A Comparative Evaluation of Matlab, Octave, FreeMat, Scilab, R, and IDL on Tara

Ecaterina Coman, Matthew W. Brewster, Sai K. Popuri, and Andrew M. Raim, and Matthias K. Gobbert*

Department of Mathematics and Statistics, University of Maryland, Baltimore County

*Corresponding author: gobbert@umbc.edu, www.math.umbc.edu/~gobbert

Technical Report HPCF–2012–15, www.umbc.edu/hpcf > Publications

Abstract

Matlab is the most popular commercial package for numerical computations in mathematics, statistics, the sciences, engineering, and other fields. IDL, a commercial package used for data analysis, along with the free numerical computational packages Octave, FreeMat, Scilab, and the statistical package R shares many of the same features as Matlab. They are available to download on the Linux, Windows, and Mac OS X operating systems. We investigate whether these packages are viable alternatives to Matlab for uses in research and teaching. We compare the results on the cluster tara in the UMBC High Performance Computing Facility with 86 nodes, each with two quad-core Intel Nehalem processors and 24 GB of memory. The tests focused on usability lead us to conclude that the package Octave is the most compatible with Matlab, since it uses the same syntax and has the native capability of running m-files. FreeMat, Scilab, R, and IDL were hampered by somewhat different syntax or function names and some missing functions. The tests focused on efficiency show that Matlab and Octave are fundamentally able to solve problems of the same size and with equivalent efficiency in absolute times, except in one test dealing with a very large problem. Also IDL performed equivalently in the case of iterative methods. FreeMat, Scilab, and R exhibit significant limitations on the problem size and the efficiency of the problems they can solve in our tests. The syntax of R and IDL are significantly different from that of Matlab, Octave, FreeMat, and Scilab. In summary, we conclude that Octave is the best viable alternative to Matlab because it was not only fully compatible (in terms of syntax) with Matlab in our tests, but it also performed very well.

1 Introduction

1.1 Overview

There are several numerical computational packages that serve as educational tools and are also available for commercial use. Matlab is the most widely used such package. The focus of this study is to introduce four additional numerical computational packages: Octave, FreeMat, Scilab, IDL, and the statistical package R, and provide information on which package is most compatible to Matlab users. Section 1.3 provides more detailed descriptions of these packages. To evaluate Octave, FreeMat, Scilab, R, and IDL, a comparative approach is used based on a Matlab user's perspective. To achieve this task, we perform some basic and

some complex studies on Matlab, Octave, FreeMat, Scilab, R, and IDL. The basic studies include basic operations solving systems of linear equations, computing the eigenvalues and eigenvectors of a matrix, and two-dimensional plotting. The complex studies include direct and iterative solutions of a large sparse system of linear equations resulting from finite difference discretization of an elliptic test problem. This report extends the report [8] by adding the package IDL to the comparison. More precisely, this report is the first one to report on results using the same sparse matrix storage for the system matrix resulting from the Poisson test problem for the iterative method as for Gaussian elimination. In this way, it updates the Matlab, Octave, FreeMat, and Scilab results of [3] and R results of [8] to use sparse storage mode (instead of a matrix-free implementation) in the conjugate gradient experiments, and it provides the comparison to the IDL results of [4]. That report was based on [10, 12], which considered the usability of software packages in a home computer setting, and extends them substantially by more substantive efficiency comparisons.

In Section 2, we perform the basic operations test using Matlab, Octave, FreeMat, Scilab, R, and IDL. This includes testing the backslash operator, computing eigenvalues and eigenvectors, and plotting in two dimensions in all the packages except R and IDL, which follow a significantly different syntax and functions. The backslash operator works identically for all of the packages (except R and IDL) to produce a solution to the linear system given. R uses the command `solve` to produce a solution. In IDL, instead of the backslash operator, the `LUDC` and `LUSOL` commands are used to compute the LU decomposition and use its result to solve the system. The command `eig` has the same functionality in Octave and FreeMat as in Matlab for computing eigenvalues and eigenvectors, whereas Scilab uses the equivalent command `spec`, and R uses the command `eigen` to compute them. IDL required the use of `ELMHES` to reduce the matrix to upper Hessenberg format, and then `HQR` to compute the eigenvalues. Given the eigenvalues, `EIGENVEC` gives the eigenvectors. Plotting is another important feature we analyze by an m-file containing the two-dimensional plot function along with some common annotations commands. Once again, Octave and FreeMat use the exact commands for plotting and annotating as Matlab whereas Scilab, R, and IDL require a few changes. For instance in Scilab, the `pi` command is defined using `%pi` and the command `grid on` from Matlab is replaced with `xgrid` or use the translator to create conversion. To overcome these conversions, we find that we can use the Matlab to Scilab translator, which takes care of these command differences for the most part. The translator is unable to convert the `xlim` command from Matlab to Scilab. To rectify this, we must manually specify the axis boundaries in Scilab using additional commands in `Plot2d`. This issue brings out a major concern that despite the existence of the translator, there are some functions that require manual conversion. Again for R, and IDL, one has to use quite different, but equivalent commands throughout the plotting.

Section 3 introduces a common test problem, given by the Poisson equation with homogeneous Dirichlet boundary conditions, and discusses the finite discretization for the problem in two dimensional space. In the process of finding the solution, we use a direct method, Gaussian elimination, and an iterative method, the conjugate gradient method. To be able to analyze the performance of these methods, we solve the problem on progressively finer meshes. The Gaussian elimination method built into the backslash operator successfully

solves the problem up to a mesh resolution of 4,096 × 4,096 in both Matlab and Octave, while the Gaussian elimination method built into the backslash operator in FreeMat successfully solves the problem up to a mesh resolution of 2,048 × 2,048. However, in Scilab, it is only able to solve up to a mesh resolution of 1,024 × 1,024, and in IDL there does not exist a Gaussian elimination method for sparse matrices without an expensive IDL Analyst license. For this reason, the IDL Gaussian elimination test was not conducted. R does not use the backslash operator for Gaussian elimination. Instead, it uses the `solve` command, which was able to solve up to a mesh resolution of 1,024 × 1,024. The conjugate gradient method is implemented in the `pcg` function, which is stored in Matlab and Octave as a m-file. This function is also available in Scilab as a `sci` file. Again, without the IDL Analyst license, we could not conduct a conjugate gradient test for IDL, but instead applied the biconjugate gradient method available through the command `LINBCG`. The sparse matrix implementation of the iterative method allows us to solve a mesh resolution up to 8,192 × 8,192 in Matlab, Octave, and IDL. Scilab is able to solve the system for a resolution up to 2,048 × 2,048. In FreeMat and R, we wrote our own `cg` functions because they do not have a built in `pcg` function. In FreeMat we were able to solve the system for a resolution up to 1,024 × 1,024 and in R for a resolution up to 4,096 × 4,096. The existence of the `mesh` in Matlab, Octave, and Scilab allows us to include the three-dimensional plots of the numerical solution and the numerical error. However, in FreeMat, `mesh` does not exist and we had to use `plot3`, which results in a significantly different looking plot. In R too, `mesh` does not exist and we had to use the `persp` function. In IDL, the `SURFACE` plots match those of Matlab, Octave, and Scilab. The syntax of Octave is identical to that of Matlab in our tests. We found during our tests that FreeMat lacks a number of functions, such as `kron` for Kronecker products, `pcg` for the conjugate gradient method, and `mesh` for three-dimensional plotting. Otherwise, FreeMat is very much compatible with Matlab. Even though Scilab is designed for Matlab users to smoothly utilize the package and has an m-file translator, it often still requires manual conversions. IDL showed to be most different from Matlab. Many of the commands were either entirely unavailable for our license (`kron`, `cg`) or had to be applied differently (no backslash operator resulted in the use of `LUDC` and `LUSOL`, and the lack of an `eig` function led to the use of `ELMHES`, `HQR`, and `EIGENVEC`). The tests focused on usability lead us to conclude that the packages Octave and FreeMat are most compatible with Matlab, since they use the same syntax and have the native capability of running m-files. Among these two packages, Octave is a significantly more mature software and has significantly more functions available for use.

The results of the numerical experiments on the complex test problem in Section 3 are clearly somewhat surprising. Fundamentally, Matlab, Octave, and IDL turn out to be able to solve problems of the same size, as measured by the mesh resolution of the problem they were able to solve. More precisely, in the tests of the backslash operator, it becomes clear that Matlab is able to solve the problem faster, which points to more sophisticated internal optimization of this operator. However, considering absolute run times, Octave's handling of the problem is acceptable, except for very large cases. What is surprising is that FreeMat and Scilab exhibit significant limitations on the problem size they are able to solve by the backslash operator. The tests using the iterative conjugate gradient method also give

some surprising results. Whether the function for this method is supplied by the package or hand-written, only basic matrix-vector operations are used by the method. Therefore, it is not clear why FreeMat and Scilab are less effective at solving the problems. R too exhibited limitations on the size of problems it could solve. Specifically, R could only solve for resolutions up to 4,096. Matlab, Octave, and IDL took about the same time to solve the problem, thus confirming the conclusions from the other tests. The data that these conclusions are based on are summarized in Tables 3.1, 3.2, 3.3, 3.4, 3.5, and 3.6.

The next section contains some additional remarks on features of the software packages that might be useful, but go beyond the tests performed in Sections 2 and 3. Sections 1.3 and 1.4 describe the five numerical computational packages in more detail and specify the computing environment used for the computational experiments, respectively.

The essential code to reproduce the results of our computational experiments is posted along with the PDF file of this tech. report on the webpage of the UMBC High Performance Computing Facility at `www.umbc.edu/hpcf` under Publications. The files are posted under directories for each software package, potentially with identical names and/or identical files for several packages. See specific references to files in each applicable section.

1.2 Additional Remarks

1.2.1 Ordinary Differential Equations

One important feature to test would be the ODE solvers in the packages under consideration. For non-stiff ODEs, Matlab has three solvers: `ode113`, `ode23`, and `ode45` implement an Adams-Bashforth-Moulton PECE solver and explicit Runge-Kutta formulas of orders 2 and 4, respectively. For stiff ODEs, Matlab has four ODE solvers: `ode15s`, `ode23s`, `ode23t`, and `ode23tb` implement the numerical differentiation formulas, a Rosenbrock formula, a trapezoidal rule using a "free" interpolant, and an implicit Runge-Kutta formula, respectively.

According to their documentations, Octave and Scilab solve non-stiff ODEs using the Adams methods and stiff equations using the backward differentiation formulas. These are implemented in `lsode` in Octave and `ode` in Scilab. The only ODE solver in FreeMat is `ode45` which solves the initial value problem probably by a Runge-Kutta method.

In R, two packages can be used to solve ODEs: `deSolve` for initial value problems and `bvpSolve` for boundary value problems. Many of the functions in the `deSolve` package use both Adams and Runge-Kutta methods. These functions are implemented based on `FORTAN's LSODA` implementation that switches automatically between stiff and non-stiff systems [13].

With the availability of an IDL Analyst License, IDL also has ODE solvers. For a possibly stiff ODE, `IMSL_ODE` implements the Adams-Gear method. If known that the problem is not stiff, a keyword to this command can use the Runge-Kutta-Verner methods or orders 4 and 6 instead. There also exist outside functions, such as `DDEABM` written by Craig Markwardt using an adaptive Adams-Bashford-Moulton method to solve primarily non-stiff and mildly stiff ODE problems or `RunKut_step` written by Frank Varosi using a fourth order Runge-Kutta method.

It becomes clear that all software packages considered have at least one solver. Matlab, Octave, Scilab, R, and IDL have state-of-the-art variable-order, variable-timestep methods for both non-stiff and stiff ODEs available, with Matlab's implementation being the richest and its stiff solvers being possibly more efficient. FreeMat is clearly significantly weaker than the other packages in that it does not provide a state-of-the-art ODE solver, particularly not for stiff problems.

1.2.2 Parallel Computing

Parallel Computing is a well-established method today. It takes in fact two forms: shared-memory and distributed-memory parallelism.

On multi-core processors or on multi-core compute nodes with shared memory among all computational cores, software such as the numerical computational packages considered here automatically use all available cores, since the underlying libraries such as BLAS, LAPACK, etc. use them; studies for Matlab in [11] demonstrated the effectiveness of using two cores in some but not all cases, but the generally very limited effectiveness of using more than two cores. Since the investigations in [11], the situation has changed such that the number of cores used by Matlab, for instance, cannot even be controlled by the user any more, since that control was never respected by the underlying libraries anyway.[1] Thus, even the 'serial' version for Matlab and the other numerical computational packages considered here are parallel on the shared memory of a compute node, this has at least potential for improved performance, and this feature is included with the basic license fee for Matlab. More recently, Matlab has started to provide built-in support for graphics processing units (GPUs). This is cutting-edge and should give a very significant advantage of Matlab over any other packages.

Still within the first form of parallel computing, Matlab offers the *Parallel Computing Toolbox* for a reasonable, fixed license fee (a few hundred dollars). This toolbox provides commands such as `parfor` as an extension of the `for` loop that allow for the farming out of independent jobs in a master-worker paradigm. This is clearly useful for parameter studies, parametrized by the `for` loop. The `parfor` extension uses all available compute nodes assigned to the Matlab job, thus it does go beyond using one node, but each job is 'serial' and lives only on one node.

The shared-memory parallelism discussed so far limits the size of any one job also that of a worker in a master-worker paradigm, to the memory available on one compute node. The second form of parallelism given by distributed-memory parallelism by contrast pools the memory of all nodes used by a job and thus enables the solution of much larger problems. It clearly has the potential to speed up the execution of jobs over 'serial' jobs, even if they use all computational cores on one node. Matlab offers the *MATLAB Distributed Computing Server*, which allows exactly this kind of parallelism. However, this product requires a substantial additional license fee that — moreover — is not fixed, but scales with the number of compute nodes (in total easily in the many thousands of dollars for a modest number of nodes).

[1] Cleve Moler, plenary talk at the SIAM Annual Meeting 2009, Denver, CO, and ensuing personal communication.

The potential usefulness of distributed-memory parallelism, the historically late appearance of the Matlab product for it, and its very substantial cost has induced researchers to create alternatives. These include for instance pMatlab[2] that can be run on top of either Matlab or other "clones" such as Octave. Particularly for pMatlab, a tutorial documentation and introduction to parallel computing is available [7]. In turn, since this distributed-memory parallel package is known to run in conjunction with Octave, this is one more reason to stir the reader to this numerical computational package to consider as alternative to Matlab.

The statistical software R also supports parallel computing on a distributed system with the addition of optional open-source packages[3]. In particular, the SNOW (Simple Network of Workstations) package provides a master-worker paradigm. Functions such as `clusterCall` (evaluate a function on every worker with identical arguments) and `parApply` (apply a function to the rows or columns of a matrix) are used to distribute computations to workers in a cluster. Another package Rmpi allows MPI communications to be used within R, either in master-worker or single program multiple data (SPMD) paradigms. In addition, R has added packages in recent years [4] that target GPUs too.

1.2.3 Applicability of this Work

Numerical computational packages see usage both in research and in teaching.

In a research context, an individual researcher is often very concerned with the portability of research code and reproducibility of research results obtained by that code. This concern applies over long periods of time, as the researcher changes jobs and affiliations. The software Matlab, while widely available at academic institutions, might not be available at some others. Or even if it is available, it is often limited to a particular computer (as fixed-CPU licenses tend to be cheaper than floating license keys). Freely downloadable packages are an important alternative, since they can be downloaded to the researcher's own desktop for convenience or to any other or to multiple machines for more effective use. The more complex test case in Section 3 is thus designed to give a feel for a research problem. Clearly, the use of alternatives assumes that the user's needs are limited to the basic functionalities of Matlab itself; Matlab does have a very rich set of toolboxes for a large variety of applications or for certain areas with more sophisticated algorithms. If the use of one of them is profitable or integral to the research, the other packages are likely not viable alternatives.

In the teaching context, two types of courses should be distinguished: There are courses in which Matlab is simply used to let the student solve larger problems (e.g., to solve eigenvalue problems with larger matrices than 4×4) or to let the student focus on the application instead of mathematical algebra (e.g., solve linear system quickly and reliably in context of a physics or a biology problem). We assume here that the instructor or the textbook (or its supplement) gives instructions on how to solve the problem using Matlab. The question for the present recommendation is then, whether instructions written for Matlab would be applicable nearly word-for-word in the alternative package. Thus, you would want function

[2] `http://www.ll.mit.edu/mission/isr/pmatlab/pmatlab.html`

[3] `http://cran.r-project.org/web/views/HighPerformanceComputing.html`

[4] `http://gpgpu.org/2009/06/14/r-gpgpu`

names to be the same (e.g., `eig` for the eigenvalue calculation) and syntax to behave the same. But it is not relevant if the underlying numerical methods used by the package are the same as those of Matlab. And due to the small size of problems in a typical teaching context, efficiency differences between Matlab and its alternatives are not particularly crucial.

Another type of course, in which Matlab is used very frequently, is courses on numerical methods. In those courses, the point is to explain — at least for a simpler version — the algorithms actually implemented in the numerical package. It becomes thus somewhat important what the algorithm behind a function actually is, or at least its behavior needs to be the same as the algorithm discussed in class. Very likely in this context, the instructor needs to evaluate himself/herself on a case-by-case basis to see if the desired learning goal of each programming problem is met. We do feel that our conclusions here apply to most homework encountered in typical introductory numerical analysis courses, and the alternatives should work fine to that end. But as the above discussion on ODE solver shows, there are limitations or restrictions that need to be incorporated in the assignments. For instance, to bring out the difference in behavior between stiff and non-stiff ODE solvers, the ones available in Octave are sufficient, even if their function names do not agree with those in Matlab and their underlying methods are not exactly the same; but FreeMat's single ODE solver is not sufficient to conduct the desired computational comparison between methods from two classes of solvers.

1.3 Description of the Packages

1.3.1 Matlab

"MATLAB is a high-level language and interactive environment that enables one to perform computationally intensive tasks faster than with traditional programming languages such as C, C++, and Fortran." The web page of the MathWorks, Inc. at `www.mathworks.com` states that Matlab was originally created by Cleve Moler, a Numerical Analyst in the Computer Science Department at the University of New Mexico. The first intended usage of Matlab, also known as Matrix Laboratory, was to make LINPACK and EISPACK available to students without facing the difficulty of learning to use Fortran. Steve Bangert and Jack Little, along with Cleve Moler, recognized the potential and future of this software, which led to establishment of MathWorks in 1983. As the web page states, the main features of Matlab include high-level language; 2-D/3-D graphics; mathematical functions for various fields; interactive tools for iterative exploration, design, and problem solving; as well as functions for integrating MATLAB-based algorithms with external applications and languages. In addition, Matlab performs the numerical linear algebra computations using for instance Basic Linear Algebra Subroutines (BLAS) and Linear Algebra Package (LAPACK).

1.3.2 Octave

"GNU Octave is a high-level language, primarily intended for numerical computations," as the reference for more information about Octave is `www.octave.org`. This package was developed by John W. Eaton and named after Octave Levenspiel, a professor at Oregon

State University, whose research interest is chemical reaction engineering. At first, it was intended to be used with an undergraduate-level textbook written by James B. Rawlings of the University of Wisconsin-Madison and John W. Ekerdt of the University of Texas. This book was based on chemical reaction design and therefore the prime usage of this software was to solve chemical reactor problems. Due to complexity of other softwares and Octave's interactive interface, it was necessary to redevelop this software to enable the usage beyond solving chemical reactor design problems. The first release of this package, primarily created by John W. Eaton, along with the contribution of other resources such as the users, was on January 4, 1993.

Octave, written in C++ using the Standard Template Library, uses an interpreter to execute the scripting language. It is a free software available for everyone to use and re-distribute with certain restrictions. Similar to Matlab, Octave also uses for instance the LAPACK and BLAS libraries. The syntax of Octave is very similar to Matlab, which allows Matlab users to easily begin adapting to the package. Octave is available to download on different operating systems like Windows, Mac OS X, and Linux. To download Octave go to `http://sourceforge.net/projects/octave`. A unique feature included in this package is that we can create a function by simply entering our code on the command line instead of using the editor.

1.3.3 FreeMat

FreeMat is a numerical computational package designed to be compatible with other numerical packages such as Matlab and Octave. The supported operating systems for FreeMat include Windows, Linux, and Mac OS X. Samit Basu created this program with the hope of constructing a free numerical computational package that is Matlab friendly. The web page of FreeMat at `www.freemat.org`. states that some of features for FreeMat include eigenvalue and singular value decompositions, 2D/3D plotting, parallel processing with MPI, handle-based graphics, function pointers, etc. To download FreeMat, go to `http://sourceforge.net/projects/freemat`.

1.3.4 Scilab

"Scilab is an open source, cross-platform numerical computational package as well as a high-level, numerically oriented programming language." Scilab was written by INRIA, the French National Research Institution, in 1990. The web page for Scilab at `www.scilab.org` states the syntax is largely based on Matlab language. This software is also intended to allow Matlab users to smoothly utilize the package. In order to help in this process, there exists a built in code translator which assists the user in converting their existing Matlab codes into a Scilab code. According to Scilab's web page, the main features of Scilab include hundreds of mathematical functions; high-level programming language; 2-D/3-D visualization; numerical computation; data analysis; and interface with Fortran, C, C++, and Java. Just like Octave, Scilab is also a free software distributed under CeCILL licenses.

Scilab is fully compatible with Linux, Mac OS X, and Windows platforms. Like Octave, the source code is available for usage as well as for editing. To download Scilab go to

www.scilab.org/products/scilab/download Scilab also uses, for instance, the numerical libraries, LAPACK and BLAS. Unlike Octave, the syntax and built-in Scilab functions may not entirely agree with Matlab.

1.3.5 R

R is an open source, cross-platform numerical computational and statistical package as well as a high-level, numerically oriented programming language used for developing statistical software and data analytics. The R programming language is based on the S programming language, which was developed at Bell Laboratories. R was not developed as a Matlab clone, and therefore has a different syntax than the numerical analysis software packages previously introduced. It is part of the GNU project and can be downloaded from www.r-project.org. Add-on packages contributed by the R user community can be downloaded from CRAN (The Comprehensive R Archive Network) at cran.r-project.org.

1.3.6 IDL

IDL (Interactive Data Language) was created by Exelis Visual Information Solutions, formerly ITT Visual Information Solutions. The webpage for Excelis at www.exelisvis.com states that IDL can be used to create meaningful visualizations out of complex numerical data, processing large amounts of data interactively. IDL is an array-oriented, dynamic language with support for common image formats, hierarchical scientific data formats, and .mp4 and .avi video formats. IDL also has functionality for 2D/3D gridding and interpolation, routines for curve and surface fitting, and the capability of performing multi-threaded computations.

IDL can be run on Windows, Mac OS X, Linux, and Solaris platforms. It can be downloaded from the Exelis webpage at http://www.exelisvis.com/language/en-US/Downloads/ProductDownloads.aspx after registering.

1.4 Description of the Computing Environment

The computations for this study are performed using Matlab R2011a, Octave 3.6.2, FreeMat v4.0, Scilab-5.3.1, R 2.13.0, and IDL 8.1 under the Linux operating system Redhat Enterprise Linux 5. The cluster tara in the UMBC High Performance Computing Facility (www.umbc.edu/hpcf) is used to carry out the computations and has a total of 86 nodes, with 82 used for computation. Each node features two quad-core Intel Nehalem X5550 processors (2.66 GHz, 8,192 kB cache per core) with 24 GB of memory.

2 Basic Operations Test

This section examines a collection of examples inspired by some basic mathematics courses. This set of examples was originally developed for Matlab by the Center for Interdisciplinary Research and Consulting (CIRC). More information about CIRC can be found at `www.umbc.edu/circ`. This section focuses on the testing of basic operations using Matlab, Octave, FreeMat, Scilab, R, and IDL. We will (i) first begin by solving a linear system; (ii) then finding eigenvalues and eigenvectors of a square matrix; (iii) and finally 2-D functional plotting from data given in a file and the full annotation of plots from computed data, both of which are also typical basic tasks.

2.1 Basic Operations in Matlab

This section discusses the results obtained using Matlab operations. To run Matlab on the cluster tara, enter `matlab` at the Linux command line. This starts up Matlab with its complete Java desktop interface. Useful options to Matlab on tara include `-nodesktop`, which starts only the command window within the shell, and `-nodisplay`, which disables all graphics output. For complete information on options, use `matlab -h`.

2.1.1 Solving Systems of Equations in Matlab

The first example that we will consider in this section is solving a linear system. Consider the following system of equations:

$$\begin{aligned} -x_2 + x_3 &= 3 \\ x_1 - x_2 - x_3 &= 0 \\ -x_1 - x_3 &= -3 \end{aligned}$$

where the solution to this system $(1, -1, 2)^T$ can be found by row reduction techniques from basic linear algebra courses, referred to by its professional name Gaussian elimination. To solve this system with Matlab, let us express this linear system as a single matrix equation

$$Ax = b, \tag{2.1}$$

where A is a square matrix consisting of the coefficients of the unknowns, x is the vector of unknowns, and b is the right-hand side vector. For this particular system, we have

$$A = \begin{bmatrix} 0 & -1 & 1 \\ 1 & -1 & -1 \\ -1 & 0 & -1 \end{bmatrix}, \quad b = \begin{bmatrix} 3 \\ 0 \\ -3 \end{bmatrix}.$$

To find a solution for this system in Matlab, left divide (2.1) by A to obtain $x = A \backslash b$. Hence, Matlab use the backslash operator to solve this system. First, the matrix A and vector b are entered using the following:

```
A = [0 -1 1; 1 -1 -1; -1 0 -1]
b = [3;0;-3].
```

Then use the backslash operator to solve the system by Gaussian elimination by `x = A\b`. The resulting vector which is assigned to x is

```
x =
    1
   -1
    2
```

which agrees with the known exact solution.

2.1.2 Calculating Eigenvalues and Eigenvectors in Matlab

Here, we will consider another important function: computing eigenvalues and eigenvectors. Finding the eigenvalues and eigenvectors is a concept first introduced in a basic Linear Algebra course and we will begin by recalling the definition. Let $A \in \mathbb{C}^{n\times n}$ and $v \in \mathbb{C}^n$. A vector v is called the eigenvector of A if $v \neq 0$ and Av is a multiple of v; that is, there exists a $\lambda \in \mathbb{C}$ such that

$$Av = \lambda v$$

where λ is the eigenvalue of A associated with the eigenvector v. We will use Matlab to compute the eigenvalues and a set of eigenvectors of a square matrix. Let us consider a matrix

$$A = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$$

which is a small matrix that we can easily compute the eigenvalues to check our results. Calculating the eigenvalues using $\det(A - \lambda I) = 0$ gives $1 + i$ and $1 - i$. Now we will use Matlab's built in function `eig` to compute the eigenvalues. First enter the matrix A and then calculate the eigenvalues using the following commands:

```
A = [1 -1; 1 1];
v = eig(A)
```

The following are the eigenvalues that are obtained for matrix A using the commands stated above:

```
v =
           1.0000 + 1.0000i
           1.0000 - 1.0000i
```

To check if the components of this vector are identical to the analytic eigenvalues, we can compute

```
v - [1+i;1-i]
```

and it results in

```
ans =
      0
      0
```

This demonstrates that the numerically computed eigenvalues have in fact the exact integer values for the real and imaginary parts, but Matlab formats the output for general real numbers.

In order to calculate the eigenvectors in Matlab, we will still use the `eig` function by slightly modifying it to `[P,D] = eig(A)` where P will contain the eigenvectors of the square matrix A and D is the diagonal matrix containing the eigenvalues on its diagonals. In this case, the solution is:

```
P =
    0.7071            0.7071
         0 - 0.7071i       0 + 0.7071i
```

and

```
D =
     1.0000 + 1.0000i   0
                   0   1.0000 - 1.0000i
```

It is the columns of the matrix P that are the eigenvectors and the corresponding diagonal entries of D that are the eigenvalues, so we can summarize the eigenpairs as

$$\left(1+i, \begin{bmatrix} 0.7071 \\ 0-0.7071i \end{bmatrix}\right), \qquad \left(1-i, \begin{bmatrix} 0.7071 \\ 0+0.7071i \end{bmatrix}\right).$$

Calculating the eigenvector enables us to express the matrix A as

$$A = PDP^{-1} \tag{2.2}$$

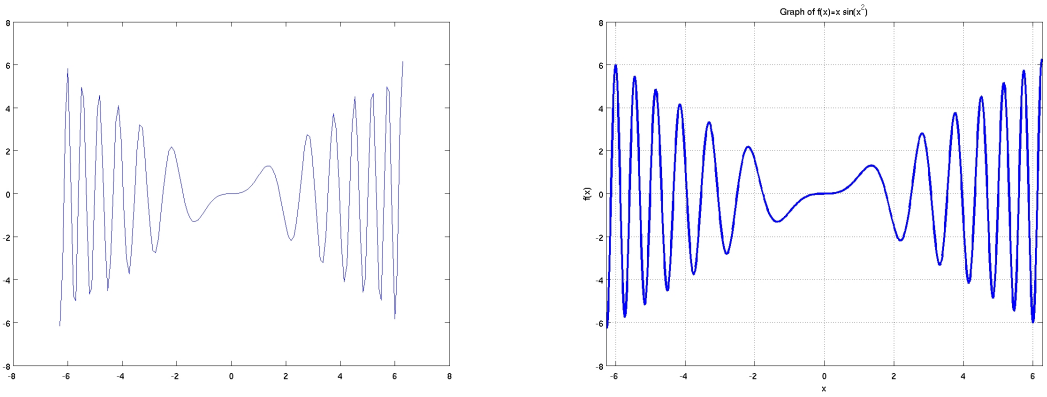
where P is the matrix of eigenvectors and D is a diagonal matrix as stated above. To check our solution, we will multiply the matrices generated using `eig(A)` to reproduce A as suggested in (2.2).

```
A = P*D*inv(P)
```

produces

```
A=
    1   -1
    1    1
```

where `inv(P)` is used to obtain the inverse of matrix P. Notice that the commands above lead to the expected solution, A.



(a) (b)

Figure 2.1: Plots of $f(x) = x\sin(x^2)$ in Matlab using (a) 129 and (b) 1025 equally spaced data points.

2.1.3 2-D Plotting from a Data File in Matlab

2-D plotting is a very important feature as it appears in all mathematical courses. Since this is a very commonly used feature, let us examine the 2-D plotting feature of Matlab by plotting $f(x) = x\sin(x^2)$ over the interval $[-2\pi, 2\pi]$. The data set for this function is given in the data file `matlabdata.dat` and is posted along with the tech. report at `www.umbc.edu/hpcf` under Publications. Noticing that the data is given in two columns, we will first store the data in a matrix A. Second, we will create two vectors, x and y, by extracting the data from the columns of A. Lastly, we will plot the data.

```
A =  load ('matlabdata.dat');
x = A(:,1);
y = A(:,2);
plot(x,y)
```

The commands stated above result in the Figure 2.1 (a). Looking at this figure, it can be noted that our axes are not labeled; there are no grid lines; and the peaks of the curves are rather coarse.

2.1.4 Annotated Plotting from Computed Data in Matlab

The title, grid lines, and axes labels can be easily created. Let us begin by labeling the axes using `xlabel('x')` to label the x-axis and `ylabel('f(x)')` to label the y-axis. `grid on` can be used to create the grid lines. Let us also create a title for this graph using `title ('Graph of f(x)=x sin(x^2)')`. We have taken care of the missing annotations, so let us try to improve the coarseness of the peaks in Figure 2.1 (a). We use `length(x)` to determine that 129 data points were used to create the graph of $f(x)$ in Figure 2.1 (a). To improve this outcome, we can begin by improving our resolution using

```
x = [-2*pi : 4*pi/1024 : 2*pi];
```

to create a vector 1025 equally spaced data points over the interval $[-2\pi, 2\pi]$. In order to create vector y consisting of corresponding y values, use

```
y = x .* sin(x.^2);
```

where `.*` performs element-wise multiplication and `.^` corresponds to element-wise array power. Then, simply use `plot(x,y)` to plot the data. Use the annotation techniques mentioned earlier to annotate the plot. In addition to the other annotations, use `xlim([-2*pi 2*pi])` to set limit is for the x-axis. We can change the line width to 2 by `plot(x,y,'LineWidth',2)`. Finally, Figure 2.1 (b) is the resulting figure with higher resolution as well as the annotations. Observe that by controlling the resolution in Figure 2.1 (b), we have created a smoother plot of the function $f(x)$. The Matlab code used to create the annotated figure is as follows:

```
x = [-2*pi : 4*pi/1024 : 2*pi];
y = x.*sin(x.^2);
H = plot(x,y);
set(H,'LineWidth',2)
axis on
grid on
title ('Graph of f(x)=x sin(x^2)
xlabel ('x')
ylabel ('f(x)')
xlim ([-2*pi 2*pi])
```

Here we will look at a basic example of Matlab programming using a script file. Let's try to plot Figure 2.1 (b) using a script file called `plotxsinx.m`. The extension `.m` indicates to Matlab that this is an executable m-file. Instead of typing multiple commands in Matlab, we will collect these commands into this script. The result is posted in the file `plotxsinx.m` along with the tech. report at `www.umbc.edu/hpcf` under Publications. Now, call `plotxsinx` (without the extension) on the command-line to execute it and create the plot with the annotations for $f(x) = x\sin(x^2)$. The plot obtained in this case is Figure 2.1 (b). This plot can be printed to a graphics file using the command

```
print -djpeg file_name_here.jpg
```

2.2 Basic Operations in Octave

In this section, we will perform the basic operations on Octave. To run Octave on the cluster tara, enter `octave` at the command line. For more information on Octave and available options, use `man octave` and `octave -h`.

2.2.1 Solving Systems of Equations in Octave

Let us begin by solving a system of linear equations. Just like Matlab, Octave defines the backslash operator to solve equations of the form $Ax = b$. Hence, the system of equations mentioned in Section 2.1.1 can also be solved in Octave using the same commands:

```
A = [0 -1 1; 1 -1 -1; -1 0 -1];
b = [3;0;-3];
x= A\b
```

which results in

```
x =
    1
   -1
    2
```

Clearly the solution is exactly what was expected. Hence, the process of solving the system of equations is identical to Matlab.

2.2.2 Calculating Eigenvalues and Eigenvectors in Octave

Now, let us consider the second operation of finding eigenvalues and eigenvectors. To find the eigenvalues and eigenvectors for matrix A stated in Section 2.1.2, we will use Octave's built in function `eig` and obtain the following result:

```
v =
    1 + 1i
    1 - 1i
```

This shows exactly the integer values for the real and imaginary parts. To calculate the corresponding eigenvectors, use `[P,D] = eig(A)` and obtain:

```
P =
   0.70711 + 0.00000i   0.70711 - 0.00000i
   0.00000 - 0.70711i   0.00000 + 0.70711i

D =
   1 + 1i        0
        0   1 - 1i
```

After comparing this to the outcome generated by Matlab, we can conclude that the solutions are same but they are formatted slightly differently. For instance, matrix P displays an extra decimal place when generated by Octave. The eigenvalues in Octave are reported exactly the same as the calculated solution, where as Matlab displays them using four decimal places for real and imaginary parts. Hence, the solutions are the same but presented slightly differently from each other. Before moving on, let us determine whether $A = PDP^{-1}$ still holds. Keeping in mind that the results were similar to Matlab's, we can expect this equation to hold true. Let us compute PDP^{-1} by entering `P*D*inv(P)`. Without much surprise, the outcome is

```
ans =
  1  -1
  1   1
```

An important thing to notice here is that to compute the inverse of a matrix, we use the `inv` command. Thus, the commands for computing the eigenvalues, eigenvectors, inverse of a matrix, as well as solving a linear system, are the same for Octave and Matlab.

2.2.3 2-D Plotting from a Data File in Octave

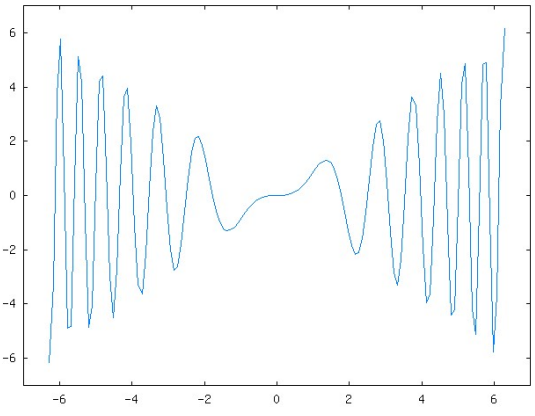
Now, we will look at plotting $f(x) = x\sin(x^2)$ using the given data file. The `load` command is used to store the data in the file into a matrix A. use `x = A(:,1)` to store the first column as vector x and `y = A(:,2)` to store the second column as vector y. We can create a plot using these vectors by entering `plot(x,y)` command in the prompt. Note that to check the number of data points, we can still use the `length` command. It is clear that this process is identical to the process in Section 2.1.3 that was used to generate Figure 2.1 (a). It would not be incorrect to assume that the figure generated in Octave could be identical to Figure 2.1 (a).

Clearly, the Figure 2.2 (a) is not labeled at all; the grid is also not on; as well as the coarseness around the peaks exists. Therefore, the only difference between the two graphs is that in Figure 2.2 (a) the limits of the axes are different than in Figure 2.1 (a). The rest appears to be same in both of the plots.

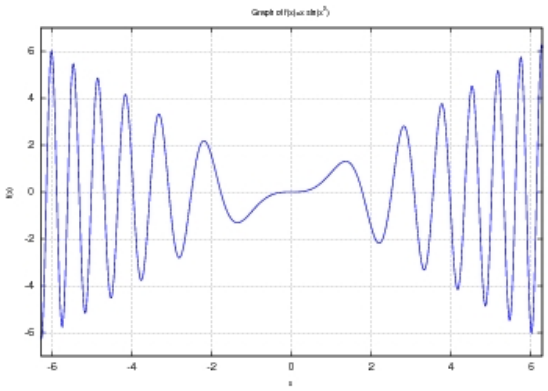
2.2.4 Annotated Plotting from Computed Data in Octave

Let us try to label the axes of this figure using the `label` command and create the title using the `title` command. In order to create a smooth graph, like before; we will consider higher resolution. Hence, `x = [-2*pi : 4*pi/1024 : 2*pi];` can be used to create a vector of 1025 points and `y = x .* sin(x.^2);` creates a vector of corresponding nfunctional values. By examining the creation of the y vector, we notice that in Octave `.*` is known as the "element by element multiplication operator" and `.^` is the "element by element power operator." After using `label` to label the axes, `title` to create a title, and `grid on` to turn on grid, we obtain Figure 2.2 (b).

Clearly, Figure 2.2 (b) and Figure 2.1 (b) are identical. We can simply put together all the commands in a script file exactly as in Matlab and generate the Figure 2.1 (b). This



(a)



(b)

Figure 2.2: Plots of $f(x) = x\sin(x^2)$ in Octave using (a) 129 and (b) 1025 equally spaced data points.

results in the same m-file `plotxsinx.m`, which is posted in the file `plotxsinx.m` along with the tech. report at `www.umbc.edu/hpcf` under Publications. One additional command we can use to print the plot to a graphics file is

```
print -djpeg file_name_here.jpg
```

2.3 Basic Operations in FreeMat

In this section, we perform the basic operations in FreeMat. To run FreeMat on tara, you have to load two modules first by entering `module load qt/4.5.2` and `module load freemat` at the command line. Then start FreeMat by `FreeMat`. A useful option to FreeMat on tara is `-noX`, which starts only the command window within the shell. For complete information on options, use `FreeMat -help`.

2.3.1 Solving Systems of Equations in FreeMat

We will begin by first solving a linear system. Let us consider matrix A as defined in Section 2.1.1. We can use the same commands a Matlab to produce a result.

```
A = [0 -1 1; 1 -1 -1; -1 0 -1];
b = [3;0;-3];
x = A\b
```

which results in

```
x =
    1
   -1
    2
```

as we had expected. Like Matlab and Octave, FreeMat also uses the backslash operator to solve linear systems.

2.3.2 Calculating Eigenvalues and Eigenvectors in FreeMat

Now, we will consider the second important operation, computing eigenvalues and eigenvectors. For our computations, let us use matrix A stated in Section 2.1.2. We will use FreeMat's built in function `eig` and obtain the following result:

```
P =
   0.7071 + 0.0000i   0.7071 - 0.0000i
   0.0000 - 0.7071i   0.0000 + 0.7071i
D =
     1.0000 + 1.0000i   0
                     0   1.0000 - 1.0000i
```

The outcome is identical to Matlab's results. Just to confirm, we compute $A = PDP^{-1}$ which results in the matrix A as following:

```
ans =
  1.0000+0.0000i  -1.0000+0.0000i
  1.0000+0.0000i   1.0000+0.0000i
```

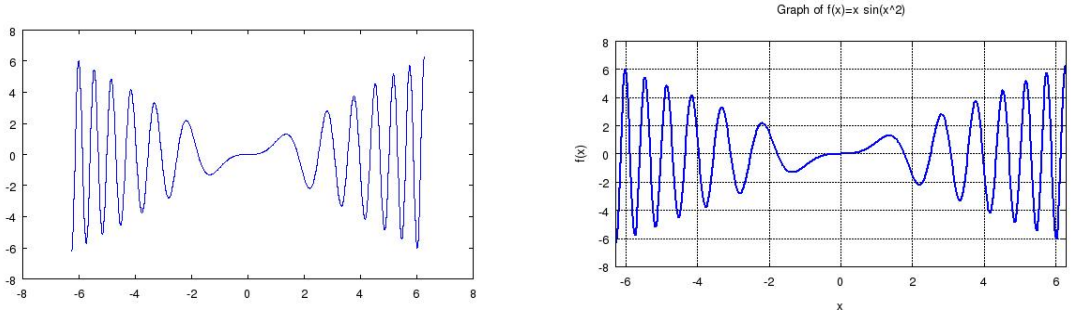
A key point here is that FreeMat uses `inv` to compute inverse of matrices. So the commands used to solve systems of operations, calculate eigenvalues and eigenvectors, and computing matrix inverse are same as Matlab.

2.3.3 2-D Plotting from a Data File in FreeMat

Now we would hope to see an agreement in the plotting and annotation commands. To examine the plotting feature of FreeMat, we will consider $f(x) = x\sin(x^2)$. Let us begin by examining the `load` command. Just like Matlab and Octave, we can load the data in a matrix A with `A = load('matlabdata.dat')` command and use `x = A(:,1)` to create vector x and `y = A(:,2)` to create y. Now, use `plot(x,y)` to generate Figure 2.3 (a) using vector x and y. Clearly, the `load` command and `plot` command have same functionality as in Matlab. Without much surprise, Figure 2.3 (a) and Figure 2.1 (a) are same.

2.3.4 Annotated Plotting from Computed Data in FreeMat

To annotate Figure 2.3 (a), we will use the same commands as Matlab. So to label the axes use `label` command, `grid on` create grid lines, and `title` command to create title. To create a smooth graph, we will create another vector x consisting of more equally spaced data points and a vector y for the corresponding functional values. Use `x = [-2*pi : 4*pi/1024 : 2*pi];` to create x and `y = x .* sin(x.^2);` to create vector y. As in the earlier sections, we hope that higher resolution will improve our plot. Let us plot this data using `plot(x,y);`. Applying the annotation techniques, we generate Figure 2.3 (b). Like in Matlab, we can also



(a) (b)

Figure 2.3: Plots of $f(x) = x\sin(x^2)$ in FreeMat using (a) 129 and (b) 1025 equally spaced data points.

put together these commands in an m-file. This results in the same m-file `plotxsinx.m`, which is posted in the file `plotxsinx.m` along with the tech. report at `www.umbc.edu/hpcf` under Publications. Use

```
print('file_name_here.jpg')
```

to print the plot to a graphics file.

2.4 Basic Operations in Scilab

In this section, we will perform the basic operations in Scilab. To run Scilab on tara, enter `scilab` at the command line, which opens its command window. A useful option to Scilab on tara is `-nogui`, which starts only the command window within the shell. For complete information on options, use `scilab -h`.

2.4.1 Solving Systems of Equations in Scilab

Once again, let us begin by solving the linear system from Section 2.1.1. Scilab follows the same method as Octave and Matlab in solving the system of equations, i.e., it uses the backslash operator to find the solution using the system mentioned in Section 2.1.1, we use the following commands in Scilab:

```
A = [0 -1 1; 1 -1 -1; -1 0 -1];
b = [3;0;-3];
x= A\b
```

to set up the matrix A and vector b. Using the backslash operator, we obtain the result:

```
x =
    1.
   -1.
    2.
```

Once again, the result is exactly what is obtained when solving the system using an augmented matrix.

2.4.2 Calculating Eigenvalues and Eigenvectors in Scilab

Now, let us determine how to calculate the eigenvalues and eigenvectors for the matrix A stated in Section 2.1.2. Scilab uses the `spec` command which has the same functionality as `eig` command to compute eigenvalues. Hence, `v = spec(A)` results in

```
v =
     1. + i
     1. - i
```

Clearly, the outcome is exactly what we had expected but the outcome is formatted slightly different from Matlab. When we calculate the a set of corresponding eigenvectors using `[P,D] = spec(A)` and the following result is obtained:

```
D =
   1 + i        0
         0   1 - 1i
P =
    0.7071068   0.7071068
   -0.7071068i  0.7071068i
```

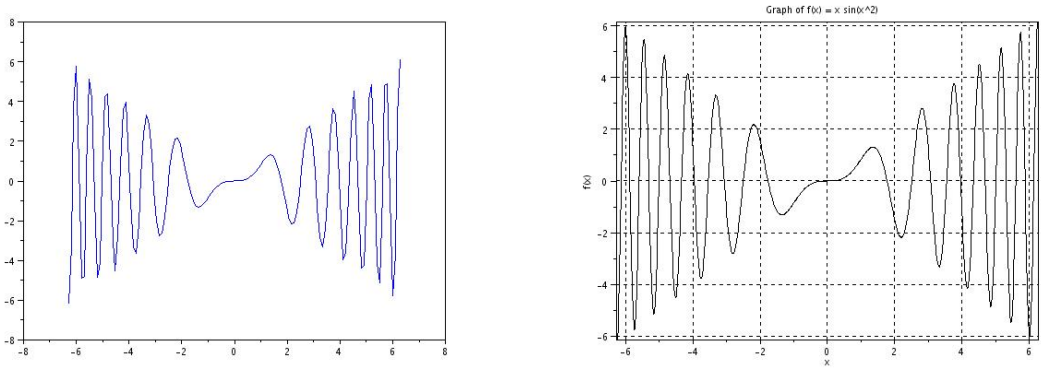
By comparing P, the matrix of eigenvectors computed in Scilab, to P, the matrix in Section 2.1.2, we can see that both packages produce same results but they are formatted differently. Let us check our solution by computing PDP^{-1} using the `inv` command to compute the inverse of the matrix.

```
P*D*inv(P)
ans =
      1. - 1.
      1.   1.
```

which is our initial matrix A. Note that one important factor in computing the eigenvalues and eigenvectors is the command used in these computations, `spec`, and that the eigenvectors found in Scilab and Matlab agree up to six decimal places.

2.4.3 2-D Plotting from a Data File in Scilab

Now, we will plot $f(x) = x\sin(x^2)$ in Scilab. To load the text file `matlabdata.dat` into a matrix, we use the Scilab command `A = fscanfMat('matlabdata.dat')`. This is specifically a command to read text files, while Scilab's `load` command is only for reading binary files; by contrast, Matlab uses `load` for both purposes. Then we use `x = A(:,1)` to store the first column vector as `x` and `y = A(:,2)` to store the second column as a vector `y`. We can create a plot using these vectors via entering `plot(x,y)`. Notice that the Figure 2.4 (a) is not labeled and it is rather coarse.



(a) (b)

Figure 2.4: Plots of $f(x) = x\sin(x^2)$ in Scilab using (a) 129 and (b) 1025 equally spaced data points.

2.4.4 Annotated Plotting from Computed Data in Scilab

Let us improve our resolution by creating vector x using

```
x = [-2*%pi : 4*%pi/1024 : 2*%pi]
```

and let `y = x .* sin(x.^2)` to create a corresponding y vector. Unlike Matlab and Octave, we have to use `%pi` to enter π in Scilab. In addition, `.*` and `.^` are still performing the element-wise operations called the "element-wise operators" in Scilab. Another factor that remains unchanged is the `length` command. We can generate the plot using the `plot(x,y)` command which creates the Figure 2.4 (a). Once again, we can use `xlabel` and `ylabel` to label the axes; title`('Graph of f(x)=x sin(x^2)')` to create a title; and `xgrid` to turn on grid. To plot and create x-axis bounds, use

```
plot2D(x,y,1,'011','',[-2*%pi,y(1),2,*%pi,y($)])
```

Notice that we can put together these commands into a sci-file in Scilab to generate a plot. The resulting script for creating the a plot is as follows:

```
x = -2*%pi:(4*%pi)/1024:2*%pi;
y = x .* sin(x.^2);
plot2d(x,y,1,'011','',[-2*%pi,y(1),2*%pi,y($)])
set(gca(),"axes_visible","on")
set(gca(),"grid",[1,1])
title("Graph of f(x) = x sin(x^2)")
xlabel("x")
ylabel("f(x)")
```

Notice that some of the Matlab commands are not compatible with Scilab. One easier approach to handle this issue is to use the "Matlab to Scilab translator" under the Applications menu or by using `mfile2sci` command. The translator is unable to convert

`xlim([-2*pi 2*pi]);` which we can take care of replacing the `plot` with `plot2d` command stated earlier. The converted code using the translator is referred to as `plotxsinx.sci` and is posted along with the tech. report at `www.umbc.edu/hpcf` under Publications. Using this script file, we obtain Figure 2.4 (b) which is similar to Figure 2.1 (b). To send these graphics to `jpg` file, we can use

```
xs2jpg(gcf(),'file_name_here.jpg')
```

2.5 Basic Operations in R

In this section, we will perform the basic operations in R. To run R on the cluster tara, enter `R` at the Linux command line. This starts up the command window of R. For the complete list of options available with the command `R`, type `R --help` at the Linux command line. One useful command in R is `help()`, which provides a detailed description and usage of commands. For example, to view the usage of the command `source` in R, type `help("source")`. To quit the R interface, type `quit()`, which will ask if you wish to save the workspace you have been working on. Typing `y` will save your command history and all objects created in the session. This will allow you to resume your session next time you open R. Refer to `http://cran.r-project.org/manuals.html` to learn more about programming in R.

2.5.1 Solving Systems of Equations in R

Once again, let us begin by solving the linear system from Section 2.1.1. R has a command called `solve` to solve the linear system of equations $Ax = b$. The command takes two matrices A and b as arguments and returns x as a matrix. To solve the system mentioned in Section 2.1.1, we use the following commands in R:

```
A = array(c(0,1,-1,-1,-1,0,1,-1,-1),c(3,3))
b = c(3, 0, -3)
```

to set up the matrix A and vector b, respectively. The `c()` function is prevalent in R code; it concatenates its arguments into a vector. A vector may be used to enumerate the elements of an array, as we have done here. In R, vectors are not the same as matrices. In order to instruct R to treat the above vector b as a column vector, we use the following command:

```
dim(b) = c(3,1)
```

This command sets the dimension of the vector b to 3 rows and 1 column. We have used the `array()` command to create the A matrix. The first argument to this command is a vector containing the data and the second argument is again a vector containing the number of rows and columns of the matrix we want to create. When creating matrices, R follows the column-ordering scheme by default, that is, the entries in `c(0,1,-1,-1,-1,0,1,-1,-1)` specify the entries of A along the columns; notice that this is different than the treatment of command-line input in Matlab and other packages. Using the `solve` command, we obtain the result

```
solve(A, b)
     [,1]
[1,]    1
[2,]   -1
[3,]    2
```

Once again, the result is exactly what is obtained when solving the system using an augmented matrix.

In R, matrices can also be created using the `matrix` command. The above system can also be solved using the following commands:

```
A <- matrix(c(0,1,-1,-1,-1,0,1,-1,-1), nrow=3)
b <- matrix(c(3,0,-3))
solve(A, b)
     [,1]
[1,]    1
[2,]   -1
[3,]    2
```

Although both the approaches work in this example, it is advisable to use the `matrix` command when working with matrices in R. Also, notice that we have used `=` assignment operator in the first method and `<-` operator in the second method. Both are assignment operators in R. However, `=` cannot be used to assign values inline unless the assignment is a named parameter. In other words, `=` can only be used as part of an independent top level expression or a sub-expression in a list of braced expressions. `<-`, on the other hand, can be used for inline assignments too, but has a potentially undesirable effect of either overwriting an existing variable with the same name in the workspace or creating a new variable with the same name if it does not exist[5]. The common practice is to use `<-` for assignments, `=` for named parameters and to altogether avoid inline assignments. Since the details of the R language are outside the scope of this report, we will not focus on exploring such subtle differences. Interested users should refer to R manuals for language specific details.

2.5.2 Calculating Eigenvalues and Eigenvectors in R

Now, let us determine how to calculate the eigenvalues and eigenvectors for the matrix A stated in Section 2.1.2. R uses the `eigen` command to compute eigenvalues and eigenvectors. The command returns a data structure that holds both eigenvalues and eigenvectors. The return value can be captured into a variable as `eig = eigen(A)`. Eigenvalues can then be accessed as `eig$values` and eigenvectors can be accessed as `eig$vectors`. So, eigenvalues and eigenvectors can be calculated and displayed as

[5]Another assignment operator `<<-` in R assigns values in global scope if the variable does not exist or overwrites the global variable if it exists. Keeping these operator behaviors in mind, one can argue that the `=` operator is probably safer to use. However, we will continue to use `<-` as it is the common practice to assign values in R

```
A <- matrix(c(1,1,-1,1),nrow=2)
eig <- eigen(A)
eig$values
[1] 1+1i 1-1i
eig$vectors
                      [,1]                  [,2]
[1,] 0.7071068+0.0000000i 0.7071068+0.0000000i
[2,] 0.0000000-0.7071068i 0.0000000+0.7071068i
```

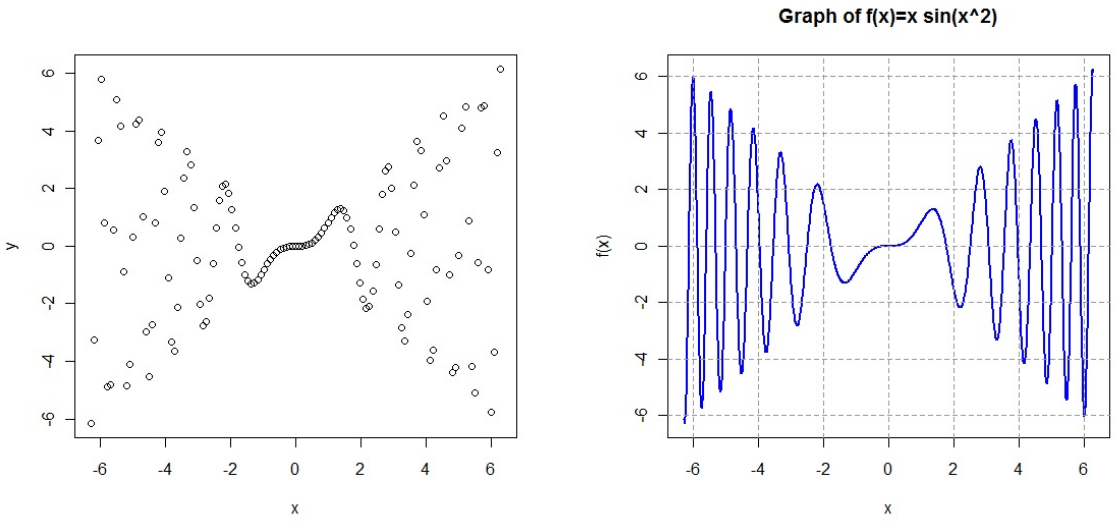
Clearly, the outcome is exactly what we had expected but the format of the outcome is slightly different from that of Matlab. Notice that Matlab's `eig` command returns both eigenvalues and eigenvectors in matrix form, with diagonal matrix D and an invertible matrix P, where D consists of eigenvalues as diagonal elements and P consists of eigenvectors as columns. By contrast, R returns a heterogeneous list data structure encapsulating both eigenvalues and eigenvectors that can be accessed using the operator `$`. It is common practice in R to return composite data structures from functions that need to return multiple values and to use the access operator `$` as shown above to access individual components in the data structure returned. But matrices D and P can be easily formed from these data structures, for example to verify that the diagonalization yields A again:

```
D <- diag(eig$values)
P <- eig$vectors
P %*% D %*% solve(P)
     [,1]  [,2]
[1,] 1+0i -1+0i
[2,] 1+0i  1+0i
```

Notice that the output of PDP^{-1} is same as the matrix A. In the above command, `%*%` is used to perform matrix multiplications and the `solve()` command is used to calculate the inverse of the non-singular matrix P (note that in R, the `solve()` command is also used to solve a linear system when called using two arguments A and b, as shown at the beginning of this section).

2.5.3 2-D Plotting from a Data File in R

Now, we will plot $f(x) = x\sin(x^2)$ in R. To load the text file `matlabdata.dat` into a matrix, we use the R command `pd = read.table('matlabdata.dat')`. There are several ways one can load data into R. Some of the commands provided by R are `read.csv` (to read comma-delimited files), `scan`, etc. R also has database/source specific commands such as `read.dta` (to load Stata binary files), `read.octave` (to load text files saved in the Octave format), etc. We use the command `read.table` to read rectangular data into an R variable. Rectangular data is read into R as data frames, which look like matrices, and their elements can be accessed using `[]` brackets just as with matrices. Below we plot the data using R's `plot` command.



(a) (b)

Figure 2.5: Plots of $f(x) = x\sin(x^2)$ in R using (a) 129 and (b) 1025 equally spaced data points.

```
x <- pd[,1]
y <- pd[,2]
plot(x, y, xlab="x", ylab="y")
dev.new()
```

`dev.new()` instructs R to generate plots in seperate windows and not to overwrite the current plot with future plots (notice how we have used `=` operator for named parameters in the `plot` command). Notice that the Figure 2.5 (a) is not labeled and it is rather coarse. Also notice that by default, R uses circular markers instead of line markers as used in other packages. This behavior can be interpreted as R's distinguishing feature as a statistical package compared to other numerical packages.

2.5.4 Annotated Plotting from Computed Data in R

Let us improve the resolution by creating vector x using

```
x  <- seq(-2*pi, 2*pi, 4*pi/1024)
```

and let `y <- x * sin(x^2)` to create a corresponding y vector. Like Matlab and Octave, we use `pi` for the constant π in R. Unlike Matlab, Octave, and Scilab, we do not need to use `.*` and `.^` operators to perform element-wise operations. Instead, we simply use `*` and `^`.

We now plot using the new `x` and `y` values again using the `plot` command, but with additional arguments:

```
plot(x,y, type="l",xlab="x", ylab="f(x)", col="blue",
```

```
    main="Graph of f(x)=x sin(x^2)", lwd=2)
grid(lwd=1, col="gray60", equilogs=TRUE)
```

The `grid` command above generates a fine grid against the plot. Note that R can handle line breaks in the code, as shown in the plot command above. However, users should ensure that the interpreter knows that the broken line is "to be continued". For example:

```
y <- 1 + 2 + 3 +
    4 + 5
z <- 1 + 2 + 3
    + 4 + 5
```

would correctly assign the value of 15 to `y`, but would assign 6 to `z` which is probably not desired. Notice that the difference between the two assignment statements is that in the first command, the + operator appears at the end of the command, indicating to R that you intend to continue the command to the following line. The above `plot` command generates the Figure 2.5 (b) which is similar to Figure 2.1 (b). Notice that compared to Figure 2.5 (a), Figure 2.5 (b) shows solid line markers in the line plot. This is achieved with the argument `type="l"`. To send these graphics to `jpg` file, we can use

```
savePlot("file_name_here.jpg", type = "jpeg")
```

We can save all the above commands to a text file with extension `.R` or `.r`. The code is posted in the file `plotxsinx.r` along with the tech. report at `www.umbc.edu/hpcf` under Publications. The file can then be executed in R, similar to Matlab's m-files, using the command

```
source("plotxsinx.r")
```

2.6 Basic Operations in IDL

In this section, we will perform the basic operations using IDL. From the command line, enter `idl` to run just the IDL console, or `idlde` to run the full development environment. To enter IDL online help, use `idlhelp` from the Linux command line or `?` at the IDL command line.

2.6.1 Solving Systems of Equations in IDL

We will again begin by solving the linear system $Ax = b$ as in Section 2.1.1. Create the matries in IDL using the commands:

```
A=[[0.0,-1.0,1.0],[1.0,-1.0,-1.0],[-1.0,0.0,-1.0]]
b=[3.0,0.0,-3.0]
```

Unlike Matlab, Octave, FreeMat, and Scilab, IDL does not have a dedicated backslash command to solve systems of linear equations. Instead, IDL uses the `LUDC` procedure along with the `LUSOL` function. The `LUDC` procedure finds the LU decomposition of the matrix of equations and outputs a vector that contains a record of the row permutations. Following the `LUDC` procedure, the `LUSOL` function is called, with inputs including the matrix A, the vector of row permutations, and the column vector b. All matrices must be of type `float` or `double` to run the `LUDC` and `LUSOL` commands, and b must be a column vector to run `LUDC`. The following commands will solve the system of equations:

```
LUDC,A,index
x=LUSOL(A,index,b)
```

Here, `index` is an input vector the LUDC creates which contains a record of the row permutations. These commands give a result

```
1.00000 -1.00000 2.00000
```

which can be written similar to the Matlab output as

```
1.00000
-1.00000
2.00000
```

which is consistent with the manual results and previous computational software results.

2.6.2 Calculating Eigenvalues and Eigenvectors in IDL

Next, let us consider the operation of computing eigenvalues and eigenvectors. We will use the matrix A introduced in Section 2.1.2. We will first use IDL's `ELMHES` function to reduce the matrix A to upper Hessenberg format, after which we will compute the eigenvalues of this new, upper Hessenberg matrix using the IDL function `HQR`. To compute eigenvectors and residuals, we will use the `EIGENVEC` function which takes inputs of the matrix A and the new matrix of eigenvalues, and outputs the residual. The commands to obtain the eigenvalues and eigenvectors are the following:

```
HesA = ELMHES(A)
eval = HQR(hesA)
evec = EIGENVEC(A,eval, /DOUBLE, RESIDUAL=residual)
```

The eigenvalues calculated by `HQR` are

```
(1.00000, -1.00000)(1.00000,1.00000)
```

and the eigenvectors are

```
(0.70710678,1.7677670e-21)(1.7677670e-21,0.70710678)
(-1.7677670e-21,0.70710678)(0.70710678,-1.7677670e-21)
```

IDL outputs complex numbers in parentheses, with the first number being the real part and the second the imaginary part. So, if we consider double-precision floating-point numbers on the scale of 10^{-21} as 0, then the above output specifies the following eigenpairs in standard mathematical notation for complex numbers and rounded to fewer digits:

$$\left(1-i, \begin{bmatrix} 0.7071 \\ 0+0.7071i \end{bmatrix}\right), \qquad \left(1+i, \begin{bmatrix} 0+0.7071i \\ 0.7071 \end{bmatrix}\right).$$

The crucial thing to notice here was that IDL output each eigenvector as a row vector; this is very different in appearance than the output based on a matrix P, whose columns are eigenvector, that was produced by other packages. We notice that the eigenvalues were output in reverse order than by Matlab. Taking this into account and also noting that multiplying by $-i$ shows $(0+0.7071i, 0.7071)^T = (0.7071, 0-0.7071i)^T$, we see that the eigenpairs are the same as those computed by Matlab.

We are able to express the matrix A as PDP^{-1}, where P is the matrix of eigenvectors and D is the diagonal matrix of eigenvalues, to reproduce the matrix A. We used `INVERT(P)` to obtain the inverse of P. This gives us a result of

```
(0.50000000,-0.50000000)(0.50000000,0.50000000)
```

Which is A scaled by .5 and to 8 significant digits.

2.6.3 2-D Plotting from a Data File in IDL

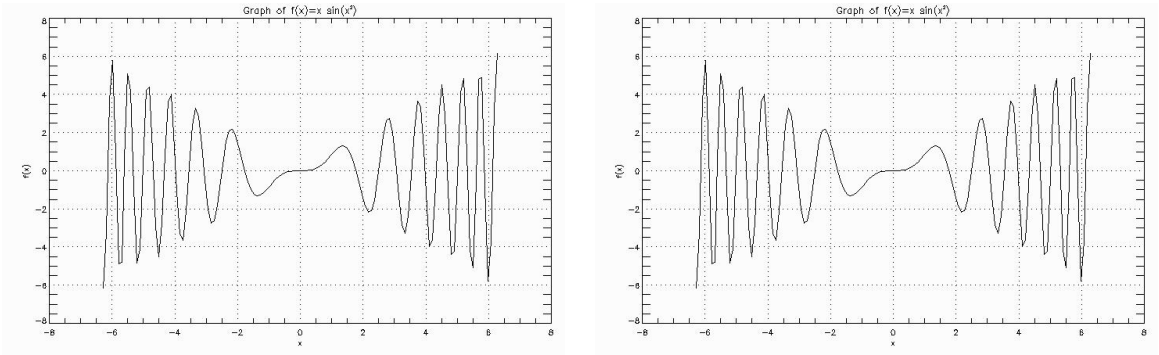
Next, we will examine the similarities between the plotting commands using $f(x) = x\sin(x^2)$. To obtain data from a file, the following commands will get the number of lines in the file and create a two-column double-type array to hold the data, open the file and create a logical unit number which will be associated with that opened file, read the data into the double array, and close the file and free the logical unit number:

```
nlines=FILE_LINES('matlabdata.dat')
data=DBLARR(2,nlines)
openr, lun, 'matlabdata.dat', /get_lun
readf, lun, data
free_lun, lun
```

Using `x=data(0,*)` and `y=data(1,*)` will create vectors x and y The plot procedure, `PLOT, x, y` will then create Figure 2.6 (a) using vectors x and y. These commands have an obvious agreement to the `LOAD` and `PLOT` commands in Matlab as Figure 2.6 (a) is the same as Figure 2.1 (a).

2.6.4 Annotated Plotting from Computed Data in IDL

To annotate this graph, use the `TITLE` command to add a graph title, the `XTITLE` and `YTITLE` commands to add a horizontal and vertical axis title, respectively. To create a grid, we use `XTICKLEN` and `YTICKLEN` and edit the line style using `XGRIDSTYLE` and `YGRIDSTYLE`. Now,



(a) (b)

Figure 2.6: Plots of $f(x) = x\sin(x^2)$ in IDL using (a) 129 and (b) 1025 equally spaced data points.

we want to have a smoother graph made of more equally spaced data points. We create a second vector x using `x=(4*!pi)*FINDGEN(1025)/1024 - (2*!pi)`, where `FINDGEN` creates an array of 1025 values from 0 to 1024, to get a range of numbers from -2π to 2π at intervals of /pi. Using this new x, make the vector y with the command `y=x*sin(x^2)`. This annotated data can be plotted using

```
PLOT, x, y, XSTYLE=1, THICK=2, TITLE='Graph of f(x)=xsin(x!E2!N)', $
   XTITLE='x', YTITLE='f(x)',XTICKLEN=1.0, YTICKLEN=1.0,$
   XGRIDSTYLE=1, YGRIDSTYLE=1
```

The command `XSTYLE=1` makes sure IDL is displaying the exact interval, as it normally wants to produce even tick marks which create a wider range than desired. Also, the command `THICK` changes the thickness of the grid lines and `$` is the line continuation character. This annotated plot procedure generates Figure 2.6 (b). These commands can all be placed into a procedure file, which can run them all at once from the IDL command line. This file, named `plotxsinx.pro`, is posted along with the tech. report at `www.umbc.edu/hcpf` under Publications. To save this plot to a `jpg` file, save the contents of the current Direct Graphics window into a variable and write that variable using

```
image=tvrd()
write_jpeg, 'filename.jpg', image
```

3 Complex Operations Test

3.1 The Test Problem

This section tests the software packages on a classical test problem given by the numerical solution with finite differences for the Poisson problem with homogeneous Dirichlet boundary conditions [1, 5, 6, 14], given as

$$\begin{aligned} -\triangle u &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega. \end{aligned} \tag{3.1}$$

This problem is a popular textbook example, see, among others, [5, 14]. We have studied it with several different emphases in, among others, [1, 3, 8–10, 12].

Here, $\partial\Omega$ denotes the boundary of the domain Ω while the Laplace operator is defined as

$$\triangle u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$$

This partial differential equation can be used to model heat flow, fluid flow, elasticity, and other phenomena [14]. Since $u = 0$ at the boundary in (3.1), we are looking at a homogeneous Dirichlet boundary condition. We consider the problem on the two-dimensional unit square $\Omega = (0, 1) \times (0, 1) \subset \mathbb{R}^2$. Thus, (3.1) can be restated as

$$\begin{aligned} -\tfrac{\partial^2 u}{\partial x^2} - \tfrac{\partial^2 u}{\partial y^2} &= f(x, y) \quad \text{for } 0 < x < 1, \quad 0 < y < 1, \\ u(0, y) = u(x, 0) = u(1, y) = u(x, 1) &= 0 \quad \text{for } 0 < x < 1, \quad 0 < y < 1, \end{aligned} \tag{3.2}$$

where the function f is given by

$$f(x, y) = -2\pi^2 \cos(2\pi x) \sin^2(\pi y) - 2\pi^2 \sin^2(\pi x) \cos(2\pi y).$$

The problem is designed to admit a closed-form solution as the true solution

$$u(x, y) = \sin^2(\pi x) \sin^2(\pi y).$$

3.2 Finite Difference Discretization

Let us define a grid of mesh points $\Omega_h = \{(x_i, y_j) = (ih, jh), i, j = 0, \ldots, N+1\}$ with uniform mesh width $h = \frac{1}{N+1}$. By applying the second-order finite difference approximation to the x-derivative at all the interior points of Ω_h, we obtain

$$\frac{\partial^2 u}{\partial x^2}(x_i, y_i) \approx \frac{u(x_{i-1}, y_j) - 2u(x_i, y_j) + u(x_{i+1}, y_j)}{h^2}. \tag{3.3}$$

If we also apply this to the y-derivative, we obtain

$$\frac{\partial^2 u}{\partial y^2}(x_i, y_i) \approx \frac{u(x_i, y_{j-1}) - 2u(x_i, y_j) + u(x_i, y_{j+1})}{h^2}. \tag{3.4}$$

Now, we can apply (3.3) and (3.4) to (3.2) and obtain

$$\begin{aligned} &- \frac{u(x_{i-1}, y_j) - 2u(x_i, y_j) + u(x_{i+1}, y_j)}{h^2} \\ &- \frac{u(x_i, y_{j-1}) - 2u(x_i, y_j) + u(x_i, y_{j+1})}{h^2} \approx f(x_i, y_j). \end{aligned} \tag{3.5}$$

Hence, we are working with the following equations for the approximation $u_{i,j} \approx u(x_i, y_j)$:

$$\begin{aligned} -u_{i-1,j} - u_{i,j-1} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1} = h^2 f_{i,j} \quad i,j = 1, \ldots, N \\ u_{0,j} = u_{i,0} = u_{N+1,j} = u_{i,N+1} = 0 \end{aligned} \tag{3.6}$$

The equations in (3.6) can be organized into a linear system $Au = b$ of N^2 equations for the approximations $u_{i,j}$. Since we are given the boundary values, we can conclude there are exactly N^2 unknowns. In this linear system, we have

$$A = \begin{bmatrix} S & -I & & \\ -I & S & -I & \\ & \ddots & \ddots & \ddots \\ & & -I & S & -I \\ & & & -I & S \end{bmatrix} \in \mathbb{R}^{N^2 \times N^2},$$

where

$$S = \begin{bmatrix} 4 & -1 & & \\ -1 & 4 & -1 & \\ & \ddots & \ddots & \ddots \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{bmatrix} \in \mathbb{R}^{N \times N} \quad \text{and} \quad I = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \\ & & & & 1 \end{bmatrix} \in \mathbb{R}^{N \times N}$$

and the right-hand side vector $b_k = h^2 f_{i,j}$ where $k = i + (j-1)N$. The matrix A is symmetric and positive definite [5,14]. This implies that the linear system has a unique solution and it guarantees that the iterative conjugate gradient method converges.

To create the matrix A, we use the observation that it is given by a sum of two Kronecker products [5, Section 6.3.3]: Namely, A can be interpreted as the sum

$$A = \begin{bmatrix} T & & & \\ & T & & \\ & & \ddots & \\ & & & T \\ & & & & T \end{bmatrix} + \begin{bmatrix} 2I & -I & & \\ -I & 2I & -I & \\ & \ddots & \ddots & \ddots \\ & & -I & 2I & -I \\ & & & -I & 2I \end{bmatrix} \in \mathbb{R}^{N^2 \times N^2},$$

where

$$T = \begin{bmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & \ddots & \ddots & \ddots \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix} \in \mathbb{R}^{N \times N}$$

and I is the $N \times N$ identity matrix, and each of the matrices in the sum can be computed by Kronecker products involving T and I, so that $A = I \otimes T + T \otimes I$. To store the matrix A efficiently, all packages provide for a sparse storage mode, in which only the non-zero entries are stored. These formulas are the basis for the code in the function `setupA` contained in the codes posted along with the tech. report at `www.umbc.edu/hpcf`.

One of the things to consider to confirm the convergence of the finite difference method is the finite difference error. The finite difference error is defined as the difference between the true solution $u(x, y)$ and the numerical solution u_h defined on the mesh points by $u_h(x_i, y_j) = u_{i,j}$. Since the solution u is sufficiently smooth, we expect the finite difference error to decrease as N gets larger and $h = \frac{1}{N+1}$ gets smaller. Specifically, the finite difference theory predicts that the error will converge like $\|u - u_h\|_{L^\infty(\Omega)} \leq C\, h^2$, as the mesh width h tends to zero $h \to 0$, where C is a constant independent of h [2,6]. For sufficiently small h, we can then expect that the ratio of errors on consecutively refined meshes behaves like

$$\text{Ratio } = \frac{\|u - u_{2h}\|}{\|u - u_h\|} \approx \frac{C\,(2h)^2}{C\,h^2} = 4 \tag{3.7}$$

Thus, we will print this ratio in the following tables in order to confirm convergence of the finite difference method. Here, the appropriate norm for the theory of finite differences is the $L^\infty(\Omega)$ function norm, defined by $\|u - u_h\|_{L^\infty(\Omega)} = \sup_{(x,y)\in\Omega} |u(x, y) - u_h(x, y)|$.

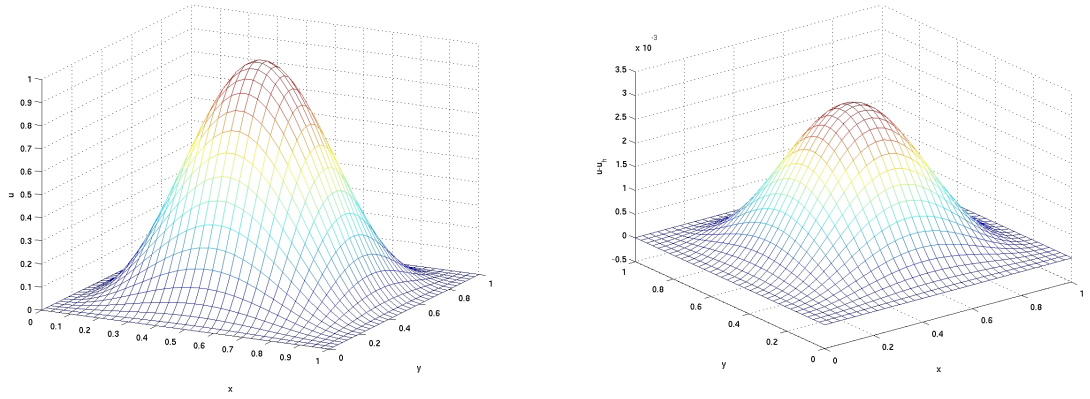
3.3 Matlab Results

3.3.1 Gaussian Elimination

Let us begin solving the linear system arising from the Poisson problem by Gaussian elimination in Matlab. We know that this is easiest approach for solving linear systems for the user of Matlab, although it may not necessarily be the best method for large systems. To create matrix A, we make use of the Kronecker tensor product, as described in Section 3.2. This can be easily implemented in Matlab using the `kron` function. The system is then solved using the backslash operator. Figure 3.1 shows the results of this for a mesh with $N = 32$. Figure 3.1 (a) shows the mesh plot of the numerical solution vs. (x, y). The error at each mesh point is computed by subtracting the numerical solution from the analytical solution and is plotted in Figure 3.1 (b). Notice that the maximum error occurs at the center. The code to solve this system for $N = 32$ and produce the plots is contained in `driver_ge.m`, which is posted along with the tech. report at `www.umbc.edu/hpcf` under Publications.

Table 3.1 (a) shows the results of a study for this problem using Gaussian elimination with mesh resolutions $N = 2^\nu$ for $\nu = 1, 2, 3, \ldots, 13$. The table lists the mesh resolution N, the number of degrees of freedom (DOF) N^2, the norm of the finite difference error $\|u - u_h\|$, the ratio of consecutive error norms (3.7), and the observed wall clock time in HH:MM:SS. To create this table, we use a version of `driver_ge.m` with the graphics commands commented out.

The norms of the finite difference errors clearly go to zero as the mesh resolution N increases. The ratios between error norms for consecutive rows in the table tend to 4 in



(a) (b)

Figure 3.1: Mesh plots for $N = 32$ in Matlab (a) of the numerical solution and (b) of the numerical error.

agreement with (3.7), which confirms that the finite difference method for this problem is second-order convergent with errors behaving like h^2, as predicted by the finite difference theory. By looking at this table, it can be concluded that Gaussian elimination runs out of memory for $N = 8{,}192$. Hence, we are unable to solve this problem for N larger than 4,096 via Gaussian elimination. This leads to the need for another method to solve larger systems. Thus, we will use an iterative method known as the conjugate gradient method to solve this linear system.

3.3.2 Conjugate Gradient Method

Now, we use the conjugate gradient method to solve the Poisson problem [1, 14]. This iterative method is an alternative to using Gaussian elimination to solve a linear system and is accomplished by replacing the backslash operator by a call to the `pcg` function. We use the zero vector as the initial guess and a tolerance of 10^{-6} on the relative residual of the iterates. This is implemented in a Matlab code `driver_cg.m` that is posted along with the tech. report at `www.umbc.edu/hpcf`. This code is equivalent to `driver_ge.m` from above, with the backslash operator replaced by a call to the `pcg` function.

The system matrix A that we use to solve the conjugate gradient method is a sparse matrix and is created using the function `setupA`. We use this method to create the system matrix over others such as the matrix-free method because it can be implemented in every package using the current licenses that are available to us for all packages. This will allow us to use a uniform method across all numerical computation packages and enable us to make accurate evaluations.

Table 3.1 (b) shows results of a study using the conjugate gradient method with this sparse matrix implementation. The column `#iter` lists the number of iterations taken by the iteration method to converge. The finite difference error shows the same behavior as in Table 3.1 (a) with ratios of consecutive errors approaching 4 as for Gaussian elimination;

Table 3.1: Convergence results for the test problem in Matlab using (a) Gaussian elimination and (b) the conjugate gradient method. The tables list the mesh resolution N, the number of degrees of freedom (DOF), the finite difference norm $\|u - u_h\|_{L^\infty(\Omega)}$, the ratio of consecutive errors, and the observed wall clock time in HH:MM:SS.

(a) Gaussian Elimination

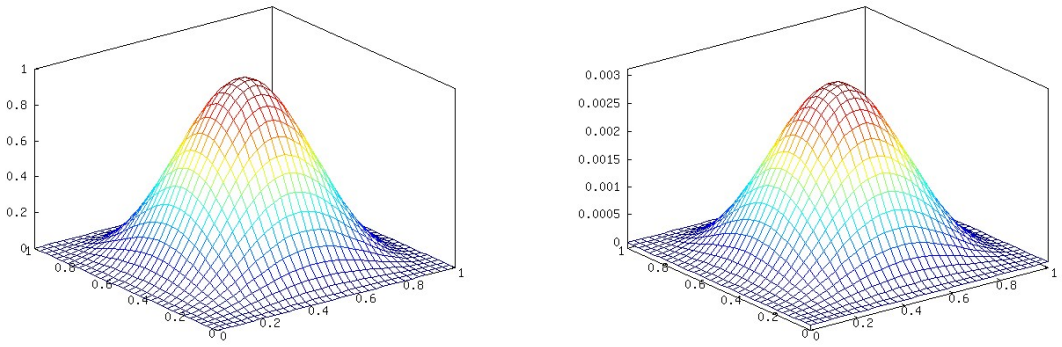
N	DOF	$\|u - u_h\|$	Ratio	Time
32	1,024	3.0128e-3	N/A	<00:00:01
64	4,096	7.7812e-4	3.8719	<00:00:01
128	16,384	1.9766e-4	3.9366	<00:00:01
256	65,536	4.9807e-5	3.9685	<00:00:01
512	262,144	1.2501e-5	3.9843	00:00:01
1,024	1,048,576	3.1313e-6	3.9923	00:00:05
2,048	4,194,304	7.8362e-7	3.9959	00:00:21
4,096	16,777,216	1.9610e-7	3.9960	00:01:39
8,192	out of memory			

(b) Conjugate Gradient Method

N	DOF	$\|u - u_h\|$	Ratio	`#iter`	Time
32	1,024	3.0128e-3	N/A	48	<00:00:01
64	4,096	7.7811e-4	3.8719	96	<00:00:01
128	16,384	1.9765e-4	3.9368	192	<00:00:01
256	65,536	4.9797e-5	3.9690	387	00:00:01
512	262,144	1.2494e-5	3.9856	783	00:00:11
1,024	1,048,576	3.1266e-6	3.9961	1,581	00:01:32
2,048	4,194,304	7.8019e-7	4.0075	3,192	00:13:27
4,096	16,777,216	1.9366e-7	4.0313	6,452	01:49:50
8,192	67,777,216	4.7400e-8	4.0829	13,033	14:28:27

this confirms that the tolerance on the relative residual of the iterates is tight enough. To create this table, we use a version of `driver_cg.m` with the graphics commands commented out.

Tables 3.1 (a) and (b) indicate that Gaussian elimination is faster than the conjugate gradient method in Matlab, whenever it does not run out of memory. For problems greater than 4,096, the results show that the conjugate gradient method is able to solve for larger mesh resolutions.



(a) (b)

Figure 3.2: Mesh plots for $N = 32$ in Octave (a) of the numerical solution and (b) of the numerical error.

3.4 Octave Results

3.4.1 Gaussian Elimination

In this section, we will solve the Poisson problem discussed in Section 3.2 via Gaussian elimination method in Octave. Just like Matlab, we can solve the equation using the backslash operator. Using the same m-file we used in Matlab, `driver_ge.m`, which is posted along with the tech. report at `www.umbc.edu/hpcf` under Publications, we create Figure 3.2 which is identical to Figure 3.1.

The numerical results in Table 3.2 (a) are identical to the results in the Table 3.1 (a), but the timing results show that Matlab is significantly faster than Octave when solving a system of linear equations using the method of Gaussian elimination.

3.4.2 Conjugate Gradient Method

Now, let us try to solve the problem using the conjugate gradient method in Octave. Just like Matlab, there exists a `pcg.m` function. The input requirements for the `pcg` function are identical to the Matlab `pcg` function. Once again, we use the zero vector as the initial guess and the tolerance is 10^{-6} on the relative residual of the iterations. We can use the m-file `driver_cg.m`, which is posted along with the tech. report at `www.umbc.edu/hpcf` under Publications.

Just like in Matlab, Tables 3.2 (a) and (b) indicate that Gaussian elimination is faster than the conjugate gradient method in Octave, whenever it does not run out of memory. For problems greater than 4,096, the results show that the conjugate gradient method is able to solve for larger mesh resolutions.

Comparing Table 3.2 (b) to Table 3.1 (b), we see that the conjugate gradient method in Octave solves a problem slower than Matlab.

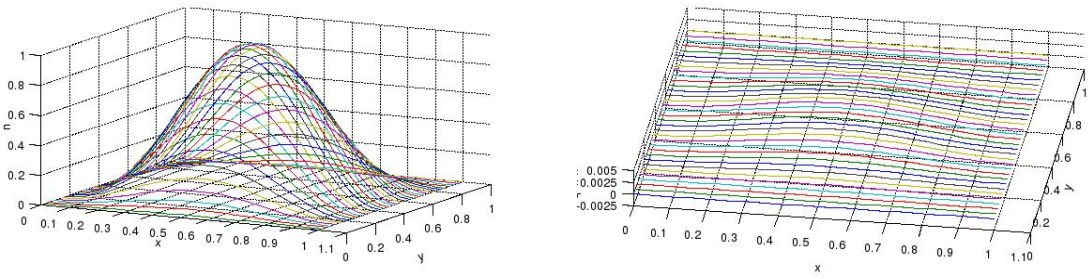
Table 3.2: Convergence results for the test problem in Octave using (a) Gaussian elimination and (b) the conjugate gradient method. The tables list the mesh resolution N, the number of degrees of freedom (DOF), the finite difference norm $\|u - u_h\|_{L^\infty(\Omega)}$, the ratio of consecutive errors, and the observed wall clock time in HH:MM:SS.

(a) Gaussian Elimination

N	DOF	$\|u - u_h\|$	Ratio	Time
32	1,024	3.0128e-3	N/A	<00:00:01
64	4,096	7.7812e-4	3.8719	<00:00:01
128	16,384	1.9766e-4	3.9366	<00:00:01
256	65,536	4.9807e-5	3.9685	<00:00:01
512	262,144	1.2501e-5	3.9843	00:00:02
1,024	1,048,576	3.1313e-6	3.9922	00:00:15
2,048	4,194,304	7.8362e-7	3.9959	00:01:53
4,096	16,777,216	1.9610e-7	3.9960	00:15:50
8,192	out of memory			

(b) Conjugate gradient method

N	DOF	$\|u - u_h\|$	Ratio	`#iter`	Time
32	1,024	3.0128e-3	N/A	48	<00:00:01
64	4,096	7.7811e-4	3.8719	96	<00:00:01
128	16,384	1.9765e-4	3.9368	192	<00:00:01
256	65,536	4.9797e-5	3.9690	387	00:00:02
512	262,144	1.2494e-5	3.9856	783	00:00:14
1,024	1,048,576	3.1266e-6	3.9961	1,581	00:01:56
2,048	4,194,304	7.8019e-7	4.0075	3,192	00:17:50
4,096	16,777,216	1.9354e-7	4.0313	6,452	02:34:29
8,192	67,108,864	4.6775e-8	4.1355	13,033	20:01:27



(a) (b)

Figure 3.3: Plot3 plots for $N = 32$ in FreeMat (a) of the numerical solution and (b) of the numerical error.

3.5 FreeMat Results

3.5.1 Gaussian Elimination

Once again, we will solve the Poisson problem via Gaussian elimination method, but using FreeMat this time. To create the system matrix A using `setupA`, we need to write our own implementation of the `kron` function, since it is not implemeneted in FreeMat. Furthermore, `mesh` does not exist in FreeMat, so we used the `plot3` command to create Figure 3.3 (a) the plot3 of the numerical solution vs. (x, y) and Figure 3.3 (b) the plot3 of the error vs. (x, y). Despite the view adjustments for these figures using the `view` command, Figure 3.3 (b) is still very hard to read. The code to solve this system for $N = 32$ and produce the plots is contained in `driver_ge.m`, which is posted along with the tech. report at `www.umbc.edu/hpcf` under Publications.

The numerical results in Table 3.3 (a) are identical to the results in Tables 3.1 (a) and 3.2 (a) for the mesh resolutions that FreeMat can solve for. However, Gaussian elimination in FreeMat ran significantly slower than Octave, which in turn ran slower than Matlab, in those cases. Moreover, Gaussian elimination in FreeMat could only solve the problem up to $N = 2{,}048$ and ran out of memory for $N = 4{,}096$.

3.5.2 Conjugate Gradient Method

Unlike Matlab and Octave, a `pcg` function does not exist in FreeMat. To address this issue, we wrote our own `cg` function to use in FreeMat. The code to solve this system and produce the plots for $N = 32$ using conjugate gradients is contained in `driver_cg.m`, which is posted along with the tech. report at `www.umbc.edu/hpcf` under Publications. To create the table of results for larger N, we used a version of this function with graphics commands commented out.

We encountered some issues in FreeMat when we were solving for a mesh resolution of $2{,}048 \times 2{,}048$, namely the time it took to solve the problem was excessively long. The numerical results in Table 3.3 (b) are identical to the results in Tables 3.1 (b) and 3.2 (b) for the mesh resolutions that FreeMat can solve for. However, the conjugate gradient method

Table 3.3: Convergence results for the test problem in FreeMat using Gaussian elimination. The tables list the mesh resolution N, the number of degrees of freedom (DOF), the finite difference norm $\|u - u_h\|_{L^\infty(\Omega)}$, the ratio of consecutive errors, and the observed wall clock time in HH:MM:SS.

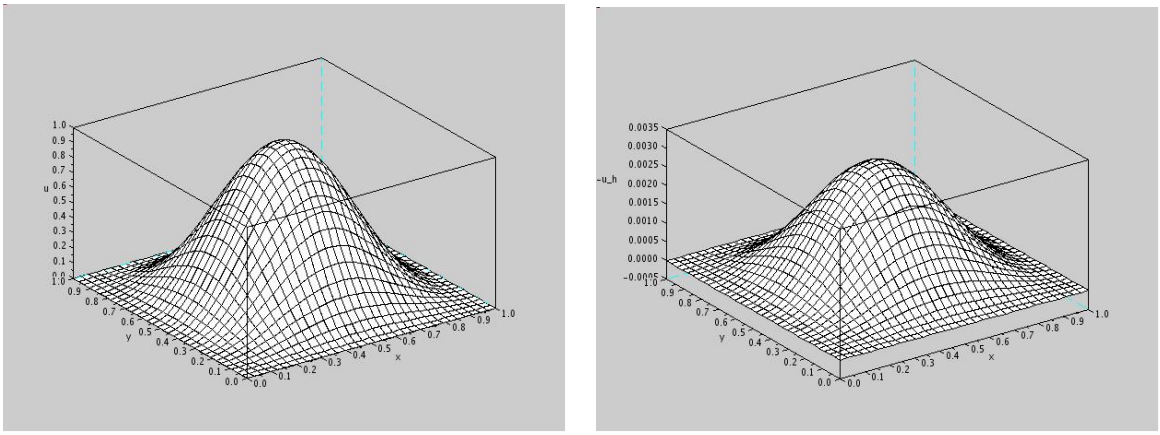
(a) Gaussian Elimination

N	DOF	$\|u - u_h\|$	Ratio	Time
32	1,024	3.0128e-3	N/A	<00:00:01
64	4,096	7.7812e-4	3.8719	<00:00:01
128	16,384	1.9766e-4	3.9366	<00:00:01
256	65,536	4.9807e-5	3.9685	00:00:04
512	262,144	1.2501e-5	3.9843	00:00:28
1,024	1,048,576	3.1313e-6	3.9922	00:03:15
2,048	4,194,304	7.8362e-7	3.9959	00:14:29
4,096	out of memory			
8,192	out of memory			

(b) Conjugate Gradient Method

N	DOF	$\|u - u_h\|$	Ratio	`#iter`	Time
32	1,024	3.0128e-3	N/A	48	<00:00:01
64	4,096	7.7810e-4	3.8719	96	00:00:02
128	16,384	1.9765e-4	3.9368	192	00:00:17
256	65,536	4.9797e-5	3.9689	387	00:02:29
512	262,144	1.2494e-5	3.9856	783	00:21:16
1,024	1,048,576	3.1266e-6	3.9961	1,581	02:59:08
2,048	excessive time requirement				
4,096	excessive time requirement				
8,192	excessive time requirement				

in FreeMat ran significantly slower than Octave and Matlab in those cases. As explained in the previous paragraph, simulations with $N \geq 2{,}048$ would take an excessive amount of time, and we did not run them to completion.



(a) (b)

Figure 3.4: Mesh plots for $N = 32$ in Scilab (a) of the numerical solution and (b) of the numerical error.

3.6 Scilab Results

3.6.1 Gaussian Elimination

Once again, we will solve the Poisson equation via Gaussian elimination, this time using Scilab. Scilab also uses the backslash operator to solve the linear system. To compute the Kronecker tensor product of matrix X and Y in Scilab, we can use the `X.*.Y` command when setting up the system matrix A. Figure 3.4 (a) is a mesh plot of numerical solution for a mesh resolution of $N = 32$, and Figure 3.4 (b) is a plot of the error associated with the numerical solution. The mesh plots are equivalent to the Matlab mesh plots in Figure 3.1. The code to solve this system for $N = 32$ and produce the plots is contained in `driver_ge.sci`, which is posted along with the tech. report at `www.umbc.edu/hpcf` under Publications. The initial version of this Scilab sci-file is obtained using the "Matlab to Scilab translator" under the Applications menu or by using `mfile2sci` command in Scilab.

To create the tables in Scilab we used a version `driver_ge.sci` with the graphics commands commented out. The Scilab code used to create the tables also utilizes the Scilab command `stacksize("max")`. The `stacksize("max")` command allows Scilab to use all available memory when running the code. The numerical results in Table 3.4 (a) are identical to the results in Tables 3.1 (a), 3.2 (a), and 3.3 (a) for the mesh resolutions that Scilab can solve for. However, Gaussian elimination in Scilab ran significantly slower than Octave, which in turn ran slower than Matlab, in those cases. Moreover, Gaussian elimination in Scilab can only solve the problem up to $N = 1{,}024$ and runs out of memory for $N = 2{,}048$, despite the use of `stacksize("max")`. In fact, without `stacksize("max")`, Scilab can only solve the problem up to $N = 256$ and runs out of memory for $N = 512$ [3]. That report also attempted to get better results by switching to UMFPack or TAUCS as alternative linear solvers available in Scilab. These solvers can be accessed by replacing the backslash operator in the linear solve. The results in [3] show that neither UMFPack nor TAUCS were able to solve for $N = 2,048$ either, but they were both much faster than the default backslash

Table 3.4: Convergence results for the test problem in Scilab using (a) Gaussian elimination and (b) the conjugate gradient method. The tables list the mesh resolution N, the number of degrees of freedom (DOF), the finite difference norm $\|u - u_h\|_{L^\infty(\Omega)}$, the ratio of consecutive errors, and the observed wall clock time in HH:MM:SS.

(a) Gaussian elimination

N	DOF	$\|u - u_h\|$	Ratio	Time
32	1,024	3.0128e-3	N/A	<00:00:01
64	4,096	7.7812e-4	3.8719	<00:00:01
128	16,384	1.9766e-4	3.9366	00:00:11
256	65,536	4.9807e-5	3.9685	00:03:19
512	262,144	1.2500e-5	3.9846	00:39:04
1,024	1,048,576	3.1313e-6	3.9920	08:32:20
2,048	out of memory			
4,096	out of memory			
8,192	out of memory			

(b) Conjugate gradient method

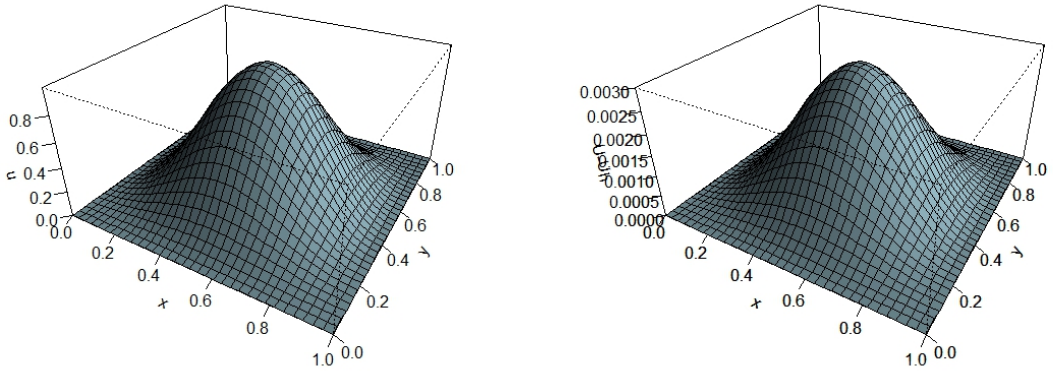
N	DOF	$\|u - u_h\|$	Ratio	`#iter`	Time
32	1,024	3.0128e-3	N/A	48	<00:00:01
64	4,096	7.7811e-4	3.8719	96	<00:00:01
128	16,384	1.9765e-4	3.9368	192	<00:00:01
256	65,536	4.9797e-5	3.9690	387	00:00:02
512	262,144	1.2494e-5	3.9856	783	00:00:22
1,024	1,048,576	3.1266e-6	3.9960	1,581	00:03:19
2,048	4,194,304	7.8018e-7	4.0075	3,192	00:26:57
4,096	out of memory				
8,192	out of memory				

operator that was used in Table 3.4 (a). At this point, we hope to solve a larger system using the conjugate gradient method.

3.6.2 Conjugate Gradient Method

Let us use the conjugate gradient method to solve the Poisson problem in Scilab. Here, we will use Scilab's `pcg` function. In order to solve, the initial guess is the zero vector and the tolerance is 10^{-6} on the relative residual of the iterates. We can use the sci-file `driver_cg.sci` which is posted along with the tech. report at `www.umbc.edu/hpcf`.

The numerical results in Table 3.4 (b) are identical to the results in Tables 3.1 (b), 3.2 (b), and 3.3 (b) for the mesh resolutions that Scilab can solve for. Also the run times are comparable for these cases. However, the conjugate gradient method in Scilab ran out of memory for $N = 4{,}096$, despite the use of `stacksize("max")`.



(a) (b)

Figure 3.5: Mesh plots for $N = 32$ in R (a) of the numerical solution and (b) of the numerical error.

3.7 R Results

3.7.1 Gaussian Elimination

Once again, we will solve the Poisson equation via Gaussian elimination, this time using R. As mentioned earlier, R uses the `solve` command to solve the linear system. To compute the Kronecker tensor product of matrix X and Y in R, we have used the `kronecker()` command while setting up the system matrix A. Figure 3.5 (a) is a mesh plot of numerical solution for a mesh resolution of $N = 32$, and Figure 3.5 (b) is a plot of the error associated with the numerical solution. The mesh plots are equivalent to the Matlab mesh plots in Figure 3.1. The code to solve this system for $N = 32$ and to produce the plots is contained in `driver_ge.r`, which is posted along with this report at `www.umbc.edu/hpcf` under Publications.

To create the tables in R, we have used a version of the driver script `driver_ge.r` with the graphics commands commented out. Comparing the results in Table 3.5 (a) with Tables 3.1 (a), 3.2 (a), 3.3 (a), and 3.4 (a), one can say that R's performance was similar to FreeMat's and better than Scilab's but worse than Matlab's and Octave's performances. Also, R could not solve for the resolutions N = 2,048, 4,096, and 8,192, as it ran out of allocated memory.

3.7.2 Conjugate Gradient Method

Unlike Matlab, Octave, and Scilab, R does not have a `pcg` function. As a result, we have written our own `cg` function in R. The code to solve this system and produce the plots for $N = 32$ using conjugate gradients is contained in `driver_cg.r`, which is posted along with

Table 3.5: Convergence results for the test problem in R using (a) Gaussian elimination and (b) the conjugate gradient method. The tables list the mesh resolution N, the number of degrees of freedom (DOF), the finite difference norm $\|u - u_h\|_{L^\infty(\Omega)}$, the ratio of consecutive errors, and the observed wall clock time in HH:MM:SS.

(a) Gaussian elimination

N	DOF	$\|u - u_h\|$	Ratio	Time
32	1,024	3.0128e-3	N/A	<00:00:01
64	4,096	7.7812e-4	3.8719	<00:00:01
128	16,384	1.9766e-4	3.9366	<00:00:01
256	65,536	4.9807e-5	3.9685	00:00:03
512	262,144	1.2501e-5	3.9843	00:00:25
1,024	1,048,576	3.1313e-6	3.9921	00:04:26
2,048	out of memory			
4,096	out of memory			
8,192	out of memory			

(b) Conjugate gradient method

N	DOF	$\|u - u_h\|$	Ratio	`#iter`	Time
32	1,024	3.0128e-3	N/A	48	<00:00:01
64	4,096	7.7810e-4	3.8719	96	00:00:01
128	16,384	1.9765e-4	3.9368	192	00:00:03
256	65,536	4.9797e-5	3.9690	387	00:00:13
512	262,144	1.2494e-5	3.9856	783	00:01:10
1,024	1,048,576	3.1266e-6	3.9961	1,581	00:07:29
2,048	4,194,304	7.8019e-7	4.0075	3,192	01:06:11
4,096	16,777,216	1.9366e-7	4.0286	6,452	09:20:20
8,192	out of memory				

the tech. report at `www.umbc.edu/hpcf` under Publications.

The numerical results in Table 3.5 (b) are identical to the results in Tables 3.1 (b), 3.2 (b), 3.3 (b), and 3.4 (b). While the conjugate gradient method in R ran significantly slower than Matlab and Octave, the performance is comparable to Scilab and better than FreeMat. Also, R could solve for resolutions until 4,096, whereas FreeMat and Scilab could only solve for resolutions until 1,024 and 2,048, respectively. After investigating the results, we found out that a significant amount of run-time was being spent in performing matrix algebra (specifically, subtractions on sparse matrices). There might be more efficient alternate ways in R to perform such matrix operations and there might be a significant scope for improvement.

3.8 IDL Results

3.8.1 Gaussian Elimination

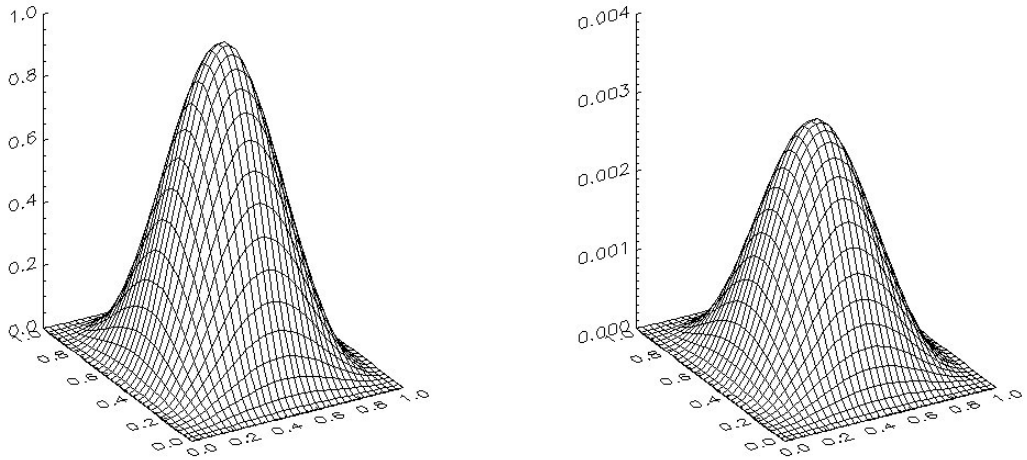
Unfortunately, there is no function or routine in IDL written to compute Gaussian elimination for sparse matrices without the purchase of a separate license. With the IDL Analyst license, `IMSL_SP_LUSOL` or `IMSL_SP_BDSOL` can be used to solve a sparse system of linear equations in various storage modes.

3.8.2 Biconjugate Gradient Method

No function or routine exists in the basic IDL license to compute the conjugate gradient method to solve the Poisson problem. With an extra IDL Analyst license, the routine `IMSL_SP_CG` is available to solve a real symmetric definite linear system using a conjugate gradient method. However, the biconjugate gradient method is available with our current license of IDL, and this method is mathematically equivalent to the conjugate gradient method for symmetric matrices. Thus, the biconjugate gradient method, just like the conjugate gradient method, can replace the use of LU decomposition in IDL with the `LINBCG` function. The zero vector is used as our initial guess. We also set the keyword `ITOL=1`, specifying to stop the iteration when $|Ax - b|/|b|$ is less than the tolerance. This convergence test with the tolerance 10^{-6} matches those used in the other languages.

To create the matrix A, we use three arrays holding the column and row locations, and the specific values at those locations, in the `SPRSIN` function to create a sparse matrix A. In addition, we used `CMREPLICATE`, created by Craig B. Markwardt, to replicate an array and create a full grid similar to the command `ndgrid` in Matlab. The `SURFACE` routine was used to create Figure 3.6 (a), the surface plot of the numerical solution vs. (x, y) and Figure 3.6 (b) the surface plot of the error vs. (x, y). The code to solve this system for $N = 32$ and produce the plots is contained in `driver_bcg.pro`, which is posted along with the tech. report at `www.umbc.edu/hcpf` under Publications.

To create the table, we used a version of `driver_bcg.pro` with the graphics commands commented out. The numerical results in Table 3.6 are identical to those in Table 3.1 (b), 3.2 (b), 3.3 (b), 3.4 (b), and 3.5 (b) (except for the largest case $N = 8{,}192$, which is not very different), and the performance of the biconjugate gradient method in IDL is comparable to the conjugate gradient methods in Matlab and Octave.



(a) (b)

Figure 3.6: Surface plots for $N = 32$ in IDL (a) of the numerical solution and (b) of the numerical error.

Table 3.6: Convergence results for the test problem in IDL using the biconjugate gradient method. The tables list the mesh resolution N, the number of degrees of freedom (DOF), the finite difference norm $\|u - u_h\|_{L^\infty(\Omega)}$, the ratio of consecutive errors, and the observed wall clock time in HH:MM:SS.

Biconjugate Gradient Method					
N	DOF	$\|u - u_h\|$	Ratio	`#iter`	Time
32	1,024	3.0128e-3	N/A	48	<00:00:01
64	4,096	7.7811e-4	3.8719	96	<00:00:01
128	16,384	1.9764e-4	3.9368	192	<00:00:01
256	65,536	4.9798e-5	3.9690	387	00:00:01
512	262,144	1.2494e-5	3.9856	783	00:00:12
1,024	1,048,576	3.1267e-6	3.9961	1,581	00:01:41
2,048	4,194,304	7.8029e-7	4.0070	3,192	00:17:26
4,096	16,777,216	1.9395e-7	4.0232	6,452	02:44:48
8,192	67,777,216	4.9022e-8	3.9563	13,044	20:08:46

Acknowledgments

The second author acknowledges financial support from the Department of Mathematics and Statistics at UMBC. The third and fourth authors acknowledge financial support from the UMBC High Performance Computing Facility. We are indebted to Neeraj Sharma, whose M.S. thesis first formalized the comparison between some of the software packages. The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant no. CNS–0821258) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See `www.umbc.edu/hpcf` for more information on HPCF and the projects using its resources.

References

[1] Kevin P. Allen. Efficient parallel computing for solving linear systems of equations. *UMBC Review: Journal of Undergraduate Research and Creative Works*, vol. 5, pp. 8–17, 2004.

[2] Dietrich Braess. *Finite Elements*. Cambridge University Press, third edition, 2007.

[3] Matthew Brewster and Matthias K. Gobbert. A comparative evaluation of Matlab, Octave, FreeMat, and Scilab on tara. Technical Report HPCF–2011–10, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2011.

[4] Ecaterina Coman. IDL: A possible alternative to Matlab? Senior Thesis, Department of Mathematics and Statistics, University of Maryland, Baltimore County, 2012.

[5] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[6] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge Texts in Applied Mathematics. Cambridge University Press, second edition, 2009.

[7] Jeremy Kepner. *Parallel MATLAB for Multicore and Multinode Computers*. SIAM, 2009.

[8] Sai K. Popuri, Andrew M. Raim, Matthew W. Brewster, and Matthias K. Gobbert. A comparative evaluation of Matlab, Octave, FreeMat, Scilab, and R on tara. Technical Report HPCF–2012–7, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2012.

[9] Andrew M. Raim and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the cluster tara. Technical Report HPCF–2010–2, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010.

[10] Neeraj Sharma. A comparative study of several numerical computational packages. M.S. thesis, Department of Mathematics and Statistics, University of Maryland, Baltimore County, 2010.

[11] Neeraj Sharma and Matthias K. Gobbert. Performance studies for multithreading in Matlab with usage instructions on hpc. Technical Report HPCF–2009–1, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2009.

[12] Neeraj Sharma and Matthias K. Gobbert. A comparative evaluation of Matlab, Octave, FreeMat, and Scilab for research and teaching. Technical Report HPCF–2010–7, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010.

[13] Karline Soetaert, Thomas Petzoldt, and R. Woodrow Setzer. Package deSolve: Solving initial value differential equations in R, CRAN R project documentation, 2010.

[14] David S. Watkins. *Fundamentals of Matrix Computations*. Wiley, third edition, 2010.