SCI INSTITUTE

TECHNICAL REPORT

Large Scale Biomedical Modeling and Simulation From Concept to Results

C.S. Henriquez, C.R. Johnson, K.A. Henneberg, L.J. Leon, and A.E. Pollard

UUSCI-1995-001

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

1995

Abstract:

The complexity of biomedical modeling problems advances concurrently with the state of the art in high performance computing. Many problems of interest are moving beyond the capability of the single workstation environment and investigators are finding the need to use supercomputers with more memory and greater performance to run their simulations. The tasks of large and small scale modeling are generally the same. First, the appropriate numerical method for the given problem must be determined. This choice involves a number of factors including which computing environment will be used. Once a method has been chosen and the solution domain has been discretized, the method must be designed, coded and optimized to fully utilize the power of vector and/or parallel processing. The algorithm must include features to manage large amounts of numerical output and extract one, two, and three-dimensional variables for postprocessing and meaningful visualization. This chapter examines large scale biomedical computing in a supercomputer environment. The main focus is on problems in the areas of model development, implementation and optimization, data management and visualization that are likely to differ from those encountered in a single-processor workstation environment.

THE UNIVERSITY OF UTAH

Large Scale Biomedical Modeling and Simulation: From Concept to Results

by

C.S. Henriquez[1], C.R. Johnson[2], K.A. Henneberg[3], L.J. Leon[3], and A.E. Pollard[4]

[1] Department of Biomedical Engineering, Duke University, Durham, N.C. 27706, [2] Departments of Computer Science, Bioengineering, and Mathematics, University of Utah, Salt Lake City, UT 84112, [3] Institute of Biomedical Engineering, University of Montreal and Ecole Polytechnique, Montreal, Quebec, Canada, H3C 3J7, [4] Department of Biomedical Engineering, Tulane University, New Orleans, LA, and The Cardiovascular Research and Training Institute, University of Utah, Salt Lake City, UT 84112.

Abstract

The complexity of biomedical modeling problems advances concurrently with the state of the art in high performance computing. Many problems of interest are moving beyond the capability of the single workstation environment and investigators are finding the need to use supercomputers with more memory and greater performance to run their simulations. The tasks of large and small scale modeling are generally the same. First, the appropriate numerical method for the given problem must be determined. This choice involves a number of factors including which computing environment will be used. Once a method has been chosen and the solution domain has been discretized, the method must be designed, coded and optimized to fully utilize the power of vector and/or parallel processing. The algorithm must include features to manage large amounts of numerical output and extract one, two, and three-dimensional variables for postprocessing and meaningful visualization. This chapter examines large scale biomedical computing in a supercomputer environment. The main focus is on problems in the areas of model development, implementation and optimization, data management and visualization that are likely to differ from those encountered in a single-processor workstation environment.

1 Introduction

Academic scientific computing has changed dramatically over the past decade. In the early eighties, most computing was performed on large centralized mainframes, maintained by the university or a group of universities. These systems were plagued with awkward operating systems, poor or absent facilities for displaying data and restricted remote access. As personal computers became

more popular, there was a drive to provide a desktop environment for scientific computing. The user benefited from the reduction of job turn around time and an increased ability to visualize and manipulate data. By the late eighties, workstations were on the market with computing power greater than the typical mainframe used at the beginning of the decade. However, the decentralization of computing resources put a greater burden on the users to manage and maintain the hardware and software. Without University cost-sharing, individual labs found it difficult to improve their resources to meet their rapidly changing needs.

Over the last few years, there has been a significant attempt to standardize operating systems, improving the portability of algorithms and tools from machine to machine, and to provide a fast communication network to connect the user to machines and other users across the globe. These two factors have lured many users back to a modernized mainframe environment comprised of supercomputers with large memories and special architectures to excel in floating point operations. With improved networking, the workstation can serve as both a local computing resource for algorithm development and as a terminal for accessing remote supercomputers running memory and operation intensive simulations. While the mainframe has enjoyed a comeback of sorts, some believe it is short lived. These supercomputers are expensive to purchase and operate and thus general access is limited. The supercomputer environment of the future for academic research is expected to exploit the growing numbers of faster and relatively inexpensive workstations within the university. Software tools like Linda and PVM are being developed that permit networked workstations to operate as an effective parallel computer. Although the communication between processors is slower than on dedicated parallel machines, the performance of a modest number of workstations is expected to approach that of the largest mainframe supercomputer.

In a recent review article, Board [7] notes that as a group, biomedical researchers have been slower to take advantage of supercomputers than scientists in other scientific and engineering disciplines. However, it has been increasingly evident that the size and scope of biomedical models are moving beyond the capabilities of current single workstation technologies and will demand that biomedical modelers take greater advantage of vector or parallel processing to reduce the computation time of their simulations. Unfortunately, even with greater standardization, the transition from a workstation to a supercomputer environment is not always easy or straightforward. Supercomputers, from the expensive mainframe to the cluster of networked workstations, have not only expanded what we can model but also changed how we must model. Some of the increased performance of these machines is obtained by using hardware like vector processors that prefer certain program constructions. Hence, existing algorithms usually must be rewritten or restructured to take maximal advantage of the special architecture. In a parallel or distributed environment, algorithms must be constructed to distribute the load and minimize wait times between individual processor computations. Large scale applications usually involve a vast amount of data transfer and manipulation. Researchers often find their local resources are inadequate to store or visualize the large data sets generated by large scale simulations and must seek alternatives. The goal of this chapter is to aid the transition from the single workstation to a supercomputer environment for

large scale simulation and modeling. We will discuss some of the expected problems and provide some possible solutions in the areas of model development, implementation and optimization, data management and visualization.

2 Approximation Methods

With the increasing availability of high performance computers, it is possible to more accurately represent anatomical structures in physiological models. Figure 1 illustrates a model for the computation of electrocardiographic potentials on the body surface (the forward problem) and on the epicardium (the inverse problem). Although this particular model was designed for solving bioelectric field problems, it is apparent that models of similar complexity are often required in problems arising in for example biomechanics [3], [8], [18], [23], [37], [41], [42], [44], [49], [58].

Due to the complexity of the geometry and the numerous inhomogeneities, this type of model is only tractable if numerical approximation methods such as the Finite Difference (FDM), the Finite Element (FEM), and the Boundary Element (BEM) methods are used. Consequently, we have chosen to limit our discussion in this chapter to the large scale implementation of these approximation schemes.

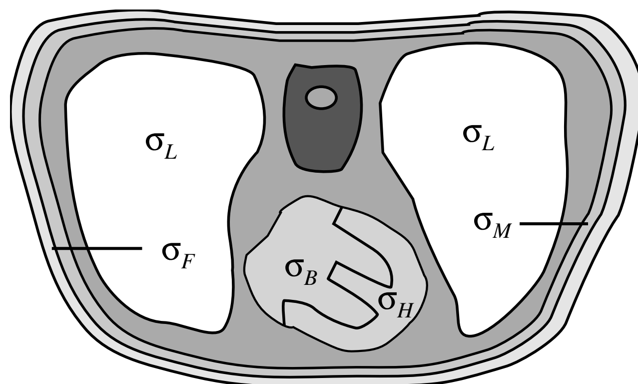


Figure 1: Cross section of human torso with heart, lungs, skeletal muscle and fat outlined. Each material has unique electrical conduction properties.

For the purpose of identifying modeling issues common to many simulation studies in biomedical engineering, we consider a bioelectric boundary value problem easily associated with the torso model illustrated in Figure 1. Assuming a model for the bioelectric sources in the heart is known, we will consider the problem of evaluating the potential distribution in the torso and on the body surface. Regions in the thorax containing active sources are governed by Poisson's equation:

$$\nabla \cdot (\sigma \nabla \Phi) = -I_v \quad \text{in } \Omega_H \tag{1}$$

where Φ denotes the potential distribution and I_v denotes the volume source within the heart. σ is the conductivity tensor and Ω_H represents the domain of the heart. In regions with no sources

(e.g. lungs, skeletal muscle and fat) Poisson's equation simplifies to Laplace's equation:

$$\nabla \cdot (\sigma \nabla \Phi) = 0 \quad \text{in } \Omega_i \tag{2}$$

where Ω_i denotes the source free regions. The distinction between regions with and without sources is natural when a solution approach based on integral equations (e.g. the boundary element method) is chosen because Green's second identity can be applied to each homogeneous region. The solutions in the subregions are constrained by the conditions of continuity in potential and current at the interfaces between inhomogeneities. If a volume discretization method such as the finite element method is used, the subdivision into homogeneous subdomains is not required because the finite element method can handle inhomogeneous material poperties very elegantly. In the latter approach it is therefore more tractable to model the entire thorax as a single domain governed by Poisson's equation and include the inhomogeneities in the conductivity tensor.

The boundary value problem above only serves to illustrate that the numerical method chosen for the problem must be able to deal with complex shapes, anisotropic material properties, domain decomposition, and varying degrees of inhomogeneity (i.e. large homogeneous regions bounded by strongly heterogeneous regions with continuously changing anisotropic conductivity tensors). These requirements stem from the complexity of the body and are typical in electrical as well as other types of biomedical problems. Many researchers have found that commercial FE and BE packages are inadequate for solving biomedical problems and are often forced to write their own algorithms. These algorithms usually take many months to write and debug and are often platform specific. Although each of the numerical methods has its own advantages and disadvantages for a particular problem, user experience and the availability of existing algorithms may greatly influence the choice of a particular method. In this section we discuss each of these methods and examine those features within each method that will likely require special attention when implemented on a supercomputer. Ultimately, the method (or methods) chosen must be able to solve a set of partial differential equations in a heterogeneous domain; however, in the interest of keeping this chapter reasonably short and general, we will only consider Poisson's equation (1) in the subsequent discussion of numerical methods.

2.1 The Galerkin Method

The sample problem in (1) can be solved using any of these approximation schemes. All three techniques (FEM, FDM, and BEM) can be derived by the Galerkin method for the discretization of the spatial portion of the time-dependent parabolic problem. To express our problem in a Galerkin form, we begin by rewriting (1), as:

$$A\Phi = -I_v \tag{3}$$

where A is the differential operator, $A = \nabla \cdot (\sigma \nabla)$. An equivalent statement of (3) is, find Φ such that $(A\Phi + I_v, \overline{\Phi}) = 0$. Here, $\overline{\Phi}$ is an arbitrary *test function*, which can be thought of physically as a

virtual potential field, and the notation, $(\phi_1, \phi_2) \equiv \int_\Omega \phi_1 \phi_2 \, d\Omega$, denotes the inner product in $L_2(\Omega)$, i.e. the space of square integrable functions. Applying Green's theorem, we can equivalently write,

$$(\sigma \nabla \Phi, \nabla \overline{\Phi}) - \langle \frac{\partial \Phi}{\partial n}, \overline{\Phi} \rangle = -(I_v, \overline{\Phi}) \tag{4}$$

where the notation $\langle \phi_1, \phi_2 \rangle \equiv \int_S \phi_1 \phi_2 \, dS$, denotes the inner product on the boundary S. When the Dirichlet, $\Phi = \Phi_0$ and Neumann, $\sigma \nabla \Phi \cdot \mathbf{n} = 0$ boundary conditions are specified on S, we obtain the *weak form* of (1):

$$(\sigma \nabla \Phi, \nabla \overline{\Phi}) = -(I_v, \overline{\Phi}) \tag{5}$$

It is understood that this equation must hold for all test functions, $\overline{\Phi}$, which must vanish at the boundaries where $\Phi = \Phi_0$. The Galerkin approximation ϕ to the weak form solution Φ in (5) can be expressed as:

$$\phi(\mathbf{x}) = \sum_{i=0}^{N} \phi_i \psi_i(\mathbf{x}) \tag{6}$$

The trial functions $\psi_i, i = 0, 1, \ldots, N$ form a basis for an N+1 dimensional space $\mathcal{S}$. We define the *Galerkin approximation* to be that element $\phi \in \mathcal{S}$ which satisfies:

$$(\sigma \nabla \phi, \nabla \psi_j) = -(I_v, \psi_j) \qquad (\forall \psi_j \in \mathcal{S}) \tag{7}$$

Since our differential operator A is positive definite and self adjoint (i.e., $(A\Phi, \Phi) \geq \alpha(\Phi, \Phi) > 0$ for some non-zero positive constant α and $(A\Phi, \overline{\Phi}) = (\Phi, A\overline{\Phi})$, respectively), then we can define a space E with an inner product defined as $(\Phi, \overline{\Phi})_E = (A\Phi, \overline{\Phi}) \equiv a(\Phi, \overline{\Phi})$ and norm (the so-called energy norm) equal to:

$$\|\Phi\|_E = \{\int_\Omega (\nabla \Phi)^2 d\,\Omega\}^{\frac{1}{2}} = (\Phi, \Phi)_E^{\frac{1}{2}} \tag{8}$$

The solution Φ of (3) satisfies:

$$(A\Phi, \psi_i) = -(I_v, \psi_i) \qquad (\forall \psi_i \in \mathcal{S}) \tag{9}$$

and the approximate Galerkin solution obtained by solving (7) satisfies:

$$(A\phi, \psi_i) = -(I_v, \psi_i) \qquad (\forall \psi_i \in \mathcal{S}) \tag{10}$$

Subtracting (9) from (10) yields:

$$(A(\phi - \Phi), \psi_i) = (\phi - \Phi, \psi_i)_E = 0 \qquad (\forall \psi_i \in \mathcal{S}) \tag{11}$$

The difference $\phi - \Phi$ denotes the error between the solution in the infinite dimensional space V and the $N+1$ dimensional space $\mathcal{S}$. Equation (11) states that the error is orthogonal to all basis functions spanning the space of possible Galerkin solutions. Consequently, the error is orthogonal to all elements in $\mathcal{S}$ and must therefore be the minimum error. Thus the Galerkin approximation is an orthogonal projection of the true solution Φ onto the given finite dimensional space of possible approximate solutions. This leads to the statement that the Galerkin approximation is the best

approximation in the energy space E. Since the operator is positive definite the approximate solution is unique. Assume for a moment there are two solutions, ϕ_1 and ϕ_2, satisfying:

$$(A\phi_1, \psi_i) = -(I_v, \psi_i) \ ; \quad (A\phi_2, \psi_i) = -(I_v, \psi_i) \quad (\forall \psi_i \in \mathcal{S}) \tag{12}$$

respectively, then the difference yields:

$$(A(\phi_1 - \phi_2), \psi_i) = (A\Delta\phi, \psi_i) = 0 \quad (\forall \psi_i \in \mathcal{S}) \tag{13}$$

The function $\Delta\phi = \phi_1 - \phi_2$ arising from subtracting one member from another member in $\mathcal{S}$ also belongs in $\mathcal{S}$, hence $\Delta\phi$ can be expressed by the set of A orthogonal basis functions spanning $\mathcal{S}$:

$$\Delta\phi = \sum_{j=0}^{N} \Delta\phi_j \psi_j \tag{14}$$

Thus, inserting (14) in (13) yields:

$$\sum_{j=0}^{N} \Delta\phi_j (A\psi_j, \psi_i) = 0 \quad (\forall \psi_i \in \mathcal{S}) \tag{15}$$

When $i \neq j$, the terms vanish due to the basis functions being orthogonal with respect to A. Since A is positive definite:

$$(A\psi_i, \psi_i) > 0 \quad i = 0, \ldots, N \tag{16}$$

Thus, $\Delta\phi_i = 0$, $i = 0, \ldots, N$, and by virtue of (14) $\Delta\phi = 0$, such that $\phi_1 = \phi_2$. The identity contradicts the assumption of two distinct Galerkin solutions, thus the proof of uniqueness is complete.

2.2 The Finite Difference Method

Perhaps the most traditional way of solving (1) using the finite difference approach is to discretize the solution domain Ω using a grid of quadrilaterals (for 2D) or cubes (for 3D). The coordinates of a typical grid point are $x = lh, y = mh, z = nh$ ($l, m, n = integers$), and the value of $\Phi(x, y, z)$ at a grid point is denoted by $\Phi_{l,m,n}$. Taylor's theorem is then used to provide the difference equations. For example:

$$\Phi_{l+1,m,n} = (\Phi + h\frac{\partial \Phi}{\partial x} + \frac{1}{2}h^2\frac{\partial^2 \Phi}{\partial x^2} + \frac{1}{6}h^3\frac{\partial^3 \Phi}{\partial x^3} + \ldots)_{l,m,n} \tag{17}$$

with a similar equations for $\Phi_{l-1,m,n}, \Phi_{l,m+1,n}, \Phi_{l,m-1,n,\ldots}$. The finite difference representation of (1) is:

$$\frac{\Phi_{l+1,m,n} - 2\Phi_{l,m,n} + \Phi_{l-1,m,n}}{h^2} + \frac{\Phi_{l,m+1,n} - 2\Phi_{l,m,n} + \Phi_{l,m-1,n}}{h^2} + \frac{\Phi_{l,m,n+1} - 2\Phi_{l,m,n} + \Phi_{l,m,n-1}}{h^2} = -I_{l,m,n}(v) \tag{18}$$

or equivalently,

$$\Phi_{l+1,m,n} + \Phi_{l-1,m,n} + \Phi_{l,m+1,n} + \Phi_{l,m-1,n} + \Phi_{l,m,n+1} + \Phi_{l,m,n-1} - 6\Phi_{l,m,n} = -h^2 I_{l,m,n}(v) \tag{19}$$

If we define the vector $\mathbf{\Phi}$ to be $[\Phi_{1,1,1} \ldots \Phi_{1,1,N-1}; \ldots \Phi_{1,N-1,1} \ldots \Phi_{N-1,N-1,N-1}]^T$ to designate the $(N-1)^3$ unknown grid values, and pull out all the known information from (19), we can reformulate (1) by its finite difference approximation in the form of the matrix equation, $A\mathbf{\Phi} = \mathbf{b}$, where $\mathbf{b}$ is a vector which contains the sources and modifications due to the Dirichlet boundary condition.

Unlike the traditional Taylor's series expansion method, the Galerkin approach uses basis functions, such as linear piecewise polynomials, to approximate the true solution. For example, the Galerkin approximation to the sample problem, (1), would require evaluating (7) for the specific grid formation and specific choice of basis function:

$$\int_\Omega (\sigma_x \frac{\partial \phi}{\partial x}\frac{\partial \psi_i}{\partial x} + \sigma_y \frac{\partial \phi}{\partial y}\frac{\partial \psi_i}{\partial y} + \sigma_z \frac{\partial \phi}{\partial z}\frac{\partial \psi_i}{\partial z}) d\Omega = -\int_\Omega I_v \psi_i d\Omega \tag{20}$$

Difference quotients are then used to approximate the derivatives in (20). We note that if linear basis functions are used in (20), one obtains a formulation which corresponds exactly with the standard finite difference operator. Regardless of the difference scheme or order of basis function, the approximation results in a system of linear equations of the form, $A\mathbf{\Phi} = \mathbf{b}$, subject to the appropriate boundary conditions.

2.3 The Finite Element Method

As we have seen above, the classical numerical treatment of partial differential equations is the finite difference method, where the solution domain is approximated by a grid of uniformly spaced nodes. At each node, the governing differential equation is approximated by an algebraic expression which references adjacent grid points. A system of equations is obtained by evaluating the previous algebraic approximations for each node in the domain. Finally, the system is solved for each value of the dependent variable at each node. In the finite element method, the solution domain is discretized into a number of non-uniform finite elements that are connected via nodes. The change of the dependent variable with regard to location is approximated within each element by an interpolation function. The interpolation function is defined relative to the values of the variable at the nodes associated with each element. The original boundary value problem is then replaced with an equivalent integral formulation (such as (7)). The interpolation functions are then substituted into the integral equation, integrated and combined with the results from all other elements in the solution domain. The results of this procedure can be reformulated into a matrix equation of the form, $A\mathbf{\Phi} = \mathbf{b}$, which is subsequently solved for the unknown variable [2], [24].

The formulation of the finite element approximation starts with the Galerkin approximation, $(\sigma\nabla\Phi, \nabla\overline{\Phi}) = -(I_v, \overline{\Phi})$, where $\overline{\Phi}$ is our test function. We now use the finite element method to turn the continuous problems into a discrete formulation. First we discretize the solution domain, $\Omega = \cup_{e=1}^{E}\Omega_e$ and define a finite dimensional subspace, $V_h \subset V = \{\overline{\Phi} : \overline{\Phi} \text{ is continuous on } \Omega , \nabla\overline{\Phi} \text{ is piecewise continuous on } \Omega\}$. One usually defines parameters of the function $\overline{\Phi} \in V_h$ at node points, $\alpha_i = \overline{\Phi}(x_i)$, $i = 0, 1, \ldots, N$. We now define the basis functions, $\psi_i \in V_h$ as linear continuous functions, where each basis functionis zero everywhere except on its element of support.

In the supporting element the basis function takes the value 1 at one node points and the value 0 at all other node points. We can then represent the function $\overline{\Phi} \in V_h$ as:

$$\overline{\Phi}(x) = \sum_{i=0}^{N} \alpha_i \psi_i(x) \tag{21}$$

such that each $\overline{\Phi} \in V_h$ can be written in a unique way as a linear combination of the basis functions $\psi_i \in V_h$. Now the finite element approximation of the original boundary value problem can be stated as:

$$\text{Find } \Phi_h \in V_h \text{ such that } (\sigma \nabla \Phi_h, \nabla \overline{\Phi}) = -(I_v, \overline{\Phi}) \tag{22}$$

Furthermore, if $\Phi_h \in V_h$ satisfies (22), then we have $(\sigma \nabla \Phi_h, \nabla \psi_i) = -(I_v, \psi_i)$ [33]. Finally, since Φ_h itself can be expressed as the linear combination:

$$\Phi_h = \sum_{i=0}^{N} \xi_i \psi_i(x) \qquad \xi_i = \Phi_h(x_i) \tag{23}$$

we can then write (22) as:

$$\sum_{i=0}^{N} \xi_i (\sigma_{ij} \nabla \psi_i, \nabla \psi_j) = -(I_v, \psi_j) \qquad j = 0, \ldots, N \tag{24}$$

subject to the Dirichlet boundary condition. This is the finite element approximation of (1) which can equivalently be expressed as a system of N equations with N unknowns $\xi_i, \ldots, \xi_N$ (the electrostatic potentials, for example). In matrix form, the above system can be written as $A\xi = b$, where $A = (a_{ij})$ is called the global stiffness matrix and has elements $(a_{ij}) = (\sigma_{ij} \nabla \psi_i, \nabla \psi_j)$, while $b_i = -(I_v, \psi_i)$ and is usually termed the load vector.

For volume conductor problems, A contains all of the geometry and conductivity information of the model. The matrix A is symmetric and positive definite, thus it is nonsingular and has a unique solution. Because the basis function differs from zero for only a few intervals, A is sparse - (only a few of its entries are nonzero). These features of the matrix A, resulting from a FEM formulation, will significantly affect the strategy for solving the system of equations.

2.4 The Boundary Element Method

For differential operators the system response at any given point to sources and boundary conditions only depends on the response at neighboring points. In the FDM and FEM approximate differential operators are defined on subregions (volume elements) in the domain, hence direct mutual influence (connectivity) only exists between neighboring elements and the coefficient matrices generated by these methods have relatively few non-zero coefficients in any given matrix row. As is clearly demonstrated by Maxwell's laws [27], equations in differential forms can often be replaced by equations in integral forms, e.g. the potential distribution in a domain is uniquely defined by the volume sources and the potential and current density on the boundary. The boundary element method uses this fact by transforming the differential operator defined in the domain to integral

operators defined on the boundary. In the boundary element method [10],[28] only the boundary is discretized, hence the mesh generation is considerably simpler for this method than for the volume methods. The BEM approximates the potential and the normal derivative of the potential by series expansions in basis functions defined on the surface elements. Boundary solutions are obtained directly by solving the set of linear equations; however, potentials and gradients in the domain can only be evaluated after the boundary solution has been obtained.

For the boundary element formulation a weak form of (3) is obtained by choosing the anisotropic Green's function as the test function $\overline{\Phi} = 1/\sigma_e R$, where R is the distance function between the field point $\vec{p}$ and the source point $\vec{q}$:

$$R = \sqrt{\frac{(x_q - x_p)^2}{\sigma_x} + \frac{(y_q - y_p)^2}{\sigma_y} + \frac{(z_q - z_p)^2}{\sigma_z}} \tag{25}$$

and:

$$\sigma_e = \sqrt{\sigma_x \sigma_y \sigma_z} \tag{26}$$

Integration of $A\Phi = -I_v$ by parts twice yields Green's third identity:

$$\frac{1}{2}\Phi + H\Phi - G\frac{\partial \Phi}{\partial n} - \Phi_a = 0 \tag{27}$$

where

$$\Phi_a = (I_v, \overline{\Phi}) \tag{28}$$

and G and H are the single and double layer operators:

$$G\frac{\partial \Phi}{\partial n} = \frac{1}{4\pi}\int_S \frac{\partial \Phi}{\partial n}\overline{\Phi} dS \tag{29}$$

and:

$$H\Phi = \frac{1}{4\pi}\int_S \Phi \frac{\partial \overline{\Phi}}{\partial n} dS \tag{30}$$

The Galerkin approximations to the weak form solutions Φ and $\frac{\partial \Phi}{\partial n}$ in (27) are expressed as:

$$\phi(\mathbf{s}) = \sum_{i=0}^{N} \phi_i \psi_i(\mathbf{s}) \; ; \qquad \frac{\partial \phi}{\partial n}(\mathbf{s}) = \sum_{i=0}^{N} \left(\frac{\partial \phi}{\partial n}\right)_i \psi_i(\mathbf{s}) \tag{31}$$

where $\mathbf{s}$ denotes a parameterization of the surface. The functions $\phi(\mathbf{s})$ and $\frac{\partial \phi}{\partial n}(\mathbf{s})$ are members of the finite dimensional space V_h and their coefficients ϕ_i and $\left(\frac{\partial \phi}{\partial n}\right)_i$ are determined by the set of linear equations:

$$\sum_{j=0}^{N} \langle \frac{1}{2}\delta_{i,j} + H\psi_j, \psi_i \rangle \phi_j - \sum_{j=0}^{N} \langle G\psi_j, \psi_i \rangle \left(\frac{\partial \phi}{\partial n}\right)_j = \langle \Phi_a, \psi_i \rangle \; ; \quad i = 0, \ldots, N \tag{32}$$

where $\delta_{i,j}$ is the Kronecker delta function. The operator G is symmetric and positive definite, hence if the potential is known on the boundary, (32) yields a symmetric coefficient matrix. The operator H is non-symmetric, hence for the Neumann problem and problems with mixed boundary conditions, the coefficient matrix is non-symmetric. In general the Galerkin formulation presented here

does not satisfy (11) and only for the Dirichlet problem does the method classify as an orthogonal projection method [56].

Equation (27) defines the residual:

$$\mathcal{R} = \frac{1}{2}\Phi + H\Phi - G\frac{\partial \Phi}{\partial n} - \Phi_a \tag{33}$$

The true solutions Φ and $\frac{\partial \Phi}{\partial n}$ satisfy (27) exactly, hence the residual vanishes everywhere. For the Galerkin approximation (32) the residual only vanishes on each element in the average sense:

$$(\mathcal{R}, \psi_i) = 0 \; ; \quad i = 0, \ldots, N \tag{34}$$

thus ψ_i is acting as a weighting function in the Galerkin Weighted Residual formulation in (34).

The matrix coefficients in the Galerkin BEM requires the evaluation of double surface integrals and the method is therefore more demanding on computing resources than the collocation method which only includes single surface integrals. In the latter method the weighting function ψ_i is replaced by the Dirac delta function $\delta(|\vec{p} - \vec{p}_i|)$ and the inner product $\langle \phi_1, \phi_2 \rangle$ is replaced by the bilinear form $(\phi_1, \phi_2)_B = \int_S \phi_1 \phi_2 dS$. The latter is required since the Dirac delta function is not square integrable and consequently does not belong in the space V_h [56]. When the Dirac delta function is used as the weighting function, the residual only vanishes at node points:

$$(\mathcal{R}, \delta(|\vec{p} - \vec{p}_i|)) = 0 \; ; \quad i = 0, \ldots, N \tag{35}$$

hence in general this method (the Collocation Weighted Residual method) is less accurate than the Galerkin method. In the Collocation Weighted Residual method the basis functions can be identical to the basis functions of the Galerkin method; however, the collocation weighting function is the Dirac delta function, whereas the Galerkin weighting function must be identical to the basis function. The collocation formulation equivalent to the Galerkin formulation in (32) is obtained:

$$\begin{aligned} \sum_{j=0}^{N} \left(\frac{1}{2}\delta_{i,j} + H\psi_j, \delta(|\vec{p} - \vec{p}_i|) \right)_B \phi_j \quad & - \quad \sum_{j=0}^{N} (G\psi_j, \delta(|\vec{p} - \vec{p}_i|))_B \left(\frac{\partial \phi}{\partial n} \right)_j \\ & = \quad (\Phi_a, \delta(|\vec{p} - \vec{p}_i|))_B \; ; \quad i = 0, \ldots, N \end{aligned} \tag{36}$$

hence carrying out the outer integrations yields:

$$\sum_{j=0}^{N} \left(\frac{1}{2}\delta_{i,j} + H\psi_j \right)_i \phi_j - \sum_{j=0}^{N} (G\psi_j)_i \left(\frac{\partial \phi}{\partial n} \right)_j = (\Phi_a)_i \; ; \quad i = 0, \ldots, N \tag{37}$$

where $(\;)_i$ denotes the ith field point. The collocation method in (37) is a non-orthogonal projection method [56]. Since the Galerkin method requires the evaluation of double surface integrals it is only used if the increased accuracy is essential. We will therefore only discuss the implementation of the collocation method.

3 Comparison of Methods

As demonstrated above, the FDM, FEM, and BEM can all be used to approximate the boundary value problems which arise in biomedical research problems. The choice depends on the nature of the problem. The FEM and FDM methods are similar in that the entire solution domain must be discretized as opposed to the BEM where only the bounding surfaces is discretized. For regular domains, the FDM is generally the easiest method to code and implement. The FDM usually requires special modifications to define irregular boundaries, abrupt changes in material properties, and complex boundary conditions. While typically more difficult to implement, the BEM and FEM are preferred for problems with irregular, inhomogeneous domains and mixed boundary conditions. The FEM is superior to BEM for representing nonlinearity and true anisotropy, while the BEM is superior to FEM for problems where only the boundary solution is of interest or where solutions are wanted in a set of highly irregularly spaced points in the domain. The computational mesh is simpler for the BEM than for the FEM, hence less book-keeping is required in a BEM program than a FEM program. For this reason BE programs are often considered easier to develop than FE programs; however, the difficulties associated with singular integrals in the BEM are often highly underestimated. In general FEM is preferred for problems where the domain is highly heterogeneous whereas the BEM is preferred for highly homogeneous domains.

4 Mesh Generation

After deciding upon the particular approximation method to use (and what kind of element and number of degrees of freedom associated with each element), we need to construct a mesh of the solution domain, i.e., subdividing our model geometry into polygons. For the sake of simplicity, we will assume that we will use triangles for two-dimensional meshes (corresponding to three-dimensional BE problems or two-dimensional FD and FE problems) and tetrahedrons for three-dimensional FD and FE domains (see Figure 2). Due to the complex geometries often associated with bioelectric field problems, mesh construction and mesh adaption can turn out to be the most time consuming aspect of the modeling process.

In general, there are two basic approaches to mesh construction. The first is the “divide and conquer” strategy. Simply put, given a set of points which define the bounding surface(s), the volume is repeatedly divided into smaller regions until a satisfactory discretization level has been achieved (we will discuss stopping criteria later). Usually, the domain is broken up into eight-node cubic elements, which are then subdivided into five (minimally) or six tetrahedral elements for three-dimensional problems and the corresponding quadrilaterals and triangles for two-dimensional problems. This methodology has the advantage of being fairly easy to program (also, commercial mesh generators exist for the divide and conquer method). Its main disadvantage has to do with its inability to control the elements which overlap interior boundaries. A single element may span two different conductive regions, for example, when part of an element represents muscle tissue (which could be anisotropic) and the other part of the element falls into a region representing fat

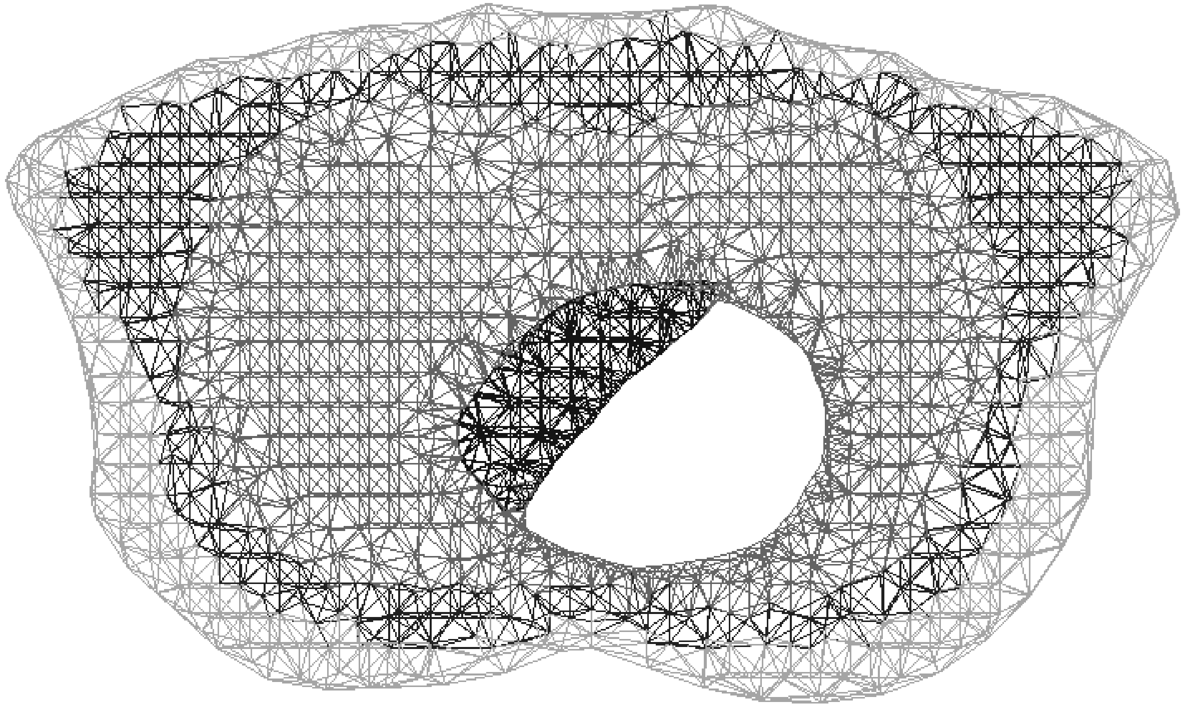


Figure 2: Finite element mesh for torso model. The gray scale coloring of tetrahedrons denotes heterogeneous regions with different conductivities.

tissue. It then becomes very difficult to assign unique conductivity parameters and at the same time accurately represent the geometry.

A second method of mesh generation is based upon the Delauney strategy. Given a two- or three-dimensional set of points which define the boundaries and interior regions, tessellate the point cloud into an optimal mesh of triangles or tetrahedra. The advantages and disadvantages tend to be exactly contrary to those arising from the divide and conquer strategy. The primary advantage is that the mesh can be developed to fit any predefined geometry, including subsurfaces, by starting with points which define all the necessary surfaces and then adding additional interior points to minimize the aspect ratio. For triangles, the aspect ratio is defined to be the ratio of the maximal horizontal length to the maximal vertical length of the element or by the ratio of the diameter of a circumscribed circle to the maximal distance between vertices, while for tetrahedra, the aspect ratio can be defined as $4\sqrt{\frac{3}{2}}\frac{\rho_k}{h_k}$ where ρ_k denotes the diameter of the sphere circumscribed about the tetrahedron and h_k is the maximum distance between two vertices. These formulations yield a value of 1 for an "isosceles" tetrahedron (triangle) and a value of 0 for a degenerate (flat) element [5]. The closer to the value of 1, the better. The Delauney criterion is basically a method for minimizing the occurrence of obtuse angles in the mesh, yielding elements which have aspect ratios as close to 1 as possible, given the available point set. While the ideas of Delauney triangulation are straightforward, the programming is nontrivial and is the primary drawback to this method. At this point in time, we are unaware of any commercially available implementations of the De-

launey tessellation method for general three- dimensional point clouds. However, there do exist several public domain, two-dimensional versions from netlib (e-mail: netlib@research.att.com with message: send index). For more information on mesh generation, see Bowyer [9] and Hoole [24].

Another drawback of the Delaunay method of mesh generation is that it produces elements in convex regions that lie outside the bounding surface. For example, if one is triangulating a two-dimensional kidney shaped object, the Delaunay method will construct triangles outside the bounding contour in the convex C-shaped region. One way to rid the mesh of unwanted triangles (or tetrahedrons) is to supplement the mesh generator with the following algorithm, based on the Gauss-Bonnet (GB) theorem of topology [55]. If one calculates the angles subtended by the points bounding a two-dimensional area about a point P (adding consecutive angles with some sign convention), the sum will equal 2π if P is inside the enclosed contour, π if P is on the boundary, and 0 if P lies outside. Thus it is a simple matter to test a questionable element by checking to see if its centroid is inside or outside the contour. In the analogous three-dimensional version, one calculates the solid angles about the centroids of each tetrahedron, which will sum to 4π for interior points and 0 for exterior points. We can exploit this idea further to complete one last bit of preprocessing before solving the system [32],[43]. We need to assign conductivity values to the various tissue regions. In the FE formulation, we need to assign a conductivity tensor to each element of the model. This is usually done by using a table look-up scheme in which each element is given a number according to the type of tissue it contains. During processing, the corresponding value from a table of conductivity tensors is used to calculate the global stiffness matrix, making it a trivial matter to change the various conductivities without changing the geometry or any other parameters of the model. One can use the GB algorithm to automatically ascertain to which conductivity group an element belongs by defining the boundaries of the various subsurfaces and determining whether the centroids of the elements are inside or outside the region in question, assigning each to an appropriate conductivity group number. Once all the conductivity tensors are assigned to the groups, the preprocessing is complete and we are ready to compute solutions.

5 Solution Techniques

Each of the FDM, FEM, and BEM involve a number of similar computational subtasks: discretizing the domain into a mesh, computing the coefficients of the matrix A, assembling the matrix A and solving the system of equations. However, because of the different ways in which the approximation schemes are formulated, the system of equations is not the same for each method. The structure of the matrix determines the computational strategy used in a high performance implementation of the schemes.

5.1 FD and FEM: Sparse Matrix Methods

In FDM, A is banded (tridiagonal in one-dimensional problems) such that one can use a number of efficient solvers operating on matrix diagonals. While it depends on the specific underlying prob-

lem, in the case of a regular two- or three-dimensional domain and a Laplace operator, fast direct methods based on FFTs are often optimal. Banded Cholesky solvers are an often used alternative choice if the number of gridpoints is not too large. Other options include an appropriately preconditioned conjugate gradient or conjugate residual method, the ADI method, or SOR. Recently, elliptic problems on regular grids have been efficiently treated using multigrid methods [11].

The FE discretization process results in a matrix A which is symmetric and very sparse, whose computational size is determined by the maximum bandwidth of the global stiffness matrix. The computational load involved in computing the matrix elements depends on the order of the element and its shape function. For linear triangles and linear tetrahedra, there exist exact forms of the necessary integrals which allows for the most expedient element matrix evaluations. For higher order elements, one must evaluate the integrals which increases the necessary element evaluation time proportionally. For large scale systems, the creation of the global stiffness matrix is often the most time consuming aspect of a finite element solution. The number of nonzero elements along any row (or column) depends on the number of nodes that interact via common elements, therefore, the bandwidth, while potentially small, can become grossly inflated due to an inefficient numbering of the nodes. As the size of the problem increases, a reduction of the matrix bandwidth becomes increasingly important as a means to reduce overall memory requirements. Since the size of the system to solve (for bandwidth solvers) is determined by the element which is furthest from the diagonal, one could imagine a worst case scenario in which there is an element on the diagonal and one at the right most element (N) in the matrix. One would then store (N-2)/2 zeros (assuming the matrix is symmetric). Thus, bandwidth minimization can provide a considerable savings in storage costs.

One method of bandwidth optimization uses an algorithm put forth by Cuthill and McKee [16]. This necessitates reordering the nodes in such a way as to minimize the elemental connectivity, and thus the bandwidth. While this strategy typically reduces the bandwidth by an order of magnitude, it still leaves numerous zeros in the matrix. A second, and often more successful strategy, involves storing the global stiffness matrix using a sparse storage scheme, such as compressed-sparse-row (CSR) format [20]. According to these schemes, only the nonzero values in the matrix are retained, along with arrays which contain the necessary pointers to locate the original elements. Sparse storage typically reduces the memory needs for the global stiffness matrix by two orders of magnitude. While such schemes require more overhead to retrieve and store the data, when used with sparse matrix solvers on large problems, the overall effect is a considerable reduction in computation time over standard solution strategies.

For problems which can be kept in memory, direct solution strategies are often preferred. Direct solutions typically consists of first optimizing the bandwidth, then storing the maximum bandwidth region of the global stiffness matrix in a one-dimensional array, and finally computing the solution using a bandwidth solver based on a modified Gaussian elimination method [30]. Estimates of the number of operations necessary to perform the various solution decompositions for direct methods can be made by considering the number of multiplications (including divisions) and additions [19].

For an LU factorization the number of multiplications is $\frac{1}{2}mn^2 - \frac{1}{6}n^3 + \frac{1}{2}mn - \frac{1}{2}n^2 + \frac{2}{3}n$ and the number of additions is $\frac{1}{2}mn^2 - \frac{1}{6}n^3 - \frac{1}{2}mn + \frac{1}{6}n$ yielding a total number of flops, $mn^2 - \frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{5}{6}n$ where m by n is the dimension of the matrix. Once an LU factorization is completed, systems can be solved by using $NRHS[2n^2 - n]$ total flops where $NRHS$ denotes the number of right hand sides. Cholesky factorization yields $\frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$ total flops and the solution of systems after Cholesky factorization requires $NRHS[2n^2]$ total flops. For large scale problems, efficient iterative schemes are the only alternative. Iterative solution strategies often include implementation of either a Jacobi method with preconditioned conjugate gradients (JCG), a successive overrelaxation (SOR) method, or a symmetric SOR with a conjugate gradient preconditioner (SSORCG) [36] while utilizing the sparse storage techniques described above.

For problems whose size exceeds the memory constraints of a single vector or parallel supercomputer (about 25 million floating point numbers on the Cray Y-MP at Pittsburgh Supercomputing Center, 51.3 million floating point numbers on the Cray Y-MP at North Carolina Supercomputing Center, and about 128 million floating point numbers on the Connection Machine CM-2 at Pittsburgh Supercomputing Center), it is often advantageous to distribute the load over a number of separate machines (usually workstations). By implementing a "distributed paradigm" from the beginning, it is relatively easy to accomodate larger and larger problems.

Parallel and distributed implementations of iterative methods are often quite attractive for large scale problems, especially when the matrix has a regular nonzero structure which can be exploited to obtain a structure that leads to independent substructures [1], [53], [47], [48], [38], [54], [52]. Unfortunately, no single iterative method is robust enough to solve all sparse linear systems accurately and efficiently. An excellent resource for considering various sparse matrix iterative solution strategies may be found in the text by Dongarra [19].

Structured parallel algorithms are usually designed to maximize the amount of computation performed by a typical task module before communication with other modules, to create an efficient intermodule communication topology, and to enforce the interactions among different task modules (ensuring correctness of the parallel algorithm). The optimal implementations of these algorithms are generally machine dependent and are beyond the expertise of the typical biomedical researcher.

Recently there have been general purpose software packages available which can help in the automatic distribution of ones solution over several processors (machines, in the case of linked workstations) and aid in developing an effective parallel scheme. These packages contain protocols for processing with sophisticated forms of message passing down to the simplistic (but still effective) form of functional replication, where several machines are working on different aspects of same problem. Examples of available software include: PVM, Express, and Linda. PVM is free and distributed through netlib. It is flexible, provides all the basic message-passing and process control functions, and has accessible source code so that one can customize it to more non-generic network interfaces. One negative aspect of PVM is that the messages are not direct; messages are first sent to an intermediary daemon which then directs the messages to the appropriate processes. Express (from Parasoft) is a more integrated package which includes automatic parallelization and a debug-

ger. Express provides direct versus indirect message-passing or synchronous versus asynchronous message-passing (the latter option enables one to *hide* the communication latency by overlapping communication with computation. Linda is another commercial package (Scientific Computing Associates), though different from PVM and Express in that it presents a simulated shared memory model to the programmer (independent of whether the physical underlying memory is shared or not). Once again, the specific package one chooses is highly dependent on the application and available hardware.

5.2 BEM: Full Matrix Methods

The BEM generates a full matrix, or if domain decomposition methods are used, a block sparse matrix with dense block submatrices. For the full matrix the direct LU factorization method is the standard approach, whereas block sparse matrices are solved with block equation solvers [15]. An alternative approach is to use iterative methods, however, since the matrix is nonsymmetric only the so called generalized iterative solvers [34] should be used. An interesting combination of direct and iterative methods is suggested by Bettess [6], who suggests a direct solver be used on a diagonal band containing the most significant terms and an iterative method is used on the smaller terms outside the band.

Another issue is whether the matrix should be stored in core memory or on secondary storage (e.g. disk). The decision depends on the size of the matrix and the amount of core memory it is tractable to allocate to the program. It is usually not prudent to allocate the full amount of physical memory to the job as the job priority will decrease dramatically on time shared computers. Out-of-core solvers only has a small slab of the matrix resident in memory hence the memory allocation is much smaller and the job priority much higher. Direct out-of-core equation solvers are very complex to program [26] and they are not available in general purpose software libraries, thus most out-of-core equation solvers are based on iterative techniques. Because BEM generates large and dense matrices, the storage and solution techniques must be designed before designing the algorithm for computing the matrix coefficients. One reason for this is that it is usually not possible to store the coefficient matrices in memory for the subsequent assemblage of the system matrix. In this section we will briefly discuss the computing and solving of BEM equations and point the attention to factors that can degrade performance.

In many BEM problems, linear or quadratic interpolation functions are needed within each element to obtain high accuracy. For a surface discretized with quadratic quadrilateral Lagrange elements containing nine interpolation nodes [10], the surface integrals in (37) can be approximated as follows:

$$\sum_{j=0}^{N}(G\psi_j)_i\left(\frac{\partial\phi}{\partial n}\right)_j \approx \sum_{j=1}^{N_e}\sum_{l=1}^{9}\left[\int_{-1}^{1}\int_{-1}^{1}\frac{\psi_l(s,t)}{4\pi\sigma_e R_{i,j}(s,t)}|\vec{J}_j(s,t)|dsdt\right]\left(\frac{\partial\phi}{\partial n}\right)_{j,l} \tag{38}$$

$$\approx \sum_{j=1}^{N_e}\sum_{l=1}^{9}\left[\sum_{m=1}^{N_s}\sum_{n=1}^{N_t}\frac{W_m W_n\psi_l(s_m,t_n)}{4\pi\sigma_e R_{i,j}(s_m,t_n)}|\vec{J}_j(s_m,t_n)|\right]\left(\frac{\partial\phi}{\partial n}\right)_{j,l} \tag{39}$$

and:

$$\sum_{j=0}^{N}(H\psi_j)_i\phi_j \approx \sum_{j=1}^{N_e}\sum_{l=1}^{9}\left[\int_{-1}^{1}\int_{-1}^{1}\psi_l(s,t)\frac{-\vec{r}_{i,j}(s,t)\cdot\vec{J}_j(s,t)}{4\pi\sigma_e R_{i,j}^3(s,t)}dsdt\right]\phi_{j,l} \tag{40}$$

$$\approx \sum_{j=1}^{N_e}\sum_{l=1}^{9}\left[\sum_{m=1}^{N_s}\sum_{n=1}^{N_t}\frac{W_m W_n \psi_l(s_m,t_n)}{4\pi\sigma_e R_{i,j}^3(s_m,t_n)}\left(-\vec{r}_{i,j}(s_m,t_n)\cdot\vec{J}_j(s_m,t_n)\right)\right]\phi_{j,l} \tag{41}$$

where N_e denotes the number of elements and $\vec{J}(s,t)$ denotes the Jacobian associated with the transformation from the cartesian coordinate system to the curvilinear (s,t) coordinate system. In (39) and (41) further approximation is introduced by employing the Gaussian Quadrature integration scheme [17]. The number of Gauss points in the s and t directions are denoted N_s and N_t, respectively, and the Gauss points are denoted by s_m and t_n, respectively. W_m and W_n denote the Gauss weights in the s and t directions, respectively.

The CPU-time required to compute the matrix coefficients is impossible to predict on a general basis because the performance on vector processors is strongly dependent on the implementation. The mathematical expressions in (39) and (41) for the matrix coefficients contain five indices (i,j,l,m,n). The j index divides the matrix into slabs each containing 9 columns associated with the same element. Index l enumerates the columns in a single slab and index i enumerates the elements in each column. Hence j and l are partitioned whereas i is sequential from 0 to N. Each matrix element is the result of a double summation (indices m and n) of terms in the Gaussian Quadrature scheme. The Gaussian double summation can be reduced to a single summation by merging the matrices of Gauss points and weights into sequential arrays.

If the summations are performed in the order written in (39) and (41) (do loops nested in the order i,j,l,m,n) each matrix coefficient is finished before the next one is computed. This order of nested summations corresponds to computing the matrix coefficients in a row-wise order. A more efficient nesting of the do loops would be j,m,n,i with the l loop unrolled (written out as 9 separate statements) inside the i loop. The j loop is chosen as the outer loop such that the Jacobian only needs to be computed once for an entire matrix slab. The i loop is chosen as the inner loop in order to obtain a long range of the index of the inner do loop (performance on a vector processor degrades for a small range in the inner loop index [40] as is discussed in the next section).

The direct and iterative methods for solving the system of linear equations are fairly standard and thus it is possible to obtain a good estimate of the cpu-time. The number of floating point operations required by the LU factorization of a nonsymmetric matrix of dimension N is on the order of $2N^3/3$. The floating point performance of the out-of-core solver RLUD [14] in the BNCHLIB library installed at North Carolina Supercomputing Center (NCSC) is approximately 210 MFLOPS (million floating point operations per second) per cpu. A 5400×5400 matrix requires about 500 CPU-seconds on a single processor. A generalized iterative solver such as the preconditioned biconjugate gradient method (PBCG) [34] requires about $2.8N^{2.5}$ floating point operations (assuming $\sqrt{N}$ iterations) and performances on the order of 245 MFLOPS have been observed during tests, hence the CPU-time will be about 24 CPU-seconds on a single processor. This performance of

the bi-conjugate gradient method was obtained by explicitly inverting the dominant block diagonal matrix containing singular integrals and use it as preconditioner. A break-even point of $N = 365$ can be estimated from:

$$T_{CPU_{\mathrm{RLUD}}} = \frac{(2N^3/3)\mathrm{FloatOps}}{210\mathrm{MFLOPS}} \qquad T_{CPU_{\mathrm{PBCG}}} = \frac{(2.8N^{2.5})\mathrm{FloatOps}}{245\mathrm{MFLOPS}} \tag{42}$$

where the performance data for the PBCG method are obtained from experiments. For most problems there is a clear speed advantage in using iterative solvers, however, iterative solvers for nonsymmetric matrices are not often found in general purpose mathematical software libraries and the user is left to implement custom algorithms optimized to the application. The performances assumed in the above analysis of out-of-core solvers is based on the application of asynchronous input/output (I/O) facilities on the Cray Y-MP. These read and write instructions will not halt program execution, hence the next slab of a matrix can be read into memory while the iterative solver operates on the current matrix slab. Utilizing this particular Cray feature can prevent the performance of the iterative solver from being I/O bound; unfortunately the I/O routines will not function on the local workstation.

Which solution strategy one uses depends on the size of the system, available computational resources, stability of the problem, and the availability of functioning computer code. For small scale BEM problems the two coefficient matrices and the assembled system matrix can be stored in memory. If the boundary conditions change, the program only has to go back and reassemble the system matrix from the two coefficient matrices. For large scale BEM problems, storage of both coefficient matrices and the assembled system matrix may exceed both memory and disk capacity. Considerable savings on storage is obtained by assembling the system matrix ad hoc without saving the coefficient matrices; unfortunately the matrix coefficients must be recomputed if the boundary conditions are changed.

6 Porting, Optimizing and Benchmarking Programs

Although it is extremely attractive to simulate models on local machines, many large scale problems require more computing power than the new generation of workstations or data servers can supply. Such large scale simulation models must be ported to a supercomputer. The types of supercomputers of most interest today is the multi processor vector computers (e.g. IBM 3090 (6 CPUs); Cray Y-MP (8 CPUs)) and the massively parallel computers (e.g. Thinking Machines' Connection Machine (CM-2: 32,768 CPUs); various hypercubes).

There are far more vector computers than massively parallel computers, thus we will only consider the problem of porting a program to a vector computer. These computers achieve their power through *vectorization* or *pipelining* of the inner most do loops. A pipeline can be likened to a small assembly line [40] which operates on vectors of operands. The idea is that at each clock cycle each site on the assembly line performs a different operation (e.g. fetch operands, normalize operands, add/multiply, normalize result, store result). A typical floating point operation might

consist of 5 subtasks, hence on a scalar machine a new result would be obtained only every five clock cycles. For the same operation on a vector processor it takes five clock cycles to obtain the first result, however, due to the pipelining of operands, subsequent results are produced every clock cycle. The startup delay while filling a pipeline for the arithmetic instruction can be reduced by linking floating point operations together into a chain. If for example two vectors are multiplied term by term and the results are added to a third vector, the multiplication pipeline is chained to an addition pipeline. The result from the multiplication is fed directly into the addition pipeline, thereby avoiding intermediate store and fetch operations.

In general, if a scalar floating point operation consisting of n subtasks can be vectorized in a pipeline, then the vectorized version should run about n times its non-vectorized speed. This ideal speedup is rarely achieved since there is some overhead with setting up the pipeline and with executing the do loop. For example the Cray Y-MP has very fast access vector registers which can hold vectors 64 elements in length. The optimal implementation of a vector operation on a set of vectors of length N makes use of the vector registers by dividing each of the original vectors into vectors of length 64, plus a set of remainder vectors. The operation is then executed 64 elements at a time with the operation on the remainder vectors executed out at the end. If the vector length N is much larger than 64, the relatively large overhead of computing a short remainder is negligible compared to the total time spent in the loop. Thus performance measured in floating point operations per second will increase with increasing vector lengths. Due to the overhead the speed up of a vectorized code segment is typically between 10 and 20.

6.1 Porting of Code

Standardization of operating systems and languages has made it easier to move a model from a workstation to a supercomputer for large scale simulations. Ideally, to take full advantage of the supercomputer, the algorithm should be designed with the supercomputer architecture in mind. In the most common scenario, an investigator has written a program for a scalar workstation but an increase in the model size demands the greater performance or memory of a larger machine. For this situation, the ported code will likely require some restructuring. There are three major steps to converting working scalar code to vector code:

1. Automatic vectorization
2. Rewriting of isolated regions of code or subroutines, inclusion of compiler directives to facilitate vectorization, and replacement of code with numerical library utilities.
3. Changes to alternative algorithms better suited for vectorization.

In some cases, the investigator may enlist the assistance of a supercomputer programmer. The first step is to remove any syntax or constructs that do not conform with standard Fortran 77, Fortran 90 or ANSI C. Before proceeding, it is important to establish that the new, standardized program is reliable and well structured. The version of the program emerging after removing all

syntax violations will provide the benchmark for all subsequent optimizations. The next step is to compile it using the automatic vectorization facilities. These facilities are available on most, if not all vector computers. In many cases, the user will find the auto-vectorized version of the program will not run much faster than the scalar version on the supercomputer did. In the worst case the auto-vectorized version may not run faster than the original program on the workstation.

In order to further improve the performance, manual analysis and modifications are usually required. The first option is to replace code segments with library routines whenever possible. Supercomputers have highly optimized routines to carry out most common mathematical operations. To solve the compatibility problems associated with library routines, a collection of general purpose Cray compatible routines have been archived in netlib. These algorithms can be ported to the workstation such that the Cray version of the program also compiles on the workstation. Secondly it is important to identify the remaining portions of the code which have not been vectorized. Some of the most common vector inhibitors are:

- Short loops— As mentioned above there is a large overhead associated with setting up a pipeline. Consequently, very short loops will not be vectorized.
- Data Dependency— A variable stored in one statement is subsequently used in another statement; hence, operations must be executed in order. Vectorization requires that several variables are operated on simultaneously. Two of the most common situations where data dependency inhibits vectorization are:
  - Recursion— In general, loops containing recursion cannot be vectorized. Although it is often possible to rewrite the relationship to allow vectorization, in some cases the programmer is forced to choose an alternative algorithm for the calculation.
  - Indirect addressing of arrays— Loops which use indirect accessing are rarely vectorized automatically by the compiler because of the possibility of data dependency.
- Conditions in loops— Although most machines have facilities for vectorizing loops with simple conditions in them, loops containing nested condition statements cannot usually be vectorized. Furthermore loops which contain conditions which are rarely true often run faster in scalar mode.
- Subroutine calls within loops— Strictly speaking it is not possible to vectorize over a change of control such as a subroutine call.
- I/O inside loops— It is usually not possible to vectorize loops which contain I/O statements.

Once the non-vectorized portions have been identified, the next step is to restructure them for vectorization. Redesigning the code to remove the vector inhibitors will often yield a considerable improvement in performance; however, restructuring alone will rarely result in peak performance. For maximal performance, the program must be modified to exploit the memory architecture and high speed input/output facilities.

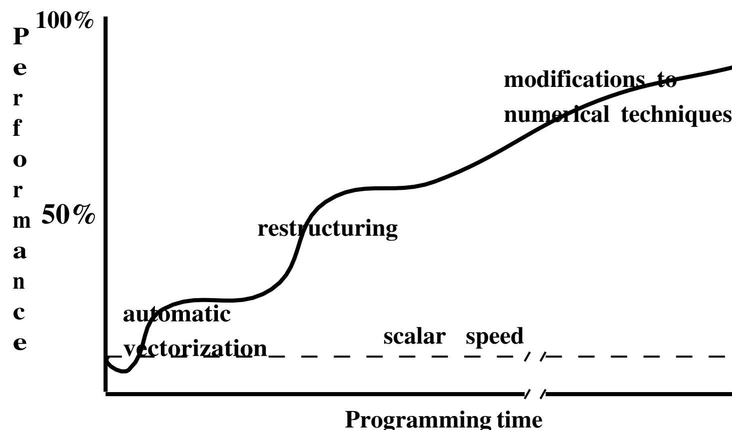


Figure 3: Performance versus programming time. The time course of porting a typical program.

The optimization process is governed by *the law of diminishing returns* (see Figure 3). The peak performance of the program is usually obtained only after making painstaking adjustments that may take weeks or even months to complete. This small increase is usually not worth the required effort. The targets of optimization has changed from the old generation of compilers to the very intelligent compilers found on modern high performance computers. When optimizing programs for compilers of the past, the programmer was targeting code structures on the line level looking for redundant storage and instructions. On modern compilers the dos and don'ts have almost been reversed, e.g. it is better to leave invariant and common subexpressions inside the loop, or even move them back in if they have been moved outside the loop by a "good meaning" programmer. On the line level, the modern day programmer is only responsible for using code constructs that makes it possible for the compiler to recognize possibilities for optimization. This usually means the programmer should use parenthesis with caution. The major targets for optimization on modern computers are global code structures, e.g. in-lining of functions and subroutines, and reorganization of conditional branching:

- Restructuring of code to minimize the nesting of condition statements within loops. Most compilers will not vectorize nested if statements. It is always possible to rewrite nested block with multiple single level blocks which vectorize.
- Removing short loops— The order of nested loops is an extremely important factor. As a general rule the longest loop should be innermost. Very short inner loops can be unrolled (*i.e.* replaced by the explicit instructions), collapsing two loops into one. Another similar vector inhibitor which is easily fixed are loops which are iterated an indefinite number of times. The compiler estimates the length of the loop and in many cases will not automatically vectorize it. If it is known that the loop is long, one only need issue a compiler directive to vectorize it.

- Ensuring indirect addressing of arrays does not lead to data dependency— If it is clear that there are no data dependencies associated with indirect accessing, it is possible to issue compiler directives to force the vectorization of loops containing indirect addressing.

- Subroutine and function calls within loops — Frequently used single line functions can be replaced with statement functions which vectorize. In many cases calls to subroutines can be replaced with the code itself. Many compilers have a so called *inline* option making this transparent to the user. Often the overhead of a call to a small subroutine is of the same order of magnitude as the instructions themselves thus even if *in-lining* does not result in vectorization there are still savings to be realized.

Because the logical structure of the scalar program may have been sacrificed in the optimization process, the new version is usually far more cumbersome to modify than the original. Hence, although the optimized program may also run efficiently on a conventional computer, it is usually better to make a new workstation version based on the original scalar program than based on the vectorized version.

6.2 Tools for Benchmarking Programs

The implementation of changes to obtain better vectorization is an iterative process. The impact of each change must be accessed with regard to performance and the correctness of the solution. Fortunately, there are a number of system tools that can be used to identify computational bottlenecks and monitor improvements in performance. The tools illustrated in this section are available on the Cray Y-MP; similar tools are available on other supercomputers. In many algorithms, the input/output portion of the code may dominate the total CPU time. The PROCSTAT program [13] collects run-time statistics for input/output including filename, maximum file size, characters processed, and waiting time. If a large amount of time is spent on I/O, alternative file formats should be considered and then tested. Table 1 [13] illustrates the relative speed up obtained for different methods of I/O. Row a denotes the reference data for the formatted write statement. For the unformatted output statements in rows b-d the speed up is relative to the row above, i.e. the speed up in going from method b to method c is 4.2 for N=1000. Option d refers to the asynchronous I/O facility, a specific Cray feature that will not halt execution during the I/O process.

When I/O bottlenecks have been removed or minimized the remainder of the program is analyzed. The peak performance of a given algorithm will depend on the percentage of the program that is vectorizable. Vectorization typically speeds up a code segment by roughly a factor of 10. If a code segment is vectorized and this segment accounts for 50% of the cpu-time of the original code, the overall execution time for the optimized program will be about 55% of the original execution time. If 90% is vectorized, the execution time is reduced to 19% of the original code. It is usually prudent to first optimize the most time consuming segments of the code. The FLOWTRACE utility [13] analyzes the dynamic flow through the program. It provides information about callers and

Table 1: I/O Performance

Examples of write statements	N			
	10	50	100	1000
a) write(10,'(E15.7)')(A(i), i=1,N)	1	1	1	1
b) write(10) (A(i), i=1,N)	10	45	66	131
c) write(10) A	1.1	1.5	1.5	4.2
d) buffer out(10,0) (A(1), A(N))	1.5	1.5	1.7	2.2

callees, such as the time spent in the routine, the number of times the routine is called, the average time per call, and a list of the routines that called a particular program unit. The FLOWVIEW utility [13] provides a graphical presentation of the information gathered by FLOWTRACE. In Figure 4 a calling tree is illustrated with the cpu-time of the respective routines in a test program for an iterative matrix solver. Routines may be ranked by time, name, number of calls, average time per call, in-line factor and called by timings.

FLOWTRACE generates a dynamic calling tree, showing the flow path actually taken during program execution. Another utility, FTREF [13], generates a static calling tree that represents the declarations of program units in the code, even though some of these may never be executed. In the example program, the algorithm MATVEC (a matrix-vector multiplication algorithm) uses about 56% of the cpu-time and should be analyzed further. A compiler listing with loopmarking is invaluable in analysing the automatic optimization of loops. If a loop is vectorized, the loopmarks will indicate if it is a regular vector loop, a short vector loop or a conditional vector loop, the latter being a loop containing IF statements. Following each subroutine is a list of all loops in the code segment and the type of vector inhibitors that prevented some loops from being vectorized.

After removing vector inhibitors a subroutine level benchmark is obtained with PERFTRACE/ PERFVIEW [13]. The information collected by PERFTRACE during execution includes general information (the number of floating point operations and the number of floating point operations per second), conditions that prevent the issue of instructions (e.g. one processor waiting for another processor to complete a task in a program executing on multiple processors), memory references and conflicts (e.g. inefficient order of accessing array elements in loops and simultaneous access to the same memory bank from different CPUs), and vector operations and number of instructions of various types (scalar, vector logical and vector integer, vector floating point, vector memory instructions, and average number of elements per vector instruction). It is clear from the PERF-TRACE output that a fair amount of knowledge about the hardware is required to make use of the information.

Figure 5 shows the PERFTRACE output for routine MATVEC and routine MATINV (inverts a matrix by repeated back substitutions). The "instruction buffer fetch rate" reports how often a new instruction was read during execution. If the number is greater than 1.0, the program is not executing optimally and may contain unnecessary GOTO statements (spaghetti code). If the

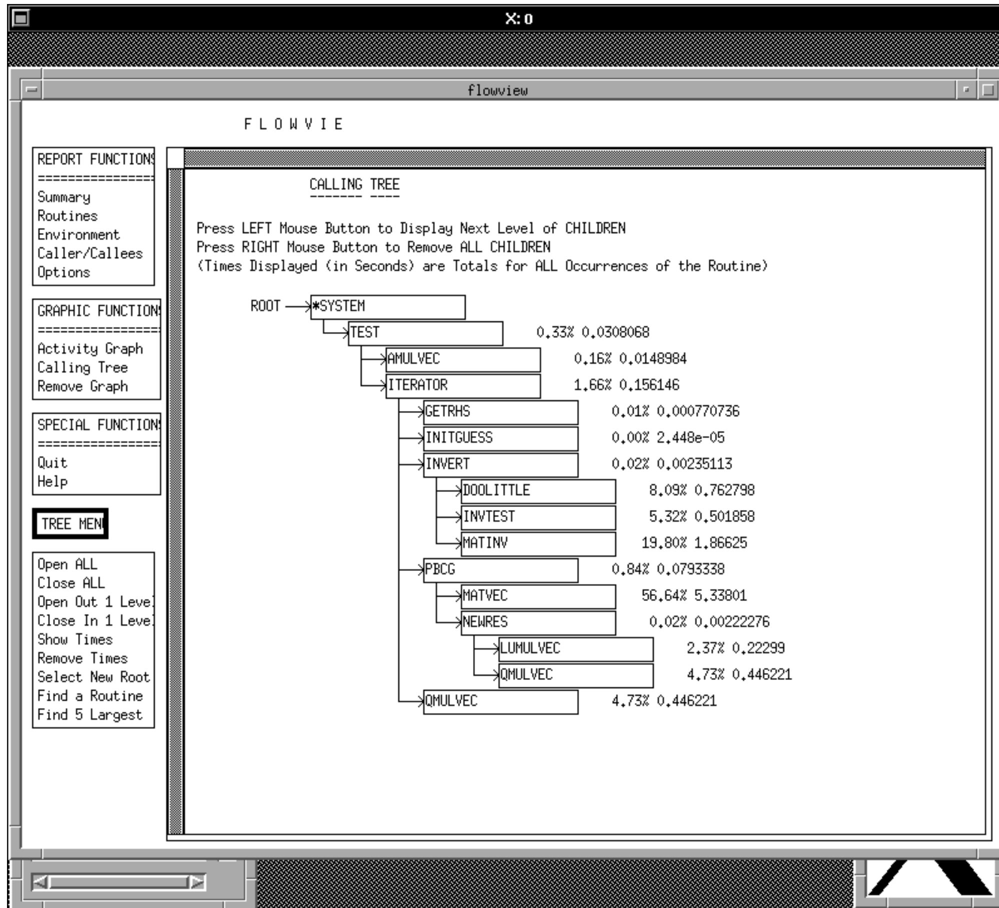


Figure 4: FLOWVIEW utility displaying a dynamic calling tree with relative and absolute cpu-time listed for each routine.

ratio "CPU mem. reference per sec./Floating ops. per sec." is much larger than 1.0, memory references prevent efficient use of CPU registers. Divisions are counted as one reciprocal and three multiplication operations, hence divisions should be eliminated as much as possible.

The most cpu intensive loops can be analyzed with the PROF/PROFVIEW utilities [13]. The profiling utility PROF provides information on a per line basis of the activity of the program. When the program executes with PROF enabled, the system subdivides the memory allocated to the program into intervals (buckets). Each time the system observes the program executing in a given area, the "hit" counter is incremented by one. The more "hits" in a given bucket, the higher the amount of CPU activity in that section of the program. PROFVIEW presents graphically or in tabular format the data gathered by PROF as illustrated in Figure 6. The core of the matrix-vector product routine MATVEC is located in the nested do loops 408, and 409, hence it is worthwhile to experiment with the loop ordering to obtain the best performance. Comparison of individual routines in a program is also possible with PERFTRACE/PERFVIEW. Figure 7 illustrates that more than 90% of the total cpu-time is spent in four subroutines; hence, the optimization should first be performed on these routines.

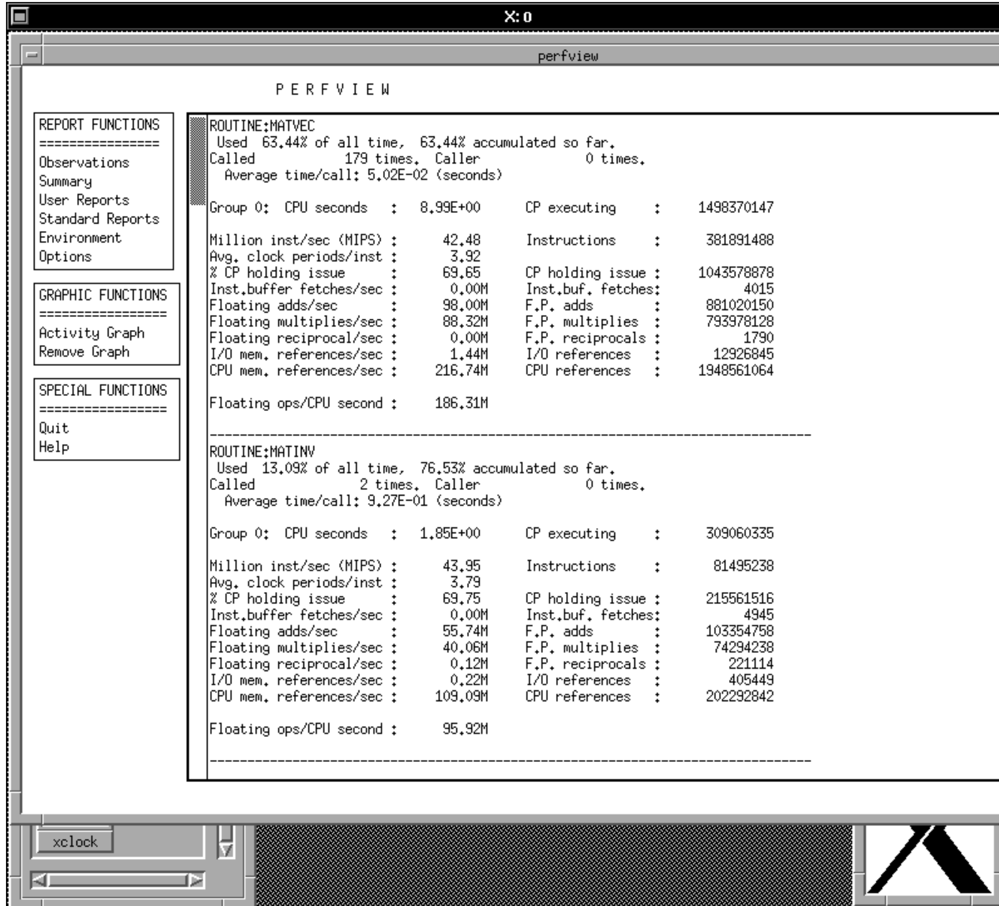


Figure 5: Subroutine level performance data displayed with PERFVIEW. The data includes information about the number and rate of floating point operations and memory operations.

Since the performance of the vector processor increases with increasing range of the innermost loop index, it is essential that the performance is benchmarked for various loop lengths to see if the ranking of CPU intensive routines changes. In Figure 8 the performance of different routines in a boundary element program is analyzed based on data obtained from a series of PERFTRACE analyses. As reviewed in the section on computational methods, a typical BE program includes a routine for computing regular integrals (off-diagonal matrix blocks), singular integrals (diagonal matrix blocks), and a matrix solver (in this case an iterative solver). The routine computing the diagonal block shows no change in performance as the problem size increases. This insensitivity to problem size results because the inner do loop is the Gaussian Quadrature summation loop (see (39) and (41)), which is independent of the number of elements. The computation of the off-diagonal blocks is affected because the inner do loop iterates over the number of field points (i.e. the i index in (39) and (41)) which increases with increasing problem size.

The iterative matrix solver (preconditioned bi-conjugate gradient solver) is strongly influenced by the size of the problem because the core task of the solver is the matrix-vector multiplication operation. The overall performance of the program is a weighted average of the performance of the

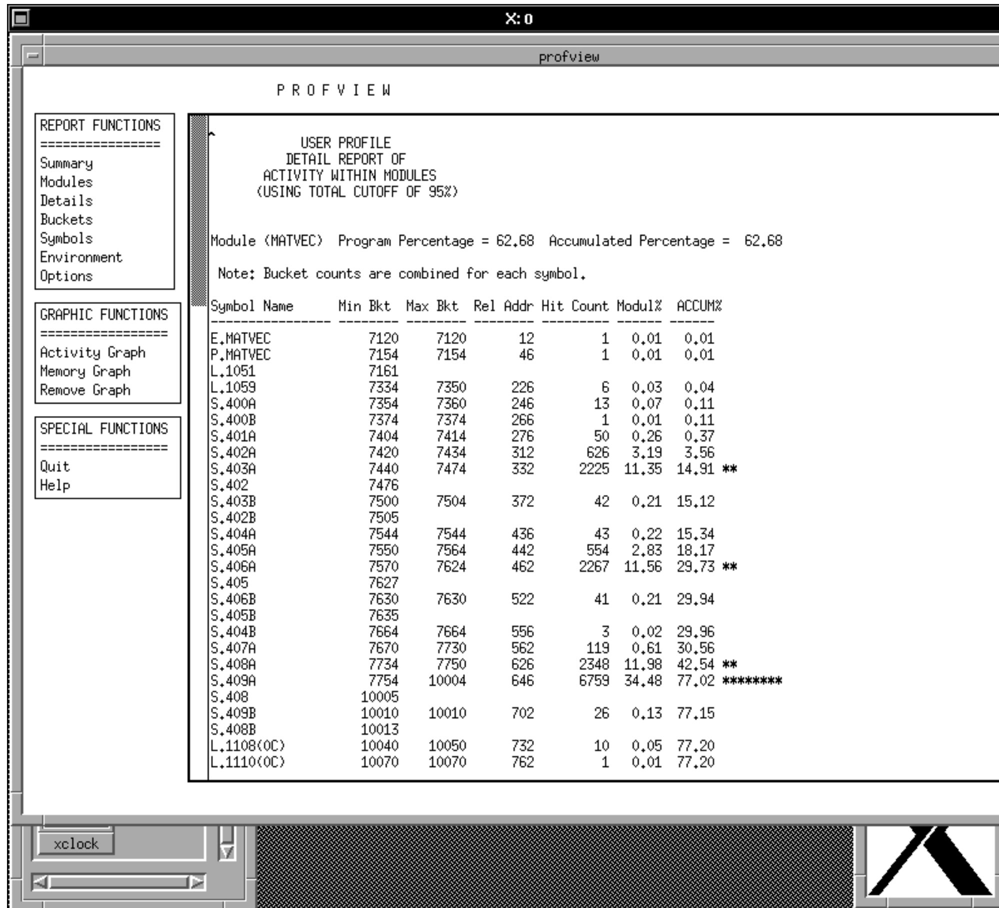


Figure 6: Line by line activity provided by PROF/PROFVIEW. Most activity is observed in the nested loops 408, and 409.

individual routines, where the weighting factor is proportional to the amount of cpu-time spent in the individual routines. In the case illustrated in Figure 8 an increasing proportion of the cpu-time is spent in the routine that computes the off-diagonal blocks, and the performance of this routine therefore tends to dominate the overall performance of the program.

7 Data Management

Although the development of optimized and vectorized software for mesh generation and problem solution in large-scale biomedical computing is a formidable task, an important practical consideration is the organization of simulation results with respect to the visualization. This part of the large-scale software development cycle is often postponed until the simulations have been completed. Postponement is unfortunate because the time required to extract and present information from the simulations can equal or exceed the time required for software development in the parts described in the preceding sections. The development process for visualization will be shortened if a 3-D display software package can be used. Unfortunately, for many problems the graphics

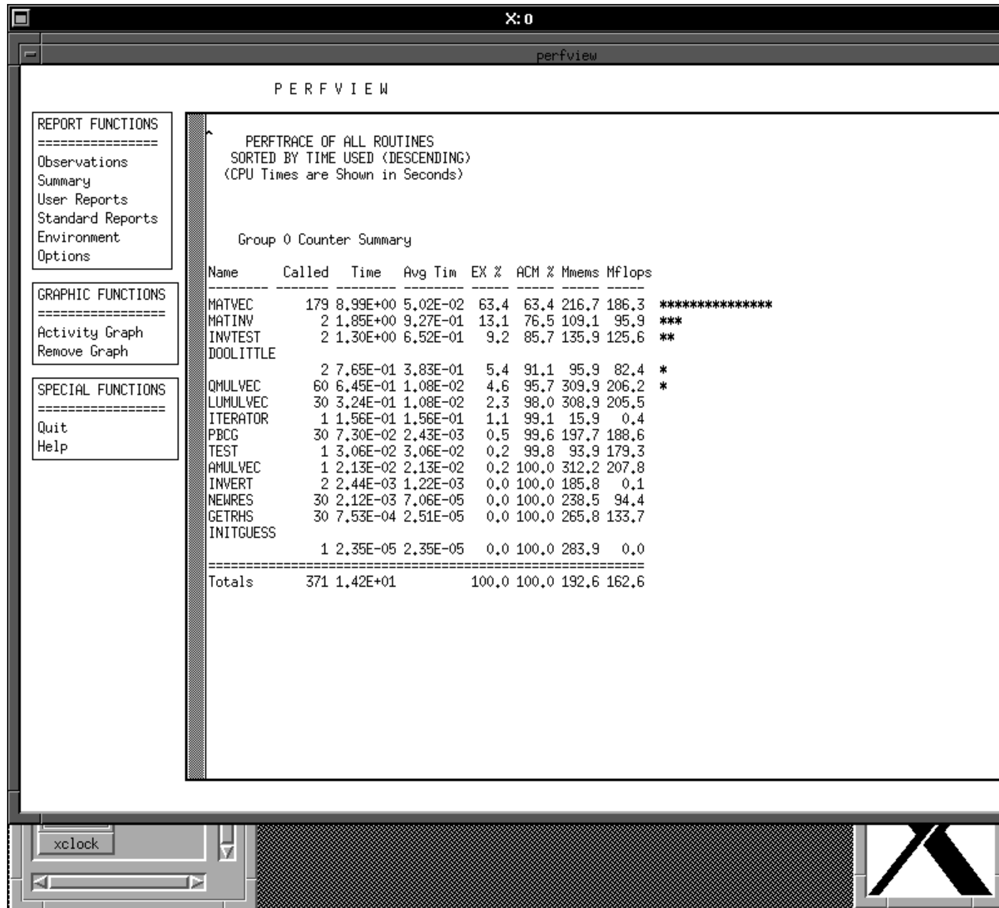


Figure 7: Performance data for the entire program. The column ACM refers to the accumulated CPU-time in percentage of the whole program. More than 90% is spent in four routines.

software must be developed by the investigator. Under these conditions, it is advantageous to identify a visualization support person early in the project. Regardless of the software used for the visualization, we have experienced that the analysis of simulation results requires a considerable amount of time, even after the visualization software has matured.

7.1 File Format Design

The challenges with regard to data management and visualization are a consequence of the aspects of large-scale computing typically considered to be advantages. Namely, supercomputers are capable of generating enormous amounts of data at high speeds. Meshes are large and often multiple variables are associated with individual nodes. In time dependent problems, unique data is generated at each node at each time step. All of these complexities pose a problem in terms of data management. For example, if we wanted to store the contents of one nodal variable from a single iteration during a simulation in a mesh with 10^6 grid points, direct storage of the results from that iteration would require space for 10^6 floating point values. If we used binary storage and 32-bit precision for each entry, a storage file would require 4 MBytes. This file size is manageable. In addition, operations

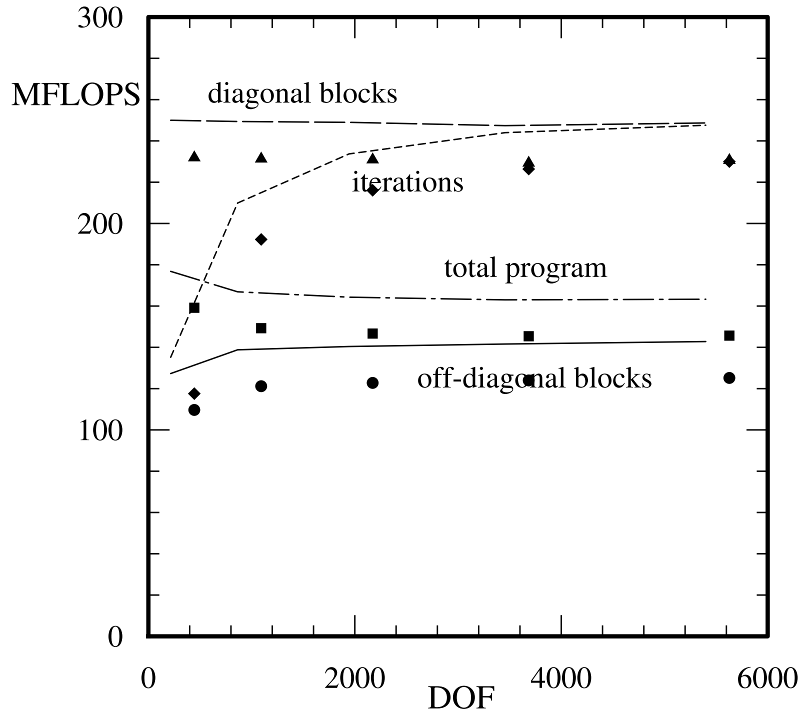


Figure 8: Performance in millions of floating point operations per second (MFLOPS) as a function of the number of degrees of freedom (DOF) (range of inner do loop) for a boundary element program employing an iterative matrix solver.

of this type can be used to construct a "nodal variable map" at one or more selected iterations from the simulation. Extending the direct storage scheme to include the contents from 10 nodal variables at one iteration would require 40 MBytes storage space. This file size is also manageable. However, the CPU time required to simply read the contents of a 40 Mbyte file will be inconvenient. To achieve a sampling rate of 50 kHz for the simulation of a 1 second period would require direct storage of the results from 50,000 iterations and 200 GBytes storage space. This file size is likely to be unmanageable for many investigators.

To design an efficient file format, the investigator must consider how much local and global information to store. The storage of too little data will mean computationally expensive simulations will need to be executed multiple times to ask different questions based on the results from the same calculations. The storage of too much data can overwhelm the storage media and make extraction for post-processing an expensive computational procedure in and of itself. A method which is appropriate for many model problems is described by Pollard and Barr [50] for the simulations of action potential propagation in an anatomically-based model of the human ventricular conduction system. In the simulation, a complete description of the transmembrane potential, V_m, for approximately 30,000 nodes and 20,000 iterations was achieved in files which were less than 20 MBytes in size. Records were stored in an ASCII format and after the application of the Unix compress utility these files occupied less than 8 MBytes. To obtain these file sizes, an adaptive

sampling scheme based on the fan algorithm [22] was implemented. Although it is not our intention to describe methods for adaptive sampling in this tutorial, we feel it is worth noting many of the goals in the development of data compression strategies for multi-channel data acquisition systems are consistent with the design of efficient storage schemes for large-scale biomedical computing. Namely, the transient nature of many biological signals mean a variety of sampling resolutions can be employed to achieve accurate signal reconstruction with limited storage facilities.

A second factor in the management of data from large scale biomedical simulations is the identification of "waveform parameters" which characterize the local behavior. Figure 9(a) shows the time course of V_m at a single node. $\dot{V}_{max}$ is the maximum upstroke velocity. The time of $\dot{V}_{max}$ is considered a good indicator of the local activation time (AT) from a transmembrane potential recording. Visualization of the manner in which AT varies spatially can provide insight into the temporal characteristics of action potential propagation during depolarization in a simulation. Similarly, the action potential duration (APD) is considered an indicator of repolarization. While both parameters can be determined from the V_m profile after the simulation has been completed, there are practical advantages to performing some post-processing while the calculations are taking place. Here it is important for the investigator to determine which parameters can be calculated at a low computational expense during the simulation.

7.2 Visualization

An appropriate visualization strategy provides the investigator with a means to ask questions. Was the mesh generated correctly? What was the local behavior in some region of the mesh? What was the global behavior at some iteration in the calculations? What was the global behavior of the waveform parameters? How did the spatial features of these parameters change over the time course of the calculations?

Each of these questions can be addressed with a visualization strategy that provides methods for the spatial examination of simulation results. Gallagher and Selker [21] outlined these methods in a recent report. To answer these questions, the visualization software should be capable of color blending based on nodal parameter values, isosurface generation, volume slicing and gradient display. Some of these features are included in software packages that facilitate visualization. Mills [45] reviewed the post-processing capabilities of a number of finite element packages which included visualization support operating in a workstation environments. Products such as IMSL/IDL, Intergraph MicroStation 4.0, Adaptive Research Corporation CFD 2000, SRAC COSMOS/M, and ANSYS 5.0 (to name a few) provide facilities for the spatial display of data which can answer the scientific questions that arise from the simulations. Work environments such as the Silicon Graphics IRIS Explorer, Cray Corporation AVS and TaraVisual Corporation's apE offer robust and flexible development tools for the generation of sophisticated graphics applications. With the latter, applications are constructed by combining software modules into flow networks. These modules include graphics primitives for data manipulation and geometric rendering. An example flow network for the apE environment is shown in Figure 10. When these tools can be used, appli-

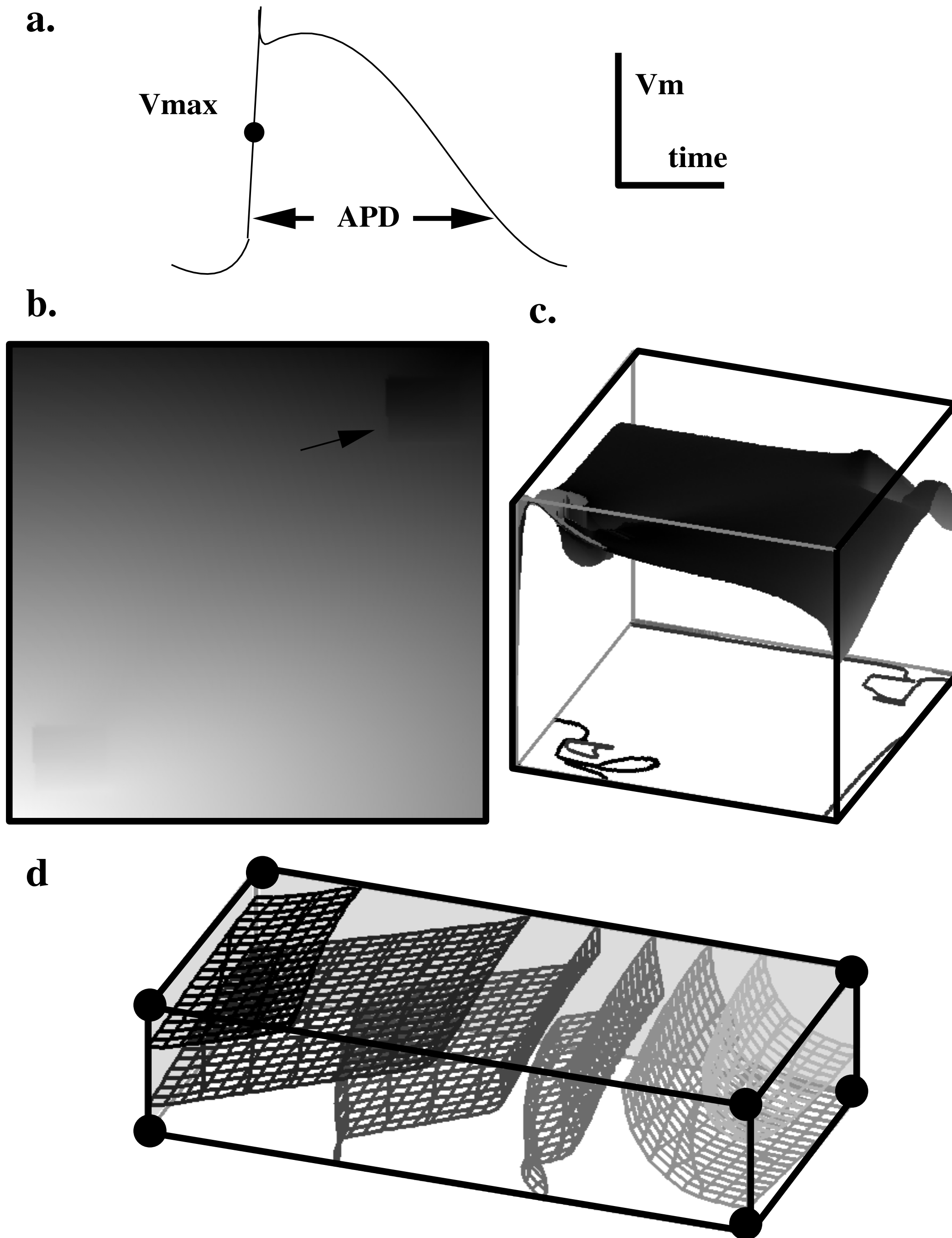
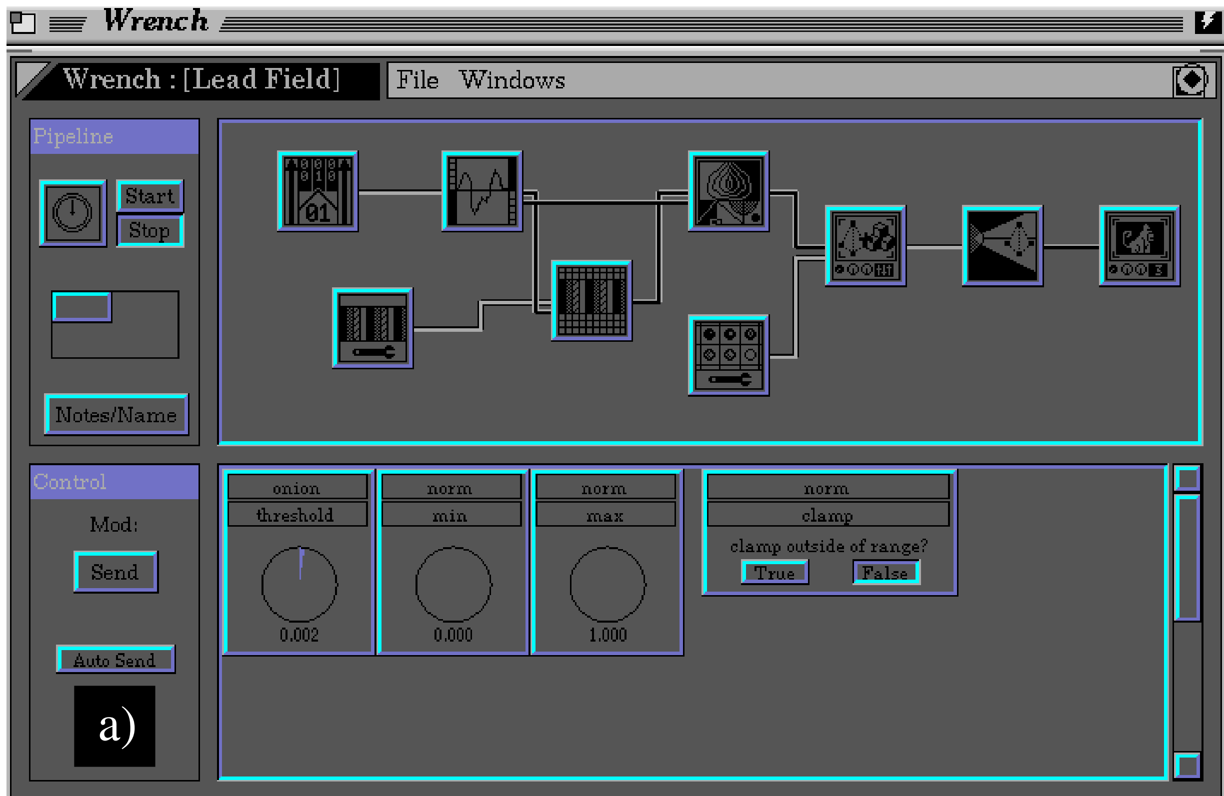
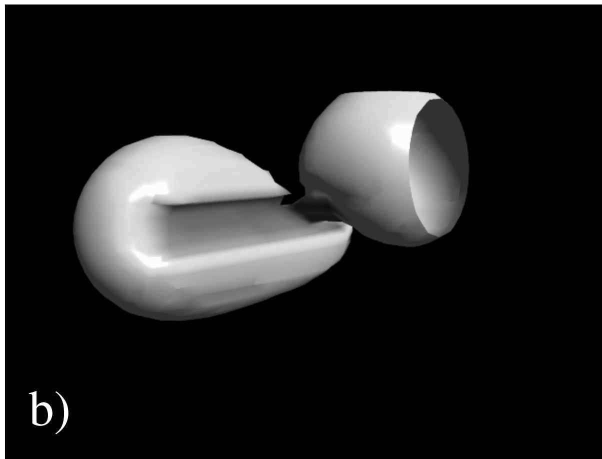


Figure 9: Examples of visualization and large-scale biomedical computing (a) a cardiac action potential and some of the parameters used to describe the action potential, (b) a contour map of AT values from a 2-D simulation of action potential propagation, (c) a raised contour map of APD values from a 2-D simulation of action potential propagation and (d) a contour map of AT values from a 3-D simulation of action potential propagation. All components were generated on a Silicon Graphics 4D/210 and annotations were placed on the images using Showcase.





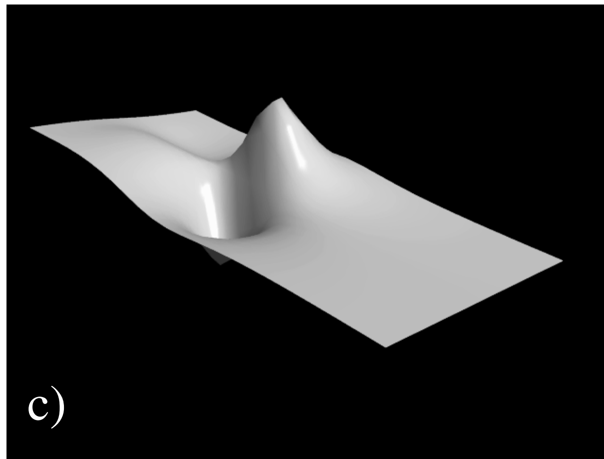


Figure 10: apE visualization environment. a) Top part shows the canvas containing visualization modules arranged in a pipeline. Bottom part shows the control dials for steering the visualization process. b) Iso-sensitivity function for a needle electrode visualized as iso-potential surfaces. c) Electrode polarity map visualized as terrain plot sliced from 3-d volume data.

cations are constructed rapidly. In many cases, however, the commercial products have difficulty accommodating large unstructured meshes and are therefore inadequate for use in a large-scale computing environment.

In the sections which follow, we will demonstrate the use of some of these features. The visualization software was developed in the C programming language using the Silicon Graphics GL library with support from the Center for Scientific Visualization (a component of the Utah Supercomputer Institute).

In most cases, the local temporal behavior at a given node is best examined with x-y graphs. In the sample problem, consider the case of the potential distribution as a function of iteration number. It is possible to obtain an appreciation of the spatial variation at a number of nodes by examining multiple x-y graph. The most difficult aspect of this process is not the presentation (there are a number of software packages for the presentation of x-y graphs in the workstation and personal computing environments) but rather the development of appropriate mechanisms to perform the local extractions at many nodes. Here it is advantageous to have redundant means for that local identification. It should be possible to complete the extraction using node numbers, node coordinates, or graphical identification tools. In practice, we have also found an adaptive sampling storage scheme to be advantageous because the time required to construct multiple x-y graphs is greatly reduced when some data trimming has been performed.

We examine mesh generation in a graphics workstation environment. For large unstructured meshes, this environment is almost required due to the volume of data displayed. The graphics workstation should provide rapid rotation and translation of the mesh which will facilitate inspection of the nodal positions and elemental connectivities. The workstation should also provide facilities for automatic color blending across elements when polygons are rendered. This will allow color blending based on nodal parameter values, which can in turn be used to examine the spatial behavior from the simulation.

Although it is possible to display 3-D data in a variety of ways, the investigator should be aware of the visualization goals. Inspection of the simulation results using a visualization tool is different from the generation of publication quality images. While there are advantages in the selection of unique colors and patterns to demonstrate dominant features of the simulation behavior during inspection, detailed image construction is time-consuming and should be postponed until the simulation results are well understood. The image in Figure 9(b) was USED DURING THE INSPECTION PROCESS and was constructed using color blending facilities of the Silicon Graphics GL library. The figure shows the spatial distribution of the waveform parameter AT from a 2-D simulation of action potential propagation. Color blending was used to show early (light) and late (dark) activation within the model during the simulation. With color blending visualization, we identified the global activation pattern in the model. In addition, we observed a region where the activation wavefront decelerated as shown by the arrow in the figure.

Isosurface generation also requires a knowledge of the internodal connections. Although this method requires more computational work than color blending, an examination of contour surfaces

provides unique information. The image in Figure 9(c) was constructed from a combination of color blending and contour lines. This figure shows the spatial distribution of APD from the same 2-D simulation depicted in Figure 9(b). Nodes in the 2-D mesh were drawn as a 3-D plot using z=APD(x,y). Color blending was introduced to demonstrate how APD varied in the simulation. APD was maximal (dark) near the region of the stimulus used to initiate action potential propagation. APD was minimal (light) near the edges of the mesh. APD varied markedly in the two regions of the mesh where we noted nonuniform activation in the activation map, Figure 9(b). Contour lines for APD were drawn in the (x,y,0) plane.

To understand the evolution of calculations from the simulation, we animate through isosurfaces from low to high values. A fixed image generated after much testing with animation is shown in Figure 9(d). Here the spatial distribution of AT from a 3-D simulation of action potential propagation is shown. Following a stimulus in the upper right corner of the model wavefronts propagated left and down. The individual contour lines on the wavefront surfaces at the times presented here were generated from triangular elements. By specifying a large number of contour levels and displaying the resultant wavefronts on a frame-by-frame basis, we were able to place 6 wavefronts on a composite image to demonstrate the spatial distribution of AT.

When the goals of the visualization move from understanding the results of a simulation to the generation of publication quality images, it is advantageous to use third-party software packages. To bridge the differences between image file formats used by different vendors and software packages, we have found the Image Conversion toolkit from the San Diego Supercomputer Center to be very useful. This toolkit is designed to translate bit-mapped images between a number of different workstation, personal computer and graphics programming language formats. Once quality images have been constructed, it is straightforward to annotate the images with third-party drawing packages. An example of a publication quality image is presented in Figure 11. This image shows the mesh from a 3-D model of the human torso used for solutions of the forward and inverse problems in cardiology [31].

8 Conclusions

As we have shown, large scale biomedical computing has moved beyond the conventional tools and methods that have dominated modeling over the past decade. A greater fraction of the investigator's time is being spent developing or identifying tools to perform the necessary subtasks of constructing the mesh of the solution domain, solving the system of equations, optimizing the algorithm and visualizing the results. This forces the investigator to keep abreast of the most recent advances in these areas.

Some of these tools already exist in many supercomputer environments. Commercial FDM, FEM, and BEM packages can be found on many supercomputers. While these packages may not be ideal for the solution phase of all biomedical problems, the pre-processors may be used for some mesh constructions. Libraries of direct and iterative solvers optimized for peak performance are

TORSO MODEL (Johnson et al, 1992)

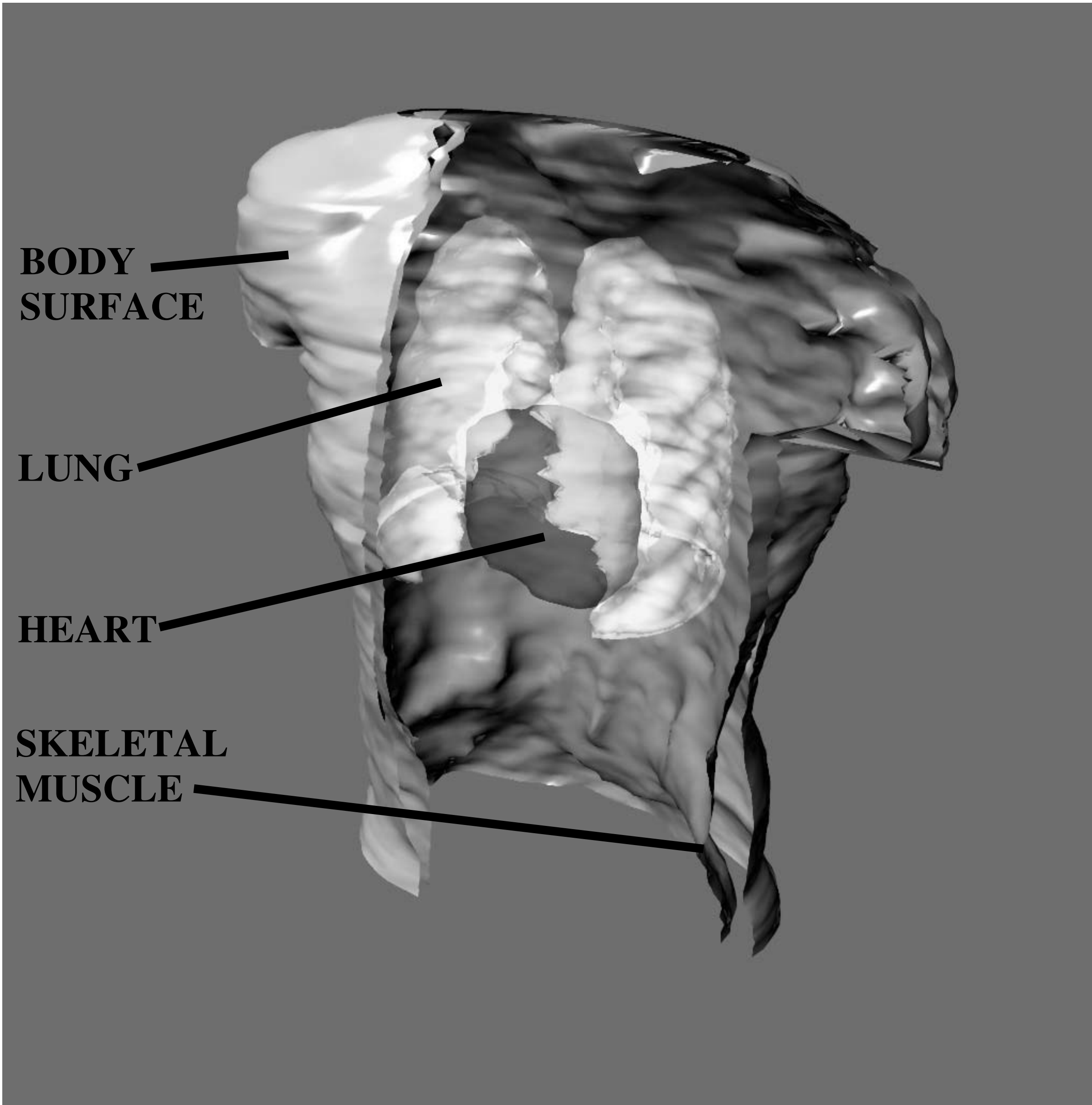


Figure 11: Example of a publication quality image. The image shows a cutaway of major components in a Finite Element model used to solve the forward and inverse problems in electrocardiology. This image was generated on an IBM/RS/6000 and archived in the Utah Raster Graphics file format. The image was then translated to a Silicon Graphics file format and imported into Showcase for annotation.

TORSO MODEL (Johnson et al, 1992)

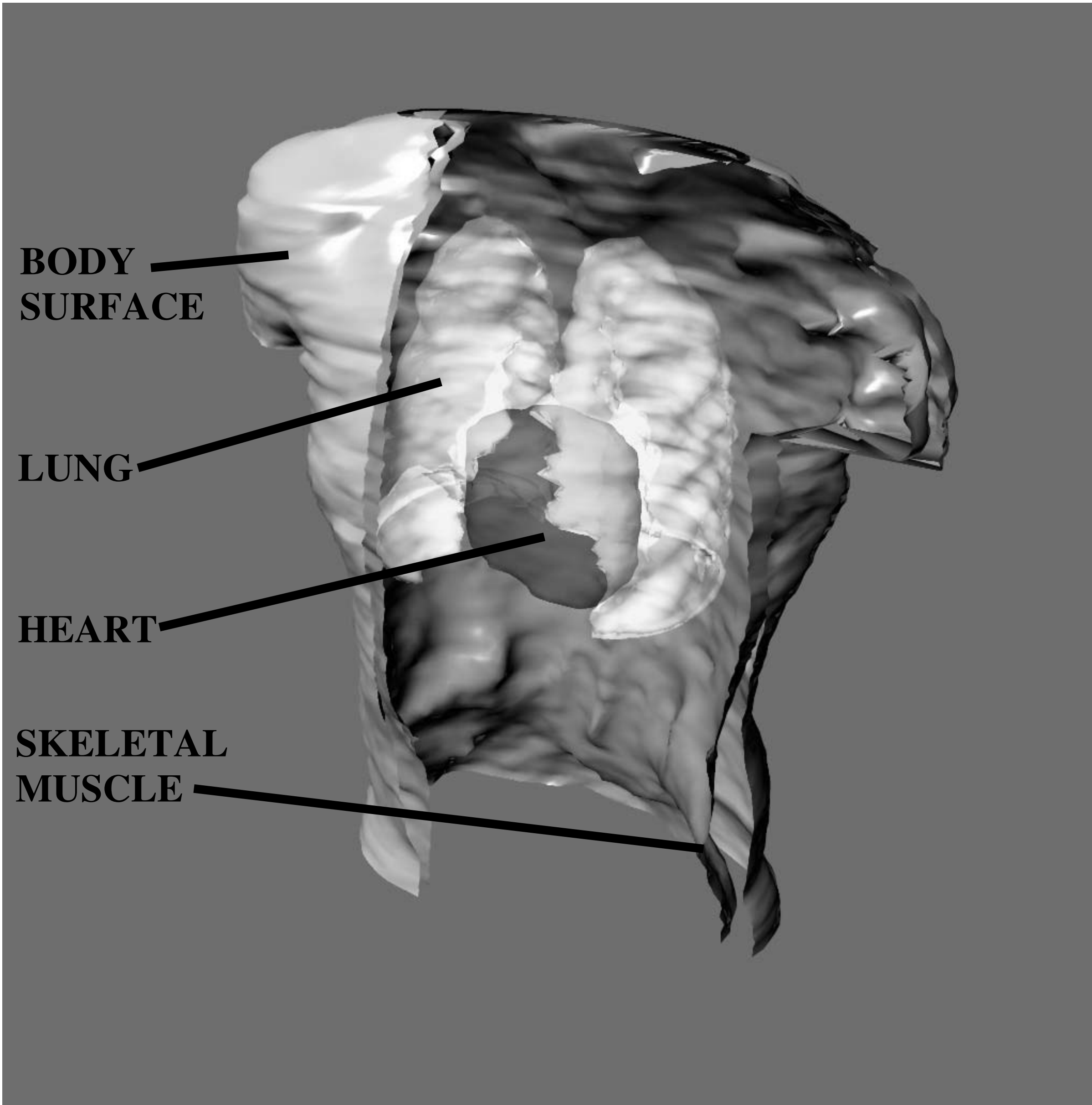


Figure 11: Example of a publication quality image. The image shows a cutaway of major components in a Finite Element model used to solve the forward and inverse problems in electrocardiology. This image was generated on an IBM/RS/6000 and archived in the Utah Raster Graphics file format. The image was then translated to a Silicon Graphics file format and imported into Showcase for annotation.

usually available on most platforms. Performance analysis tools are commonplace and their use can significantly enhance the process of identifying performance inhibitors like memory conflicts, I/O bottlenecks and poorly vectorized subroutines. Visualization tools provide the user with a comfortable interface to manipulate relatively large data sets and extract information about the spatial and temporal behavior from each simulation.

In many cases, the unique features of a given problem or the implementation of a given algorithm may fall outside the capability of existing tools. For example, the commercial general purpose mesh generators are not well suited for the complex three dimensional geometries expected for biomedical problems unless the investigator is willing to construct a mesh point by point. This tedious process prohibits the use of any scheme of adaptive mesh refinement in which the solution phase and mesh construction phase must be directly linked. For some problems, the system of equations resulting from discretization can not always be efficiently solved with library routines. The modeler is then left with the difficult task of implementing and optimizing the solver for the supercomputer. While the performance analysis tool will help identify trouble areas, the process still requires experience and a consideration of the architecture. FDM, FEM, and BEM algorithms can involve often many thousands of lines of code. The programmers of such algorithms must be cognizant not only of program constructions but also of the organization and readability. Poor readability and organization can not only significantly slow the debugging process but can make future modifications extremely difficult. Because large scale models often involve a large volume of data, the modeler must also consider an efficient data management scheme as an integral part of the algorithm. Schemes to compress and reduce data within the program will enable monitoring the behavior over the entire domain for multiple time steps and generate smaller output files for storage on local devices. Better planning in the algorithm development stage can lead to fewer costly simulations. Finally, many of the commercial module based visualization tools were written to accommodate a wide class of problems. The large overhead limits the ability to manipulate and render large meshes. This becomes increasingly difficult when the visualization tool is not resident on the display computer but must be accessed remotely via a network.

Although the supercomputer architecture will undoubtedly change over the next few years, the tasks outlined in this chapter for large scale simulation and modeling will remain essentially the same. However, if the past is any indicator, there is little guarantee that all the tools needed to perform these tasks for a particular for biomedical problem will be in place on all computers. Developing these tools de novo requires a wide range of skills. Consequently, as large scale modeling becomes the norm, we expect to find fewer bioengineers working alone; rather, the scope of the problems will demand the collaborative, inter-disciplinary effort of a "renaissance teams" of experts with a broad array of knowledge and skills in the clinical applications, the underlying physiology, the numerical analysis, programming, and visualization.

Acknowledgements

This research was supported in part by awards from the Nora Eccles Treadwell Foundation and the Richard A. and Nora Eccles Harrison Fund for Cardiovascular Research, U.S. Public Health Service Grants HL47505 and HL 34288, by the Whitaker Foundation, by the Medical Research Counsil of Canada and the Ministere de l'Enseignement Superior et de la Science du Quebec, by grants for computing resources from the Utah Supercomputing Institute, which is funded by the State of Utah and the IBM Corporation, from the North Carolina Supercomputing Center, which is funded in part by the state of North Carolina and Cray Research Inc, and from the Pittsburgh Supercomputing Center, which is funded by the National Science Foundation.

References

[1] L.M. Adams, "Iterative algorithms for large sparse linear systems on parallel computers," NASA CR-166027, NASA Langley Research Center, Hampton, VA., November 1982.

[2] J.E. Akin, *Finite Element Analysis for Undergraduates*, Academic Press, New York, 1986.

[3] A. Baumgartner and C. Mattheck, "Computer simulation of the remodelling of trabecular bone," in *Computers in Biomedicine*, K.D. Held, C.A. Brebbia, and R.D. Ciskowski (Eds.), Computational Mechanics Publications, Southampton, Boston, pp. 291-296, 1991.

[4] E.B. Becker and G.F. Carey, *Finite Elements: Mathematical Aspects*, Prentice-Hall, New Jersey, 1984.

[5] O. Betrand, 3D finite element method in brain electrical activity studies, in *Biomagnetic Localization and 3D Modeling*, J. Nenonen, H.M. Rajala, and T. Katila (Eds.), Helsinki University of Technology, Helsinki, pp. 154-171, 1991.

[6] J.A. Bettess, "Economical solution technique for boundary integral matrices," *International Journal for Numerical Methods in Engineering*, **19**, pp. 1073-1077, 1983.

[7] J.A. Board, "Grand Challenges in Biomedical Computing," in *High Performace Comuting in Biomedical Research*, CRC Press, Boca Raton, FL, 1992.

[8] M. Boulos, C. Oddou, J. Ohayon, and B. Crozatier, "Numerical model for the left ventricle pump: From the exitation of the fibers to the contraction of the whole heart," in Proc. of Ann. Int. Conf. IEEE-EMBS, Vol. 13, No. 5, pp. 2051-2053, 1991.

[9] A. Bowyer, "Computing Dirichlet Tesselations," *Computer J.*, **24**, pp. 162-166, 1981.

[10] C.A. Brebbia and J. Dominguez, *Boundary Elements, An Introductory Course*, Computational Mechanics Publications, Boston/McGraw-Hill, New York, 1989.

[11] W.L. Briggs *A Multigrid Tutorial*, Siam, 1987.

[12] P.G. Ciarlet and J.L. Lions, *Handbook of Numerical Analysis: Volume I, Finite Difference Methods, and Volume II, Finite Element Methods*, North-Holland, Amsterdam, 1991.

[13] Cray Research Inc., CFT Optimization Guide, SG-0115 1/88, 1988.

[14] Cray Research Inc., BNCHLIB Benchmarking Library and Utilities, 1990.

[15] J.M. Crotty, "A block equation solver for large unsymmetric matrices arising in the boundary integral method," *International Journal for Numerical Methods in Engineering*, **18**, pp. 997-1017, 1982.

[16] E. Cuthill and J. McKee, "Reducing the bandwith of sparse symmetric matrices," *ACM Proc. of the 24th Natl. Conf.*, 1969.

[17] P.J. Davis and P. Rabinowitz, *Methods of Numerical Integration*, Academic Press, 1984.

[18] C. Dong and R. Skalak, "A finite element model of white blood cells," in *Computers in Biomedicine*, K.D. Held, C.A. Brebbia, and R.D. Ciskowski (Eds.), Computational Mechanics Publications, Southampton, Boston, pp. 169-179, 1991.

[19] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM, Philadelphia, 1991.

[20] I.S. Duff and A.M. Erisman, *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford, 1986.

[21] R.S. Gallagher and P.J. Selker, "Three-dimensional visualization in FE analysis," *Mechanical Engineering*, **114**, pp. 54-57,1992.

[22] L.W. Gardenhire, "Data redundancy reduction for biomedical telemetry," in Biomedical Telemetry, C.A. Caceres (Ed.), Academic Press, New York, Chapter 11, pp. 255-298, 1965.

[23] R.T. Hart and D.A. Dulitz, "Calculations of the natural frequencies for a human femur," in *Computers in Biomedicine*, K.D. Held, C.A. Brebbia, and R.D. Ciskowski (Eds.), Computational Mechanics Publications, Southampton, Boston, pp. 253-258, 1991.

[24] S.R.H. Hoole, *Computer-Aided Analysis and Design of Eletromagnetic Devices*, Elsevier, New York, 1989.

[25] T.J.R. Hughes, *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Prentice-Hall, New Jersey, 1987.

[26] N. Ida and W. Lord, "Solution of linear equations for small computer systems," *International Journal for Numerical Methods in Engineering*, **20**, pp. 625-641, 1984.

[27] J.D. Jackson, *Classical Electrodynamics*, John Wiley, 1975.

[28] M.A. Jaswon and G.T. Symm, *Integral Equation Methods in Potential Theory and Elastostatics*, Academic Press, London, 1977.

[29] C.R. Johnson and R.S. MacLeod, Mathematical modeling of bioelectric fields, in *Neural Engineering*, Y.I. Kim and N.V. Thakor, eds., Springer-Verlag, New York, 1992 (to appear).

[30] C.R. Johnson, R.S. MacLeod, and P.R. Ershler, A computer model for the study of electrical current flow in the human thorax, *Comp. in Bio. Med.*, 1992 (to appear).

[31] C.R. Johnson, R.S. MacLeod, and M.A. Matheson, Computer simulations reveal complexity of electrical activity in the human thorax, *Comp. in Phys.*, pp. 230-237, May/June, 1992.

[32] C.R. Johnson and R.S. MacLeod, "Nonuniform spatial mesh adaption using *a posteriori* error estimates: applications to direct and inverse problems," in *Adaptive Methods for Partial Differential Equations,* J.E. Flaherty and M.S. Shephard (Eds.), Elsevier Science, New York, 1992 (to appear).

[33] C. Johnson, *Numerical Solution of Partial Differential Equations by the Finite Element Method*, Cambridge University Press, Cambridge, 1990.

[34] J. Joubert and T. Manteuffel, "Iterative methods for non-symmetric linear systems," In *Iterative Systems for Large Linear Systems*, D. Kincaid and L. Hayes (Eds.), Academic Press. pp. 149-172, 1990.

[35] H. Kardestuncer (Ed.), *The Finite Element Handbook*, McGraw-Hill, New York, 1987.

[36] D.R. Kincaid and J.R. Respess, "ITPACK 2C: a FORTRAN package for solving large sparse linear systems by adaptive iterative methods," Center for Numerical Analysis, University of Texas:1-20, 1991.

[37] S. Krucinski, I. Vesely, M.A. Dokainish, and G. Campbell, "On mathematical modelling of heart valve function," in Proc. of Ann. Int. Conf. IEEE-EMBS, Vol. 13, No. 5, pp. 1986-1988, 1991.

[38] H.T. Kung, "The structure of parallel algorithms," in *Advances in Computers*, Academic Press, New York, vol. 19, pp. 65-112, 1980.

[39] C. L. Lawson, "Software for C1 surface interpolation," in *Mathematical Software II*, J.R. Rice, ed., Academic Press, New York, pp. 161-194, 1977.

[40] J.M. Levesque and J.W. Williamson, *A Guidebook to Fortran on Supercomputers*, Academic Press, San Diego, 1989.

[41] C.G. Lewis, D.J. Leone, and M.D. Nowak, "Finite element analysis and mechanical verification of an orthotropic femoral implant model," in *Computers in Biomedicine*, K.D. Held, C.A. Brebbia, and R.D. Ciskowski (Eds.), Computational Mechanics Publications, Southampton, Boston, pp. 231-242, 1991.

[42] G.M. Luo, S.C. Cowin, and A.M. Sadegh, "A boundary element method investigation of different frictional boundary conditions on bone ingrowth," in *Computers in Biomedicine*, K.D. Held, C.A. Brebbia, and R.D. Ciskowski (Eds.), Computational Mechanics Publications, Southampton, Boston, pp. 218-230, 1991.

[43] R.S. MacLeod, C.R. Johnson, and M.A. Matheson, "Visualization tools for computational electrocardiography," *Visualization in Biomedical Computing,* 1992, (to appear).

[44] C. Mattheck and H. Huber-Betzer "CAO: Computer simulation of adaptive growth in bones and trees," in *Computers in Biomedicine*, K.D. Held, C.A. Brebbia, and R.D. Ciskowski (Eds.), Computational Mechanics Publications, Southampton, Boston, pp. 243-252, 1991.

[45] R. Mills, "Finite element modelers: friendly faces for FEA," *Comp. Aided Eng.*, **11**, pp. 36-54, 1992.

[46] A.R. Mitchell and D.F. Griffiths, *The Finite Difference Method in Parital Differential Equations*, John Wiley and Sons, New York, 1980.

[47] J.M. Ortega and R.G. Voigt, "Solution of partial differential equations on vector computers," *SIAM Review*, **27**(2), pp. 149-240, 1985.

[48] J.M. Ortega, *Numerical Analysis: A Second Course*, SIAM, Philadelphia, 1990.

[49] S. Ortloff, H.M. Tensi, H. Gese, and G. Wynarsky, "The effect of porous coating and material properties on micromotion of a cementless femoral total hip replacement," in Proc. of Ann. Int. Conf. IEEE-EMBS, Vol. 13, No. 5, pp. 1976-1977, 1991.

[50] A.E. Pollard and R.C. Barr, "Computer simulations of activation in an anatomically-based model of the human ventricular conduction system," *IEEE Trans. Biomed. Eng.*, **38**, pp. 982-96, 1991.

[51] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes*, Cambridge University Press, Cambridge, 1987.

[52] G. Rodrique (Ed.), *Parallel Computations*, Academic Press, New York, 1982.

[53] M. Salama, S. Utku, and R. Melosh, "Parallel solution of finite element equations," *Proc. of the Eighth ASCE Conf. on Electronic Computation*, University of Houston, Houston TX., pp. 526-539, Feb., 1983.

[54] A.H. Sameh, "Numerical Parallel Algorithms – a survey," in *High Speed Computer and Algorithm Orgainization*, Academic Press, New York, pp. 217-228, 1977.

[55] I.M. Singer and J.A. Thorpe, *Lecture Notes on Elementary Topology and Geometry*, Springer-Verlag, New York, 1967.

[56] C.W. Steele, *Numerical Computation of Electric and Magnetic Fields*, Van Nostrand Reinhold, 1987.

[57] J.C. Strikwerda, *Finite Difference Schemes and Partial Differential Equations*, Wadsworth and Brooks/Cole, California, 1989.

[58] C.C. Vesier, R.A. Levine, and A.P. Yoganathan, "Simulation of blood flow in the left ventricle: The effect of papillary muscle geometry on mitral valve function," in *Computers in Biomedicine*, K.D. Held, C.A. Brebbia, and R.D. Ciskowski (Eds.), Computational Mechanics Publications, Southampton, Boston, pp. 144-156, 1991.