

A Comparative Evaluation of Matlab, Octave, R, and Julia on Maya

Sai K. Popuri and Matthias K. Gobbert*

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

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

Technical Report HPCF-2017-3, hpcf.umbc.edu > Publications

Abstract

Matlab is the most popular commercial package for numerical computations in mathematics, statistics, the sciences, engineering, and other fields. Octave is a freely available software used for numerical computing. R is a popular open source freely available software often used for statistical analysis and computing. Julia is a recent open source freely available high-level programming language with a sophisticated compiler for high-performance numerical and statistical computing. They are all available to download on the Linux, Windows, and Mac OS X operating systems. We investigate whether the three freely available software are viable alternatives to Matlab for uses in research and teaching. We compare the results on part of the equipment of the cluster maya in the UMBC High Performance Computing Facility. The equipment has 72 nodes, each with two Intel E5-2650v2 Ivy Bridge (2.6 GHz, 20 MB cache) processors with 8 cores per CPU, for a total of 16 cores per node. All nodes have 64 GB of main memory and are connected by a quad-data rate InfiniBand interconnect. The tests focused on usability lead us to conclude that Octave is the most compatible with Matlab, since it uses the same syntax and has the native capability of running m-files. R was hampered by somewhat different syntax or function names and some missing functions. The syntax of Julia is closer to that of Matlab than it is to R. The tests focused on efficiency show that while Matlab, Octave, and Julia were fundamentally able to solve problems of the same size, Matlab and Julia were found to be closer in terms of efficiency in absolute run times, especially for large sized problems.

1 Introduction

1.1 Overview

Several numerical computational packages exist that serve as educational tools and are also available for commercial and research use. Matlab is the most widely used such package. The focus of this study is to compare Matlab to three alternative numerical computational packages: Octave; the statistical package R; and Julia; 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, R, and Julia, 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, R, and Julia. 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. Matlab, Octave, FreeMat, Scilab, R, and IDL were compared previously [5] on the cluster tara. This reports extends [5] in two aspects. The recent programming language Julia that is gaining popularity in the scientific computing community is added to our comparative study. All our numerical studies are performed on the cluster maya, which was made available after the previous comparative study [5] was conducted. Since the previous study concluded that Octave was the closest to Matlab in terms of syntax and performance, we leave out FreeMat, Scilab, and IDL from our study. However, we include the statistical software R since Julia is also viewed as a competitor to R by statistical community and therefore there might be audience for the comparison between these two software. Moreover, Julia as a very young package is promoted from the start as highly efficient computationally, and thus it is interesting to quantify this claim. In all packages, we use the sparse matrix storage for the system matrix resulting from the Poisson test problem for both the direct and the iterative solvers.

In Section 2, we perform the basic operations test using Matlab, Octave, R, and Julia. This includes testing the backslash operator, computing eigenvalues and eigenvectors, and plotting in two dimensions in all the packages except R, which follows a significantly different syntax and functions. The backslash operator works identically for Matlab, Octave, and Julia to produce a solution to the linear system given, although its syntax in Julia is that of a function with arguments. R uses the command `solve` to produce a solution. The command `eig` has the same functionality in Octave as in Matlab for computing eigenvalues and eigenvectors. R uses the command `eigen` to compute them. Julia has functions `eigvecs` and `eigvals` to compute eigenvectors and eigenvalues, respectively. 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 uses the exact commands for plotting and annotating as Matlab whereas R and Julia require a few changes.

Section 3 considers the classical test problem of the two-dimensional Poisson equation discretized by the finite difference method, as used in many Numerical Linear Algebra textbooks to test linear solvers, e.g., [6, Section 6.3], [7, Subsection 9.1.1], [8, Chapter 12], and [19, Section 8.1]. To find 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 $4,096 \times 4,096$ in Matlab, Octave, and Julia. 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 $2,048 \times 2,048$. The conjugate gradient method is implemented in the `pcg` function, which is stored in Matlab and Octave as a m-file. The method is implemented in the `cg` function of the `IterativeSolvers` package in Julia. The sparse matrix implementation of the iterative method allows us to solve a mesh resolution up to $8,192 \times 8,192$ in Matlab, Octave, and Julia. Since there does not seem to be an available function in R that is similar to the `pcg` function, we wrote our own function. In R we were

able to solve the system for a resolution up to $4,096 \times 4,096$ using our implementation of the conjugate gradient method. The existence of the `mesh` in Matlab and Octave allows us to include the three-dimensional plots of the numerical solution and the numerical error. In Julia, we used the `surface` attribute of the `figure` function in the package `PyPlot`. In R, we used the `persp` function instead. The syntax of Octave is identical to that of Matlab in our tests. The syntax of R is distinctly different from that of Matlab. We found the syntax of Julia to be more closer to that of Matlab than R. The tests focused on usability lead us to conclude that the package Octave is most compatible with Matlab, since they use the same syntax and have the native capability of running m-files.

The results of the numerical experiments on the complex test problem in Section 3 are clearly somewhat surprising. Fundamentally, Matlab, Octave, and Julia 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, Julia is found to be quite close to Matlab, except for very large problems. Although Octave’s handling of the problem is acceptable, it was found to be much slower than Julia for large problems. R exhibited some limitations on the size of problems it could solve. Specifically, R could only solve for resolutions up to $4,096 \times 4,096$. Matlab, Julia, and Octave 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, and 3.4.

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 four 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 hpcf.umbc.edu/publications 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.

Octave solves non-stiff ODEs using the Adams methods and stiff equations using the backward differentiation formulas. These are implemented in `lsode` in Octave. According

to their documentations, Octave selects the method used automatically, depending on the problem.

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 [17].

At the time of writing this technical report, Julia has two packages to solve ODEs: `DifferentialEquations` and `ODE`. Between these packages, a variety of stiff and non-stiff ODE solvers are available using several methods including the Adams-Bashforth-Moulton solver, Runge-Kutta methods, Rosenbrock method, and more.

It becomes clear that all software packages considered have at least one solver each for non-stiff and for stiff problems. All 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. Due to the vast differences between the packages in implementations of these methods, numerical comparisons of the ODE solvers are not straightforward, and we did not test them for this study.

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 [15] 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 [15], 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 could give a very significant advantage for Matlab over 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 to (up to) all cores of a shared-memory node. This is clearly useful for parameter studies, parametrized by the `for` loop.

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

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).

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 pMatlab² 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 [9]. A test for the same test problem as in this report is available [18]. However, this software and its webpage given in the footnote appear to be stale at the time of this writing.

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 that target GPUs too.⁴

Julia provides a suite of packages⁵ and offers an environment to harness the available multiple processes in a shared or separate memory setting using message passing. For example, the package `MPI` offers an interface to the MPI functions. A comparison of this package with R’s `Rmpi` package in a recent study [11] showed that Julia’s `MPI` performed better in terms of runtime. In addition, Julia offers packages to program targeting GPUs.⁶

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

²<http://www.ll.mit.edu/mission/cybersec/softwaretools/pmatlab/pmatlab.html>

³<http://cran.r-project.org/web/views/HighPerformanceComputing.html>

⁴<http://gpgpu.org/2009/06/14/r-gpgpu>

⁵<https://github.com/JuliaParallel>

⁶<http://juliacomputing.com/blog/2016/06/09/julia-gpu.html>

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 an engineering 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 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, are 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, for bringing out the difference in behavior between stiff and non-stiff ODE solvers, the ones available in Octave would be sufficient, even if their function names do not agree with those in Matlab and their underlying methods are not exactly the same, but we found it difficult to control the choice of solver explicitly.

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 programming; 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 redistribute 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 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`.

1.3.4 Julia

Julia is a recently developed programming language that is gaining popularity in scientific computing, data analysis, and high performance computing [2]. It is an open source, high-level compiled language that uses the Low Level Virtual Machine Just-in-Time technology [10] to generate an optimized version of the source code compiled to the machine level. As a result, the resulting executable is expected to run in a much efficient way compared to interpreted code (like Matlab, R etc.). Julia provides a number of computational and statistical capabilities, both in the core environment and through packages contributed by the user community. All the packages in Julia are available as projects on the `Git` repository system.⁷

1.4 Description of the Computing Environment

The computations for this study are performed using Matlab R2016a, Octave 3.4.3, R 3.2.2, and Julia 0.5.0 under the Linux operating system CentOS 6.8. A part of the equipment in the cluster maya in the UMBC High Performance Computing Facility (`hpcf.umbc.edu`) is used to carry out the computations and has a total of 72 nodes. Each node features two Intel E5-2650v2 Ivy Bridge (2.6 GHz, 20 MB cache) processors with 8 cores per CPU, for a total of 16 cores per CPU. All the nodes have 64 GB of main memory and are connected by a quad-data rate InfiniBand interconnect.

⁷<https://git-scm.com>

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 `circ.umbc.edu`. This section focuses on the testing of basic operations using Matlab, Octave, R, and Julia. 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 $\mathbf{x} = \mathbf{A} \backslash \mathbf{b}$. 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

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

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 .

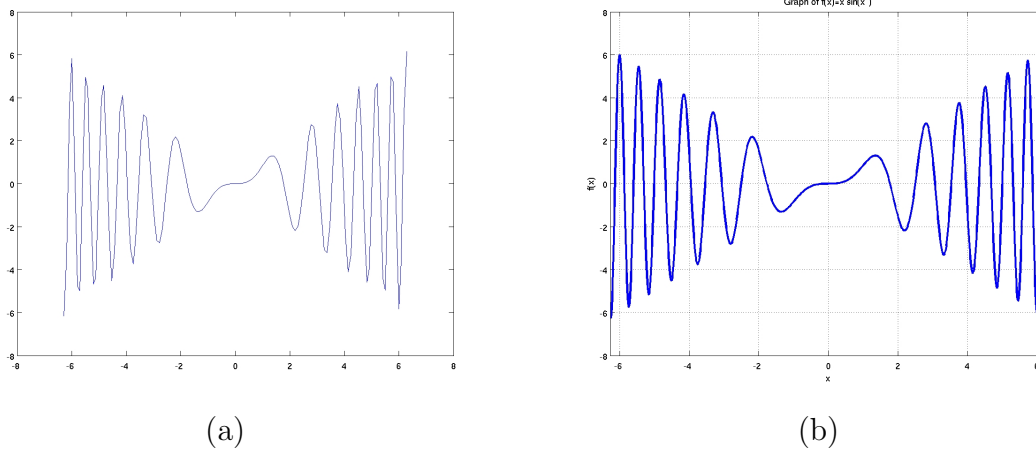


Figure 2.1: Plots of $f(x) = x \sin(x^2)$ in Matlab using (a) 129 and (b) 1025 equally spaced data points.

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 `hpcf.umbc.edu` 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, you can 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 hpcf.umbc.edu 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 functional 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

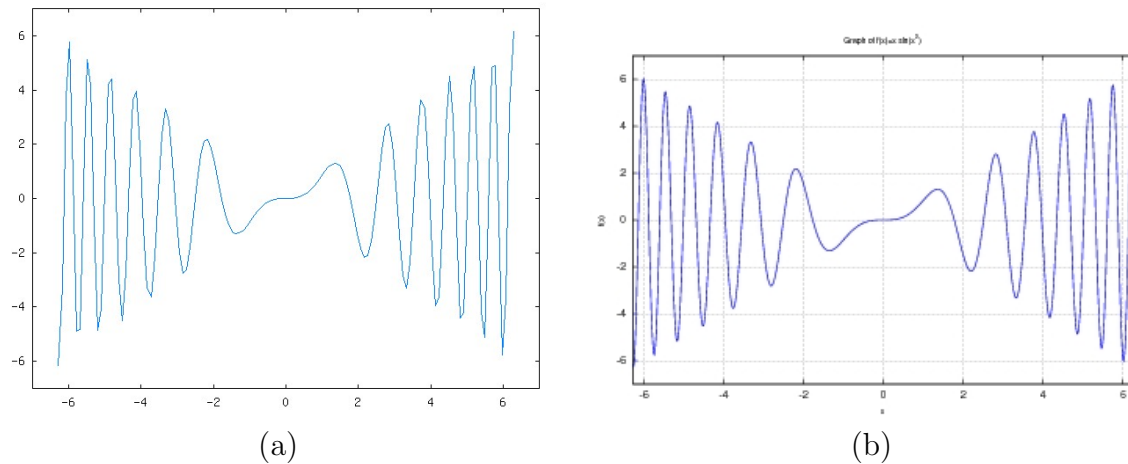


Figure 2.2: Plots of $f(x) = x \sin(x^2)$ in Octave using (a) 129 and (b) 1025 equally spaced data points.

results in the same m-file `plotxsinx.m`, which is posted in the file `plotxsinx.m` along with the tech. report at hpcf.umbc.edu 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 R

In this section, we will perform the basic operations in R. To run R on the cluster maya, 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.

2.3.1 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 exist⁸. 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.

2.3.2 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

```
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
```

⁸Another 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

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).

2.3.3 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.

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

`dev.new()` instructs R to generate plots in separate 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 2.3 (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.

2.3.4 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 and Octave, 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", equilog=TRUE)
```

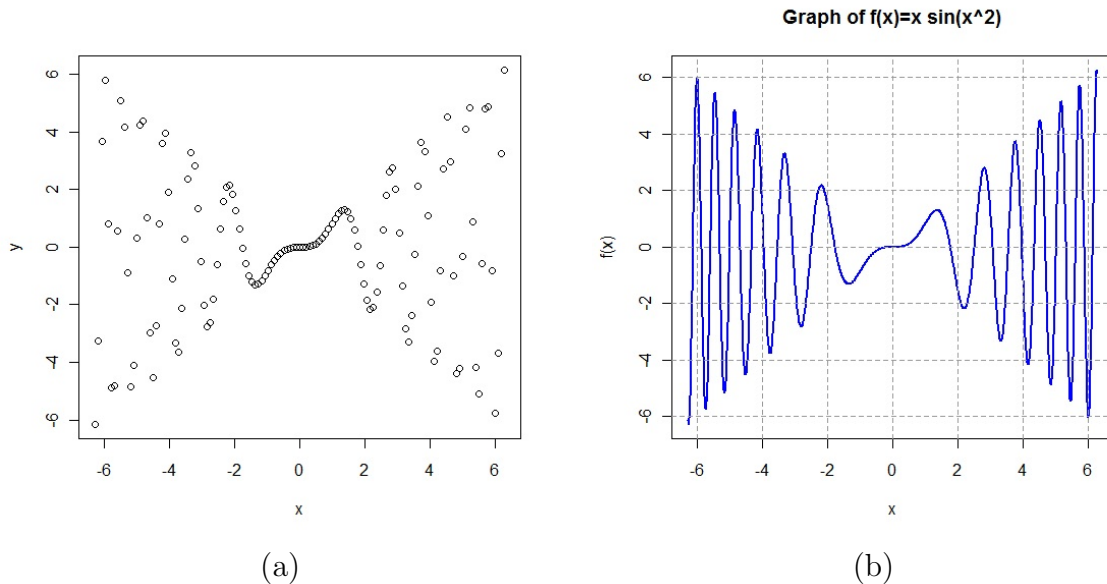


Figure 2.3: Plots of $f(x) = x \sin(x^2)$ in R using (a) 129 and (b) 1025 equally spaced data points.

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 2.3 (b) which is similar to Figure 2.1 (b). Notice that compared to Figure 2.3 (a), Figure 2.3 (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 hpcf.umbc.edu under Publications. The file can then be executed in R, similar to Matlab’s `m`-files, using the command

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

2.4 Basic Operations in Julia

In this section, we will perform the basic operations in Julia. To run Julia on the cluster maya, run the command ‘module load julia/0.5.0’ in your Linux command line. Once the module is loaded, enter `julia` at the Linux prompt. This starts up the command window of Julia with the prompt `julia>` in the Linux shell. For the complete list of options available with the command `julia`, type `julia --help` at the Linux prompt. One useful command in Julia is `?`, which switches the Julia command terminal to the ‘help’ environment, where you can type a keyword to find the related useful information. For example, to view the usage of the command `import`, type `import` in the ‘help’ environment. To quit the ‘help’ environment, just hit enter. To quit the Julia interface, type `quit()`. Refer to <http://julialang.org/> to learn more about Julia.

2.4.1 Solving Systems of Equations in Julia

Once again, let us begin by solving the linear system from Section 2.1.1. Julia uses the backslash operator 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. In Julia, vectors and matrices are aliases to one dimensional and two dimensional arrays, respectively. To solve the system mentioned in Section 2.1.1, we use the following commands in the Julia command window:

```
julia> A = [0 -1 1; 1 -1 -1; -1 0 -1]
3x3 Array{Int64,2}:
 0  -1   1
 1  -1  -1
-1   0  -1
julia> b = [3; 0; -3];
```

to set up the matrix A and vector b , respectively. When creating matrices, Julia follows the row-ordering scheme by default, that is, the entries in `[0 -1 1; 1 -1 -1; -1 0 -1]` delimited by the semicolon specify the entries of A along the rows. Notice that the above commands to create the matrix A and vector b are the same exact commands as in Matlab. In Julia, if a command is not suffixed by a semicolon, the output from the command is displayed immediately on the command window. The output shows the 3×3 matrix, which is a two-dimensional Array in Julia, A that was created. Since the command to create the vector b is suffixed with a semicolon, the created vector is not displayed. Using the function `\` with two arguments in parentheses, we obtain the result

```
julia> \ (A, b)
3-element Array{Float64,1}:
 1.0
-1.0
 2.0
```

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

The above approach to create the matrix A and vector b translates to explicitly using the `Array` command. The above system can also be solved using the following commands:

```
julia> A = Array{Int64, 3, 3};
julia> A = [0 -1 1; 1 -1 -1; -1 0 -1];
julia> b = Array{Int64, 3, 1};
julia> b = [3; 0; -3];
julia> \ (A, b)
3-element Array{Float64,1}:
 1.0
-1.0
 2.0
```

2.4.2 Calculating Eigenvalues and Eigenvectors in Julia

Now, let us determine how to calculate the eigenvalues and eigenvectors for the matrix A stated in Section 2.1.2. Julia uses the `eigvals` and `eigvecs` commands to compute eigenvalues and eigenvectors, respectively. The code below shows how to compute eigenvalues and eigenvectors for a matrix.

```
julia> A = [1 -1; 1 1];
julia> eigvals(A)
2-element Array{Complex{Float64},1}:
 1.0+1.0im
 1.0-1.0im
julia> eigvecs(A)
2x2 Array{Complex{Float64},2}:
 0.707107+0.0im      0.707107-0.0im
 0.0-0.707107im     0.0+0.707107im
```

The function `eigvals` returns the eigenvalues of A as an array and the function `eigvecs` returns the corresponding eigenvectors as a matrix, where the i^{th} column contains the eigenvector corresponding to the i^{th} element in the array returned by the function `eigvals`. Clearly, the outcome is exactly what we expect 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. Matrices D and P can be easily formed from the values returned by Julia's `eigvals` and `eigvecs`, for example to verify that the diagonalization yields A again as

```
julia> D = diagm(eigvals(A));
julia> P = eigvecs(A);
julia> P*D*inv(P)
2x2 Array{Complex{Float64},2}:
 1.0+0.0im -1.0+0.0im
 1.0+0.0im  1.0+0.0im
```

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 `inv()` command is used to calculate the inverse of the non-singular matrix P .

2.4.3 2-D Plotting from a Data File in Julia

Now, we will plot $f(x) = x \sin(x^2)$ in Julia. To load the text file `matlabdata.dat` into a matrix, we use the Julia command `pd = readdlm("matlabdata.dat");`. Graphical capabilities in Julia are available through several external packages like `Plots`, `PyPlot`, `Winston`, `Gadfly`, and more, that provide access to the available ‘backend’ visualization devices. We plot the data using the `plot` command in the `PyPlot` package, which is an interface to the `Matplotlib` Python library (<http://matplotlib.org>). In order to use the `PyPlot` package on `mayavi`, Julia must be started at the Linux prompt using the command `LD_LIBRARY_PATH=/usr/cluster/contrib/julia/0.5.0/share/julia/site/v0.5/Conda/deps/usr/lib/ julia`. This ensures that the libraries needed by `PyPlot` are made available.

```
using PyPlot
plt = PyPlot;
plt.figure();
plt.plot(datt[:,1],datt[:,2]);
plt.savefig("julia_fig_1.png");
```

The command `using` loads the package `PyPlot` into Julia’s environment. In order to resolve possible name conflicts of the package’s functions with similarly named functions from other packages loaded, we qualify the function calls with the name of the package. Aliasing the package is a convenient way to do this. The package `PyPlot` is provided an alias name as `plt` and subsequent function calls are accordingly qualified. Assuming an X11 display server is setup on the computer with a remote connection to `mayavi`, the function `figure` opens a new window and the `plot` function then displays the plot on it. The `savefig` function saves the plot as a `png` file to the current directory. Figure 2.4 (a) shows the plot created by the code above. Notice that the plot does not have a title and is not labeled.

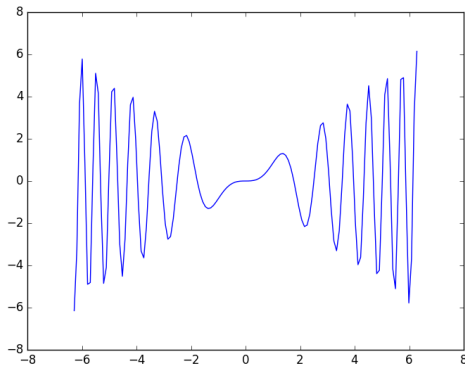
In order to reuse the display window for further plotting, the function `clf` must be called to clear the plotting area as:

```
plt.clf();
```

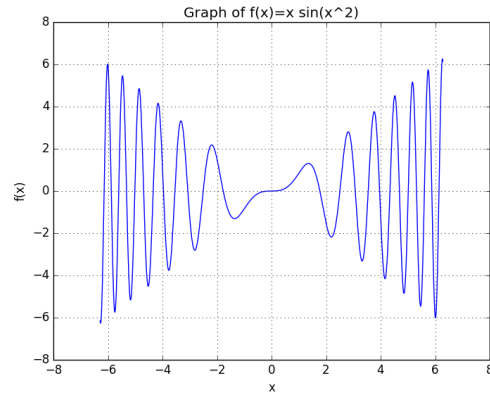
2.4.4 Annotated Plotting from Computed Data in Julia

We will improve the plot in Figure 2.4 (a) by increasing the resolution of the data and by annotating the plot with a title and labels. Let us improve the resolution by creating vector x using

```
x = collect(-2*pi:4*pi/1024:2*pi);
```



(a)



(b)

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

and let $y = x \cdot \sin(x.^2)$; to create a corresponding y vector. Like Matlab and Octave, we use `pi` for the constant π in Julia.

We now plot using the new x and y values again using the `plot` command. The displayed plot is annotated by subsequent function calls.

```
plt.figure();
plt.plot(x1, y1);
plt.ylabel("f(x)");
plt.xlabel("x");
plt.title("Graph of f(x)=x sin(x^2)");
plt.grid(true);
plt.savefig("julia_fig_2.png");
```

Note that the code above creates a new window using the `figure` function. Therefore, the `clf` function need not be called. The `grid` command above generates a fine grid against the plot. The above code generates the Figure 2.4 (b) which is similar to Figure 2.1 (b). To send these graphics to a `png` file, we have used the `savefig` function. We can save all the above commands to a text file with extension `.jl`. The code is posted in the file `plotxsinx.jl` along with the tech. report at hpcf.umbc.edu under Publications. The file can then be executed in Julia, similar to Matlab's `m`-files, using the command

```
include("plotxsinx.jl").
```


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, 6, 8, 19], given as

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

This problem is a popular textbook example, see, among others, [6, 19]. We have studied it with several different emphases in, among others, [1, 4, 5, 12–14, 16].

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

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

This partial differential equation can be used to model heat flow, fluid flow, elasticity, and other phenomena [19]. 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

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

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 $\Omega_h = \{(x_i, y_j) = (ih, jh), i, j = 0, \dots, N+1\}$ 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

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

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

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

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

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

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

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

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 [6, 19]. 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 [6, 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 hpcf.umbc.edu.

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 $h \rightarrow 0$, where C is a constant independent of h [3, 8]. For sufficiently small h , we can then expect that the ratio of errors on consecutively refined meshes behaves like

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

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 hpcf.umbc.edu 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 $\nu = 1, 2, 3, \dots, 13$. 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

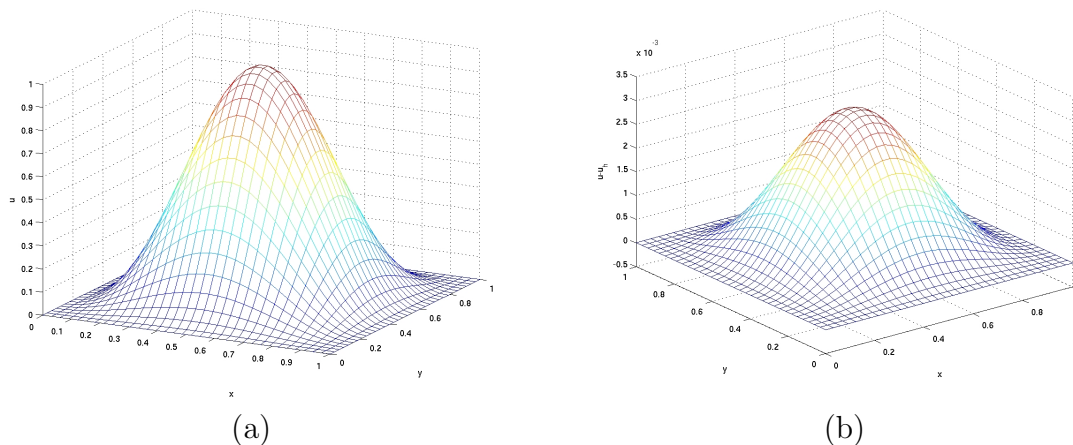


Figure 3.1: Mesh plots for $N = 32$ in Matlab (a) of the numerical solution and (b) of the numerical error.

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, 19]. 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 hpcf.umbc.edu. 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; this

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	3.9368=6		<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	3.9922		00:00:04
2,048	4,194,304	7.8362e-7	3.9960		00:00:16
4,096	16,777,216	1.9608e-7	3.9964		00:01:06
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	00:00:06
1,024	1,048,576	3.1266e-6	3.9961	1,581	00:00:46
2,048	4,194,304	7.8019e-7	4.0075	3,192	00:07:13
4,096	16,777,216	1.9366e-7	4.0286	6,452	00:57:26
8,192	67,777,216	4.7401e-8	4.0856	13,033	07:21:53

confirms that the tolerance on the relative residual of the iterates is tight enough. For the problems it was able to solve, Gaussian elimination was faster than the conjugate gradient method. 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.

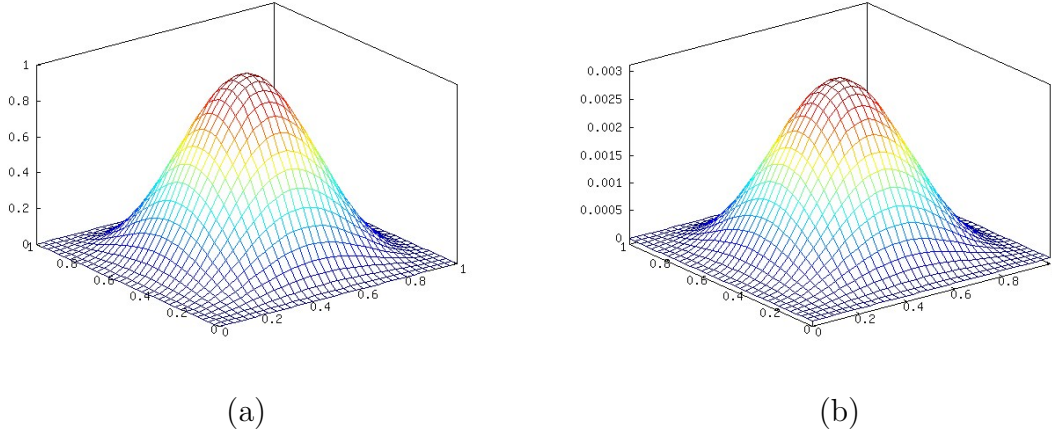


Figure 3.2: Mesh plots for $N = 32$ in Octave (a) of the numerical solution and (b) of the numerical error.

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 hpcf.umbc.edu 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 hpcf.umbc.edu 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	00:00:01
1,024	1,048,576	3.1313e-6	3.9922	00:00:06
2,048	4,194,304	7.8362e-7	3.9960	00:00:38
4,096	16,777,216	1.9607e-7	3.9966	00:04:27
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	00:00:07	
1,024	1,048,576	3.1266e-6	3.9961	1,581	00:01:03	
2,048	4,194,304	7.8019e-7	4.0075	3,192	00:12:52	
4,096	16,777,216	1.9366e-7	4.0286	6,452	01:45:34	
8,192	67,108,864	4.7402e-8	4.0856	13,033	14:06:29	

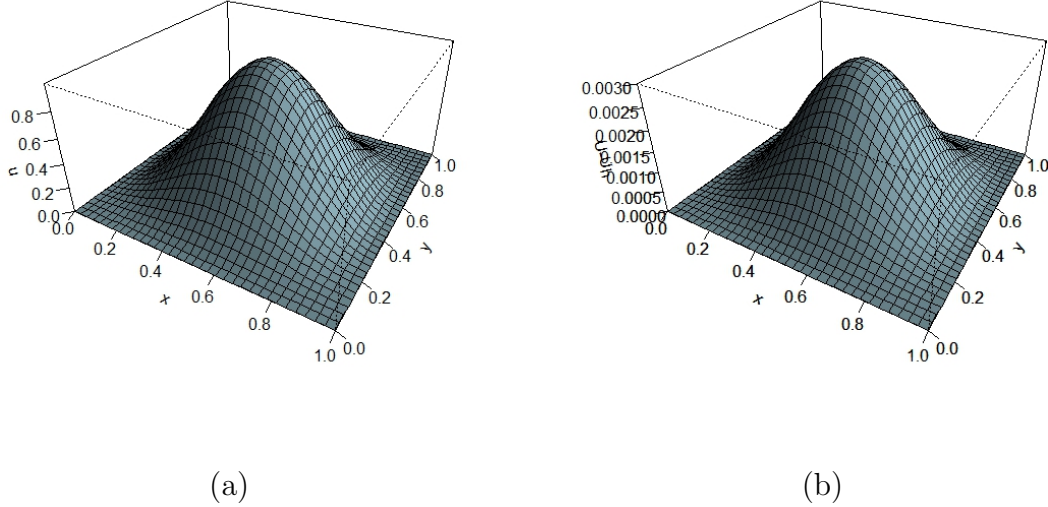


Figure 3.3: Mesh plots for $N = 32$ in R (a) of the numerical solution and (b) of the numerical error.

3.5 R Results

3.5.1 Gaussian Elimination

Once again, we will solve the Poisson equation via Gaussian elimination, this time using R. We have used the function `solve` to solve the linear system in the Poisson problem. The Kronecker tensor product of matrix X and Y is calculated using the `kronecker()` function while setting up the system matrix A . Figure 3.3 (a) is a mesh plot of the numerical solution for the resolution of $N = 32$ and Figure 3.3 (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 hpcf.umbc.edu/publications under Publications.

Comparing the results in Table 3.3 (a) with Tables 3.1 (a) and 3.2 (a), we can say that our program in R was not as efficient as our problems in Matlab and Octave. Also, our R program could not solve for the resolutions $N = 4,096$ and $8,192$, as it ran out of allocated memory.

3.5.2 Conjugate Gradient Method

Unlike Matlab and Octave, 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` and `cg.r`, which are posted along with the this report at hpcf.umbc.edu/publications under Publications.

Table 3.3: 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		00:00:02
512	262,144	1.2501e-5	3.9843		00:00:17
1,024	1,048,576	3.1313e-6	3.9922		00:03:16
2,048	4,194,304	7.8359e-7	3.9961		00:57:06
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	00:00:07
512	262,144	1.2494e-5	3.9856	783	00:00:40
1,024	1,048,576	3.1266e-6	3.9961	1,581	00:05:28
2,048	4,194,304	7.8019e-7	4.0075	3,192	00:53:59
4,096	16,777,216	1.9366e-7	4.0286	6,452	07:36:53
8,192	67,108,864	4.7402e-8	4.0856	13,033	61:57:11

The numerical results in Table 3.3 (b) are identical to the results in Tables 3.1 (b) and 3.2 (b). Our implementation of the conjugate gradient method in R was found to be significantly slower than Matlab and Octave. Just like Matlab and Octave, the Gaussian elimination method was faster than the conjugate gradient method for the problems it was able to solve. 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). We note that it is possible to improve the performance results for R as more packages become available that perform algebraic operations on sparse matrices more efficiently.

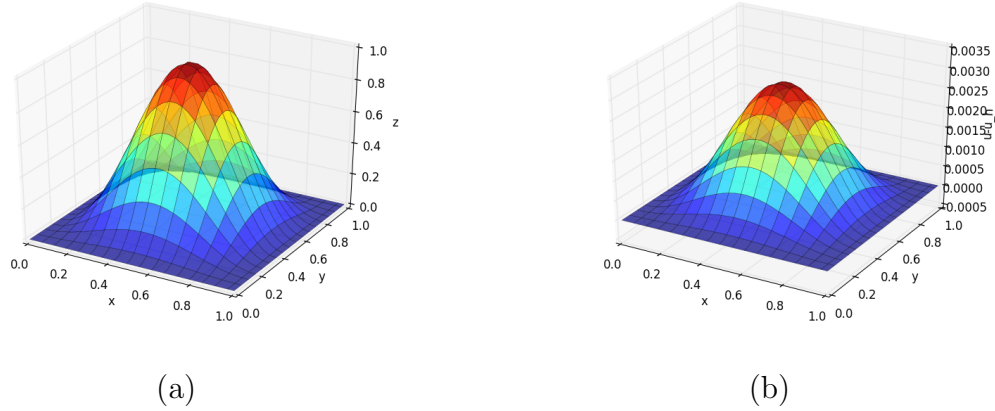


Figure 3.4: Mesh plots for $N = 32$ in Julia (a) of the numerical solution and (b) of the numerical error.

3.6 Julia Results

3.6.1 Gaussian Elimination

Julia provides the backward slash operator as a function, which we use to solve the linear system in the Poisson equation. To compute the Kronecker tensor product of matrix X and Y , we have used the `kron()` command while 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 to produce the plots is contained in `driver_ge.jl`, which is posted along with this report at hpcf.umbc.edu/publications under Publications.

Comparing the results in Table 3.4 (a) with Tables 3.1 (a), 3.2 (a), and 3.3 (a), we can say that the performance of our program in Julia is similar to that of our program in Matlab, better than our Octave implementation, and significantly better than our R implementation. However, like Matlab and Octave, Julia could not solve for the resolution $N = 8,192$, as it ran out of allocated memory.

3.6.2 Conjugate Gradient Method

We use the function `cg` in the package `IterativeSolvers` to solve the linear system of equations using the conjugate gradient method. This function is similar to the `pcg` function in Matlab. The code used to solve this system and produce the plots for $N = 32$ using conjugate gradients is contained in `driver_cg.jl`, which is posted along with this report at hpcf.umbc.edu/publications under Publications.

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). From the numerical results, we can say that our Julia implementation performed significantly better than R and is better than our Octave implementation, especially for large

Table 3.4: Convergence results for the test problem in Julia 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	3.9922		00:00:05
2,048	4,194,304	7.8363e-7	3.9960		00:00:19
4,096	16,777,216	1.9610e-7	3.9961		00:01:19
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	00:00:07
1,024	1,048,576	3.1266e-6	3.9961	1,581	00:01:12
2,048	4,194,304	7.8019e-7	4.0075	3,192	00:15:45
4,096	16,777,216	1.9366e-7	4.0287	6,452	01:38:34
8,192	67,108,864	4.7392e-8	4.0864	13,033	11:42:43

problems. Although Julia was able to solve problems of all sizes like Matlab, it was slower in comparison. Also, just like Matlab, Octave, and R, our Gaussian elimination method was much faster than the conjugate gradient method for the problems it was able to solve.

Acknowledgments

The first author acknowledges financial support from the UMBC High Performance Computing Facility (HPCF). The hardware used in the computational studies is part of HPCF, which is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS-0821258 and CNS-1228778) and the SCREMS program (grant no. DMS-0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See hpcf.umbc.edu 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*, 5:8–17, 2004.
- [2] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing, 2014. [arXiv:1411.1607](#).
- [3] Dietrich Braess. *Finite Elements*. Cambridge University Press, third edition, 2007.
- [4] 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.
- [5] Ecaterina Coman, Matthew W. Brewster, Sai K. Popuri, Andrew M. Raim, and Matthias K. Gobbert. A comparative evaluation of Matlab, Octave, FreeMat, Scilab, R, and IDL on tara. Technical Report HPCF–2012–15, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2012.
- [6] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.
- [7] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*, volume 17 of *Frontiers in Applied Mathematics*. SIAM, 1997.
- [8] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge Texts in Applied Mathematics. Cambridge University Press, second edition, 2009.
- [9] Jeremy Kepner. *Parallel MATLAB for Multicore and Multinode Computers*. SIAM, 2009.
- [10] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*, Palo Alto, California, Mar 2004.
- [11] S. K. Popuri, A. M. Raim, N. K. Neerchal, and M. K. Gobbert. Parallelizing Computation of Expected Values in Recombinant Binomial Trees. *ArXiv e-prints*, January 2017.
- [12] 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.

- [13] 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.
- [14] Neeraj Sharma. *A Comparative Study of Several Numerical Computational Packages*. M.S. Thesis, Department of Mathematics and Statistics, University of Maryland, Baltimore County, 2010.
- [15] 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.
- [16] 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.
- [17] Karline Soetaert, Thomas Petzoldt, and R. Woodrow Setzer. Package deSolve: Solving initial value differential equations in R, CRAN R project documentation, 2010.
- [18] Sarah Swatski. Investigating the use of pmatlab to solve the poisson equation on the cluster maya. Technical Report HPCF–2014–24, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2014.
- [19] David S. Watkins. *Fundamentals of Matrix Computations*. Wiley, third edition, 2010.