IDL: A Possible Alternative to Matlab?

Ecaterina Coman

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

Senior Thesis, Spring 2012

Thesis mentor Dr. Matthias K.Gobbert

Abstract

Created by Exelis Visual Information Solutions, IDL (Interactive Data Language) is a commercial package used for data analysis. We compared the usability and efficiency of IDL to that of Matlab to determine if IDL is a viable substitute. Two studies were performed for this analysis. The first, a basic test inspired by the CIRC Tutorial for Basic Matlab, consisted of solving a system of linear equations using basic operations, computing eigenvalues and eigenvectors, and creating two-dimensional plots. It showed identical results between the two packages, though it is important to note that the syntax and display of output between IDL and Matlab differs greatly. The second test focused on direct and iterative solutions of a large sparse system of linear equations. This system arises from the finite difference discretization of the Poisson equation with homogeneous Dirichlet boundary conditions and is prototypical for linear systems in many related contexts. In Matlab, Gaussian elimination was used as the direct method of solving the Poisson test problem. Unfortunately, such a method was not available with our license of IDL to solve sparse systems as it required a more expensive IDL Analyst License. Originally, we aimed to solve the problem iteratively using the conjugate gradient method, but, though a function was available in Matlab for solving a sparse system this way, none existed in IDL. Instead, we turned to the biconjugate gradient method. The numerical results of this method in IDL are identical to those in Matlab, but IDL runs the code slightly faster for finer meshes. Those looking to make a switch from Matlab to IDL might have a difficult time encountering a different syntax, output display, and the need for a more expensive license to run a larger breadth of functions or procedures, but if efficiency is of concern, IDL can potentially be faster than Matlab.

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 an additional numerical computational package, IDL, and provide information on whether or not this package is compatible to Matlab and can be used as an alternative. Section 1.3 provides a more detailed description of these two packages. To evaluate IDL, a comparative approach is used based on a Matlab user's perspective. To achieve this task, we perform some basic and some complex studies on Matlab and IDL.

The basic studies include basic operations solving systems of linear equations, computing the eigenvalues and eigenvectors of a matrix, and two-dimensional plotting. The complex studies include direct and iterative solutions of a large sparse system of linear equations resulting from finite difference discretization of an elliptic test problem.

In Section 2, we perform the basic operations test using Matlab and IDL. This includes testing the backslash operator, computing eigenvalues and eigenvectors, and plotting in two dimensions. The backslash operator works easily in Matlab to produce a solution to the linear system given. Instead of the backslash operator, the `LUDC` and `LUSOL` commands are used in IDL to compute the LU decomposition and use its result to solve the system. The command `eig` computes eigenvalues and eigenvectors in Matlab, whereas IDL requires a three-step process. IDL uses `ELMHES` to reduce the matrix to upper Hessenberg format, and then `HQR` to compute the eigenvalues. Given the eigenvalues, `EIGENVEC` then gives the eigenvectors. Plotting is another important feature we analyze by an m-file containing the two-dimensional plot function along with some common annotation commands. Like Matlab, IDL uses a basic `plot` command to create the 2D plot, but one has to use quite different, but equivalent commands for annotation.

Section 3 introduces a common test problem, given by the Poisson equation with homogeneous Dirichlet boundary conditions, and discusses the finite discretization for the problem in two dimensional space. In the process of finding the solution, we use a direct method, Gaussian elimination, and an iterative method, the biconjugate gradient method. To be able to analyze the performance of these methods, we solve the problem on progressively finer meshes. The Gaussian elimination method built into the backslash operator successfully solves the problem up to a mesh resolution of $4{,}096 \times 4{,}096$ in Matlab, but in IDL there does not exist a Gaussian elimination method for sparse matrices without an expensive IDL Analyst license. For this reason, the IDL Gaussian elimination test was not conducted. We had originally planned a comparison using the conjugate gradient iterative method in both Matlab and IDL, but such a method did not exist in IDL for solving sparse systems. For this reason, the biconjugate gradient method was used in both packages for comparison. The biconjugate gradient method is available in the `bicg` function in Matlab and in IDL through the command `LINBCG`. The sparse matrix implementation of the iterative method allows us to solve a mesh resolution up to $8{,}192 \times 8{,}192$ in Matlab and IDL. The existence of the `mesh` in Matlab allows us to include the three-dimensional plots of the numerical solution and the numerical error. In IDL, the `SURFACE` plots match those of Matlab. IDL showed to be quite different from Matlab when considering syntax. Many of the commands were either entirely unavailable for our license (`kron`) or had to be applied differently (no backslash operator resulted in the use of `LUDC` and `LUSOL`, and the lack of an `eig` function led to the use of `ELMHES`, `HQR`, and `EIGENVEC`). The tests focused on usability lead us to conclude that the package is not too compatible with Matlab, since it does not use the same syntax and some functions are not available. Is it important to note that, even though syntax is different, IDL is able to solve the same problems as Matlab.

Fundamentally, Matlab and IDL turn out to be able to solve complex problems of the same size, as measured by the mesh resolution of the problem. Matlab and IDL also took about the same time to solve the problem, though IDL was slightly faster. The data that

these conclusions are based on are summarized in Tables 3.1 and 3.2.

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

1.2 Additional Remarks: 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.

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

It becomes clear that all software packages considered have at least one solver. Matlab and IDL have state-of-the-art variable-order, variable-timestep methods for both non-stiff and stiff ODEs available, with Matlab's implementation being the richest and its stiff solvers being possibly more efficient.

1.3 Description of the Packages

1.3.1 Matlab

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

1.3.2 IDL

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

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

1.4 Description of the Computing Environment

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

2 Basic Operations Test

This section examines a collection of examples inspired by some basic mathematics courses. This set of examples was originally developed for Matlab by the Center for Interdisciplinary Research and Consulting (CIRC). More information about CIRC can be found at `www.umbc.edu/circ`. This section focuses on the testing of basic operations using Matlab and IDL. 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. To run Matlab on the cluster tara, enter `matlab` at the Linux command line. This starts up Matlab with its complete Java desktop interface. Useful options to Matlab on tara include `-nodesktop`, which starts only the command window within the shell, and `-nodisplay`, which disables all graphics output. For complete information on options, use `matlab -h`.

2.1.1 Solving Systems of Equations

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

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

whose solution is $(1, -1, 2)^T$. 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 $x = A \backslash b$. Hence, Matlab use the backslash operator to solve this system. First, the matrix A and vector b are entered using the following:

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

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 + 0.7071i
```

and

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

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

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

Calculating the eigenvector enables us to express the matrix A as

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

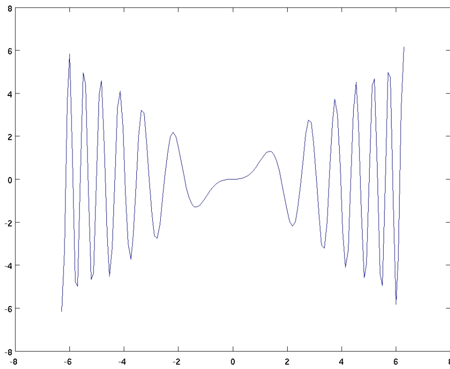
where P is the matrix of eigenvectors and D is a diagonal matrix as stated above. To check our solution, we will multiply the matrices generated using `eig(A)` to reproduce A as suggested in (2.2).

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

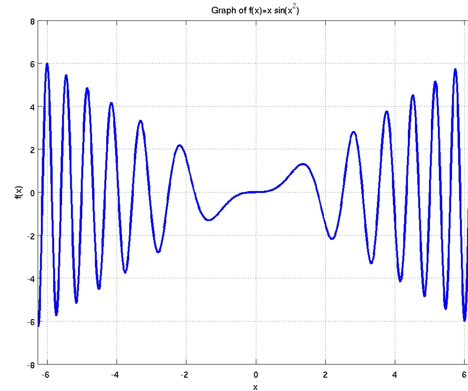
produces

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

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



(a)



(b)

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

2.1.3 2-D Plotting

2-D plotting is a very important feature as it appears in all mathematical courses. Since this is a very commonly used feature , let us examine the 2-D plotting feature of Matlab by plotting $f(x) = x\sin(x^2)$ over the interval $[-2\pi, 2\pi]$. The data set for this function is given in the data file `matlabdata.dat` and is posted along with the tech. report [3] at `www.umbc.edu/hpcf` under Publications. Noticing that the data is given in two columns, we will first store the data in a matrix A. Second, we will create two vectors, x and y, by extracting the data from the columns of A. Lastly, we will plot the data.

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

The commands stated above result in the Figure 2.1 (a). Looking at this figure, it can be noted that our axes are not labeled; there are no grid lines; and the peaks of the curves are rather coarse. 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 `axis off`. 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 annotation techniques mentioned earlier to annotate the plot. In addition to the other annotations, use `xlim([-2*pi 2*pi])` to set limit is for the x-axis. We can change the line width to 2 by `plot(x,y,'LineWidth',2)`. Finally, Figure 2.1 (b) is the resulting figure with higher resolution as well as the annotations. Observe that by controlling the resolution in Figure 2.1 (b), we have created a smoother plot of the function $f(x)$. The Matlab code used to create the annotated figure is as follows:

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

2.1.4 Programming

Here we will look at a basic example of Matlab programming using a script file. Let's try to plot Figure 2.1 (b) using a script file called `plotxsinx.m`. The extension `.m` indicates to Matlab that this is an executable m-file. Instead of typing multiple commands in Matlab, we will collect these commands into a script. Now, call the script from the command-line to execute it and create the plot with the annotations for $f(x) = x\sin(x^2)$. The plot obtained in this case is Figure 2.1 (b). This plot can be printed to a graphics file using the command

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

2.2 Basic operations in IDL

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

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

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

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

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

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

```
1.00000 -1.00000 2.00000
```

which is a row representation of the solution vector

$$x = \begin{bmatrix} 1.00000 \\ -1.00000 \\ 2.00000 \end{bmatrix}$$

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

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

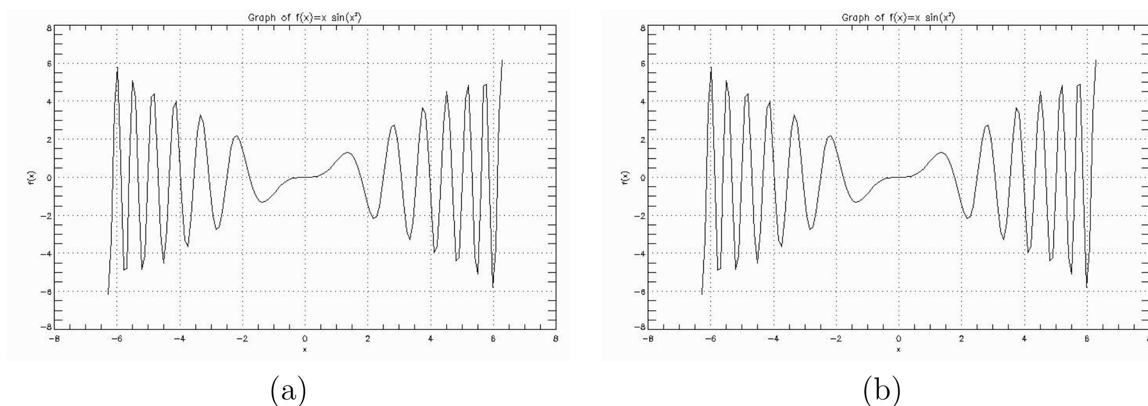
```
A=[[1.0,-1.0],[1.0,1.0]]
hesA = ELMHES(A)
eval = HQR(hesA)
evec = EIGENVEC(A,eval, /DOUBLE, RESIDUAL=residual)
```

The eigenvalues calculated by `HQR` are

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

and the eigenvectors are

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



(a) (b)

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

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

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

IDL outputs each eigenvector as a row vector, which is very different in appearance from the output produced by the other packages whose output is based on the matrix P, whose columns are the eigenvectors. We notice that the eigenvalues were output in reverse order than by Matlab. Taking this into account and also noting that multiplying by $-i$ shows $[0.7071i, 0.7071]^T = [0.7071, -0.7071i]^T$, we see that the eigenpairs are the same as those computed by Matlab.

We are able to express the matrix A as PDP^{-1}, where P is the matrix of eigenvectors and D is the diagonal matrix of eigenvalues, to reproduce the matrix A. Remembering that IDL outputs eigenvectors as row vectors, we turn the output matrix of eigenvectors into the matrix P by taking the transpose with the command `P=transpose(evec)` then use `INVERT(P)` to obtain the inverse of P. To create D, we create a matrix with the eigenvalues on the diagonal. This can easily be created with the command `D=[[eval[0],0],[0,eval[1]]]`, which takes the complex values in the vector of eigenvalues found using HQR and puts them on the diagonal. In IDL, the `##` operator is used for matrix multiplication, and computing PDP^{-1} with the command `P##diag##invert(P)` gives a result of

```
(1.00000000,-0.00000000)(-1.00000000,0.00000000)
(1.00000000,0.00000000)(1.00000000,0.00000000)
```

which is A to at least 8 significant digits.

Next, we will examine the similarities between the plotting commands using $f(x) = x\sin(x^2)$. To obtain data from a file, the following commands will get the number of lines in

the file and create a two-column double-type array to hold the data, open the file and create a logical unit number which will be associated with that opened file, read the data into the double array, and close the file and free the logical unit number:

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

Using `x=data(0,*)` and `y=data(1,*)` will create vectors x and y The plot procedure, `PLOT, x, y` will then create Figure 2.2 (a) using vectors x and y. These commands have an obvious agreement to the `LOAD` and `PLOT` commands in matlab as Figure 2.2 (a) is the same as Figure 2.1 (a). To annotate this graph, use the `TITLE` command to add a graph title, the `XTITLE` and `YTITLE` commands to add a horizontal and vertical axis title, respectively. To create a grid, we use `XTICKLEN` and `YTICKLEN` and edit the line style using `XGRIDSTYLE` and `YGRIDSTYLE`. Now, we want to have a smoother graph made of more equally spaced data points. We create a second vector x using `x=(4*!pi)*FINDGEN(1025)/1024 - (2*!pi)`, where `FINDGEN` creates an array of 1025 values from 0 to 1024, to get a range of numbers from -2π to 2π at intervals of pi. Using this new x, make the vector y with the command `y=x*sin(x^2)`. This annotated data can be plotted using

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

The command `XSTYLE=1` makes sure IDL is displaying the exact interval, as it normally wants to produce even tick marks which create a wider range than desired. Also, the command `THICK` changes the thickness of the grid lines and `$` is the line continuation character. This annotated plot procedure generates Figure 2.2 (b). These commands can all be placed into a procedure file, which can run them all at once from the IDL command line. To save this plot to a `jpg` file, save the contents of the current Direct Graphics window into a variable and write that variable using

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

3 Complex Operations Test

3.1 The Test Problem

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, 4, 5, 9], given as

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

This problem was studied before in, among other sources, [1, 3, 4, 6–9]. 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 [9]. Since $u = 0$ at the boundary in (3.1), we are looking at a homogeneous Dirichlet boundary condition. We consider the problem on the two-dimensional unit square $\Omega = (0,1) \times (0,1) \subset \mathbb{R}^2$. Thus, (3.1) can be restated as

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

where the function f is given by

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

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

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

3.2 Finite Difference Discretization

Let us define a grid of mesh points $\Omega_h = (x_i, y_j)$ 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 $u_{i,j} \approx u(x_i, y_j)$:

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

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 [4, 9]. This implies that the linear system has a unique solution and it guarantees that the iterative Krylov subspace methods converge, of which the conjugate gradient method as well as the biconjugate gradient method are examples.

To create the matrix A in Matlab effectively, 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

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

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 $A = I \otimes T + T \otimes I$.

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 as $\|u - u_h\|_{L^\infty(\Omega)} \leq C\,h^2$, as $h \to 0$, 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

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

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 the linear system arising from the Poisson problem by Gaussian elimination in Matlab. We know that this is easiest approach for solving linear systems for the user of Matlab, although it may not necessarily be the best method for large systems. To create matrix A, we make use of the Kronecker tensor product, as described in Section 3.2. This can be easily implemented in Matlab using the `kron` function. The matrix A is stored in sparse storage mode, which does not store the zeros in the matrix. The system is then solved using the backslash operator. Figure 3.1 shows the results of this for a mesh with $N = 32$.

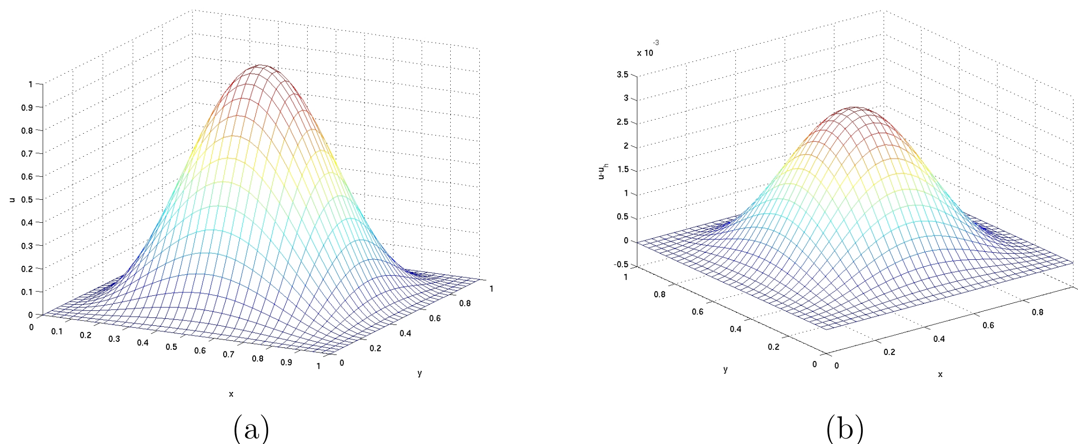


Figure 3.1: Mesh plots for $N = 32$ in Matlab (a) of the numerical solution and (b) of the numerical error.

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

(a) Gaussian Elimination

N	DOF	$\|u - u_h\|$	Ratio	Time
32	1,024	3.0128e-3	N/A	<00:00:01
64	4,096	7.7812e-4	3.8719	<00:00:01
128	16,384	1.9766e-4	3.9366	<00:00:01
256	65,536	4.9807e-5	3.9685	<00:00:01
512	262,144	1.2501e-5	3.9843	00:00:01
1,024	1,048,576	3.1313e-6	3.9923	00:00:31
2,048	4,194,304	7.8362e-7	3.9959	00:00:27
4,096	16,777,216	1.9610e-7	3.9960	00:02:07
8,192		out of memory		

(b) Biconjugate Gradient Method

N	DOF	$\|u - u_h\|$	Ratio	`#iter`	Time
32	1,024	3.0128e-3	N/A	48	<00:00:01
64	4,096	7.7811e-4	3.8719	96	<00:00:01
128	16,384	1.9765e-4	3.9368	192	<00:00:01
256	65,536	4.9797e-5	3.9690	387	00:00:02
512	262,144	1.2494e-5	3.9856	783	00:00:18
1,024	1,048,576	3.1266e-6	3.9961	1,581	00:02:07
2,048	4,194,304	7.8019e-7	4.0075	3,192	00:20:38
4,096	16,777,216	1.9366e-7	4.0286	6,452	02:34:19
8,192	67,777,216	4.7400e-8	4.0857	13,033	21:31:20

Figure 3.1 (a) shows the mesh plot of the numerical solution vs. (x, y). The error at each mesh point is computed by subtracting the numerical solution from the analytical solution and is plotted in Figure 3.1 (b). Notice that the maximum error occurs at the center.

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

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 $N = 8{,}192$. Hence,

we are unable to solve this problem for N larger than 4,096 via Gaussian elimination. This leads to the need for another method to solve larger systems. Thus, we will use an iterative method to solve this linear system.

3.3.2 Biconjugate Gradient Method

Now, we use the biconjugate gradient method to solve the Poisson problem [1, 9]. This iterative method is an alternative to using Gaussian elimination to solve a linear system and is accomplished by replacing the backslash operator by a call to the `bicg` function. We use the zero vector as the initial guess and a tolerance of 10^{-6} on the relative residual of the iterates.

Table 3.1 (b) shows results of a study using the biconjugate gradient method with the system matrix A in sparse storage mode. The column `#iter` lists the number of iterations taken by the iteration method to converge. The finite difference error shows the same behavior as in Table 3.1 (a) with ratios of consecutive errors approaching 4 as for Gaussian elimination; this confirms that the tolerance on the relative residual of the iterates is tight enough.

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

3.4 IDL Results

3.4.1 Gaussian Elimination

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

3.4.2 Biconjugate Gradient Method

No function or routine exists in the basic IDL license to compute the conjugate gradient method to solve the Poisson problem. With an extra IDL Analyst license, the routine `IMSL_SP_CG` is available to solve a real symmetric definite linear system using a conjugate gradient method. As this routine is not available with our current license, we will use the biconjugate gradient method available instead. The biconjugate gradient method is an iterative method which can replace the use of LU decomposition in IDL with the `LINBCG` function. The zero vector is used as our initial guess. We also set the keyword `ITOL=1`, specifying to stop the iteration when $|Ax - b|/|b|$ is less than the tolerance. This convergence test with the tolerance 10^{-6} matches those used in the other languages.

To create the matrix A, we use three arrays holding the column and row locations, and the specific values at those locations, in the `SPRSIN` function to create a sparse matrix A.

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

Biconjugate Gradient Method					
N	DOF	$\|u - u_h\|$	Ratio	`#iter`	Time
32	1,024	0.0030128	N/A	48	<00:00:01
64	4,096	7.7811e-4	3.8719	96	<00:00:01
128	16,384	1.9764e-4	3.9368	192	<00:00:01
256	65,536	4.9798e-5	3.9690	387	00:00:01
512	262,144	1.2494e-5	3.9856	783	00:00:12
1,024	1,048,576	3.1267e-6	3.9961	1,581	00:01:55
2,048	4,194,304	7.8029e-7	4.0070	3,192	00:16:28
4,096	16,777,216	1.9395e-7	4.0232	6,452	02:22:06
8,192	67,777,216	4.9022e-8	3.9563	13,044	18:21:34

In addition, we used `CMREPLICATE`, created by Craig B. Markwardt, to replicate an array and create a full grid similar to the command `ndgrid` in Matlab. The `SURFACE` routine was used to create Figure 3.2 (a), the surface plot of the numerical solution vs. (x, y) and Figure 3.2 (b) the surface plot of the error vs. (x, y).

To create Table 3.2, we used a version of the IDL procedure file with the plotting and

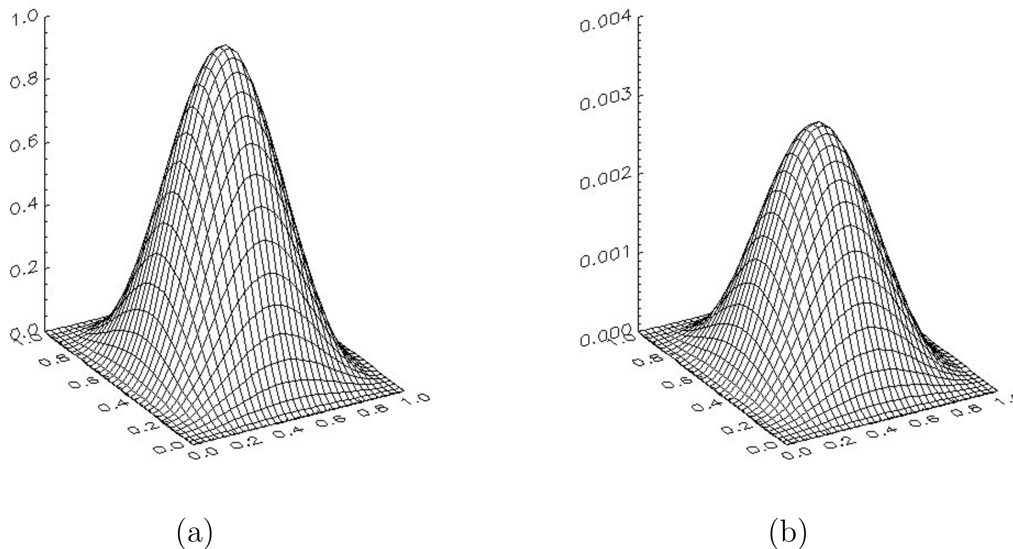


Figure 3.2: Surface plots for $N = 32$ in IDL (a) of the numerical solution and (b) of the numerical error.

graphics commands commented out. This IDL biconjugate gradient method solves the problem for all mesh sizes without running out of memory and is slightly faster than the Matlab results in Table 3.1 (b).

Acknowledgments

This work was made possible with the help of Andrew Raim at the University of Maryland, Baltimore County (UMBC). The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant no. CNS–0821258) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See `www.umbc.edu/hpcf` for more information on HPCF and the projects using its resources.

References

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

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

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

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

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

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

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

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

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