A COMPARATIVE EVALUATION OF MATLAB, OCTAVE, FREEMAT, AND SCILAB FOR RESEARCH AND TEACHING

Neeraj Sharma and Matthias K. Gobbert

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

{nsharma1,gobbert}@umbc.edu

Technical Report HPCF–2010–7, `www.umbc.edu/hpcf` > Publications

Abstract: The commercial software package Matlab is the most widely used numerical computational package. GNU Octave, FreeMat, and Scilab are other numerical computational packages that have many of the same features as Matlab. Unlike Matlab, the other three packages are free of charge to the user and easily available to download to Linux, Windows, or Mac OS X operating systems. Are these packages viable alternatives to Matlab for uses in research or teaching? We review in depth the basic operations of Matlab, such as solving a system of linear equations, computing eigenvalues and eigenvectors, and plotting in two-dimensions. In addition, we research a more complex test problem resulting from a finite difference discretization of the Poisson equation in two spatial dimensions. Then, we compare the results we receive from GNU Octave, FreeMat, and Scilab to our previously found Matlab results. All packages gave identical numerical results, though Scilab exhibited a limitation in the size of the linear system it could solve in the complex test problem and FreeMat was hampered by the lack of a conjugate gradient function. The available graphical functions differ in functionality, but give equivalent plots, though FreeMat has limited three-dimensional graphics capabilities at present. Based on these tests, we conclude that GNU Octave is the most compatible with Matlab due to its numerical abilities and the similarity of its syntax. Another reason to consider Octave is that free parallel computing extensions are available that are known to work with this package. Therefore, for uses in research, Octave's maturity and resulting richness of functions make it a viable alternative to Matlab. For uses in teaching, the similarity of its syntax with Matlab will likely make textbook instructions directly applicable, and we thus recommend consideration of Octave for this purpose, though the particular assignments of each course will need to be tested in detail.

Contents

Chapter 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 three additional numerical computational packages: GNU Octave, FreeMat, and Scilab, and provide information on which package is most compatible to Matlab users. Section 1.3 provides more detailed descriptions of these packages. To evaluate GNU Octave, FreeMat, and Scilab, 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, GNU Octave, FreeMat, and Scilab. 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 work is an extension of the M.S. thesis [6].

In Chapter 2, we perform the basic operations test using Matlab, GNU Octave, FreeMat, and Scilab. This includes testing the backslash operator, computing eigenvalues and eigenvectors, and plotting in two dimensions. The backslash operator works identically for all of the packages to produce a solution to the linear system given. 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` to compute them. 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 similar for annotating as Matlab whereas Scilab requires 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 the major concern that despite the existence of the translator, there are some functions that require manual conversion.

Chapter 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 a 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 1,024 × 1,024 in Matlab, Octave, and FreeMat. However, in Scilab, it is only able to solve up to a mesh resolution of 256 × 256. The conjugate gradient method is implementation 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. The matrix free implementation of the conjugate gradient method allows us to solve a mesh resolution up to 2,048 × 2,048 for Matlab and Octave. However, even with the matrix free implementation, Scilab is still not able to solve the system for a resolution more than 256×256. In FreeMat, we are unable to report any results using conjugate gradient method due to the lack of conjugate gradient function `pcg`. 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.

The syntax of Octave and FreeMat is identical to that of Matlab in our tests. However, 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 a m-file translator, it often still requires manual conversions.

The tests in this work 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 next section contains some additional remarks on features of the software packages that might be useful, but go beyond the tests performed in Chapters 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. Finally, the Appendices A through D collect download information for the packages and the code used in the experiments.

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.

It becomes clear that all software packages considered have at least one solver. Matlab, Octave, and Scilab 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 by these authors for Matlab demonstrated in the past 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 [7]. Since our investigations in [7], 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.

Still within the first form of parallel computing, Matlab offers the *Parallel Computing Toolbox* for a reasonable, fixed license fee (few hundreds of 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 also 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 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 [5]. 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.

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

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

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

than floating license keys). Freely downloadable packages are an important alternative, since they can be downloaded to the researchers own desktop for convenience or to any other or to multiple machines for more effective use. The more complex test case in Chapter 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) give 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, 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 Bangaret 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 Basic Linear Algebra Subroutines (BLAS) and Linear Algebra Package (LAPACK).

1.3.2 GNU Octave

"GNU Octave is a high-level language, primarily intended for numerical computations," as the reference for more information about GNU Octave is `www.gnu.org/software/octave`. 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, GNU Octave also uses the LAPACK and BLAS libraries. The syntax of Octave is very similar to Matlab, which allows Matlab users to easily begin adapting the package. Octave is available to download on different operating systems like Windows, Mac OS, and Linux. The downloading instructions for Windows can be found in Appendix A.1. 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 commercial 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 `freemat.sourceforge.net`. 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. The downloading instructions for FreeMat are in Appendix A.2

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, and Windows platforms. Like Octave, the source code is available for usage as well as for editing. For downloading instructions, refer to Appendix A.3. Scilab uses the two numerical libraries, LAPACK and BLAS. Unlike Octave, the syntax and built-in Scilab functions may not entirely agree with Matlab.

1.4 Description of the Computing Environment

The computations for this study are performed using Matlab R2007a Student, GNU Octave 3.2.4, FreeMat v4.0, and Scilab-5.2.2 under the Windows XP operating system. The computer used has an Intel Core 2 Duo E6300 DC 1.86 GHz with 1.5 GB of memory. The studies were performed in July 2010.

Chapter 2

Basic Operations Test

This chapter 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, and Scilab. We will first begin by solving a linear system; then finding eigenvalues and eigenvectors of a square matrix; and finally, 2-D functional plotting.

2.1 Basic operations in Matlab

This section discusses the results obtained using Matlab operations.

2.1.1 Solving Systems of Equations

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 is $(1, -1, 2)$. In order to use Matlab to solve this system, 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 $x = A \setminus b$. 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].
```

Now use `x = A\b` to solve this system. The resulting vector which is assigned to x is:

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

Notice the solution is exactly what was expected based on our hand computations.

2.1.2 Calculating Eigenvalues and Eigenvectors

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 + 0.7071it
```

and

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

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.

2.1.3 2-D Plotting

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 a data file located at `www.umbc.edu/circ/workshops`. 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)
```

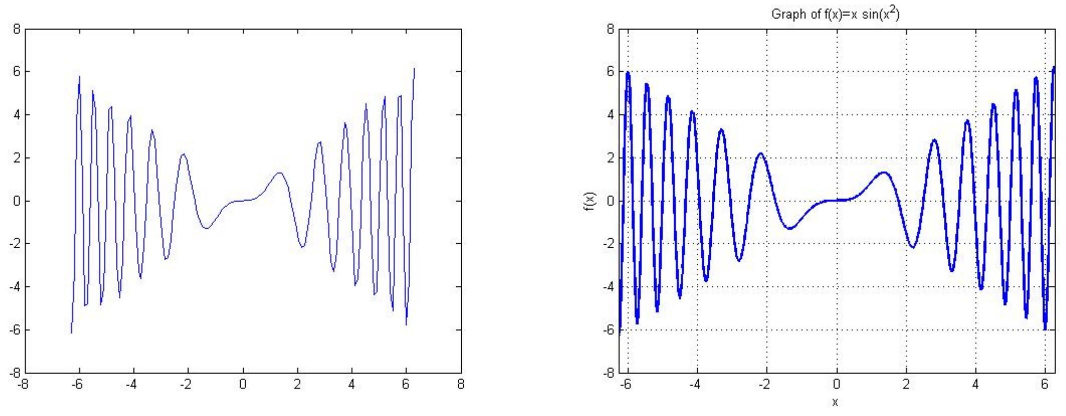


Figure 2.1: Plots of $f(x) = x\sin(x^2)$ in Matlab using (a) 129 and (b) 1025 equally spaced data points.

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. 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. `axis on` and `grid on` can be used to create the axis and the grid lines. The axes are on by default and we can turn them off if necessary using `axisoff`. 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 and lets 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 $f(x)$. 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 annotations 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 $f(x)$.

2.1.4 Programming

Here we will look at a basic example of Matlab programming using the script file. Lets try to plot Figure 2.1 (b) using a script file called `plotxsinx.m`. Instead of typing multiple commands in Matlab, we will collect these commands into this script. First, open the editor using `edit plotxsinx.m`, and then type all of the commands in this script file to create the plot with the annotations for

$f(x) = x\sin(x^2)$. Refer to Appendix B.1 for this m-file. Now, call `plotxsinx` on the command-line to execute it. The plot obtained in this case is Figure 2.1 (b). This plot can be saved using the following command:

```
print -djpeg file_name_here.jpeg
```

2.2 Basic operations in Octave

In this section, we will perform the basic operations on GNU Octave as mentioned at the beginning of the chapter. Let us begin by solving the linear system. 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.

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:

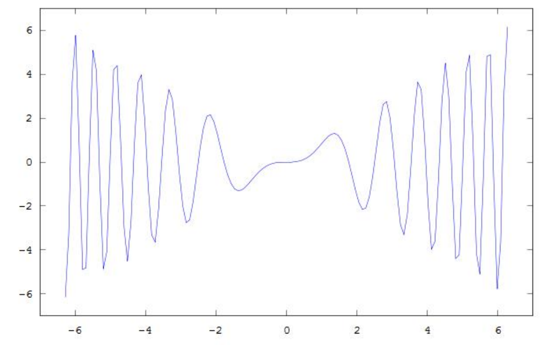
```
v =
    1 + 1i
    1 - 1i
```

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

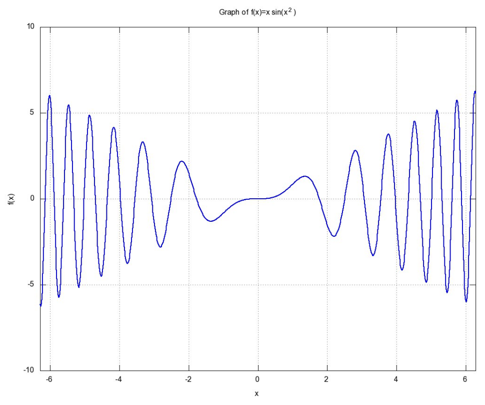
```
P =
   0.70711 + 0.00000i   0.70711 - 0.00000i
   0.00000 - 0.70711i   0.00000 + 0.70711i


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 same as the calculated solution, where as Matlab displays them using four decimal places for real and imaginary parts. Hence, the solution is 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



(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.

```
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.

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 via 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. 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 the `label` to label the axes; `title` to create a title; and `gridon` 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 the way described in Section 2.1.4 and generate the Figure 2.1 (b). One additional command we can use to save the plot is:

```
print -djpeg file_name_here.jpeg
```

Like Matlab, the script file can be saved as a m-file in Appendix B.1 to produce this outcome.

2.3 Basic operations in FreeMat

We will begin by first solving a linear system in FreeMat. 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.

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.

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. 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, `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

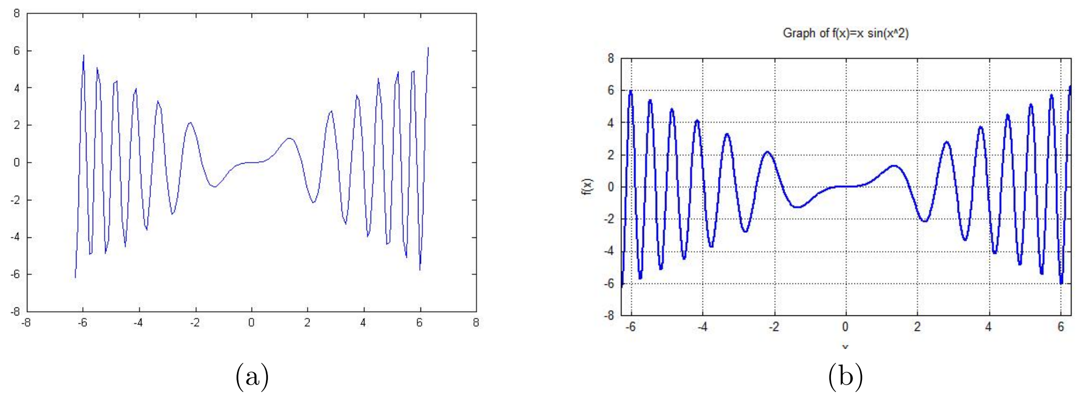


Figure 2.3: Plots of $f(x) = x\sin(x^2)$ in FreeMat using (a) 129 and (b) 1025 equally spaced data points.

plot this data using `plot(x,y);`. Applying the annotation techniques, we generate Figure 2.3 (b). Use `print('file_name_here.jpeg')` to save the plot.

Like Matlab and Octave, we can also put together these commands in an m-file. Use the m-file in Appendix B.1 to create Figure 2.3 (b).

2.4 Basic operations in Scilab

Once again, let us begin by solving the linear system from Section 2.1.1. Scilab follows the same method as GNU 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
```

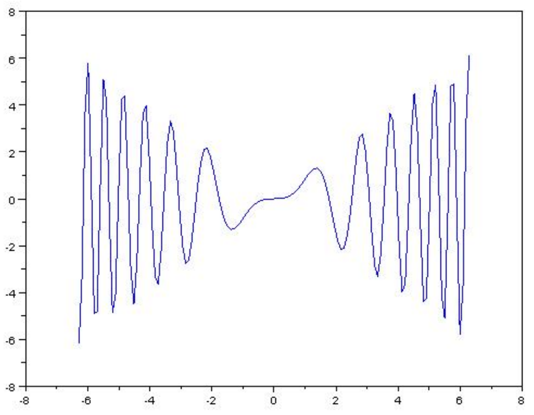
to set up the matrix A and vector b. Using the backslash operator, we obtain the result:

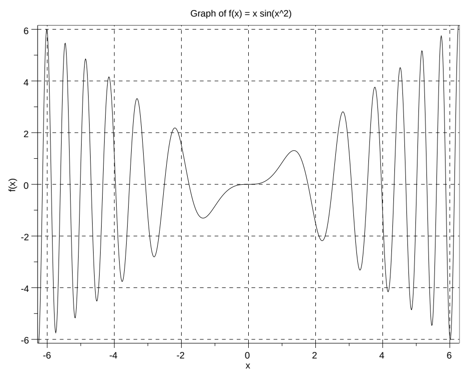
```
x =
     1
    -1
     2
```

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

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
```


(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.

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.

Now, we will plot $f(x) = x\sin(x^2)$ in Scilab. Unlike Matlab, we are unable load the existing data file into Scilab and create vectors from each column. Scilab requires that each column of the stored file has been saved as vectors. So, we will have to store each column of this file as a vector. In another words, create vector `x` and `y` using the columns from the data file `matlabdata.dat` and then save it using `save('matlabdata.dat',x,y,)`. Once the file is saved, it can be loaded using the command `load('matlabdata.dat','x','y')` and create the plot using `plot(x,y)`. Notice that the Figure 2.4 (a) is not labeled and it is rather coarse. Let us improve our resolution 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. Let us use the script file in Section 2.1.4 to create a sci-file. 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. Appendix B.2 includes the code converted using the translator. Using this script file, we are able to obtain a graph of $f(x)$ which is similar to Figure 2.1 (b). To send these graphics to JPG file, we can use `xs2jpg(gcf(),'file_name_here.jpg')`.

Chapter 3

Complex Operations Test

3.1 The Test Problem

Starting in this section, we study a classical test problem given by the numerical solution with finite differences for the Poisson problem with homogeneous Dirichlet boundary conditions [1, 3, 4, 8], given as

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

We have studied this problem before in a technical report [7]. 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 [8]. 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

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

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 $\Omega_h = (x_i, y_j)$ with $x_i = ih, i = 0, \ldots, N + 1, y_j = jh, j = 0, \ldots, N + 1$ where $h = \frac{1}{N+1}$. 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 $u_{ij} \approx u(x_i, y_j)$:

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

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

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

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 [3, 8]; hence, there exists a unique solution to the linear system.

To create the matrix A in Matlab effectively, we use the observation that it is given by a sum of two Kronecker products [3, 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 $A = I \otimes T + T \otimes I$. These formulas are the basis for the code in the function `setupA` shown in Appendix C.1.

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 and the approximation u_h defined on the mesh points. 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 as $\|u - u_h\|_{L^\infty(\Omega)} \leq C\,h^2$, as $h \to 0$, where C is a constant independent of h [2, 4]. For sufficiently small h, we can then expect that the ratio of errors on consecutively refined meshes behaves like

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

Thus, we will print this ratio in the following tables in order to confirm convergence of the finite difference method.

3.3 Matlab Results

3.3.1 Gaussian Elimination

Let us begin solving this linear system via Gaussian elimination. We know that this is easiest approach for solving linear systems for the user of Matlab, but it may not necessarily be the best method for large systems. To create matrix A, we make use of the Kronecker tensor product. This can be easily performed in Matlab using `kron` command. Refer to Appendix C.1 for Matlab code use to solve the linear system. In the process of solving this problem, we create Table 3.1 (a), which lists the mesh resolution N, where $N = 2^\nu$ for $\nu = 1, 2, 3, \ldots, 11$, for mesh of size $N \times N$; the number of degrees of freedom (DOF) which is dimension of the linear system N^2; the norm of the finite difference error; the ratio of consecutive error norms; and the observed wall clock time in HH:MM:SS.

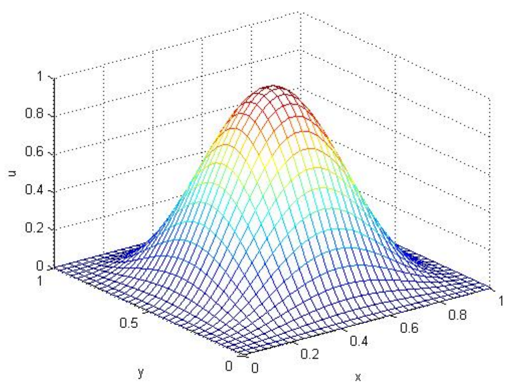
The norms of the finite difference errors clearly go to zero as the mesh resolution N increases. The ratios between error norms for consecutive rows in the table tend to 4, which confirms that the finite difference method for this problem is second-order convergent with errors behaving like h^2, as predicted by the finite difference theory. By looking at this table, it can be concluded that Gaussian elimination runs out of memory for N = 2,048. Hence, we are unable to solve this problem for N larger than 1,024 via Gaussian elimination. This leads to the need of another method to solve larger systems. Thus, we will use an iterative method known as the conjugate gradient method to solve this linear system.

Figure 3.1 (a) shows the mesh plot of the numerical solution vs. (x, y). The error at each point of the mesh is computed by comparing the numerical solution to the actual solution by $u - u_h$ and plotted in Figure 3.1 (b). Notice that the maximum error occurs at the center.

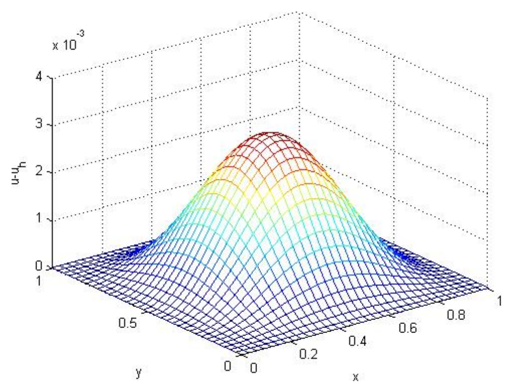
3.3.2 Conjugate Gradient Method

Now, we will use the conjugate gradient method to solve the Poisson problem [1, 8]. In the process of solving, we will use Matlab's `pcg` function which is a m-file stored at

```
\MATLAB\R2007a Student\toolbox\matlab\sparfun\pcg.m
```



(a)



(b)

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

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.

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	00:00:03
1,024	1,048,576	3.1313e-6	3.9922	00:00:31
2,048		out of memory		

(a)

N	DOF	$\|u - u_h\|$	Ratio	`#iter`	Time
32	1,024	3.0128e-3	N/A	48	00:00:06
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	00:00:08
512	262,144	1.2494e-5	3.9856	783	00:01:10
1,024	1,048,576	3.1266e-6	3.9961	1,581	00:09:26
2,048	4,194,304	7.8019e-7	4.0075	3,192	01:18:46

(b)

To use `pcg`, we supply a symmetric positive definite matrix A and a right hand side vector b. We also use the initial guess as the zero vector and the tolerance to be 10^{-6} on the relative residual of the iterates.

The system matrix A accounts for the largest amount of memory used by the conjugate gradient method. Hence, to solve the problem for larger meshes, we can use a so-called matrix-free implementation of the method that avoids storing A. The only place, where A enters into the iterative method is the matrix-vector multiplication $q = Ap$ in each iteration. Hence, for a matrix-free implementation, instead of supplying A itself, the user supplies a function that computes the matrix-vector product $q = Ap$ directly for a given input vector p without ever storing a system matrix A. Thus, the matrix-free implementation returns a vector q for the input vector p by performing the component-wise product on p and the matrix A using the knowledge of the components of A. Thus, we can expect to save memory significantly.

Now, we will use this matrix-free implementation to solve the problem and report the results in Table 3.1 (b). Refer to Appendix C.2 for Matlab code used to solve the linear system using the conjugate gradient method. We must compare this outcome to Table 3.1 (a) to confirm the convergence of this method. In order to guarantee the convergence, we must compare the finite difference error and convergence order for each value of N. The finite difference error shows the same behavior with ratios of consecutive errors approaching 4 as for Gaussian elimination; this confirms that the tolerance on the relative residual of the iterates is tight enough. The comparison of the table concludes that these parameters match exactly, hence the matrix-free method accurately converges. By eliminating the storage of A, we should have boosted our memory savings.

The comparison of Table 3.1 (a) and Table 3.1 (b) also shows that we are able to solve the problem on a mesh with a finer resolution than by Gaussian elimination, namely up to $N = 2{,}048$.

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 on Octave. Just like Matlab, we can solve the equation using the backslash operator as discussed in Section 2.2. Let us the m-file from Appendix C.1 to create Table 3.2 (a). This table lists the mesh resolution N, where $N = 2^\nu$ for $\nu = 1, 2, 3, \ldots, 11$, for mesh of size $N \times N$; the number of degrees of freedom (DOF) which is dimension of the linear system N^2; the norm of the finite difference error; the ratio of consecutive error norms; and the observed wall clock time in HH:MM:SS.

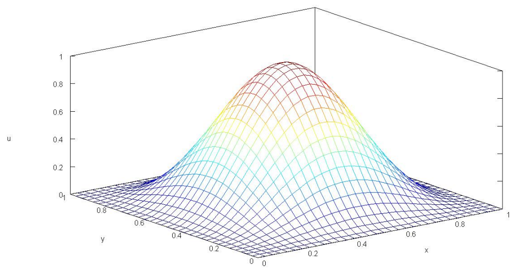
The results in the Table 3.2 (a) are identical to results in the Table 3.1 (a). Hence, the backslash operator works the same way. Figure 3.2 (a) is the mesh plot of the solution vs. (x, y) and Figure 3.2 (b) is the mesh plot of the error vs. (x, y).

3.4.2 Conjugate Gradient Method

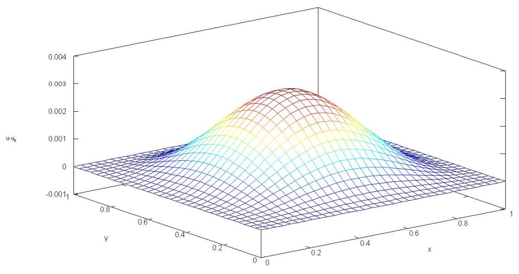
Now, let us try to solve the problem using the conjugate gradient method. Just like Matlab, there exists a `pcg.m` file that can be located at:

```
Octave\3.2.4_gcc-4.4.0\share\octave\3.2.4\m\sparse.
```

According to this m-file, it has been created by Piotr Krzyzanowki and later modified by Vittoria Rezzonio. The input requirements for the `pcg` function are identical to the Matlab's `pcg` function. Once again, we use the zero vector as the initial guess and the tolerance is 10^{-6} on the relative



(a)



(b)

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

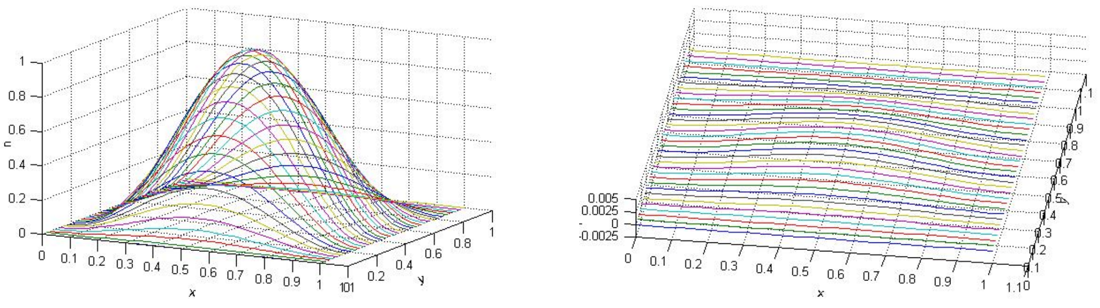
Table 3.2: Convergence results for the test problem in Octave using (a) Gaussian elimination and (b) the conjugate gradient method. The tables list the mesh resolution N, the number of degrees of freedom (DOF), the finite difference norm $\|u - u_h\|_{L^\infty(\Omega)}$, the ratio of consecutive errors,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

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	00:00:05
1,024	1,048,576	3.1313e-6	3.9922	00:00:40
2,048		out of memory		

(a)

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	00:00:06
512	262,144	1.2494e-5	3.9856	783	00:00:45
1,024	1,048,576	3.1266e-6	3.9961	1,581	00:06:25
2,048	4,194,304	7.8019e-7	4.0075	3,192	00:50:01

(b)



(a) (b)

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

residual of the iterates. We can use the m-file in Appendix C.2 to create a table which lists the mesh resolution; dimension of the linear system; norm of the finite difference error; convergence ratio; `#iter`, which is the number of iterations it takes the method to converge; the run time. In order to confirm the convergence, we must compare the finite difference error, convergence order, and the iteration count for each value of N to the results reported in Table 3.1 (b).

The comparison of Table 3.1 (b) and Table 3.2 (b) concludes that we are able to solve the problem on a mesh with a finer resolution that by Gaussian elimination in Matlab and Octave, namely up to $N = 2{,}048$. Just like before, the finite difference error shows the same behavior with ratios of consecutive errors approaching 4; this confirms that the tolerance on the relative residual of the iterates is tight enough. In the results for the conjugate gradient method, as printed here, Octave solves the problem faster than Matlab. Studying the `pcg` m-files in our version of Matlab and Octave, we observe that Matlab uses two matrix vector products per iteration in this 2007 version (this has since been improved), where as Octave uses only one. It is for that reason that Octave is faster than Matlab here. This issue brings out the importance of having an optimal implementation of this important algorithm.

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. As mentioned in Section 2.3, the linear system can be solved using the backslash operator. We can use the m-file in Appendix C.1 to solve the problem using the direct solver. However, while creating matrix A, we are unable to use the `kron` function in FreeMat since it does not exist. In order to perform Kronecker tensor product, we use the `kron.m` file from Matlab to compute this product. In the process of solving via Gaussian eliminations, let us create a Table 3.3, which lists the mesh resolution N, where $N = 2^\nu$ for $\nu = 1, 2, 3, \ldots, 11$, for mesh of size $N \times N$; the number of degrees of freedom (DOF) which is dimension of the linear system N^2; the norm of the finite difference error; the ratio of consecutive error norms; and the observed wall clock time in HH:MM:SS.

The results in the Table 3.3 are basically identical to Table 3.1 (a) except namely the time that it took to solve the problem.

Since `mesh` does not exist, we used the `plot3` command to create Figure 3.3 (a) the plot3 of the

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.

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	00:00:19
1,024	1,048,576	3.1313e-6	3.9922	00:01:07
2,048		out of memory		

numerical solution vs. (x, y) and Figure 3.3 (b) the plot3 of the error vs. (x, y). Despite the view adjustments for these figure using the `view` command, Figure 3.3 (b) is still very hard to study.

3.5.2 Conjugate Gradient Method

Unlike Matlab and Octave, a `pcg` function does not exist in FreeMat. Since the purpose of this study is to examine the existing functions, we did not study this problem using the conjugate gradient method.

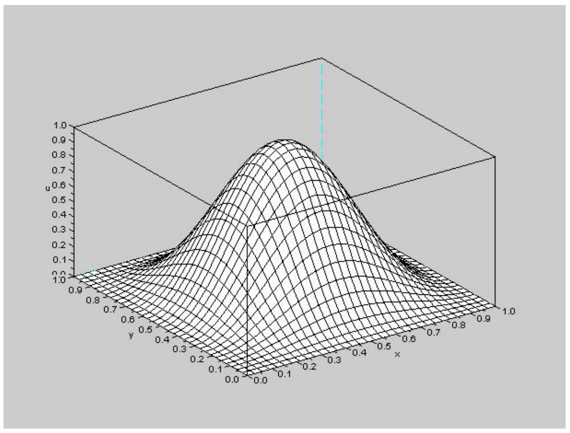
3.6 Scilab Results

3.6.1 Gaussian Elimination

Once again, we will solve the Poisson equation via Gaussian elimination on 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. So we make a use of this command while creating matrix A. Let us use the sci-file from Appendix D.1 to solve the problem. In the process of solving this problem, we create Table 3.4 (a), which lists the mesh resolution N, where $N = 2^\nu$ for $\nu = 1, 2, 3, \ldots, 11$, for mesh of size $N \times N$; the number of degrees of freedom (DOF) which is dimension of the linear system N^2; the norm of the finite difference error; the ratio of consecutive error norms; and the observed wall clock time in HH:MM:SS.

As expected, 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, 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. Unlike Matlab and Octave, Gaussian elimination runs out of memory for $N = 512$ which is much earlier. Hence, we are unable to solve this problem for N larger than 256 via Gaussian elimination in Scilab. At this point, we hope to solve a larger system using the conjugate gradient method.

Figure 3.4 (a) is a mesh plot of numerical solution for a mesh resolution of 32×32. and Figure 3.4 (b) is a plot of the error associated with each numerical solution.


(a)

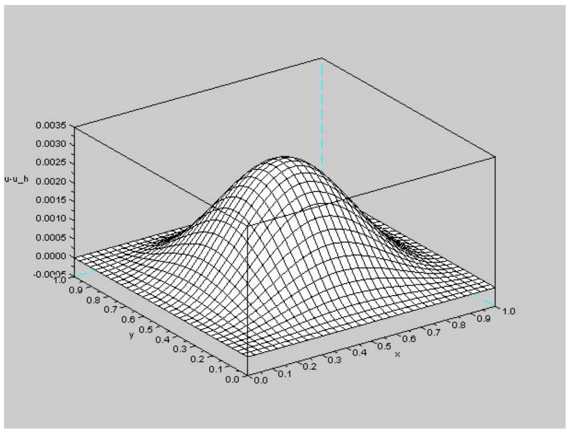

(b)

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

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.

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	00:00:17
256	65,536	4.9807e-5	3.9685	00:04:26
512		out of memory		
1,024		out of memory		
2,048		out of memory		

(a)

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	00:00:02
128	16,384	1.9765e-4	3.9368	192	00:00:02
256	65,536	4.9797e-5	3.9690	387	00:00:07
512		out of memory			
1,024		out of memory			
2,048		out of memory			

(b)

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. This is a `sci` file that can be located using the following information:

`\scilab-5.2.2\modules\sparse\macros.`

In order to solve, the initial guess is the zero vector and the tolerance is 10^{-6} on the relative residual of the iterates. Just like before, we can use the sci-file in Appendix D.2 and create a table which lists the mesh resolution; dimension of the linear system; norm of the finite difference error; convergence ratio; `#iter`, which is the number of iterations it takes the method to converge; and the run time. Notice that the finite difference error shows the same behavior with ratios like all of the other methods. The consecutive errors approach 4 which confirms that the tolerance on the relative residual of the iterates is tight enough.

The analysis of Table 3.4 (a) and Table 3.4 (b) shows that we are not able to solve the problem on a mesh with a finer resolution than by Gaussian elimination. The conjugate gradient method does not seem to help in this matter.

Appendix A

Downloading Instructions

A.1 Octave

As mention earlier, Octave is a free software that can be easily downloaded using the following steps:

1. Download GNU Octave from `sourceforge.net/projects/octave` by clicking on download and download again on the next page. The file size is 69.6 MB which takes about 1 minute to download.
2. Once the download is completed, click on run to install.
3. Accept and then decide whether or not to install Softonic Toolbar. Click on Next at the end.
4. Now GNU Octave 3.2.4 Setup will begin. Select Next.
5. Accept the License Agreement by clicking on Next.
6. Choose Installation location.
7. Choose the features of GNU Octave 3.2.4 you want to install and proceed.
8. The installation takes several minutes and select Finish.

A.2 FreeMat

FreeMat can be easily installed on a windows operating system using the following instructions:

1. Download `FreeMat-4.0-svn_rev4000-win-.exe` from `sourceforge.net/projects/freemat/files`.
2. After the download is completed, run the software by double clicking on it.
3. Click on Next to continue.
4. Accept the agreement and click on Next to continue.
5. Now, select the installation location.
6. Now you are ready to install. Just click on Install and it will take a few minutes.
7. Installation is complete.

A.3 Scilab

Scilab can be easily down on a windows operating system using the following instructions:

1. Download `Scilab 5.2.2 for 32 bits (Recommended)` from `www.scilab.org/products/scilab/download`.
2. After the download is completed, run the software by double clicking on it.
3. Select the language used the setup the program.
4. Click on Next to continue.
5. Accept the agreement and click on Next to continue.
6. Now, select the destination location for the installation.
7. To install all of the components, select full installation.
8. Create program shortcuts.
9. Select any additional tasks that should be performed.
10. Now you are ready to install. Just click on Install and it will take several minutes.
11. Installation is complete.

Appendix B

Matlab and Scilab code used to create 2-D plot

B.1 Matlab code

```
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])
```

B.2 Scilab code

The following is the Scilab file for plotxsinx that corresponds to the Matlab code in Section B.1.

```
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)")
```

Appendix C

Matlab Codes Used to Solve the Test Problem

C.1 Gaussian elimination

The Matlab code for solving the test problem via Gaussian elimination method is shown in the following. This code is used for Matlab, Octave, and FreeMat. Since the latter one does not have the `mesh` function for three-dimensional plotting, it is replaced by the `plot3` function and also the view of the plot manually adjusted; see the comments in the m-file.

```
function driver_ge(N)

  h = 1 / (N+1);
  x = [h : h : 1-h];
  y = x;
  [X, Y] = ndgrid(x,y);
  F = (-2*pi^2) * (cos(2*pi*X).*(sin(pi*Y)).^2 + (sin(pi*X)).^2.*cos(2*pi*Y));
  b = h^2 * F(:);
  clear X Y F;

  tic;
  A = setupA(N);
  u = A \ b;
  Uint = reshape(u, [N N]); % N.B.: Uint has only solutions on interior points
  timesec = toc;

  % append boundary to x, y, and to U:
  x = [0 x 1];
  y = [0 y 1];
  [X, Y] = ndgrid(x,y);
  U = zeros(size(X));
  U(2:end-1,2:end-1) = Uint;

  % plot numerical solution:
  figure
  H = mesh(X,Y,U);   % for Matlab and Octave
  %plot3(X,Y,U)      % for FreeMat
  xlabel('x');
  ylabel('y');
  zlabel('u');
  %view(27.5,20)     % for FreeMat
  % compute and plot numerical error:
  Utrue = (sin(pi*X)).^2 .* (sin(pi*Y)).^2;
  E = U - Utrue;
  figure
  H = mesh(X,Y,E);   % for Matlab and Octave
  %plot3(X,Y,E)      % for FreeMat
  xlabel('x');
  ylabel('y');
  zlabel('u-u_h');
  %view(8.5,2)       % for FreeMat

  % compute L^inf norm of error and print:
  enorminf = max(abs(E(:)));
  fprintf('N = %5d\n', N);
  fprintf('h                 = %24.16e\n', h);
  fprintf('h^2               = %24.16e\n', h^2);
  fprintf('enorminf          = %24.16e\n', enorminf);
  fprintf('C = enorminf / h^2 = %24.16e\n', (enorminf/h^2));
  fprintf('wall clock time    = %10.2f seconds\n', timesec);


function A = setupA(N)
  I = speye(N);
  s = [-1*ones(1,N-1) 2*ones(1,N) -1*ones(1,N-1)]';
  i = [2:N    1:N  1:N-1]';
  j = [1:N-1  1:N  2:N  ]';
  T = sparse(i,j,s);
  A = kron(I,T) + kron(T,I);
```

C.2 Conjugate gradient method

The Matlab code for solving the test problem via the conjugate gradient method is shown in the following.

```
function driver_cg(N)

  h = 1 / (N+1);
  x = [h : h : 1-h];
  y = x;
  [X, Y] = ndgrid(x,y);
  F = (-2*pi^2) * (cos(2*pi*X).*(sin(pi*Y)).^2 + (sin(pi*X)).^2.*cos(2*pi*Y));
  b = h^2 * F(:);
  clear X Y F;

  tic;
  A = @Ax;
  tol = 1.0e-6;
  maxit = 9999;
  u = zeros(N^2,1);
  [u,flag,relres,iter,resvec] = pcg(A,b,tol,maxit,[],[],u);
  Uint = reshape(u, [N N]); % N.B.: Uint has only solutions on interior points
  timesec = toc;

  % append boundary to x, y, and to U:
  x = [0 x 1];
  y = [0 y 1];
  [X, Y] = ndgrid(x,y);
  U = zeros(size(X));
  U(2:end-1,2:end-1) = Uint;

  % plot numerical solution:
  figure;
  H = mesh(X,Y,U);
  xlabel('x');
  ylabel('y');
  zlabel('u');

  % compute and plot numerical error:
  Utrue = (sin(pi*X)).^2 .* (sin(pi*Y)).^2;
  E = U - Utrue;
  figure;
  H = mesh(X,Y,E);
  xlabel('x');
  ylabel('y');
  zlabel('u-u_h');

  % compute L^inf norm of error and print:
  enorminf = max(abs(E(:)));
  fprintf('N = %5d, tol = %10.1e, maxit = %d\n', N, tol, maxit);
  fprintf('flag = %1d, iter = %d, relres = %24.16e\n', flag, iter, relres);
  fprintf('h                  = %24.16e\n', h);
  fprintf('h^2                = %24.16e\n', h^2);
  fprintf('enorminf           = %24.16e\n', enorminf);
  fprintf('C = enorminf / h^2 = %24.16e\n', (enorminf/h^2));
  fprintf('wall clock time    = %10.2f seconds\n', timesec);


function v = Ax(u)

  N = sqrt(length(u));
  U = reshape (u, [N N]);
  V = 4 * U;                                % V_{ij}  = 4 u_{ij} initialized

  V(:,2:end  ) = V(:,2:end  ) - U(:,1:end-1); % V_{ij} -= u_{ij-1} accumulated
  V(2:end  ,:) = V(2:end  ,:) - U(1:end-1,:); % V_{ij} -= u_{i-1j} accumulated
  V(1:end-1,:) = V(1:end-1,:) - U(2:end  ,:); % V_{ij} -= u_{i+1j} accumulated
  V(:,1:end-1) = V(:,1:end-1) - U(:,2:end  ); % V_{ij} -= u_{ij+1} accumulated

  v = V(:);
```

Appendix D

Scilab Conversion of the Matlab Codes Used to Solve the Test Problem

D.1 Gaussian elimination

The Scilab code for solving the test problem via Gaussian elimination method is shown in the following.

```
function driver_ge(N)
h = 1/(N+1);
x = h:h:1-h;
y = x;
[X,Y] = ndgrid(x,y);
F = (-2*(%pi^2))*mtlb_a(cos((2*%pi)*X) .*(sin(%pi*Y) .^2),(sin(%pi*X) .^2)
     .*cos((2*%pi)*Y));
b = (h^2)*F(:);
clear("X","Y","F");

tic;
A = setupA(N);
u = A\b;
Uint = matrix(u,[N N]); // N.B.: Uint has only solutions on interior points
timesec = toc();

// append boundary to x, y, and to U:
x = [0,x,1];
y = [0,y,1];
[X,Y] = ndgrid(x,y);
%v0 = size(mtlb_double(X));U = zeros(%v0(1),%v0(2));
U(2:$-1,2:$-1) = mtlb(Uint);

// plot numerical solution:
figure
H = mesh(X,Y,U)
xlabel("x");
ylabel("y");
zlabel("u");

// compute and plot numerical error:
Utrue = (sin(%pi*X) .^2) .*(sin(%pi*Y) .^2);
E = mtlb_s(U,Utrue);
figure
```

```
H = mesh(X,Y,E)
xlabel("x");
ylabel("y");
zlabel("u-u_h");

// compute L^inf norm of error and print:
%v0 = abs(E(:));enorminf = max(%v0,firstnonsingleton(%v0));
// L.46: No simple equivalent, so mtlb_fprintf() is called.
mtlb_fprintf("N = %5d\n",N);
// L.47: No simple equivalent, so mtlb_fprintf() is called.
mtlb_fprintf("h                  = %24.16e\n",h);
// L.48: No simple equivalent, so mtlb_fprintf() is called.
mtlb_fprintf("h^2                = %24.16e\n",h^2);
// L.49: No simple equivalent, so mtlb_fprintf() is called.
mtlb_fprintf("enorminf           = %24.16e\n",enorminf);
// L.50: No simple equivalent, so mtlb_fprintf() is called.
mtlb_fprintf("C = enorminf / h^2 = %24.16e\n",enorminf/(h^2));
// L.51: No simple equivalent, so mtlb_fprintf() is called.
mtlb_fprintf("wall clock time    = %10.2f seconds\n",timesec);
endfunction

function A = setupA(N)
s = [-1*ones(1,N-1) 2*ones(1,N) -1*ones(1,N-1)]'
i = [2:N 1:N 1:N-1]'
j = [1:N-1  1:N  2:N  ]';
ij = [i j]
T = sparse(ij,s)
I = speye(N,N);
A = mtlb_a(mtlb_double(I) .*. T,T .*. mtlb_double(I));
endfunction
```

D.2 Conjugate gradient method

The Scilab code for solving the test problem via the conjugate gradient method is shown in the following.

```
function driver_cg(N)
h = 1/(N+1);
x = h:h:1-h;
y = x;
[X,Y] = ndgrid(x,y);
F = (-2*(%pi^2))*mtlb_a(cos((2*%pi)*X) .*(sin(%pi*Y) .^2),(sin(%pi*X) .^2)
    .*cos((2*%pi)*Y));
b = (h^2)*F(:);
clear("X","Y","F");

tic;
A = Ax;
tol = 0.000001;
maxit = 9999;
u = zeros(N^2,1);
[u,flag,iter,res] = pcg(A,b,tol,maxit,[],[],u);
Uint = matrix(u,[N,N]);// N.B.: Uint has only solutions on interior points
timesec = toc();
relres = res($)

// append boundary to x, y, and to U:
x = [0,x,1];
```

```
y = [0,y,1];
[X,Y] = ndgrid(x,y);
%v0 = size(mtlb_double(X));U = zeros(%v0(1),%v0(2));
U(2:$-1,2:$-1) = Uint;

// plot numerical solution:
figure
H = mesh(X,Y,U)
xlabel("x");
ylabel("y");
zlabel("u");

// compute and plot numerical error:
Utrue = (sin(%pi*X) .^2) .*(sin(%pi*Y) .^2);
E = mtlb_s(U,Utrue);
figure
H = mesh(X,Y,E)
xlabel("x");
ylabel("y");
zlabel("u-u_h");

// compute L^inf norm of error and print:
%v0 = abs(E(:));enorminf = max(%v0,firstnonsingleton(%v0));
// L.51: No simple equivalent, so mtlb_fprintf() is called.
mtlb_fprintf("N = %5d, tol = %10.1e, maxit = %d\n",N,tol,maxit);
// L.52: No simple equivalent, so mtlb_fprintf() is called.
mtlb_fprintf("flag = %1d, iter = %d, res = %24.16e\n",flag,iter,relres);
// L.53: No simple equivalent, so mtlb_fprintf() is called.
mtlb_fprintf("h                  = %24.16e\n",h);
// L.54: No simple equivalent, so mtlb_fprintf() is called.
mtlb_fprintf("h^2                = %24.16e\n",h^2);
// L.55: No simple equivalent, so mtlb_fprintf() is called.
mtlb_fprintf("enorminf           = %24.16e\n",enorminf);
// L.56: No simple equivalent, so mtlb_fprintf() is called.
mtlb_fprintf("C = enorminf / h^2 = %24.16e\n",enorminf/(h^2));
// L.57: No simple equivalent, so mtlb_fprintf() is called.
mtlb_fprintf("wall clock time    = %10.2f seconds\n",timesec);
endfunction

function v = Ax(u)
N = sqrt(max(size(mtlb_double(u))));
U = matrix(u,[N,N]);
size(U)
V = 4*U;// V_(ij).entries = 4 u_(ij).entries initialized
V(:,2:$) = mtlb_s(V(:,2:$),U(:,1:$-1));
// V_(ij).entries -= u_(ij-1).entries accumulated
V(2:$,:) = mtlb_s(V(2:$,:),U(1:$-1,:));
// V_(ij).entries -= u_(i-1j).entries accumulated
V(1:$-1,:) = mtlb_s(V(1:$-1,:),U(2:$,:));
// V_(ij).entries -= u_(i+1j).entries accumulated
V(:,1:$-1) = mtlb_s(V(:,1:$-1),U(:,2:$));
// V_(ij).entries -= u_(ij+1).entries accumulated
v = V(:)
endfunction
```

Bibliography

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

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

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

[6] Neeraj Sharma. A comparative study of several numerical computational packages. M.S. thesis, University of Maryland, Baltimore County, 2010.

[7] 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.

[8] David S. Watkins. *Fundamentals of Matrix Computations.* Wiley, second edition, 2002.