

Study of Free Alternative Numerical Computation Packages

Matthew W. Brewster (bmatt3@umbc.edu)

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

Abstract

Matlab is the most popular commercial package for numerical computations in mathematics, statistics, the sciences, engineering, and other fields. Octave, FreeMat, and Scilab are free numerical computational packages that have many of the same features as Matlab. They are available to download on the Linux, Windows, and Mac OS X operating systems. We investigate whether these packages are viable alternatives to Matlab for uses in teaching and research. We compare the packages under Linux on one compute node with two quad-core Intel Nehalem processors (2.66 GHz, 8 MB cache) and 24 GB of memory that is part of an 86-node distributed-memory cluster. After performing both usability and performance tests on Matlab, Octave, FreeMat, and Scilab, we conclude that Octave is the most usable and most powerful freely available numerical computation package. Both FreeMat and Scilab exhibited some incompatibility with Matlab and some performance problems in our tests. Therefore, we conclude that Octave is the best viable alternative to Matlab because not only was it fully compatible with Matlab, but it also exhibited the best performance. This paper reports on work done while working for the REU Site: Interdisciplinary Program in High Performance Computing at the University of Maryland, Baltimore County.

1 Introduction

There are several numerical computation packages that serve as educational tools and are also available for commercial use. Of the available packages, Matlab is the most widely used. The focus of this study is to introduce three additional numerical computation packages: Octave, FreeMat, and Scilab, and provide information on which package is most compatible to Matlab users. To evaluate Octave, FreeMat, and Scilab we use a comparative approach based on a Matlab user's perspective. To achieve this task, we perform both basic and complex studies on Matlab, GNU Octave, FreeMat, and Scilab. The basic studies in Section 2 include testing basic operations such as solving systems of linear equations, computing the eigenvalues and eigenvectors of a matrix, and two-dimensional plotting. The basic operations test also serves to highlight the differences in syntax between the all of the numerical computation packages. The complex studies in Section 3 include direct and iterative solutions of a large sparse system of linear equations resulting from finite difference discretization of an elliptic test problem illustrating the differences in power and performance between each package.

Numerical computation packages see usage both in teaching and in research. In teaching, the use of Matlab is becoming widespread in many college courses in mathematics and other fields. For instance, professors might demonstrate facts using Matlab or assign homework/projects that state "Use Matlab . . . ". But the commercial package Matlab is often only available on campus in specific computer labs maintained by the institution. Most

students find it much more convenient to do homework on their own laptops or other computers. The question is then if one of the free alternatives such as Octave, FreeMat, or Scilab could be used by the student, since these can readily be downloaded to the student's own computer. Section 2 is designed to study this.

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 many academic institutions, might not be available outside of academia due to its high license fees there. Or even if it is available, it is often limited to a particular computer (as fixed-CPU licenses tend to be cheaper than floating license keys). Freely downloadable packages are an important alternative, since they can be downloaded to the researchers own desktop for convenience 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 has 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.

The computations for this study are performed using Matlab R2011a, Octave 3.0.4, FreeMat v4.0, and Scilab-5.3.1 under the Linux operating system RedHat Enterprise Linux 5 on one node of the

(www.umbc.edu/hpcf). This cluster has a total of 86 nodes, each node featuring two quad-core Intel Nehalem X5550 processors (2.66 GHz, 8 MB cache) with 24 GB of memory.

2 Basic Operations Test

In this section we report on a basic operations test using Matlab, Octave, FreeMat, and Scilab. Basic functionalities of these software packages include (i) the solution of a system of linear equations by Gaussian elimination, (ii) finding eigenvalues and eigenvectors of a matrix, and (iii) plotting in two dimensions [3]. These operations were chosen to highlight the operations that one might see in a basic linear algebra course for which Matlab was originally designed, such as linear system solutions and eigenvalue computations. Additionally, we demonstrate and compare the 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

2.1.1 Solving Systems of Equations in Matlab

The first example we will consider in this section is solving a linear system. Consider the 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

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

First, the matrix A and vector b are entered using the commands

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

Then use the backslash operator to solve the system by Gaussian elimination by $x = A \backslash b$. 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

Now, 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 the 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
```

Calculating the diagonalization enables us to express the matrix A as

$$A = PDP^{-1}, \quad (2.2)$$

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 from a Data File in Matlab

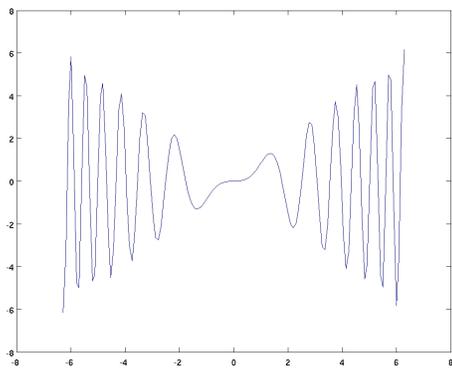
Two-dimensional plotting is a very important feature as it appears in all mathematical courses. Since this is a very commonly used feature, let us examine the 2-D plotting feature of Matlab by plotting $f(x) = x \sin(x^2)$ over the interval $[-2\pi, 2\pi]$. The data set for this function is given in a data file `matlabdata.dat` and is posted along with the technical report [3] at www.umbc.edu/hpcf under Publications. Noticing that the data are 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)
```

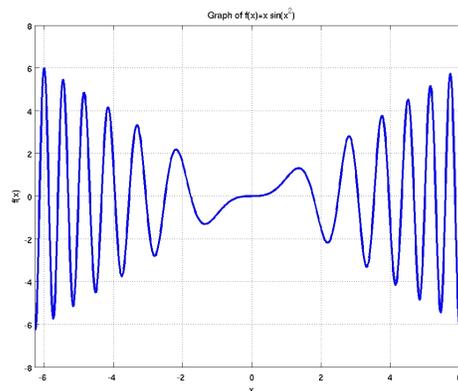
The commands stated above result in the Figure 2.1 (a). Looking at this figure, it can be noted that our axes are not labeled; there are no grid lines; and the peaks of the curves are rather coarse.

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



(a)



(b)

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

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

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

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

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

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

2.2 Basic operations in Octave

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
```

```
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:

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

```
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

```
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 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).

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 the `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 to generate Figure 2.1 (b).

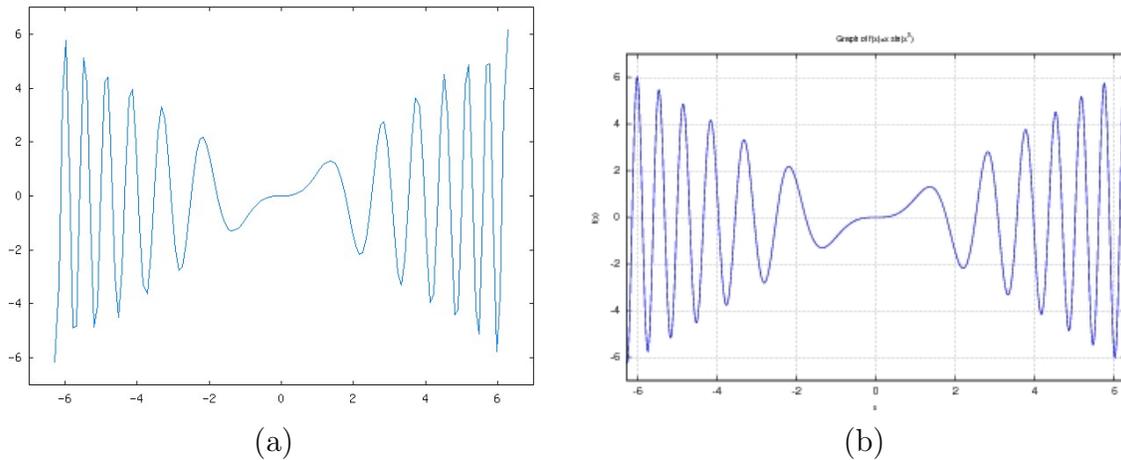


Figure 2.2: Plots of $f(x) = x \sin(x^2)$ in Octave using (a) 129 and (b) 1025 equally spaced data points.

2.3 Basic operations in FreeMat

2.3.1 Solving Systems of Equations in FreeMat

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

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

which results in

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

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

2.3.2 Calculating Eigenvalues and Eigenvectors in FreeMat

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

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

```

1.0000 + 1.0000i    0
                0    1.0000 - 1.0000i

```

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

```

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

```

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

2.3.3 2-D Plotting from a Data File in FreeMat

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

2.3.4 Annotated Plotting from Computed Data in FreeMat

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

2.4 Basic operations in Scilab

2.4.1 Solving Systems of Equations in Scilab

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

```

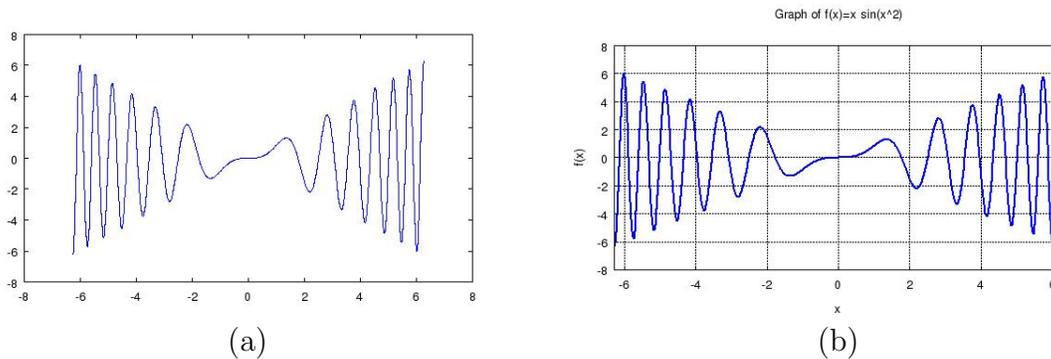


Figure 2.3: Plots of $f(x) = x \sin(x^2)$ in FreeMat using (a) 129 and (b) 1025 equally spaced data points.

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

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

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

2.4.2 Calculating Eigenvalues and Eigenvectors in Scilab

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

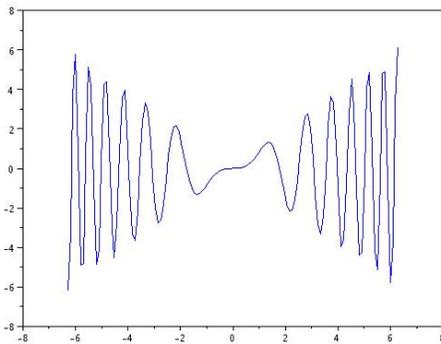
```
v =
    1. + i
    1. - i
```

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

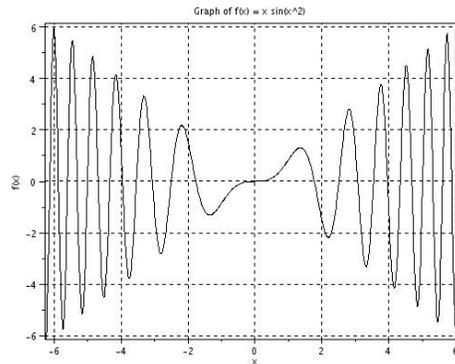
```
D =
    1 + i      0
      0    1 - i

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.



(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.

```
P*D*inv(P)
```

```
ans =
```

```
  1. - 1.
  1.  1.
```

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

2.4.3 2-D Plotting from a Data File in Scilab

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

2.4.4 Annotated Plotting from Computed Data in Scilab

Let us improve our resolution by creating vector x using

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

and let `y = x .* sin(x.^2)` to create a corresponding y vector. Unlike Matlab and Octave, we have to use `%pi` to enter π in Scilab. The operator, `.*` and `.^` still perform the element-wise operations in Scilab. Another factor that remains unchanged is the `length` command. We can generate the plot using the `plot(x,y)` command which creates the Figure 2.4 (a). Once again, we can use `xlabel` and `ylabel` to label the axes;

title('Graph of $f(x)=x \sin(x^2)$ ') to create a title; and `xgrid` to turn on grid. To plot and create x-axis bounds, use

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

Notice that we can put together these commands into a sci-file in Scilab to generate a plot. The resulting script for creating Figure 2.4 (a) is as follows:

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

Notice that some of the Matlab commands are not compatible with Scilab. One easier approach to handle this issue is to use the “Matlab to Scilab translator” under the Applications menu or by using `mfile2sci` command. The translator is unable to convert `xlim([-2*pi 2*pi])`; which we can take care of replacing the `plot` with `plot2d` command above. Using this script file, we obtain Figure 2.4 (b) which is similar to Figure 2.1 (b).

2.5 Comparison

After performing the basic operations in Matlab, we repeated the same operations in the other numerical computation packages. 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 of the common annotation commands discussed above. Once again, Octave and FreeMat use the exact commands for plotting and similar ones for annotating as Matlab, whereas Scilab requires several changes. For instance in Scilab, the constant `pi` is defined using `%pi`, and the command `grid` from Matlab is replaced with `xgrid`. To accomplish these conversions, we find that we can use the Matlab-to-Scilab translator provided by Scilab, which takes care of these command differences for the most part. However, 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 arguments in `Plot2d`. This issue brings out the major concern for Scilab that despite the existence of the translator, there are some functions that require manual conversion.

The small sample code below tests whether basic programming elements including `if`-statements and `for`-loops work the same way in all the numerical computation packages, with both snippets outputting the same sequence of numbers 1, 0.75, 0.5, 0.25, 0:

```

for c = 1.0 : -0.25 : 0.0
    c
end

c = 1.0
while (c > 0.0)
    c = c - 0.25
end

```

Executing these snippets in all packages also confirms that non-integer and negative loop increments are legal and work in the same way in all of them. The format of a function interface in all languages is `[out1, out2] = function foo (in1, in2, in3)` with output arguments on the left [in brackets] and input arguments on the right (in parentheses). Other programming elements like `break` from a loop and `return` from a function are also available in all packages and work in the same way.

Next, to assess how compatible the languages are, we compiled a list of 30 common mathematical functions in Table 2.1. The table divides the functions into six categories: basic linear algebra functions, matrix decomposition/factorization functions, iterative solvers for systems of equations, sparse matrix functions, polynomial, interpolation, and other functions, and basic statistics functions. The results show that Octave is capable of performing all of the functions listed and has essentially the same syntax. FreeMat lacks several basic commands that Matlab and Octave are capable of executing but it uses the same syntax as Matlab and Octave for the functions that it is able to execute. Scilab was capable of executing all of the basic functions, but some of its functions are named differently and have a different syntax than the functions in Matlab, Octave, and FreeMat.

Another 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. Since the ODE capabilities of the packages vary too widely and use very different numerical methods, a detailed, performance-oriented comparison between them is not practical.

Table 2.1: This table shows equivalent commands in Matlab, Octave, FreeMat, and Scilab. The notation N/A indicates that the specific function does not exist in the respective software package.

Function	Matlab	Octave	FreeMat	Scilab
Basic linear algebra functions:				
Determinant	det	det	det	det
Matrix inverse	inv	inv	inv	inv
Kronecker product	kron	kron	N/A	kronck
Matrix rank	rank	rank	rank	rank
Matrix condition number	cond	cond	cond	cond
Orthogonal basis	orth	orth	N/A	orth
Kernel/nullspace of a matrix	null	null	N/A	kernel
Pseudoinverse	pinv	pinv	pinv	pinv
Matrix decomposition/factorization functions:				
Eigenvalues of a matrix	eig	eig	eig	spec
Schur decomposition	schur	schur	N/A	schur
Singular value decomposition	svd	svd	svd	svd
QR decomposition	qr	qr	qr	qr
LU decomposition	lu	lu	lu	lu
Cholesky factorization	chol	chol	N/A	chol
Iterative solvers for systems of equations:				
Conjugate gradient method	pcg	pcg	N/A	pcg
GMRES	gmres	gmres	N/A	gmres
BiCG-STAB	bicgstab	bicgstab	N/A	bicgstab
Sparse matrix functions:				
Create sparse matrix	sparse	sparse	sparse	sparse
Sparse identity matrix	speye	speye	speye	speye
Sparse random matrix	sprand	sprand	sprand	sprand
Polynomial, interpolation, and other functions:				
Roots of polynomial	roots	roots	roots	roots
Linear interpolation	interp1	interp1	interplin1	interpln
Cubic spline	spline	spline	N/A	splin
Greatest common demoninator	gcd	gcd	N/A	gcd
Least common multiple	lcm	lcm	N/A	lcm
Basic statistics functions:				
Mean	mean	mean	mean	mean
Median	median	median	N/A	median
Standard deviation	std	std	std	st_deviation
Variance	var	var	var	variance
Covariance	covar	cov	N/A	covar

3 Complex Operations Test

This section tests the software packages on a classical test problem from partial differential equations both in terms of performance and memory consumption [1, 3, 4, 6–9]. This paper differs from similar work such as [6–8]. In [6], only the conjugate gradient method is implemented, and it uses the source code language C; in fact, the focus of [6] is on studying the performance of the parallel communications library MPI. In [7, 8], the studies comparing Matlab, Octave, FreeMat, and Scilab were run on a personal computer; furthermore, the present paper extended these to include the conjugate gradient method in FreeMat.

The problem considered in this section can appear in several contexts of undergraduate classes, namely either as a linear algebra test problem or as an introductory example of numerical method for partial differential equations. In both cases, students are often required to write code that solves the overall problem.

3.1 The Test Problem

The Poisson problem with homogeneous Dirichlet boundary conditions is given as

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

on a given open region $\Omega \subset \mathbb{R}^2$. Here, $\partial\Omega$ denotes the boundary of the domain Ω , and the Laplace operator is defined as

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

We consider the problem on the domain Ω given by the open two-dimensional unit square $\Omega = (0, 1) \times (0, 1) \subset \mathbb{R}^2$. Since $u = 0$ on the boundary $\partial\Omega$ in (3.1), we have a homogeneous Dirichlet boundary condition. Thus, (3.1) can be restated as

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

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

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

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

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

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

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

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 has components $b_k = h^2 f_{i,j}$ where $k = i + (j - 1)N$. The matrix A is symmetric and positive definite [4, 9]. 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 [4, 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 $A = I \otimes T + T \otimes I$. This idea is used to set up the matrix A in our codes that performs Gaussian elimination to solve the linear system of equations. To store the matrix A efficiently, all packages provide for a sparse storage mode, in which only the non-zero entries are stored.

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 $\|u - u_h\| \leq C h^2$, as the mesh width h tends to zero $h \rightarrow 0$, where C is a constant independent of h [2, 5]. 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 $\|u - u_h\| = \sup_{(x,y) \in \Omega} |u(x, y) - u_h(x, y)|$ [2, 5].

3.2 Results

3.2.1 Gaussian Elimination

Figure 3.1 (a) shows the numerical solution of the system solved with a mesh resolution of 32×32 . Figure 3.1 (b) shows the error at each mesh point, which is computed by subtracting the numerical solution u_h from the analytical solution u . Notice that the maximum error at the center. When the mesh resolution is 32×32 , the maximum error in Figure 3.1 (b) that occurs is approximately 3.0×10^{-3} .

Now we want to solve the Poisson problem on finer meshes in order to obtain a more precise solution. The results of this are summarized in Table 3.1, which lists the mesh size of the discretization $N \times N$, the dimension of the linear system N^2 , the norm of the absolute finite difference error $\|u - u_h\|$, and the ratio of the absolute errors. Table 3.1 shows that the finite difference error converges toward zero and the ratio of the consecutively refined meshes is approaching four. Because the finite difference error tends to zero and the ratio behaves as predicted by (3.7), this shows that our code is working correctly.

Since the easiest way to solve the linear system $Au = b$ is by Gaussian elimination, we begin by solving the linear system arising from the Poisson problem by this method in each

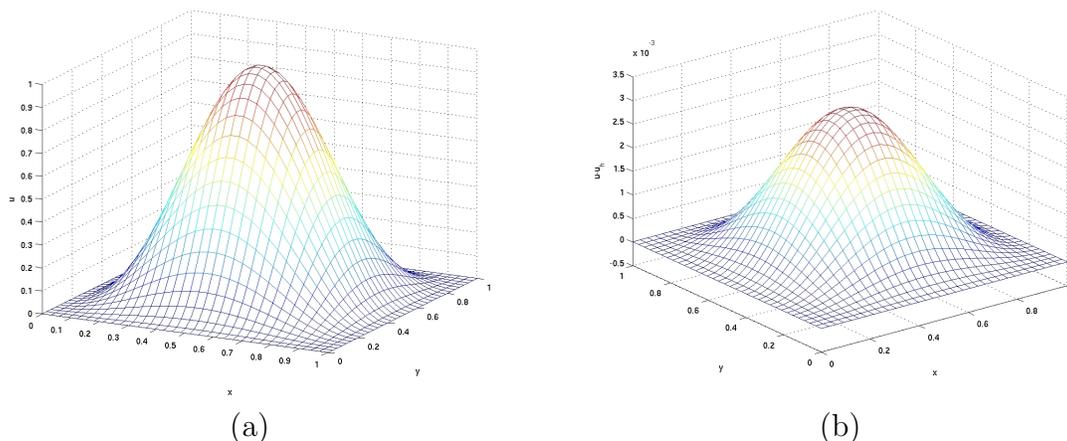


Figure 3.1: Mesh plots for $N = 32$ in Matlab (a) of the numerical solution and (b) of the numerical error.

of the different software packages. Here we use the sparse storage mode in all packages, in which only non-zero entries are stored. Table 3.2 lists the mesh resolution N , the dimension of the linear system, and the observed wall clock time is HH:MM:SS for the different software packages. By looking at Table 3.2, it can be concluded that the Gaussian method built into the backslash operator successfully solves the problem up to a mesh resolution of $4,096 \times 4,096$ in both Matlab and Octave, before running out of memory. The fundamental reason for running out of memory is that Gaussian elimination turns most of the zero entries in A to non-zero numbers, thus continuously increasing the memory requirements during the algorithm. The Gaussian elimination method built into the backslash operator in FreeMat successfully solves the problem up to a mesh resolution of $2,048 \times 2,048$. The wall clock results show that Matlab was faster than Octave, but both could solve the same mesh resolutions.

Table 3.1: Convergence results for the finite difference method. The table lists the mesh resolution N , the number of degrees of freedom N^2 , the finite difference norm $\|u - u_h\|$, and the ratio of consecutive errors.

$N \times N$	N^2	$\ u - u_h\ $	Ratio
32×32	1,024	3.0128e-3	N/A
64×64	4,096	7.7811e-4	3.8719
128×128	16,384	1.9765e-4	3.9368
256×128	65,536	4.9797e-5	3.9690
512×512	262,144	1.2494e-5	3.9856
$1,024 \times 1,024$	1,048,576	3.1266e-6	3.9961
$2,048 \times 2,048$	4,194,304	7.8019e-7	4.0075
$4,096 \times 4,096$	16,777,216	1.9353e-7	4.0313
$8,192 \times 8,192$	67,108,864	4.7400e-8	4.0829

Table 3.2: Performance comparison for Gaussian elimination. The table lists the mesh resolution N , the number of degrees of freedom N^2 , and the observed wall clock time in HH:MM:SS for Matlab, Octave, FreeMat, and Scilab. The abbreviation O.M. stands for “out of memory.”

$N \times N$	N^2	Matlab	Octave	FreeMat	Scilab
32×32	1,024	<00:00:01	<00:00:01	<00:00:01	<00:00:01
64×64	4,096	<00:00:01	<00:00:01	<00:00:01	<00:00:01
128×128	16,384	<00:00:01	<00:00:01	<00:00:01	00:00:11
256×256	65,536	<00:00:01	<00:00:01	00:00:04	00:03:19
512×512	262,144	00:00:01	00:00:02	00:00:28	00:39:04
$1,024 \times 1,024$	1,048,576	00:00:31	00:00:16	00:03:15	08:32:20
$2,048 \times 2,048$	4,194,304	00:00:27	00:01:57	00:14:29	O.M.
$4,096 \times 4,096$	16,777,216	00:02:07	00:15:37	O.M.	O.M.
$8,192 \times 8,192$	67,108,864	O.M.	O.M.	O.M.	O.M.

In turn, Octave was faster and was able to solve a higher mesh resolution than both FreeMat and Scilab. Scilab was the slowest and could not solve the same mesh resolution as the other packages. The Scilab code used to create the tables utilizes the command `stacksize("max")` to allow it to use all available memory when running the code; the other packages have this behavior as default without given an explicit command. In fact, without the use of `stacksize("max")`, Scilab can only solve the problem up to 256×256 and runs out of memory already for 512×512 [3].

3.2.2 Conjugate Gradient Method

Now, we use the conjugate gradient method to solve the Poisson problem. This iterative method is an alternative to using Gaussian elimination to solve a linear system. We use the zero vector as the initial guess and a tolerance of 10^{-6} on the relative residual of the iterates. To solve the problem for larger meshes, we use a so-called matrix-free implementation of the method to save a significant amount of memory. This implementation takes advantage of the fact that the conjugate gradient method does not need the system matrix A itself, but only its action on a vector in a matrix-vector product. Hence, we do not set up A at all, but provide a function that returns $v = Au$ for an input vector u . By saving the memory for A , we expect to solve the problem for larger meshes. Table 3.3 lists the mesh resolution $N \times N$, the degree of freedom N^2 , the number of iterations taken by the iteration method to converge (iter), and the observed wall clock times for the different software packages. All packages gives the same number of iterations and produced identical results to within round-off.

Table 3.3 shows that the conjugate gradient method is indeed able to solve for mesh resolutions as large, or larger, than those solved using Gaussian elimination. The matrix-free implementation of the conjugate gradient method allows us to solve a mesh resolution

Table 3.3: Performance comparison for the conjugate gradient method. The table lists the mesh resolution N , the number of degrees of freedom N^2 , the number of iterations (iter) indicates the number of iterations needed to solve the linear system of equations, and the observed wall clock times in HH:MM:SS for Matlab, Octave, FreeMat, and Scilab. The abbreviation E.T.R stands for “excessive time requirement.”

$N \times N$	N^2	iter	Matlab	Octave	FreeMat	Scilab
32×32	1,024	48	<00:00:01	<00:00:01	<00:00:01	<00:00:01
64×64	4,096	96	<00:00:01	<00:00:01	00:00:03	<00:00:01
128×128	16,384	128	<00:00:01	<00:00:01	00:00:23	<00:00:01
256×256	65,536	387	00:00:02	00:00:03	00:03:07	00:00:04
512×512	262,144	783	00:00:40	00:00:27	00:15:21	00:00:36
$1,024 \times 1,024$	1,048,576	1,581	00:05:47	00:04:23	03:24:25	00:06:57
$2,048 \times 2,048$	4,194,304	3,192	00:50:26	00:40:07	03:52:24	00:48:12
$4,096 \times 4,096$	16,777,216	6,452	07:46:24	05:52:11	E.T.R	06:03:58
$8,192 \times 8,192$	67,108,864	13,033	69:42:45	40:44:32	E.T.R	O.M.

up to $8,192 \times 8,192$ for Matlab and Octave. With the matrix-free implementation, Scilab is able to solve the system for a resolution up to $4,096 \times 4,096$. In FreeMat, we wrote our own `cg` function because it does not have a built-in function and we were able to solve the system for a resolution of $2,048 \times 2,048$, before the time became excessive. The wall clock times show that Octave was slightly faster than Matlab. The results also show that FreeMat was slower than Octave, Matlab, and Scilab, and was not able to solve as large of a system before the time required to solve the problem became excessively long. Scilab performed better than FreeMat and was able to solve the system comparably as fast as Octave and Matlab. However, the conjugate gradient method in Scilab ran out of memory for $8,192 \times 8,192$, despite the use of `stacksize("max")`.

4 Conclusions

We tested the four software packages Matlab, Octave, FreeMat, and Scilab for two criteria: usability and performance. In Section 2, we analyzed the software packages’ usability by comparing the syntax and functions to Matlab. The more similar a software package was to Matlab, the more usable it was. Octave was determined to be the most usable, because it uses the same commands and syntax as Matlab for all of our tests. Scilab exhibited the most differences in both syntax and commands. For example, instead of using the `eig` function like Matlab, Octave, and FreeMat to compute eigenvalues, Scilab uses a function called `spec`.

In Section 3, we tested the performance of the software packages by using both Gaussian elimination and the conjugate gradient method to solve the linear system of equations resulting from the finite difference discretization of the Poisson equation in two spatial dimensions.

- The results from Table 3.2 reveal that Matlab performed the best when solving the

system via Gaussian elimination. Octave was significantly slower than Matlab for large problems, but performed the best of all the free software packages tested and was able to solve the same size systems as Matlab, albeit somewhat slower. Scilab's performance was the weakest and was much slower, compared to Matlab and Octave.

- The results from Table 3.3 reveal that Matlab, Octave, and Scilab were all able to solve the system comparably fast, but Scilab was not able to solve as large a system as Octave or Matlab. FreeMat was the weakest and could not solve the system for mesh resolutions larger than $2,048 \times 2,048$ without requiring an excessive amount of time.

Comparing Tables 3.2 and 3.3, we find that for large sparse linear systems the conjugate gradient method can solve larger problems than Gaussian elimination, but Gaussian elimination is generally faster when it does work, potentially by orders of magnitude. Among the three free numerical computation packages, Octave is the most powerful one, even outperforming Matlab sometimes.

After considering usability and performance, it can be concluded that Octave is the most powerful of the free numerical computation packages, and the easiest to use.

Acknowledgments

The author acknowledges support as undergraduate assistant from the REU Site: Interdisciplinary Program in High Performance Computing (www.umbc.edu/hpcreu) at the University of Maryland, Baltimore County (UMBC) in Summer 2011. Furthermore, the author would like to thank Dr. Matthias K. Gobbert (gobbert@umbc.edu, director of the REU Site) for providing the research opportunity, guidance, and support. The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant no. CNS-0821258) and the SCREMS program (grant no. DMS-0821311), with additional substantial support from UMBC. See www.umbc.edu/hpcf for more information on HPCF and the projects using its resources.

References

- [1] K. P. ALLEN, *Efficient parallel computing for solving linear systems of equations*, UMBC Review: Journal of Undergraduate Research and Creative Works, 5 (2004), pp. 8–17.
- [2] D. BRAESS, *Finite Elements*, Cambridge University Press, third ed., 2007.
- [3] M. BREWSTER AND M. K. GOBBERT, *A comparative evaluation of Matlab, Octave, FreeMat, and Scilab on tara*, Tech. Rep. HPCF-2011-10, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2011.
- [4] J. W. DEMMEL, *Applied Numerical Linear Algebra*, SIAM, 1997.

- [5] A. ISERLES, *A First Course in the Numerical Analysis of Differential Equations*, Cambridge Texts in Applied Mathematics, Cambridge University Press, second ed., 2009.
- [6] A. M. RAIM AND M. K. GOBBERT, *Parallel performance studies for an elliptic test problem on the cluster tara*, Tech. Rep. HPCF–2010–2, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010.
- [7] N. SHARMA, *A comparative study of several numerical computational packages*. M.S. thesis, Department of Mathematics and Statistics, University of Maryland, Baltimore County, 2010.
- [8] N. SHARMA AND M. K. GOBBERT, *A comparative evaluation of Matlab, Octave, FreeMat, and Scilab for research and teaching*, Tech. Rep. HPCF–2010–7, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010.
- [9] D. S. WATKINS, *Fundamentals of Matrix Computations*, Wiley, third ed., 2010.