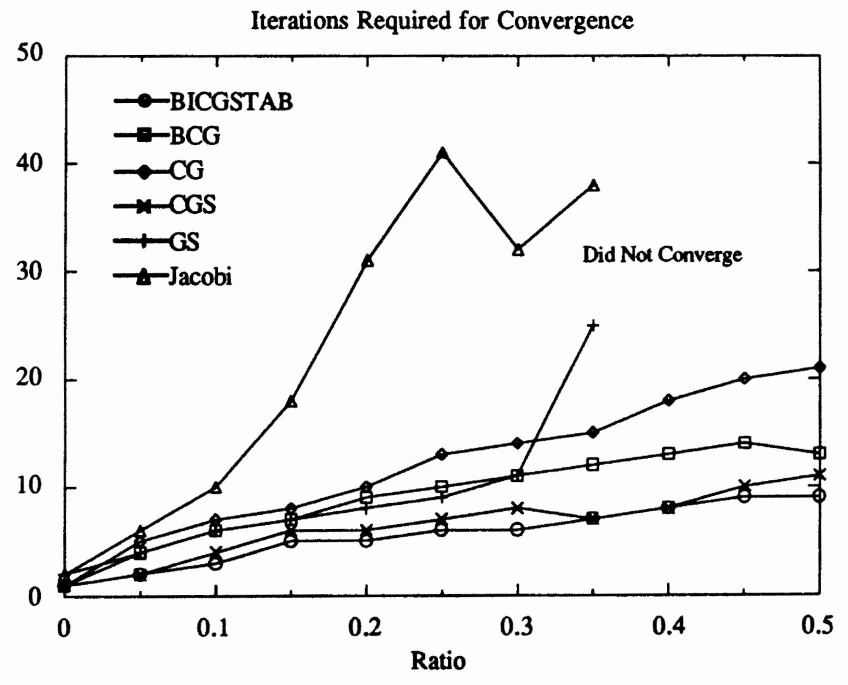

Figure 11. Effect of MM Grid Ratio on Convergence for N=320.

As the ratio approached the upper limit of the range tested, both the Gauss-Seidel method and the Jacobi method did not converge. Conversely, variants of CG converged under all surface amplitude to incident wavelength ratios.

Algorithm Parameter Changes

Missing from the experimental discussion to this point is the successive overrelaxation (SOR) method. A relative of the Gauss-Seidel method, SOR is the only iterative method that includes a relaxation parameter for refining the convergence. The relaxation factor, which can vary from 0 to 2, significantly affects the convergence of the SOR in certain circumstances. If a value of one is used, the SOR algorithm reduces to the Gauss-Seidel method [41]. To determine the optimal relaxation factor, it was varied between 0.1 and 1.6 for four different surface amplitude ratios. The results are shown in Figure 12.

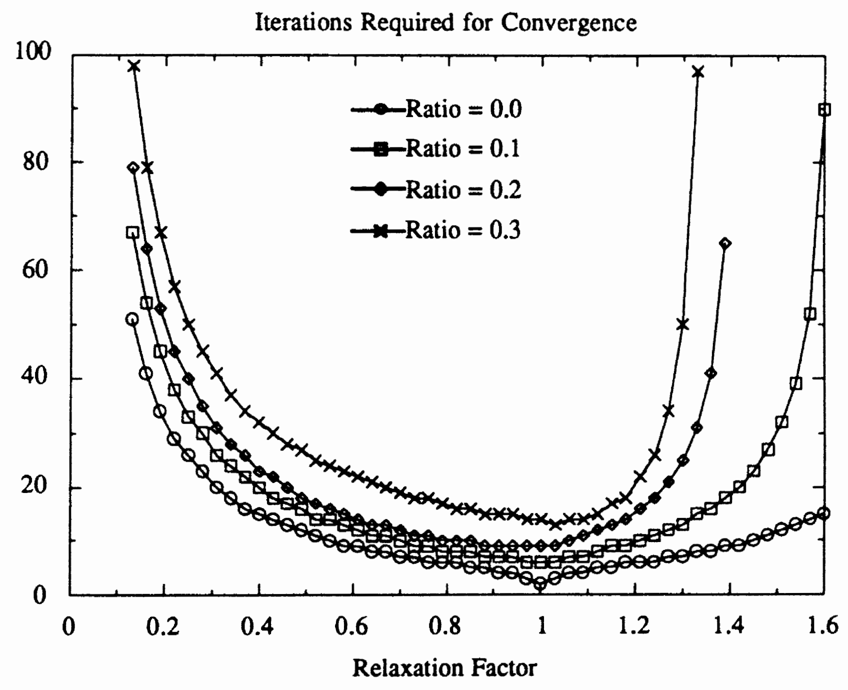

Figure 12. Effect of Relaxation Factor on SOR Convergence for N=320.

From the figure, the optimal relaxation factor for realistic sinusoidal surface heights was determined to be 1. Only for surface height ratios of 0.4 and above did the relaxation factor deviate significantly from unity. Since ratios higher than 0.3 are unlikely to produce accurate results in the moment method analysis due to MM grid limitations, the SOR can be assumed to converge no differently than Gauss-Seidel for practical scattering problems. It is for this reason that SOR was excluded from the experimental discussions until now.

Machine Dependence

The next experiments confirmed that the results shown above are valid across different computer architectures. Although primarily a function of the performance of the hardware, execution performance on different computers is important to assure validity of the experimental data. Figure 13 shows the CPU execution times for the iterative algorithms for the DEC 5000 for matrix dimensions ranging from 80 to 320.

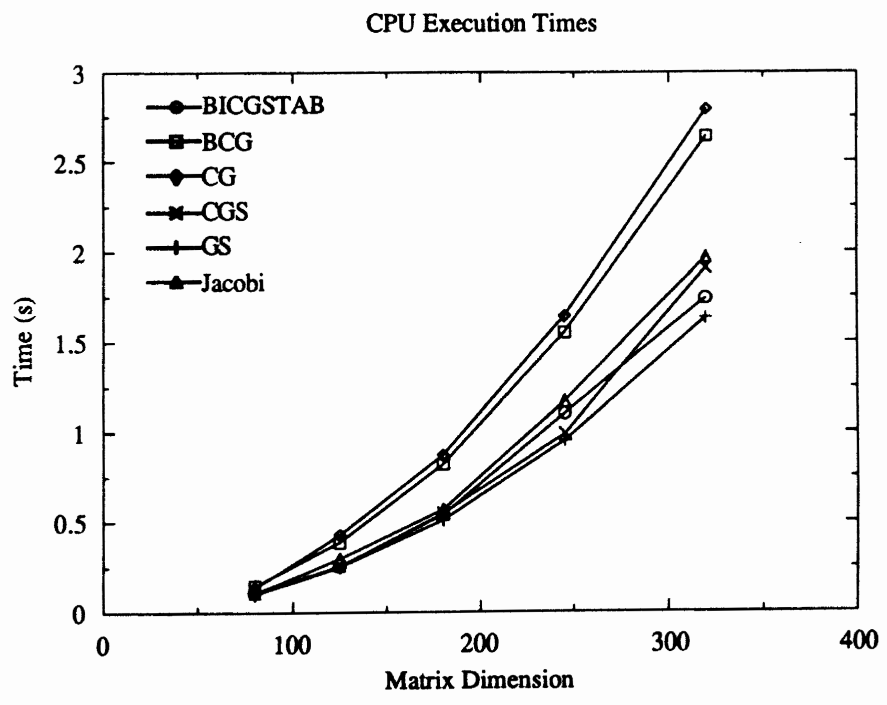

Figure 13. MES CPU Execution Times, DEC-5000.

The times include the matrix fill times. Since the DEC-5000 is a scalar RISC machine similar to the RS/6000, no significant difference was expected in the relative performance of the iterative methods. These results confirm this hypothesis. For a matrix dimension of 320, CGS required an additional iteration, so its execution performance decreased slightly.

The CPU execution times for the iterative methods using the IBM 3090 vector-processing computer are shown in Figure 14.

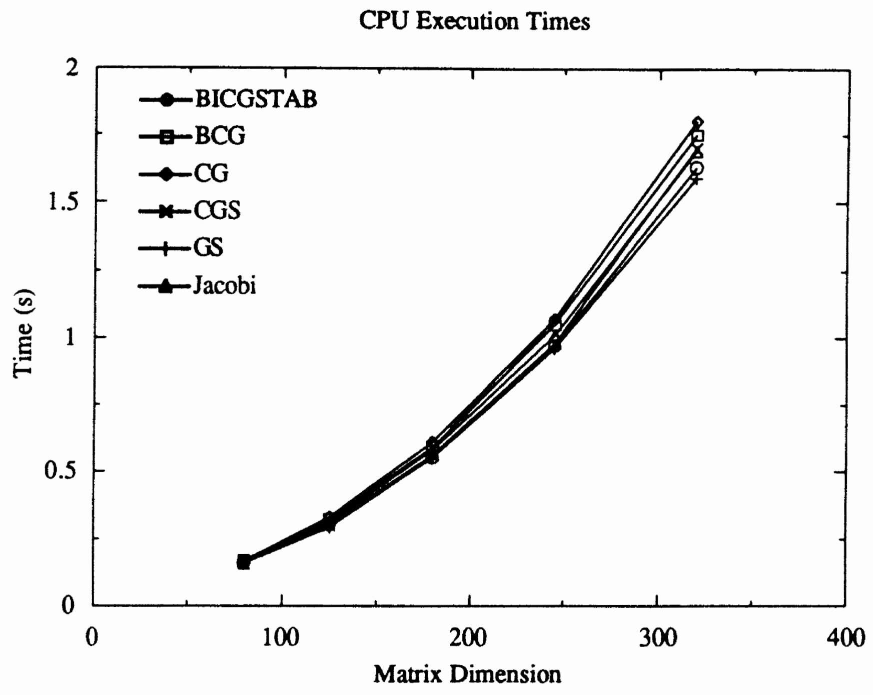

Figure 14. MES CPU Execution Times, IBM 3090.

For this computer system, a slight change in the relative performance levels of the methods is noted. The performance of the Jacobi method slightly exceeds that of the CGS method. Since the IBM 3090 possesses vector-processing capabilities, it can take advantage of parallelism inherent in the Jacobi method and thereby improve its performance level.

Efficiency Study

This section explains the preceding experimental results in terms of the iterative algorithm theory. These experiments demonstrate the relative efficiency of the solution methods but do not clearly show which algorithms require less computational work per significant digit of accuracy in the solution vector magnitude. To form this comparison, it is necessary to determine the total number of multiplications required for convergence and divide by the number of significant digits. Table V compares the iterative methods based on this technique using the number of iterations required for convergence for each algorithm as shown in the Figure 4.

Table V. Multiplications Required for Convergence for N=720.

Algorithm	Outside Loop	Inside Loop	Iterations	Total	Per Digit ($\times 10^6$)
GS	0	521281	6	3127686	1.043
BICGSTAB	518400	1044725	3	3652575	1.218
CGS	518400	1044003	4	4694412	1.565
Jacobi	0	521281	10	5212810	1.738
BCG	518400	1043283	6	6778098	2.259
CG	1037520	1041843	7	8330421	2.777

The iterative methods in this table are sorted according to efficiency, with the most efficient algorithms placed first. By providing a secondary source of comparison, this table confirms the order of efficiency of the algorithms found by execution time only.

Table VI and Table VII normalize the MES and MER CPU execution times relative to that for the GS algorithm.

Table VI. Comparison of Work to Experimental Execution Times for N=720.

Algorithm	MES Time	MER Time	MES Ratio	MER Ratio	Expected
GS	1.08	50.04	1.043	1.043	--
BICGSTAB	1.28	59.06	1.236	1.230	1.218
CGS	1.65	72.87	1.593	1.518	1.565
Jacobi	1.81	81.65	1.747	1.701	1.738
BCG	7.46	105.51	7.201	2.198	2.259
CG	9.92	132.18	9.576	2.754	2.777

The MER experiments nearly matched the predicted values, but the MES experiments deviated significantly.

In Table VI, the MES ratio for CG and BCG greatly exceeded the ratio predicted by the number of multiplications required alone. Both of these methods require the access to the matrix and the matrix transpose in each iteration. Since FORTRAN uses column major storage for matrices, accessing the rows of the transpose matrix involves selecting sixteen byte complex matrix elements separated by $16N$ bytes, where N is the matrix dimension. For larger matrices, this access involves retrieval from disk through page swapping. Other programming languages would suffer identical problems if only the matrix and not the matrix transpose is stored for page swapping configurations.

Table VII. Comparison of Work to Experimental Execution Times for N=1620.

Algorithm	MES Time	MER Time	MES Ratio	MER Ratio	Expected
GS	1106.60	254.80	1.043	1.043	--
BICGSTAB	1172.20	294.60	1.104	1.205	1.218
CGS	1145.00	371.00	1.079	1.518	1.565
Jacobi	1248.00	460.80	1.176	1.885	1.738
BCG	2090.00	536.80	1.969	2.196	2.259
CG	2045.00	685.60	1.927	2.805	2.777

Table VII shows that the advantage in execution time for GS diminishes for large matrices. The MES ratios in this case only roughly follow the trends given in Table V. Since page swapping overhead occupies much of the CPU execution time, the MES ratios and execution times have little meaning for matrices exceeding the RAM storage capacity of the computer system.

CHAPTER 6

CONCLUSIONS

Experimental results confirm the usefulness of iterative methods for solving the large linear systems in moment method calculations of electromagnetic scattering. All the iterative methods are more efficient in terms of computational time and storage space requirements than direct methods of solution for large problems and low-accuracy solutions. In addition, most of the iterative methods converge on these problems for wide ranges of parameter changes and matrix dimensions. Experiments also demonstrate the advantage of using a matrix element recalculation configuration for the iterative methods, for extremely large scattering problems. Without exception, iterative methods using the MER configuration outperform the respective matrix storage configuration for large problems where the matrix storage requirements exceed physical memory.

Of the algorithms tested, the BICGSTAB method possesses the most attractive combination of execution speed and convergence stability for use with the MM analysis of electromagnetic scattering. For well-conditioned systems, the Gauss-Seidel method effectively solves the matrix equation in less CPU time than any other method. Unfortunately, it cannot solve the equations where the MM grid is too coarse and therefore cannot be trusted for general problems. CGS provides quick and stable convergence to the solution for the experiments performed, but the literature indicates that it suffers stability problems for certain cases [35].

BIBLIOGRAPHY

[1] Axline, R. M., and Adrian K. Fung, "Numerical computation of scattering from a perfectly conducting random surface," *IEEE Transactions on Antennas and Propagation*, **vol. AP-26**, no. 3, pp. 482-488, May 1978.

[2] Balanis, Constantine A., *Advanced Engineering Electromagnetics*. New York: John Wiley and Sons, 1989, pp. 707-717.

[3] Chen, M. F., and A. K. Fung, "A numerical study of the regions of validity of the Kirchhoff and small-perturbation rough surface scattering models," *Radio Science*, **vol. 23**, no. 2, pp. 163-170, March-April 1988.

[4] Chen, Ruimin, notes for the development of the scalar Magnetic Field Integral Equation, April 1993.

[5] Elman, H. C., *Iterative Methods for Large Sparse Nonsymmetric Systems of Linear Equations*. Ph.D. thesis, Computer Science Department, Yale University, New Haven, CT, 1978.

[6] Fletcher, R., "Conjugate gradient methods for indefinite systems," *Numerical Analysis: Proceedings of the Dundee Conference on Numerical Analysis*. G.A. Watson, ed., New York: Springer-Verlag, no. 506,1976, pp. 73-89.

[7] Forsythe, George E., "Solving linear algebraic equations can be interesting," *Bull. of Am. Math. Society*, pp. 299-329, 1953.

[8] Forsythe, George E. and Wolfgang R. Wasow, *Finite-difference Methods for Partial Differential Equations*. New York: John Wiley and Sons, 1960.

[9] Fung, A. K., and M. F. Chen, "Numerical simulation of scattering from simple and composite random surfaces," *J. Opt. Soc. Am. A*, **vol. 2**, no. 12, pp. 2274-2284, December 1985.

[10] Glisson, Allen W. and Donald R.Wilton, "Simple and efficient numerical methods for problems of electromagnetic radiation and scattering from surfaces," *IEEE Transactions on Antennas and Propagation*, **vol. AP-28**, no. 5, pp. 593-603, September 1980.

[11] Golub, Gene H. and Dianne P. O'Leary, "Some history of the conjugate gradient and Lanczos algorithms: 1948-1976," *SIAM Review*, **vol. 31**, no. 1, pp. 50-102, March 1989.

[12] Golub, Gene H. and Charles F. Van Loan, *Matrix Computations*, 2nd. ed., Baltimore, Md: John Hopkins University Press, c1989.

[13] Gregory, Robert T. and David L. Karney, *A Collection of Matrices for Testing Computational Algorithms*. New York: Wiley-Interscience, 1969.

[14] Hageman, Louis A. and David M. Young, *Applied Iterative Methods*, New York: Academic Press, 1981.

[15] Harrington, R. F., *Field Computation by the Moment Method.* New York: Macmillan, 1968.

[16] Hestenes, Magnus R. and Eduard Stiefel, "Methods of conjugate gradients for solving linear systems," *Journal of Research of the National Bureau of Standards*, **vol. 49**, no. 6, pp. 409-436, December 1952.

[17] Joubert, Wayne D. and Thomas A. Manteuffel, "Iterative methods for non-symmetric linear systems," *Iterative Methods for Large Linear Systems*. Boston: Academic Press, Inc., 1990, pp. 149-171.

[18] Keller, J. B., "Geometrical Theory of Diffraction," *J. Opt. Soc. Amer.*, **vol. 52**, no. 2, pp. 116-130, February 1962.

[19] Kincaid, David R. and E. Ward Cheney, *Numerical Analysis: Mathematics of Scientific Computing*. Pacific Grove, California: Brooks/Cole Publishing Company, 1991, pp. 117-134, 152-153, 161-171, 181-201.

[20] Kronsjo, Lydia, *Algorithms: Their Complexity and Efficiency.* New York: John Wiley and Sons, 1987, pp. 88-148.

[21] Lanczos, C., "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators," *J. Res., Nat. Bur. Standards*, **vol. 45**, 1950, pp. 255-282.

[22] Miller, Webb and David Spooner, "Software for Roundoff Analysis, II," ACM Transactions on Mathematical Software, **vol. 4**, no. 4, pp. 369-387, December 1978.

[23] Ney, Michael M., "Method of moments as applied to electromagnetic problems," *IEEE Transactions on Microwave Theory and Techniques*, **vol. MTT-33**, no. 10, pp. 972-980, October 1985.

[24] Pearson,L. Wilson, "A technique for organizing large moment calculations for use with iterative solution methods," *IEEE Transactions on Antennas and Propagation*, **vol. AP-33**, no. 9, pp. 1031-1033, September 1985.

[25] Peterson, Andrew F. and Raj Mittra, "Iterative-based computational methods for electromagnetic scattering from individual or periodic structures," *IEEE Journal of Oceanic Engineering*, **vol. OE-12**, no. 2, pp. 458-465, April 1987.

[26] Peterson, Andrew F. and Raj Mittra, "Method of conjugate gradients for the numerical solution of large-body electromagnetic scattering problems," *Journal of the Optical Society of America A*. **vol. 2**, no. 6, pp. 971-977, June 1985.

[27] Poggio, A. J. and E.K. Miller, "Integral equation solutions of three-dimensional scattering problems," *Computer Techniques for Electromagnetics*, R. Mittra, ed., New York: Pergamon Press, 1973, pp. 159-261.

[28] Rao, Sadasiva M., Donald R. Wilton, and Allen W. Glisson, " Electromagnetic scattering by surfaces of arbitrary shape," *IEEE Transactions on Antennas and Propagation*, **vol. AP-30**, no. 3, pp. 409-418, May 1982.

[29] Saad, Youcef and Martin H. Schultz, "GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM J.Sci. Stat. Comput.*, **vol. 7**, no. 3, pp. 856-869, July 1986.

[30] Sarkar, Tapan K. and Sadasiva M. Rao, "The application of the conjugate gradient method for the solution of electromagnetic scattering from arbitrarily oriented wire antennas," *IEEE Transactions on Antennas and Propagation*, **vol. AP-32**, no. 4, pp. 398-403, April 1984.

[31] Sarkar, Tapan K. and Ercument Arvas, "On a class of finite step iterative methods (conjugate directions) for the solution of an operator equation arising in electromagnetics," *IEEE Transactions on Antennas and Propagation*, **vol. AP-33**, no. 10, pp. 1058-1066, October 1985.

[32] Sarkar, Tapan K., Kenneth R. Siarkiewicz, and Roy F. Stratton, "Survey of numerical methods for solution of large systems of linear equations for electromagnetic field problems," *IEEE Transactions on Antennas and Propagation*, **vol. AP-29**, no. 6, pp. 847-856, November 1981.

[33] Sonneveld, Peter, "CGS, a fast Lanczos-type solver for nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, **vol. 10**, no. 1, pp. 36-52, January 1989.

[34] Sultan, Michel F. and Raj Mittra, "An iterative moment method for analyzing the electromagnetic field distribution inside inhomogeneous lossy dielectric objects," *IEEE Transaction on Microwave Theory and Techniques*, **vol. MTT-33**, no. 2, pp. 163-168, February 1985.

[35] Van Der Vorst, H. A., "BI-CGSTAB: a fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, **vol. 13**, no. 2, pp. 631-644, March 1992.

[36] Varga, Richard S., *Matrix Iterative Analysis*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1962.

[37] Vuik, C. and H. A. Van Der Vorst, "A comparison of some GMRES-like methods," *Linear Algebra and Its Applications*, **vol. 160**, pp. 131-162, 1992.

[38] L. B. Wetzel, "Models for Electromagnetic Scattering from the Sea at Extremely Low Grazing Angles," Naval Research Laboratory Report 6098, December 31, 1987.

[39] Wilkinson, James H., *The Algebraic Eigenvalue Problem*. Oxford: Clarendon Press, 1965.

[40] Young, David M. and Kang C. Jea, "Generalized conjugate-gradient acceleration of nonsymmetrizable iterative methods," *Linear Algebra and Its Applications*, **vol. 34**, pp. 159-194, 1980.

[41] Young, David M., *Iterative Solution of Large Linear Systems*. New York: Academic Press, Inc., 1971.

APPENDIXES

APPENDIX A

BICONJUGATE GRADIENT METHOD

Source Code

```
c       Unpreconditioned Biconjugate gradient iterative method taken from
c       "Bi-CGSTAB: A Fast Variant of BI-CG", H.A. Van Der Vorst
c       SIAM J. SCI. Stat. Comput., Vol 13, No 2, pp 631-644, Mar 1992
c
c       Van Der Vorst credited the algorithm to
c       Fletcher, R., "Conjugate Gradient Methods for Indefinite Systems",
c       Numerical Analysis Dundee 1975, G.A. Watson ed., New York: Springer,
c       Lecture Notes in Mathematics, No. 506, 1976, pp. 73-89.
c
c       Note: Matrix "a" contains the transposed coefficient matrix
c
c       Passed Variables:
c
c       a       complex*16   matrix (zmn) (npxnp)
c       b       complex*16   vector containing right hand side of ax=b
c                            (will contain solution upon exit)
c       n       integer      number of unknowns
c       np      integer      physical dimension of "a" matrix
c       eps     real*8       stopping condition for iterations
c
c       Local Variables:
c
c       p       complex*16   direction vector
c       pq      complex*16   second p vector
c       x       complex*16   solution vector
c       r       complex*16   residual vector (r=b-ax)
c       rq      complex*16   second residual vector
c       nu      complex*16   holds matrix-vector product Ap
c       alpha   complex*16   bcg parameter
c       beta    complex*16   bcg parameter
c       rho     complex*16   bcg parameter
c       oldrho  complex*16   bcg parameter
c       sum     complex*16   temporary matrix-vector product
c       i,j     integer      loop counters
c       k       integer      iteration counter
c       mr      real*8       magnitude of residual vector
c       mx      real*8       magnitude of solution vector
c       gam     real*8       ratio of mr to mx
c       eps     real*8       stopping criterion for gam

        subroutine bcg(a,b,n,np,eps,nit)
        implicit none
        real*8 mr,mx,gam,eps
        integer i,j,k
        integer n,np,nit,max
        parameter (max=50)
        complex*16 a(np,np),b(np),x(1620),r(1620),p(1620)
        complex*16 rq(1620),pq(1620),nu(1620)
        complex*16 sum,rho,oldrho,sigma,alpha,beta

c       initial guess for solution (x) vector
        do i=1,n
          x(i)=0.0
        end do

c       initial residual (r) and p vectors
        do i=1,n
          p(i)=0.0
          pq(i)=0.0
          sum=0.0
          do j=1,n
```

```
            sum=sum+a(j,i)*x(j)
          end do
          r(i)=b(i)-sum
          rq(i)=r(i)
        end do

c       initialize iteration variables
        k=0
        rho=1.0
        gam=1.0

c       main iteration loop
c       repeat while the stopping criterion is not met
c       (see below for details)
        do 10 while ((gam.gt.eps).and.(k.lt.max))

c         calculate rho and beta
          oldrho=rho
          rho=0.0
          do i=1,n
               rho=rho+rq(i)*r(i)
          end do
          beta=rho/oldrho

c         calculate new p and pq
          do i=1,n
            p(i)=r(i)+beta*p(i)
            pq(i)=rq(i)+beta*pq(i)
          end do

c         calculate sigma and alpha
          sigma=0.0
          do i=1,n
            sum=0.0
            do j=1,n
              sum=sum+a(j,i)*p(j)
            end do
            nu(i)=sum
            sigma=sigma+pq(i)*sum
          end do
          alpha=rho/sigma

c         calculate new residuals
          do i=1,n
            sum=0.0
            do j=1,n
              sum=sum+a(i,j)*pq(j)
            end do
            r(i)=r(i)-alpha*nu(i)
            rq(i)=rq(i)-alpha*sum
          end do

c         calculate new solution
          do i=1,n
            x(i)=x(i)+alpha*p(i)
          end do

c         increment iteration counter
          k=k+1

c         Determine whether stopping criterion has been met
c         using the following algorithm:
c
c         mr = Sqrt(<r,r*>)
```

```
c         mx = Sqrt(<x,x*>)
c         gam = mr / mx
c         stop iterations iff gam <= eps

c         clear magnitudes of solution and residual
          mr=0.0
          mx=0.0

c         determine new magnitudes
          do j=1,n
            mr=mr+r(j)*conjg(r(j))
            mx=mx+x(j)*conjg(x(j))
          end do

          mr=sqrt(mr)
          mx=sqrt(mx)
          gam=mr/mx

10        continue

c       replace b vector with the solution
        do i=1,n
          b(i)=x(i)
        end do

        nit=k
        end
```

APPENDIX B

BICGSTAB METHOD

Source Code

```
c       Algorithm taken from "BI-CGSTAB:  A Fast and Smoothly Converging
c       Variant of BI-CG for the Solution of Nonsymmetric Linear Systems",
c       by H.A. Van Der Vorst, SIAM J. Sci. Stat. Comput., Vol. 13, No. 2,
c       pp. 631-644, March 1992
c
c       Note: Matrix "a" contains the transposed coefficient matrix
c
c       Passed Variables:
c
c       a       complex*16   matrix (zmn) (npxnp)
c       b       complex*16   vector containing right hand side of ax=b
c       n       integer      number of unknowns
c       np      integer      physical dimension of "a" matrix
c       eps     real*8       stopping condition for iterations
c
c       Local Variables:
c
c       r       complex*16   residual vector
c       x       complex*16   solution vector
c       r0      complex*16   initial residual vector
c       sum     complex*16   used in matrix-vector products
c       mr      real*8       magnitude of residual
c       mx      real*8       magnitude of solution
c       gam     real*8       ratio of mr to mx

        subroutine bicgstab(a,b,n,np,eps,nit)
        implicit none
        real*8 mr,mx,gam,eps
        integer i,j,k,n,np,max,nit
        parameter (max=50)
        complex*16 a(np,np),b(np),x(1620),r(1620)
        complex*16 sum,s(1620),t(1620),p(1620),nu(1620),r0(1620)
        complex*16 num,den,rho,oldrho,alpha,omega,beta

c       initial guess for the solution (x) vector
        do i=1,n
          x(i)=0.0
        end do

c       initial residual (r) vector
        do i=1,n
          r(i)=0.0
          do j=1,n
            sum=sum+a(j,i)*x(j)
          end do
          r(i)=b(i)-sum
          r0(i)=r(i)
        end do

c       initialize iteration variables
        gam=1.0
        rho=1.0
        alpha=1.0
        omega=1.0
        k=0
        do i=1,n
          s(i)=0.0
          t(i)=0.0
          p(i)=0.0
          nu(i)=0.0
        end do
```

```
c      main iteration loop
c      repeat while the stopping criterion is not met
c      (see below for details)
       do 10 while ((gam.gt.eps).and.(k.lt.max))
         k=k+1

c        calculate new rho
c        use conjugated r0 vector (modification for complex)
         oldrho=rho
         rho=0.0
         do i=1,n
           rho=rho+conjg(r0(i))*r(i)
         end do

c        calculate beta
         beta=rho/oldrho*alpha/omega

c        calculate new p
         do i=1,n
           p(i)=r(i)+beta*(p(i)-omega*nu(i))
         end do

c        calculate new nu
         do i=1,n
           sum=0.0
           do j=1,n
             sum=sum+a(j,i)*p(j)
           end do
           nu(i)=sum
         end do

c        calculate new alpha
c        use conjugated r0 vector (modified for complex)
         sum=0.0
         do i=1,n
           sum=sum+conjg(r0(i))*nu(i)
         end do
         alpha=rho/sum

c        calculate new s
         do i=1,n
           s(i)=r(i)-alpha*nu(i)
         end do

c        calculate new t
         do i=1,n
           sum=0.0
           do j=1,n
             sum=sum+a(j,i)*s(j)
           end do
           t(i)=sum
         end do

c        calculate new omega
c        conjugate first vector in inner products (t)
c        (modify for complex)
         num=0.0
         den=0.0
         do i=1,n
           num=num+conjg(t(i))*s(i)
           den=den+conjg(t(i))*t(i)
         end do
         omega=num/den
```

```
c        update solution vector
         do i=1,n
           x(i)=x(i)+alpha*p(i)+omega*s(i)
         end do

c        update the residual vector
         do i=1,n
           r(i)=s(i)-omega*t(i)
         end do

c        Determine whether stopping criterion has been met
c        using the following algorithm:
c
c        mr = Sqrt(<r,r*>)
c        mx = Sqrt(<x,x*>)
c        gam = mr / mx
c        stop iterations iff gam <= eps

c        clear magnitudes of solution and residual
         mr=0.0
         mx=0.0

c        determine new magnitudes
         do j=1,n
           mr=mr+r(j)*conjg(r(j))
           mx=mx+x(j)*conjg(x(j))
         end do

         mr=sqrt(mr)
         mx=sqrt(mx)
         gam=mr/mx

10       continue

c      replace b vector with the solution
       do i=1,n
         b(i)=x(i)
       end do

       nit=k
       end
```

APPENDIX C

CONJUGATE GRADIENT METHOD

Source Code

```
c       Conjugate-Gradient Iteration algorithm taken from
c       "Methods of Conjugate Gradients for Solving Linear Systems",
c       Hestenes and Stiefel, Journal of Research of the National Bureau
c       of Standards, Vol. 49, No. 6, December 1952, pp.409-436.
c
c       This subroutine uses Hestenes and Stiefel algorithm (10:2) for
c       solving the linear system Ax=k where A is a general nonsingular matrix.
c
c       Note: Matrix "a" contains the transposed coefficient matrix
c
c       Passed Variables:
c
c       a       complex*16   matrix (zmn) (npxnp)
c       b       complex*16   vector containing right hand side of ax=b
c       n       integer      number of unknowns
c       np      integer      physical dimension of "a" matrix
c       eps     real*8       stopping condition for iterations
c
c       Local Variables:
c
c       p       complex*16   direction vector (Hestenes and Stiefel)
c       ap      complex*16   matrix-vector product a*p
c       actr    complex*16   matrix-vector product a(conj.transpose)*r
c       x       complex*16   solution vector
c       r       complex*16   residual vector (r=b-ax)
c       alpha   complex*16   conjugate gradient parameter
c       beta    complex*16   conjugate gradient parameter
c       i,j     integer      loop counters
c       k       integer      iteration counter
c       mr      real*8       magnitude of residual vector
c       mx      real*8       magnitude of solution vector
c       gam     real*8       ratio of mr to mx
c       oldms   real*8       previous magnitude squared of (a*)*r
c       ms      real*8       current magnitude squared of (a*)*r
c       den     real*8       temporary variable (denominator of expression)
c       eps     real*8       stopping criterion for gam

        subroutine cg(a,b,n,np,eps,nit)
        implicit none
        integer np
        complex*16 a,b,x,r,p,ap,actr
        dimension a(np,np),b(np),x(1620),r(1620),p(1620)
        dimension ap(1620),actr(1620)
        complex*16 alpha,beta,sum
        integer i,j,k,n,nit,max
        parameter (max=50)
        real*8 mr,mx,gam,oldms,ms,den,eps

c       initial guess for solution vector
        do i=1,n
          x(i)=0.0
        end do

c       initial residual (r) vector
        do i=1,n
          sum=0.0
          do j=1,n
            sum=sum+a(j,i)*x(j)
          end do
          r(i)=b(i)-sum
        end do
```

```
c       find the initial direction (p) vector and
c       the initial magnitude of a conj. transpose * r squared (ms)
        ms=0.0
        do i=1,n
          sum=0.0
          do j=1,n
            sum=sum+conjg(a(i,j))*r(j)
          end do
          p(i)=sum
          ms=ms+sum*conjg(sum)
        end do

c       initialize iteration variables
        k=0
        gam=1.0

c       main iteration loop
        do 10 while ((gam.gt.eps).and.(k.lt.max))

c         multiply a * p to get the denominator of alpha expression
c         and save result in ap so that it can be used in finding the new
c         residual vector later
          den=0.0
          do i=1,n
            sum=0.0
            do j=1,n
              sum=sum+a(j,i)*p(j)
            end do
            ap(i)=sum
            den=den+sum*conjg(sum)
          end do

c         since the (a*)*r was already calculated for the current residual,
c         ms can be used to find alpha (n^2 multiplications saved)
          alpha=ms/den

c         since the a*p was already calculated for the current direction,
c         ap can be used to find the new residual (n^2 multiplications saved)

c         determine new solution and residual vectors
          do i=1,n
            x(i)=x(i)+alpha*p(i)
            r(i)=r(i)-alpha*ap(i)
          end do

c         update the ms vector using the new residual
c         actr holds (a*)*r result for later use in the p vector calculation

c         determine numerator and denominator for beta
          oldms=ms
          ms=0.0
          do i=1,n
            sum=0.0
            do j=1,n
              sum=sum+conjg(a(i,j))*r(j)
            end do
            actr(i)=sum
            ms=ms+sum*conjg(sum)
          end do

c         determine beta
          beta=ms/oldms
```

```
c         determine new direction vector (p)
          do i=1,n
            p(i)=actr(i)+beta*p(i)
          end do

c         increment iteration counter
          k=k+1

c         Determine whether stopping criterion has been met
c         using the following algorithm:
c
c         mr = Sqrt(<r,r*>)
c         mx = Sqrt(<x,x*>)
c         gam = mr / mx
c         stop iterations iff gam <= eps

c         clear magnitudes of solution and residual
          mr=0.0
          mx=0.0

c         determine new magnitudes
          do j=1,n
            mr=mr+r(j)*conjg(r(j))
            mx=mx+x(j)*conjg(x(j))
          end do

          mr=sqrt(mr)
          mx=sqrt(mx)
          gam=mr/mx

10        continue

c       replace b vector with the solution
        do i=1,n
          b(i)=x(i)
        end do

        nit=k
        end
```

APPENDIX D

CONJUGATE GRADIENT SQUARED METHOD

Source Code

```
c       Conjugate Gradient Squared iteration algorithm taken from
c       "CGS, A Fast Lanczos-Type Solver for Nonsymmetric Linear Systems",
c       Peter Sonneveld, SIAM J. Sci. Stat. Comput., Vol. 10, No. 1,
c       pp.36-52, January 1989.
c
c       Note: Matrix "a" contains the transposed coefficient matrix
c
c       Passed Variables:
c
c       a       complex*16      matrix (zmn) (npxnp)
c       b       complex*16      vector containing right hand side of ax=b
c                               (will contain solution upon exit)
c       n       integer         number of unknowns
c       np      integer         physical dimension of "a" matrix
c       eps     real*8          stopping condition for iterations
c
c       Local Variables:
c
c       p       complex*16
c       q       complex*16
c       u       complex*16
c       v       complex*16      matrix-vector product a*p
c       num     complex*16      temporary variable (numerator)
c       upq     complex*16      vector sum u + q
c       rho     complex*16
c       oldrho  complex*16
c       sigma   complex*16
c       alpha   complex*16
c       beta    complex*16
c       mr      real*8          magnitude of residual
c       mx      real*8          magnitude of solution
c       gam     real*8          ratio of mr to mx

        subroutine cgs(a,b,n,np,eps,nit)
        implicit none
        integer i,j,k,n,nit,np,max
        parameter (max=50)
        complex*16 alpha,beta,sigma,rho,oldrho
        complex*16 a(np,np),b(np),x(1620),r(1620),p(1620),q(1620),u(1620)
        complex*16 sum,num,r0(1620),v(1620),upq(1620)
        real*8 mr,mx,gam,eps

c       initial guess for the solution vector
        do i=1,n
          x(i)=0.0
        end do

c       initial residual vector
        num=0.0
        do i=1,n
          sum=0.0
          do j=1,n
            sum=sum+a(j,i)*x(j)
          end do
          r(i)=b(i)-sum
          r0(i)=r(i)
          q(i)=0.0
          p(i)=0.0
        end do

c       initialize variables
```

```
      gam=1.0
      k=0
      rho=1

c     main iteration loop
c     repeat loop while the ratio of the magnitude of the residual to
c     the magnitude of the solution is greater than eps

      do 10 while ((gam.gt.eps).and.(k.lt.max))
        k=k+1

c       calculate new rho
        oldrho=rho
        rho=0.0
        do i=1,n
          rho=rho+conjg(r0(i))*r(i)
        end do

c       calculate new beta
        beta=rho/oldrho

c       calculate new u vector
        do i=1,n
          u(i)=r(i)+beta*q(i)
        end do

c       calculate new p vector
        do i=1,n
          p(i)=u(i)+beta*(q(i)+beta*p(i))
        end do

c       calculate matrix-vector product a*p
        do i=1,n
          sum=0.0
          do j=1,n
            sum=sum+a(j,i)*p(j)
          end do
          v(i)=sum
        end do

c       calculate sigma
        sigma=0.0
        do i=1,n
          sigma=sigma+conjg(r0(i))*v(i)
        end do

c       calculate alpha
        alpha=rho/sigma

c       calculate new q
        do i=1,n
          q(i)=u(i)-alpha*v(i)
        end do

c       calculate u + q
        do i=1,n
          upq(i)=u(i)+q(i)
        end do

c       calculate new residual vector
        do i=1,n
          sum=0.0
          do j=1,n
            sum=sum+a(j,i)*upq(j)
            end do
            r(i)=r(i)-alpha*sum
          end do

c         calculate new solution vector
          do i=1,n
            x(i)=x(i)+alpha*upq(i)
          end do

c         Determine whether stopping criterion has been met
c         using the following algorithm:
c
c         mr = Sqrt(<r,r*>)
c         mx = Sqrt(<x,x*>)
c         gam = mr / mx
c         stop iterations iff gam <= eps

c         clear magnitudes of solution and residual
          mr=0.0
          mx=0.0

c         determine new magnitudes
          do j=1,n
            mr=mr+r(j)*conjg(r(j))
            mx=mx+x(j)*conjg(x(j))
          end do

          mr=sqrt(mr)
          mx=sqrt(mx)
          gam=mr/mx

10        continue

c       replace b vector with the solution
        do i=1,n
          b(i)=x(i)
        end do

        nit=k
        end
```

APPENDIX E

GAUSS-SEIDEL METHOD

Source Code

```
c       Gauss-Seidel Iteration algorithm taken from the textbook,
c       David Kincaid and Ward Cheney. "Numerical Analysis".
c       (Pacific Grove, California: Brooks/Cole Publishing Co., 1991),
c       p.190.
c
c       Note: Matrix "a" contains the transposed coefficient matrix
c
c       Passed Variables:
c
c       a       complex*16      matrix (zmn) (npxnp)
c       b       complex*16      vector containing
c       n       integer         number of unknowns
c       np      integer         physical dimension of "a" matrix
c       eps     real*8          stopping condition (explained in code)
c
c       Local Variables:
c
c       x       complex*16      vector containing iterative solutions
c       d       complex*16      used in matrix conditioning
c       r       complex*16      residual vector
c       sum     complex*16      totals in matrix multiplication
c       temp    complex*16      temporary variable
c       rsum    complex*16      totals for residual vect. calculation
c       i,j     integer         indicies
c       k       integer         iteration number
c       mr      real*8          magnitude of residual
c       mx      real*8          magnitude of iterative solution
c       gam     real*8          ratio of mr to mx (compare to stopping value)

        subroutine gs(a,b,n,np,eps,nit)
        implicit none
        integer np
        complex*16 a,b,x,r,sum,temp,rsum
        dimension a(np,np),b(np),x(1620),r(1620)
        integer i,j,k,n,nit,max
        parameter (max=50)
        real*8 mr,mx,gam,eps

c       initial guess for x
        do i=1,n
          x(i)=0.0
        end do

c       initialize variables
        k=0
        gam=1.0

c       main iteration loop
c       repeat while the stopping criterion is not met
c       (see below for details)
        do 10 while ((gam.gt.eps).and.(k.lt.max))
          k=k+1

c         multiply coefficient matrix by x vector
c         using column-major storage to improve execution speed

          do i=1,n

c           clear the residual sum and product sum
            rsum=0.0
            sum=0.0
```

```
c          since the matrix is stored in column-major format,
c          we must step through the rows for each column in the do loop
           do j=1,n

             temp=a(j,i)*x(j)

c            Residual vector sum requires all terms
             rsum=rsum+temp

           end do

c          Solution vector sum only includes off-diagonal terms
c          Placement here avoids "if" in inner loop -- better optimization
           sum=rsum-a(i,i)*x(i)

c          calculate new solution and residual values for this row
c          MFIE diagonal is 0.5 so 1/a(i,i) = 2.0
           x(i)=2.0*(b(i)-sum)
           r(i)=b(i)-rsum

         end do

c        Determine whether stopping criterion has been met
c        using the following algorithm:
c
c        mr = Sqrt(<r,r*>)
c        mx = Sqrt(<x,x*>)
c        gam = mr / mx
c        stop iterations iff gam <= eps

c        clear magnitudes of solution and residual
         mr=0.0
         mx=0.0

c        determine new magnitudes
         do j=1,n
           mr=mr+r(j)*conjg(r(j))
           mx=mx+x(j)*conjg(x(j))
         end do

         mr=sqrt(mr)
         mx=sqrt(mx)
         gam=mr/mx

10       continue

c       replace b vector with the solution
        do i=1,n
          b(i)=x(i)
        end do

        nit=k
        end
```

APPENDIX F

JACOBI METHOD

Source Code

```
c      Jacobi Iteration algorithm taken from the textbook,
c      David Kincaid and Ward Cheney. "Numerical Analysis".
c      (Pacific Grove, California: Brooks/Cole Publishing Co., 1991),
c      p.186.
c
c      Note: Matrix "a" contains the transposed coefficient matrix
c
c      Passed Variables:
c
c      a       complex*16       matrix (zmn) (npxnp)
c      b       complex*16       vector containing
c      n       integer          number of unknowns
c      np      integer          physical dimension of "a" matrix
c      eps     real*8           stopping condition (explained in code)
c
c      Local Variables:
c
c      r       complex*16       residual vector
c      u       complex*16       updated solution vector
c      x       complex*16       solution vector
c      d       complex*16       diagonal element for matrix preconditioning
c      sum     complex*16       used for matrix-vector products
c      rsum    complex*16       used for residual calculation
c      temp    complex*16       temporary variable
c      i,j     integer          loop counters
c      k       integer          iteration counter
c      mr      real*8           magnitude of residual vector
c      mx      real*8           magnitude of solution vector
c      gam     real*8           ratio of mr to mx

       subroutine jac(a,b,n,np,eps,nit)
       implicit none
       integer np
       complex*16 a,b,x,r,sum,u,rsum,temp
       dimension a(np,np),b(np),x(1620),r(1620),u(1620)
       integer i,j,k,n,nit,max
       parameter (max=50)
       real*8 mr,mx,gam,eps

c      initial guess for solution vector
       do i=1,n
         x(i)=0.0
       end do

c      initialize variables
       k=0.0
       gam=1.0

c      main iteration loop
c      repeat while the stopping criterion is not met
c      (see below for details)
       do 10 while((gam.gt.eps).and.(k.lt.max))
         k=k+1

c        multiply coefficient matrix by solution vector
c        using column-major storage to improve execution speed

         do i=1,n

c          clear the matrix-vector product sum and residual sum
           sum=0.0
```

```
          rsum=0.0

c         since the matrix is stored in column-major format,
c         we must step through the rows for each column in the do loop
          do j=1,n
            temp=a(j,i)*x(j)
            rsum=rsum+temp
          end do

c         Solution vector sum includes only off-diagonal terms
c         Placing it here avoids "if" inside inner loop
          sum=rsum-a(i,i)*x(i)

c         jacobi does not immediately update the solution vector "x"
c         instead, it stores the update in "u"
c         MFIE diagonal is 0.5 so 1/a(i,i) = 2.0
          u(i)=2.0*(b(i)-sum)
          r(i)=b(i)-rsum

        end do

c       form the new solution vector
        do i=1,n
          x(i)=u(i)
        end do

        mr=0.0

c       Determine whether stopping criterion has been met
c       using the following algorithm:
c
c       mr = Sqrt(<r,r*>)
c       mx = Sqrt(<x,x*>)
c       gam = mr / mx
c       stop iterations iff gam <= eps

c       clear magnitudes of solution and residual
        mr=0.0
        mx=0.0

c       determine new magnitudes
        do j=1,n
          mr=mr+r(j)*conjg(r(j))
          mx=mx+x(j)*conjg(x(j))
        end do

        mr=sqrt(mr)
        mx=sqrt(mx)
        gam=mr/mx

10      continue

c     replace b vector with the solution
      do i=1,n
        b(i)=x(i)
      end do

      nit=k
      end
```

APPENDIX G

SUCCESSIVE OVERRELAXATION METHOD

Source Code

```
c      Symmetric Overrelaxation technique based upon the
c      Gauss-Seidel Iteration algorithm taken from the textbook,
c      David Kincaid and Ward Cheney. "Numerical Analysis".
c      (Pacific Grove, California: Brooks/Cole Publishing Co., 1991),
c      p.190.
c
c      Note:  Matrix "a" contains the untransposed coefficient matrix
c
c      Passed Variables:
c
c      a       complex*16   matrix (zmn) (npxnp)
c      b       complex*16   vector containing right hand side of ax=b
c                           (will contain solution upon exit)
c      n       integer      number of unknowns
c      np      integer      physical dimension of "a" matrix
c      eps     real*8       stopping condition (explained in code)
c
c      Local Variables:
c
c      x       complex*16   vector containing iterative solutions
c      r       complex*16   residual vector
c      sum     complex*16   totals in matrix multiplication
c      temp    complex*16   temporary variable
c      rsum    complex*16   totals for residual vect. calculation
c      i,j     integer      indicies
c      k       integer      iteration number
c      mr      real*8       magnitude of residual
c      mx      real*8       magnitude of iterative solution
c      gam     real*8       ratio of mr to mx (compare to stopping value)

       subroutine sor(a,b,n,np,eps,nit,w)
       complex*16 a,b,x,r,sum,temp,rsum
       dimension a(np,np),b(np),x(1620),r(1620)
       integer i,j,k,n,nit,max
       parameter (max=50)
       real*8 mr,mx,gam,eps
       real*8 w

c      initial guess for solution vector
       do i=1,n
         x(i)=0.0
       end do

c      initialize variables
       k=0
       gam=1.0

c      main iteration loop
c      repeat while the stopping criterion is not met
c      (see below for details)
       do 10 while ((gam.gt.eps).and.(k.lt.max))
         k=k+1

c        multiply coefficient matrix by solution vector
c        using column-major storage to improve execution speed

         do i=1,n

c          clear the residual sum and product sum
           rsum=0.0
           sum=0.0
```

```
c          since the matrix is stored in column-major format,
c          we must step through the rows for each column in the do loop
           do j=1,n

             temp=a(j,i)*x(j)

c            Residual vector sum requires all terms
             rsum=rsum+temp

           end do

c          Solution vector sum only includes off-diagonal terms
           sum=rsum-a(i,i)*x(i)

c          calculate new solution and residual values for this row
c          MFIE diagonal is 0.5 so 1/a(i,i) = 2.0
           x(i)=2.0*(w*(b(i)-sum))+(1.0-w)*x(i)
           r(i)=b(i)-rsum

         end do

c        Determine whether stopping criterion has been met
c        using the following algorithm:
c
c        mr = Sqrt(<r,r*>)
c        mx = Sqrt(<x,x*>)
c        gam = mr / mx
c        stop iterations iff gam <= eps

c        clear magnitudes of solution and residual
         mr=0.0
         mx=0.0

c        determine new magnitudes
         do j=1,n
           mr=mr+r(j)*conjg(r(j))
           mx=mx+x(j)*conjg(x(j))
         end do

         mr=sqrt(mr)
         mx=sqrt(mx)
         gam=mr/mx

10       continue

c      replace b vector with the solution
       do i=1,n
         b(i)=x(i)
       end do
       nit=k
       end
```

APPENDIX H

MOMENT METHOD SOURCE CODE

```
c      ********************************************************************
c      Iterative Solution to the Moment Method Equations:  Testing Program
c      Moment Method Code:  Rumin Chen, 1992-1993
c      Testing and Iterative Solution:  Mike Sturm, 1993
c      ********************************************************************
c
c      ***********************************
c      Moment Method Comments (Rumin Chen)
c      ***********************************
c
c      This code is a Moment Method program used to calculate a conducting
c      sinusoid surface scattering problem by using Magnetic Field Integral
c      Equation.
c      The code is implemented based on the pulse interpolation basis
c      function on square patchs. The unknown current is defined
c      on the center of the patch.
c
c      The chief data structures are listed as below
c      area    vector for the area of square patch
c      el      vector for some information of the patch
c              1-- length of the patch on l direction
c              2-- y component of the unit vector on l direction
c              3-- z component of the unit vector on l direction
c      k       incident wave number
c      n       number of unknown or total number of patches
c      v       vector for right source of the moment matrix equation
c              when the equation is solved, it holds the solved current
c      wl      incident wave length
c      x       vector for x coordinate of triangular vertex
c      y       vector for y coordinate of triangular vertex
c      z       vector for z coordinate of triangular vertex
c      zmn     N by N matrix of moment matrix equation
c
c
c      ****************************************************
c      Testing and Iterative Solution Comments (Mike Sturm)
c      ****************************************************
c
c      In order to get an objective measure of the execution time for the
c      iterative algorithms, it was necessary to add the following variables
c      to Chen's program.
c
c      The added variables pertain to two methods for precise timing
c      of the execution time.  The AIX Fortran call "mclock" returns user
c      plus system time for the executing process, in one-hundredths of a
c      second.  Using "C", I wrote another subroutine called "rtime", which
c      returns the number of sec. after midnight, Greenwich Mean Time (GMT).
c      By calling these two functions before and after the matrix fill and
c      iterative solution, an accurate estimate of execution time was found.
c
c      start      integer   starting time of algorithm (mclock)
c      end        integer   ending time of algorithm (mclock)
c      loop       integer   counter for mulitple testing of execution times
c      loopcount  integer   number of tests for each matrix size
c      nit        integer   number of iterations req'd for solution
c      rtime      integer   "c" subroutine:  sec. past 0:00:00 GMT (system)
c      rstart     integer   real time for start of algorithm (rtime)
c      rend       integer   real time for end of algorithm (rtime)
c      mclock     integer   system plus user time for process
c      eps        real*8    stopping criterion for iterative algorithm
c      tfill      real*8    (mclock) time for matrix fill
c      titer      real*8    (mclock) time for iterative solution
c      rfill      real*8    (rtime) time for matrix fill
c      riter      real*8    (rtime) time for iterative solution
```

```
c       tfav        real*8     (mclock) average of fill times
c       tiav        real*8     (mclock) average of iterative solution times
c       rfav        real*8     (rtime) average of fill times
c       riav        real*8     (rtime) average of iterative solution times

        program iter
        implicit none
        complex*16 phase,angr,rcsx,rcsy
        complex*16 zmn(1620,1620),v(1620)
        integer start,end,loop,loopcount,nit
        integer rtime,rstart,rend,mclock
        integer n,i,j,nl
        real*8 x(1620),y(1620),z(1620)
        real*8 area(1620),el(1620,4)
        real*8 k,wl,r,pi,angi,a1,zx,rcs
        real*8 eps,tfill,titer,rfill,riter,tfav,tiav,rfav,riav
        parameter (eps=1.0e-3)
        parameter (loopcount=5)

c       *********************************
c       Set Up Variables and Output Files
c       *********************************

c       Moment Method Variables
        pi=acos(-1.)
        wl=1.0
        k=2.0*pi/wl

c       The execution times are stored in a sequential file
        open(5,file='timecg.dat',status='unknown')

c       Display the machine name (for error tracking)
        call system('hostname')

c       Vary the number of sections (nl) and test for each size

c       *********
c       Main Loop
c       *********

        do nl=4,14

c       Clear the average time variables
        tiav=0.0
        tfav=0.0
        riav=0.0
        rfav=0.0

c       Output the current status to the screen (for error tracking)
        write(*,*)
        write(*,*) 'Beginning nl = ',nl

c       Output the current users to the screen
c       This was useful to determine if the machine's real execution
c       times were skewed due to other user processes.
        call system('who')
        write(*,*)

c       *****************
c       Experimental Loop
c       *****************

c       Multiple experimental loops (for averaging execution time)
        do loop=1,loopcount
```

```
c      Output status to screen (progress checking)
       call system('date')

c      Set up the data needed for matrix fill
c      (Rumin Chen Subroutine)
       call dat(x,y,z,area,v,n1,n,el,wl,angi,a1,zx)

c      Time the matrix fill
       rstart=rtime()
       start=mclock()

c      Matrix fill (Rumin Chen Code)

       do 100 i=1,n
       do 100 j=1,n
       if(i.eq.j) then
       zmn(i,j)=.5
       else
       r=sqrt((x(j)-x(i))**2+(y(j)-y(i))**2+(z(j)-z(i))**2)
       phase=(0.,1.)*k*r
       zmn(j,i)=1./4./pi*(1.+phase)
     1  *((y(i)-y(j))*el(i,2)-(z(i)-z(j))*el(i,1))*exp(-phase)
     2  *area(j)/r**3
       endif
100    continue

       end=mclock()
       rend=rtime()

c      Calculate the execution times
       tfill=real(end-start)/100.

c      Correct for "negative" times which occur when execution
c      begins before midnight GMT and ends after midnight GMT
       if (rend.ge.rstart) then
         rfill=real(rend-rstart)
       else
         rfill=real(24*3600-rstart+rend)
       endif

c      Update the average times (sum)
       tfav=tfav+tfill
       rfav=rfav+rfill

c      Time the iteration time
       rstart=rtime()
       start=mclock()

c      Iterative Algorithm
       call cg(zmn,v,n,1620,eps,nit)

       end=mclock()
       rend=rtime()

c      calculate the execution times
       titer=real(end-start)/100.

c      correct for "negative" times which occur when execution
c      begins before midnight GMT and ends after midnight GMT
       if (rend.ge.rstart) then
         riter=real(rend-rstart)
       else
         riter=real(24*3600-rstart+rend)
```

```
      endif

c     Update the average times (sum)
      tiav=tiav+titer
      riav=riav+riter

c     *****  Determine the radar cross section
c     The following code by Rumin Chen determines the radar cross
c     section of the surface.  Its primary use for the iterative testing
c     is verification of the iterative solution.

      angr=0.
      rcsx=0.
      rcsy=0.

      do 200 i=1,n
      phase=(0,1.)*k*((y(i)-2.5)*sin(angr)+z(i)*cos(angr))
      rcsx=0.
      rcsy=rcsy+v(i)*area(i)*exp(phase)
200   continue

c     radar cross section calculation
      rcs=(abs(rcsx))**2+(abs(rcsy))**2
      rcs=4.*pi*rcs*(1./2./wl/wl)**2

      end do

c     calculate the average execution times for the experiments
      tfav=tfav/real(loopcount)
      tiav=tiav/real(loopcount)
      rfav=rfav/real(loopcount)
      riav=riav/real(loopcount)

c     output to file
300   format(2i6,f15.8,4f12.2)
      write(5,300) n,nit,rcs,tfav,tiav,rfav,riav

      end do

      close(5)
      end

c     Data Generation Subroutine
c     Rumin Chen
c
c     This subroutine is used to generate the data information which are
c     used for moment method on a rectangular sinusoidal surface.
c     The patches in this code are square elements.

      subroutine dat(x,y,z,area,v,n1,nodes,el,wl,angi,a1,zx)

      implicit none
      complex*16 phase
      complex*16 v(1620)
      real*8 x(1620),y(1620),z(1620)
      real*8 area(1620),el(1620,4)
      real*8 wl,angi,a1,zx,pi,k,dx,dy,z1,z2,y1,y2,xc,yc
      integer m,n1,nodes,n2,i,j

      m=1

      pi=acos(-1.)
      k=2.*pi/wl
      n2=5*n1

c     Angle of Incidence:  Varied for testing purposes
      angi=0.0

      angi=angi*pi/180.
      dx=1./n1
      dy=1./n1
      a1=0.

c     Sinusoidal Surface Height:  Varied for testing purposes
      zx=0.1

      do 10 i=1,n1
      do 10 j=1,n2
      x(m)=(i-1.)*dx+dx/2.
      y(m)=(j-1.)*dy+dy/2.
      z(m)=zx*sin(pi*y(m)/5.*10.+pi/2.)
      y1=(j-1.)*dy
      y2=j*dy
      z1=zx*sin(pi*y1/5.*10.+pi/2.)
      z2=zx*sin(pi*y2/5.*10.+pi/2.)

c     el is used to store the unit vector of l
c     1 - dy
c     2-- dz
c     3-- length of the dl
      el(m,3)=sqrt((z2-z1)**2+dy*dy)
      el(m,1)=dy/el(m,3)
      el(m,2)=(z2-z1)/el(m,3)
      area(m)=el(m,3)*dy
      a1=a1+area(m)
      phase=(0,1.)*k*((y(m)-2.5)*sin(angi)+z(m)*cos(angi))
      xc=pi*x(m)
      yc=pi*y(m)/5.
      v(m)=-(cos(angi)*el(m,1)-sin(angi)*el(m,2))
     1       *exp(phase)*(1.+cos(pi*(x(m)-0.5)/0.5))*
     2       (1.+cos(pi*(y(m)-2.5)/2.5))/4.
      m=m+1
10    continue
      nodes=n1*n2

      return
      end
```

APPENDIX I

SYSTEM CLOCK POLLING ROUTINES

Rtime Source Code

```
#include <time.h>
#include <stddef.h>
int rtime_()
{
  struct tm *gm;
  time_t t;
  int s,m,h;

  t=time(NULL);
  gm=gmtime(&t);
  s=(*gm).tm_sec;
  m=(*gm).tm_min;
  h=(*gm).tm_hour;
  return(s+m*60+h*3600);
}
```

Mclock Source Code

```
#include <time.h>
clock_t mclock_(void)
{
  return(clock());
}
```

VITA 2

James Michael Sturm

Candidate for the Degree of

Master of Science

Thesis: ITERATIVE METHODS FOR SOLVING LARGE LINEAR SYSTEMS IN THE MOMENT METHOD ANALYSIS OF ELECTROMAGNETIC SCATTERING

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Austin, Texas, February 15, 1969, the son of Dr. Gene Paul and Mrs. Phyllis Ann Sturm

Education: Graduated from Bartlesville High School, Bartlesville, Oklahoma, in May 1987; received Bachelor of Science Degree in Electrical Engineering with a Computer Engineering Option from Oklahoma State University in December, 1991; completed the requirements for the Master of Science degree at Oklahoma State University in December, 1993.

Professional Experience: Research Assistant, Department of Electrical Engineering, Oklahoma State University, January 1992 to December 1993. IIT Research Institute Undergraduate Research Fellow at National Institute for Petroleum and Energy Research in Bartlesville, Oklahoma, summers of 1988, 1989, 1991.