Coarse-Grained Parallel Matrix-Free Solution of a Three-Dimensional Elliptic Prototype Problem

Kevin P. Allen* and Matthias K. Gobbert

Department of Mathematics and Statistics,
University of Maryland, Baltimore County,
1000 Hilltop Circle, Baltimore, MD 21250, U.S.A.
{kallen1,gobbert}@math.umbc.edu

Abstract. The finite difference discretization of the Poisson equation in three dimensions results in a large, sparse, and highly structured system of linear equations. This prototype problem is used to analyze the performance of the parallel linear solver on coarse-grained clusters of workstations. The conjugate gradient method with a matrix-free implementation of the matrix-vector product with the system matrix is shown to be optimal with respect to memory usage and runtime performance. Parallel performance studies confirm that speedup can be obtained. When only an ethernet interconnect is available, best performance is limited to up to 4 processors, since the conjugate gradient method involves several communications per iteration. Using a high performance Myrinet interconnect, excellent speedup is possible for at least up to 32 processors. These results justify the use of this linear solver as the computational kernel for the time-stepping in a system of reaction-diffusion equations.

1 Introduction

This paper considers the Poisson equation

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned} \tag{1}$$

on a three-dimensional domain $\Omega \subset \mathbb{R}^3$ with a prescribed right-hand side function $f(x, y, z)$. Here, $\partial\Omega$ denotes the boundary of the domain Ω. The boundary condition $u = 0$ defines a homogeneous Dirichlet condition. Equation (1) is a model for heat flow, chemical diffusion, fluid flow, and other systems [11].

Many methods exist for the numerical solution of the elliptic problem (1), including finite difference and finite element methods as well as specialized solvers like Fast Poisson Solvers [7]. We will consider a finite difference discretization here using a conventional seven-point stencil in three dimensions. The resulting linear system of equations for the discrete unknowns is solved by the conjugate gradient method. The convergence of the method is assured, because the system matrix is guaranteed to be symmetric positive definite [2, 5]. The system matrix

* This work is part of the first author's undergraduate research.

is moreover sparse and highly-structured. We will materially use these properties to code an efficient matrix-free implementation of the matrix-vector product that is the most expensive part of the conjugate gradient method.

The reason why the solution of problem (1) with an iterative method like conjugate gradients is of interest can be explained as follows: This problem provides a prototype problem for the type of linear system the needs to be solved at each time step of a time-dependent parabolic problem like the heat equation or more general systems of reaction-diffusion equations. For every type of spatial discretization, finite difference or finite element, one has to solve a linear system with a large, sparse, highly-structured system matrix. This matrix has the same sparsity structure as the one arrived by discretizing the prototype problem (1). Hence, studying the efficiency of the linear solver applied to the linear system resulting from the prototype problem is of vital interest, because it will form the computational kernel of the time-dependent solver in the future.

Specifically, the application of interest concerns the modeling of calcium waves in human heart cells that control the heart beat [8–10]. The time-evolution of the concentration of the calcium ions is modeled by a system of coupled nonlinear reaction-diffusion equations involving n_s reactive chemical species. The model developed in [8–10] describes this evolution of the concentrations of calcium $c^{(0)}$ and of several indicator and buffer species $c^{(1)}$, ..., $c^{(n_s-1)}$ by

$$\frac{\partial c^{(0)}}{\partial t} - \nabla \cdot \left(D^{(0)} \nabla c^{(0)} \right) = \left(\sum_{j=1}^{n_s-1} R^{(j)}(c^{(0)}, c^{(j)}) \right) + R^{(0)}(c^{(0)}) + \sigma(c^{(0)}), \quad (2)$$

$$\frac{\partial c^{(i)}}{\partial t} - \nabla \cdot \left(D^{(i)} \nabla c^{(i)} \right) = R^{(i)}(c^{(0)}, c^{(i)}), \quad i = 1, \ldots, n_s - 1, \quad (3)$$

with the reaction terms $R^{(i)}(c^{(0)}, c^{(i)}) = -k_i^f c^{(0)} c^{(i)} + k_i^b \left(\bar{c}_i - c^{(i)} \right)$ for $i = 1, \ldots, n_s - 1$, and the short-hand notation $R^{(0)}(c^{(0)}) = -J^{(pump)}(c^{(0)}) + J^{(leak)}$. The term $\sigma(c^{(0)})$ models the release of the calcium from the calcium release units throughout the cell. The domain $\Omega \subset \mathbb{R}^3$ is the interior of the cell and is reasonably modeled as a parallelepiped.

In [5, 6], a specialized finite element method is developed that takes advantage of the constant coefficients in the diffusion terms of the system (2)–(3) and the regular shape of the domain. Using a semi-implicit time-discretization with lagged reaction terms yields a de-coupled linear system of equations for each of the n_s partial differential equations in (2)–(3). The choice of a uniform mesh and the constant coefficients allow for an efficient matrix-free implementation of the matrix-vector product required in each step of the iterative method. This design has allowed the solution of problems with millions of degrees of freedom, because the system matrix is not stored at all. The de-coupling of the n_s linear systems has been used to design a parallel code using n_s parallel processors [5, 6]. The parallelization of the linear solve for the prototype problem presented in this paper will apply to the linear solve in each of the n_s equations. In this way, the coarse-grainedness of each linear solve can be leveraged to make efficient use of large numbers of processors.

Results for three-dimensional simulations for the prototype problem (1) are presented here, extending earlier studies for a two-dimensional problem [1]. Performance studies show that speedup can be obtained on coarse-grained clusters of commodity workstations. When only an ethernet interconnect is available, best performance is limited to up to 4 processors, however. To obtain excellent speedup for at least up to 32 processors, a high performance interconnect like a Myrinet connection is necessary. These observations reflect the fact that the conjugate gradient method necessarily involves 4 communications per iteration and therefore requires a tight coupling of the cluster. These results validate the use of the method as computational kernel for the application problem (2)–(3).

Section 2 summarizes the numerical method used and explains the implementation of the matrix-free linear solve. Section 3 briefly summarizes serial validation studies and then presents timing results of parallel performance studies on two clusters of workstations. Section 4 summarizes our conclusions.

2 Numerical Method

2.1 The Prototype Problem

Choose the three-dimensional domain $\Omega = (0,1) \times (0,1) \times (0,1)$ and the prescribed function

$$\begin{aligned} f(x,y,z) = & -2\pi^2 \cos(2\pi x) \sin^2(\pi y) \sin^2(\pi z) \\ & -2\pi^2 \sin^2(\pi x) \cos(2\pi y) \sin^2(\pi z) \\ & -2\pi^2 \sin^2(\pi x) \sin^2(\pi y) \cos(2\pi z). \end{aligned}$$

This problem is chosen such that it admits the known solution

$$u(x,y,z) = \sin^2(\pi x) \sin^2(\pi y) \sin^2(\pi z) \tag{4}$$

to conveniently check for correctness of the numerical solution.

2.2 Finite Difference Approximation

To solve the Poisson equation numerically, the domain of the problem must be discretized by a finite mesh. Define the distance between two mesh points $h = 1/(N+1)$, where N is a chosen positive integer. From this, the three-dimensional mesh can be defined as $\Omega_h = \{(x_i, y_j, z_k) : x_i = ih, i = 0, \ldots, N+1, y_j = jh, j = 0, \ldots, N+1, z_k = kh, k = 0, \ldots, N+1\}$. Let the the approximating function be denoted as $u_h : \Omega_h \to \mathbb{R}$, then we will write for short $u_{ijk} := u_h(x_i, y_j, z_k)$ at the mesh points $(x_i, y_j, z_k) \in \Omega_h$. We use a second-order finite difference approximation to approximate the derivatives at each interior point (x_i, y_j, z_k), $i = 1, \ldots, N$, $j = 1, \ldots, N$, $k = 1, \ldots, N$. For the x-derivative, the approximation reads

$$\frac{\partial^2 u}{\partial x^2}(x_i, y_j, z_k) \approx \frac{u(x_{i-1}, y_j, z_k) - 2u(x_i, y_j, z_k) + u(x_{i+1}, y_j, z_k)}{h^2}. \tag{5}$$

An analogous approximation of y-derivative $\frac{\partial^2 u}{\partial y^2}(x_i, y_j, z_k)$ and the z-derivative $\frac{\partial^2 u}{\partial z^2}(x_i, y_j, z_k)$ can be derived. Using the notation $u_{ijk} \approx u(x_i, y_j, z_k)$, we can apply (5) to the original equation (1) to obtain

$$-u_{i-1jk} - u_{ij-1k} - u_{ijk-1} + 6u_{ij} - u_{i+1jk} - u_{ij+1k} - u_{ijk+1} = h^2 f_{ij}, \quad (6)$$
$$1 \le i, j, k \le N,$$
$$u_{0jk} = u_{i0k} = u_{ij0} = u_{N+1jk} = u_{iN+1k} = u_{ijN+1} = 0. \quad (7)$$

The finite difference error is given by $e_h := u - u_h$ between the solution $u(x, y, z)$ and the approximation $u_h(x, y, z)$ on the grid points $(x, y, z) \in \Omega_h$. Define the ℓ_∞-norm of the error $e_{ijk} := e_h(x_i, y_j, z_k)$ by $\|e_h\|_\infty = \max_{(i,j,k)} |e_{ijk}|$. Since $u \in C^4(\Omega) \cap C^0(\bar{\Omega})$ and the domain is simply connected, piecewise smooth, and convex, standard theory guarantees that this error converges as $\|e_h\|_\infty \le C\, h^2$ for h sufficiently small, where C is a constant independent of h [4, 7].

Equations (6)–(7) give a system of $n := N^3$ equations with n unknowns. The boundary conditions are not included in this system since they are given and need not be solved for. To construct a linear system $AU = b$ for the $n = N^3$ unknowns, we must order them in one-dimensional form; we use the natural ordering given by the index transformation $\ell = i + (j-1)N + (k-1)N^2$ to define the one-dimensional vector of unknowns $U = (U_\ell) \in \mathbb{R}^n$ with $U_\ell = u_{ijk}$. The right-hand side vector $b = (b_\ell) \in \mathbb{R}^n$ of the linear system for U is given in the same way by $b_\ell = h^2 f_{ijk}$, which includes the factor h^2 for convenience.

The system matrix $A \in \mathbb{R}^{n \times n}$ is a block-tridiagonal matrix of $N \times N$ blocks of submatrices of size $N^2 \times N^2$; each submatrix is in turn a $N \times N$ block matrix of matrices of size $N \times N$. Matrix A possesses exactly seven non-zero diagonal columns; the diagonal values are 6, and the off-diagonal ones are -1. The matrix A is extremely sparse and highly structured.

2.3 Matrix-Free Implementation

The most expensive operation in each iteration of the conjugate gradient or other Krylov subspace methods is the matrix-vector product $V = AU$ of the system matrix $A \in \mathbb{R}^n$ with a vector $U \in \mathbb{R}^n$.

Storing all n^2 elements of $A \in \mathbb{R}^{n \times n}$ explicitly is known as "dense storage mode." Since most of the n^2 elements are 0, this is wasteful both in memory, as a lot of zeros are stored, and in runtime, as many multiplications involving zeros are needlessly computed. Simulations using this storage mode are available in the software package Matlab and can be easily programmed in a source-code language like C.

A common idea is to take advantage of the sparsity of the matrix, i.e., the low percentage of non-zero elements. In this "sparse storage mode," only non-zero elements are stored along with integer index information to indicate the position of the element in the matrix. This reduces the memory requirements for our system matrix to approximately $7n$ and also improves performance, as only multiplications with those elements are computed that are stored explicitly. This implementation is readily available in Matlab.

To reduce the memory usage further, we take advantage of the constant coefficients in pre-determined positions of the system matrix. A function is provided that accepts a vector $U \in \mathbb{R}^n$ as input and returns the vector $V = AU \in \mathbb{R}^n$ as output; each component V_ℓ is computed as summation of the appropriate components of U multiplied with hard-coded coefficients. This matrix-vector multiplication function is inserted at the appropriate place of our implementation of the conjugate gradient method in the programming language C. This technique is known as a "matrix-free implementation," because only the vectors U and V are stored. It is the most efficient approach to memory usage.

3 Results

The conjugate gradient method is implemented in the programming language C using the Message Passing Interface (MPI) standard for communications between parallel processors. This iterative method for the solution of a linear system of equations of dimension n involves 1 matrix-vector product, 2 dot products, and 3 vector updates (saxpy operations) per iteration. Our implementation is optimized with respect to memory usage by storing exactly 4 vectors of length n, the smallest number possible, and by using a function that implements the matrix-vector multiplication in matrix-free form. The method is programmed in generic form, as the problem only enters by the right-hand side as input and by the system matrix contained in the matrix-vector multiplication function.

Each vector $U = (U_\ell)$ of length $n = N^3$ is split across the p processors by cutting its three-dimensional representation $u_{ijk} = U_\ell$ in the z-dimension. This results in N/p 'planes' of xy-data to be stored on each processor. The parallel form of the conjugate gradient algorithm requires necessarily 4 communication operations per iteration: (i) To compute the matrix-vector product, processors need to interchange one xy-plane of data each with their neighboring processors below and above. These communications are implemented with a choice between blocking `MPI_Send`/`MPI_Recv` and nonblocking `MPI_Isend`/`MPI_Irecv` commands. Here, the blocking commands are arranged such that the even-numbered processors send first and then receive, and vice versa for the odd-numbered processors; this setup avoids blocking of the code [3]. (ii) Since the dot products apply to vectors split across the processors and since the results are needed on all processors, a `MPI_Allreduce` operation is required in each of the 2 dot products. The serial part of the dot products, local to each processor, is implemented with a choice to use naive C code or to call BLAS routines.

Memory is the most important limitation to the size of the system we can solve. In the matrix-free method, the 4 vectors of length n make up the bulk of the memory used. Based on this observation, a prediction of $4n$ is almost all of the memory we need to compute the solution with a matrix-free method. For a sparse system, pieces of the system matrix need to be stored. Since our matrix is septadiagonal, we can assume that only 7 vectors of length n are stored for the system matrix. The predicted memory use is then $7n$ added to the matrix-free implementation prediction. Densely storing the system matrix requires a

vector of n^2 elements added onto the matrix-free implementation prediction. We also take into account that a value in double format in C code requires 8 bytes of memory. Table 1 shows the theoretical predictions for how much memory would be used by the C code using the dense, sparse, and the matrix-free implementation, respectively. The sparse prediction is included for comparison with an implementation of this problem using Matlab. For convenience, the column $n = N^3$ displays the number of degrees of freedom for each value of N.

Table 1. Predicted memory usage for the three storage methods in Megabytes (MB).

N	n	dense	sparse	matrix-free
8	512	2.113	0.045	0.016
16	4096	134.350	0.360	0.131
32	32768	N/A	2.884	1.049
64	262144	N/A	23.069	8.389
128	2097152	N/A	184.550	67.109
256	16777216	N/A	N/A	536.870

For our C code, the predicted values for the dense and matrix-free implementations in Table 1 compare well to observed values. An implementation in Matlab's `pcg` using sparse storage shows a large difference between predicted and observed values. For $N = 64$ and 128, memory was observed as 74 and 386 MB, respectively, much larger then the predictions. Observing from the Matlab code for the function `pcg`, several additional double precision vectors of length n are allocated (some actually storing integer values, that Matlab cannot take advantage of), giving some explanation for the disparity in the memory values.

Less than 1 MB of memory is used for a sparse or matrix-free computation compared to 134 MB when we are storing the system matrix for $N = 16$. Doubling the value to $N = 32$, we see that it takes over 8 GB using dense storage. This is not possible to compute this on our machines because we only have 1 GB of memory available for serial computations, so $N = 16$ is the largest system we can accommodate. Thus, we will use the value $N = 16$ in the following comparisons.

Table 2 compares the results of an implementation using sparse storage in Matlab and a matrix-free implementation in C code. The level-1 BLAS routine for the dot product (`DDOT`) was used in the C code. The column labeled $\|r\|_2/\|b\|_2$ shows the relative residual of the final iterate. This value will be smaller than the tolerance 10^{-6} in all convergent cases. The column #iter shows the number of iterations taken for the method to converge. We can see the values are all consistent. The values of the error $\|e_h\|_\infty$ are on the order of $h^2 \approx 3.46 \cdot 10^{-3}$ in agreement with the finite difference theory [4, 7]. We can conclude that the matrix-free implementation is correct for all values of N.

With a matrix-free implementation, systems with larger values of N can be solved quickly and efficiently. The following computations are performed on an 8-processor Beowulf cluster consisting of 4 dual-processor nodes with 1.0 GHz

Table 2. Numerical results for two different implementations of the conjugate gradient method using a system matrix A with $N = 16$, a tolerance of 10^{-6} and a maximum number of iterations of 10000.

	$\|e_h\|_\infty$	$\|r\|_2/\|b\|_2$	#iter	time [sec]
Matlab / sparse	1.1171577890e-02	9.28101896e-07	30	< 1
C / matrix-free	1.1171577890e-02	9.28101896e-07	30	< 1

Intel Pentium III processors (256 KB L2 cache) and 1.0 GB of memory per node. The machines are connected by 100 Mbps switched ethernet. Files are served from a SCSI hard drive on one of the nodes. Table 3 illustrates the convergence of the method for rising values of N. The results shown were obtained

Table 3. Numerical results for $N = 32$, 64, 128 and 256 using the matrix-free C implementation. The tolerance was set at 10^{-6} and the maximum number of iterations was 10000. The results here were computed on 1 process, but are consistent for any number of processes.

N	$\|e_h\|_\infty$	$\|e_h\|_\infty/h^2$	$\|r\|_2/\|b\|_2$	#iter
32	3.0059503665e-03	3.273479949175	8.44621010e-07	61
64	7.7764871534e-04	3.285565822329	9.11316184e-07	120
128	1.9763013098e-04	3.288763009611	9.98223337e-07	243
256	4.9807474692e-05	3.289733895922	9.71334485e-07	493

using 1 processor. Further studies confirmed that the numerical results are consistent for any number of processes. Finite difference method theory predicts that $\|e_h\|_\infty \leq Ch^2$ with a constant C for sufficiently small values of h. Therefore as $\|e_h\|_\infty \to 0$, we should see $\|e_h\|_\infty/h^2 \to C$ for each computation. Table 3 shows that the results are approaching these values. The final relative residuals and the number of iterations once again just illustrate the convergence of the method to the tolerance 10^{-6}.

Tables 4 and 5 show timings on the 8-processor cluster using blocking and non-blocking communication commands, respectively. The timings are for 1, 2, 4, and 8 processes for increasing values of N. The method is timed for $N = 32$, 64, 128, and 256. Timings for $N = 32$ were all valued at less than 1 second and are excluded from the table. The results show that the computations get

Table 4. Timings in seconds using blocking sends and receives for 1, 2, 4, and 8 processes on the 8 processor cluster.

N	1	2	4	8
64	9	5	6	5
128	165	89	77	55
256	2694	1424	788	700

Table 5. Timings in seconds using nonblocking sends and receives for 1, 2, 4, and 8 processes on the 8 processor cluster.

N	1	2	4	8
64	9	7	6	4
128	165	88	74	55
256	2689	1422	793	700

faster as more processors are used, except in the case of $N = 64$ with blocking communication. The differences in timings between blocking and nonblocking communications are less than 5 second in both directions. This does not allow us to conclude that one of the two methods is outperforming the other.

Figures 1 (a) and (b) show the observed speedup for each value of N for blocking and nonblocking communication, respectively. The optimal speedup is illustrated by the dashed lines in the plots. For the largest case $N = 256$, speedup

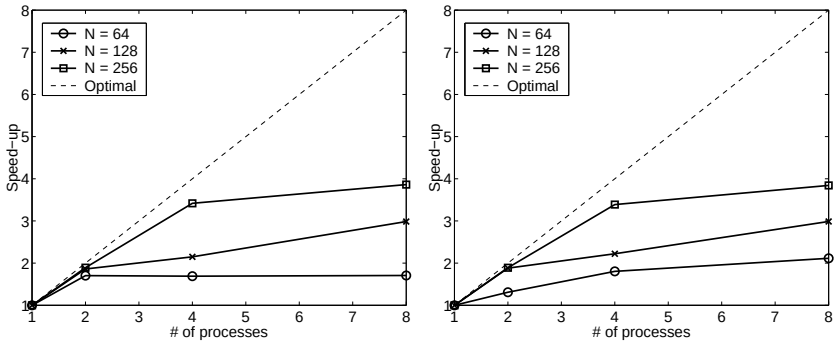

Fig. 1. Plots of the observed speedup of the conjugate gradient method for $N = 64$, 128, and 256 using (a) blocking and (b) nonblocking communication on the 8-processor cluster. The optimal case of linear speedup is shown for comparison.

is good for up to 4 processes. Performance deteriorates beyond 4 processes and for smaller values of N. As illustrated by the timing results, there is very little change in speedup between the two modes of communication. We can conclude that 4 processes performs most efficiently for large N on this cluster with ethernet interconnect.

For comparison, we investigate whether a faster interconnect gives better performance of the method. To this end, we use a 64-processor cluster with a high-performance Myrinet interconnect. The 32 dual-processor nodes have 2.0 GHz Intel Xeon chips (512 KB L2 cache) and 1 GB of memory per node. The matrix-free C code with nonblocking communication and without BLAS is used in the studies on this machine. Timing results on this larger cluster are shown in Table 6. These values are much faster then the previous result due to the faster

Table 6. Timings in seconds for 1, 2, 4, 8, 16, and 32 processes on the 64 processor cluster with Myrinet interconnect.

N	1	2	4	8	16	32
64	4	3	1	< 1	< 1	< 1
128	59	49	25	13	7	4
256	998	787	426	215	111	58

CPU with larger cache. The timings for parallel cases with 4 or more processors all decrease by nearly a factor of 2. However, the timings for 2 processors are not much faster than the 1-processor run. This particular sub-optimal behavior may be caused by some inherent startup cost associated with the use of the interconnect. Figure 2 illustrates the speedup factors for this high performance cluster. This plot demonstrates the close to linear increase in speedup for more

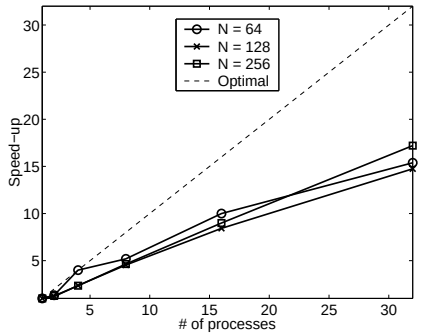

Fig. 2. Plots of the observed speedup of the conjugate gradient method for $N = 64$, 128, and 256 on the 64 processor cluster with a Myrinet interconnect. The optimal case of linear speedup is shown for comparison.

than 2 processors, the only exception being the serial case. For $N = 256$, excellent speedup is displayed for up to the maximum number of processors used. This leads to the conclusion that using 32 processors is efficient with a Myrinet interconnect.

4 Conclusions

Serial comparisons to reliable code verify, that our implementation of the conjugate gradient method computes an accurate solution. The reduction in memory due to the matrix-free implementation not only allows for larger systems, but faster runtime.

The parallel performance studies show that speedup is obtained on both clusters. When only an ethernet interconnect is available, best performance is

limited to up to 4 processors, however. To obtain excellent speedup for at least up to 32 processors, a high performance interconnect like a Myrinet connection is necessary. These observations reflect the fact that the conjugate gradient method necessarily involves 4 communications per iteration and therefore requires a tight coupling of the cluster.

Acknowledgments

The authors acknowledge the support from the University of Maryland, Baltimore County for providing the 8-processor cluster, on which the code was developed and tested. Additionally, we thank Rensselaer Polytechnic Institute for allowing us to use the 64-processor system there.

References

1. K. P. Allen and M. K. Gobbert. A matrix-free conjugate gradient method for cluster computing. Submitted.
2. J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.
3. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, second edition, 1999.
4. C. A. Hall and T. A. Porsching. *Numerical Analysis of Partial Differential Equations*. Prentice-Hall, 1990.
5. A. L. Hanhart. Coarse-grained parallel solution of a three-dimensional model for calcium concentration in human heart cells. M.S. thesis, University of Maryland, Baltimore County, 2002.
6. A. L. Hanhart, M. K. Gobbert, and L. T. Izu. A memory-efficient finite element method for a calcium concentration model in human heart cells. Submitted.
7. A. Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 1996.
8. L. T. Izu, J. R. H. Mauban, C. W. Balke, and W. G. Wier. Large currents generate cardiac Ca^{2+} sparks. *Biophysical Journal*, 80:88–102, 2001.
9. L. T. Izu, W. G. Wier, and C. W. Balke. Theoretical analysis of the Ca^{2+} spark amplitude distribution. *Biophysical Journal*, 75:1144–1162, 1998.
10. L. T. Izu, W. G. Wier, and C. W. Balke. Evolution of cardiac calcium waves from stochastic calcium sparks. *Biophysical Journal*, 80:103–120, 2001.
11. D. S. Watkins. *Fundamentals of Matrix Computations*. Wiley, second edition, 2002.