Mathematics 372 – Numerical Linear Algebra
Lab/Problem Set 1 – *due:* January 26, 2007

General Information on MATLAB

The major purpose of this assignment is to get you started working with the software package MATLAB. MATLAB is the "industry standard" for computations in numerical linear algebra and also in many of the applied areas where numerical linear algebra is used. It has some features in common with other mathematical systems such as Maple. For instance, you can use it interactively, enter commands, generate output, use the results for further computations, etc. However, there are also some differences.

First, MATLAB is *not* set up systematically the way Maple is to produce integrated mathematical documents like the *Maple worksheets* you know and love. This means that to produce annotated output explaining what you did in a computation, you will need to save the output from the command window in a file, then add comments and explanations "by hand" in a text editor. In addition, I think you will soon find that it is most convenient to generate the input commands with an editor in another text file and read them into MATLAB rather than entering them interactively. You will also need to save and print any plots you produce separately from text, input and computational results.

Second, as we will see, *essentially everything in MATLAB is a matrix* of some sort. So there are not many facilities for symbolic computation on polynomials or expressions. Generating plots in MATLAB may also take some getting used to.

Third, I think you will find that the syntax of MATLAB commands is quite a bit simpler than Maple syntax. But we will also be doing quite a bit of *programming* – writing our own MATLAB programs, or "scripts" too.

Running MATLAB in Swords 219

To run MATLAB in the Math/CS SunRay lab in Swords 219:

- Log on the system (your username and password are now the same as on the main campus Novell network – if you are doing this for the first time, there will be a short delay as the system initializes your account).
- The operating system here is *Linux*, a version of Unix, but the desktop and windows in the graphical user interface should seem familiar and similar to what you find on Windows machines. For instance, to run MATLAB, you can left-click on the "Red Hat" icon on the lower left (like the "Start Button" in Windows) then select HC Applications/MATLAB to launch the MATLAB window.
- At this point, you should take a moment to look over the default MATLAB window – notice the tool bar and pull-down menus across the top. Most of these look just like those in any Windows application. The Command Window at the right of the screen is where you will type in interactive commands and see output. The left side of the screen shows the contents of your current folder (directory) when you launched MATLAB – most likely your top-level folder. You can navigate to other folders to read in MATLAB input files using this. The lower left of the screen is a history of all

the commands you have entered. This can be very useful as you learn the syntax of various MATLAB commands – you can look back and see what you did previously. Note: This history actually includes *previous sessions* too. When the history gets too full or you just want to clean things out, you can empty it using EDIT/Clear Command History from the main pulldown menus.

MATLAB, like Maple, has extensive online Help – documentation for all the built-in commands, with examples and explanations. You should find this very useful!

A First MATLAB Session

Since MATLAB's bread and butter is performing linear algebra calculations, let's see how to solve a system of linear equations first. Let's also *save* a record of everything we do in an output file (say called first.mat), as we go along. This is done with a command like this:

```
diary 'first.mat'
```

Say we want to solve

$$\begin{pmatrix} 11.2 & 3.4 & -3.8 & 0 \\ 2.9 & 0.7 & 1.3 & 4.6 \\ 0 & 2.1 & 4.3 & 5.1 \\ -9.2 & 8.4 & 2.9 & 3.2 \end{pmatrix} x = \begin{pmatrix} 1 \\ 0 \\ -1 \\ -2 \end{pmatrix}$$

Let's call the coefficient matrix of the system A and the vector on the right-hand side b. The matrix A is entered in MATLAB as follows:

```
A = [11.2 3.4 -3.8 0; 2.9 0.7 1.3 4.6; 0 2.1 4.3 5.1; -9.2 8.4 2.9 3.2]
```

Note: no colon before the equals sign for the assignment to the name A, no commas between the numbers on the rows of the matrix, and the rows are separated by semicolons. Also, note that there is no semicolon at the end of the command as there would normally be in Maple. When you hit RETURN on this input line, you will see the matrix printed out (without parentheses). If you do put a semicolon at the end of an input line, MATLAB will *suppress the output.* Now enter the column vector b following the pattern from the input for A above (think of b as a 4×1 matrix – remember *everything in MATLAB is a matrix*).

To *solve the system of linear equations* $Ax = b$ for x, enter:

```
x = A \ b
```

(that's a "backslash" character). To check that the result is correct, we can also multiply Ax (matrix product) and check that the result is the same as the b we started with: Enter

```
A*x
```

A word about output formatting here: Note that all numbers so far have been printed out with exactly four decimal places. This is the default, "short" numerical format that

MATLAB uses for printing (*be sure you understand this – it's not the same as the number of digits used in numerical computation*). If you want to see all the digits that MATLAB has computed in a decimal number, enter

```
format long
```

Try this, then type in the name `x` to see the full computed value. (Note: MATLAB *rounds* to four decimal places in the short format.)

Next, recall from linear algebra that when the coefficient matrix of a linear system is *invertible* (that is, if the inverse matrix A^{-1} exists), then there's another way to solve the linear system. That's true for the matrix A here. To find the inverse matrix, and assign it to the name `AI`, enter

```
AI = inv(A)
```

How would you use the inverse matrix to solve the system $Ax = b$? Compute the solution this way and check with the x you found before.

We can also consider linear systems with fewer equations than variables, or more equations than variables. This leads to non-square coefficient matrices. For instance, consider the "underdetermined" system

$$\begin{pmatrix} 1 & 1 & -2 & 3 \\ 2 & -1 & 4 & 5 \end{pmatrix} x = \begin{pmatrix} 1 \\ -3 \end{pmatrix} \tag{1}$$

Enter the matrix as above, and call it `A2`. Then enter the right-hand side vector and call it `b2`.

The same sort of command as above: `A2 \ b2` will produce one *particular solution* of the system. Try it. But recall that in a case like this, there are actually *infinitely many different solutions*, and they can be written as linear combinations

$$x_p + c_1 x_1 + \cdots + c_s x_s$$

where x_p is any one particular solution, the c_i are arbitrary constants, and $\{x_1, \ldots, x_s\}$ is a basis for the kernel, or nullspace, of the matrix A. Such a basis can be computed with the command

```
null(A)
```

Try it and check your work (how?)

Saving and Printing Work

At this point, say we want to save or print out our work so far. There are several options:

- To just print out the contents of the Command Window, you can use File/Print from the pull-down menus on the MATLAB window. This isn't always the most useful form of output, though, since you might have made typing mistakes that generated error

messages, you might have had a false start in a computation that you don't really want to see, etc.

- If you have been saving the work in a file via the `diary` command as above, then you can *turn off* the saving with the command `diary off`. You can then select "Red Hat"/Accessories/Text Editor, and open the file where you were saving the diary. Then you can remove any input or output you don't want, annotate what's there with comments, etc. The file can then be printed and saved from the editor. (You will want to use this method to prepare work you hand in on this and future assignments!)
- You may have also noticed the File/Save Workspace option from the pull-down menus. This may be different from what you expect, though. What this option does is to save the current state of your computation (the most recent assignments to all names) for further use in a later session. It *does not save the input commands.* This can be useful too if you need to come back later and complete a calculation, but for most of the things we do, if you need to reconstruct the results you had, it's probably just as easy just to save the input commands in a text file and read them in as described in problem II below.

Assignment

I. In class on January 17, we discussed how discretization could be used to produce systems of linear equations whose solutions should approximate values of the solution of a 2nd order ODE boundary value problem of the form

$$\begin{cases} ay'' + by' + cy = f(x) \\ y(0) = y(1) = 0. \end{cases}$$

Recall that the $(N-1) \times (N-1)$ systems of linear equations we obtained had the general form

$$(N^2a - Nb/2)u_{i-1} + (-2N^2a + c)u_i + (N^2a + Nb/2)u_{i+1} = f_i$$

for $i = 1, \ldots, N-1$. Here $u_i \doteq y(x_i)$ and $f_i = f(x_i)$ from the points x_i in the partition of the interval $[0, 1]$ used to discretize. We use the boundary conditions $u_0 = u_N = 0$ in the first $(i = 1)$ and the last $(i = N-1)$ equation.

A) Let $a = -1, b = 0, c = 1$, and $f(x) = 6x$. Solve the systems of equations for the u_i using MATLAB in the cases $N = 6, 8, 20$, calling the solution vector u. (*Hint*: The first two matrices are not too hard to enter directly as above, but the $N = 20$ matrix will be rather tedious! Look at the online help for the `toeplitz` command to see a shortcut.)

B) The problem from part A:

$$\begin{cases} -y'' + y = 6x \\ y(0) = y(1) = 0 \end{cases}$$

is one where we can get an exact solution using some calculus and compare with the numerical solutions computed above. As in the discussion of the underdetermined system from (1) above, because this is a linear differential equation, the general solution of $-y'' + y = 6x$ is of the form

$$y = y_p + c_1 y_1 + c_2 y_2 \tag{2}$$

where y_p is any one particular solution of the equation, and y_1, y_2 are a basis for the solutions of the corresponding homogeneous equation $-y'' + y = 0$. Explain why we can take $y_p = 6x$ here, and show that $y_1 = e^x$ and $y_2 = e^{-x}$ both solve the homogeneous equation. Then, find constants c_1 and c_2 so that y from (2) satisfies the boundary conditions $y(0) = y(1) = 0$. This amounts to solving a system of linear equations – do the computations using MATLAB.

C) How good are the approximate solutions obtained by discretization here? One way to compare the approximate solutions u from part A with the exact solution from part B is via plotting. In this part, you will get a "crash course" in simple MATLAB plotting. The thing to remember here is the *everything in MATLAB is a matrix.* We think of the components of the u vector from part A as the approximate values of the solution y at the x_i. We can generate a "connect-the-dots" plot of the approximate solution in MATLAB as follows. Here are the commands for $N = 6$. You will need to figure out what to change to handle the other two cases $N = 8, 20$. First we generate a matrix with the list of x_i values. A shortcut method:

```
x = 1/6 :  1/6 :  5/6
```

To interpret this correctly, the left $1/6$ is the starting point (x_1), the middle $1/6$ is the "step" ($h = 1/N = 1/6$), and the right $5/6$ is the ending point (x_5). We will not plot the endpoints $0, 1$ since we know $y = 0$ there from the boundary conditions). Then we can generate the "connect-the-dots" plot with

```
plot(x,u)
```

The plot will be displayed in a separate window, not in the Command Window. Now, the method for plotting the exact solution from part B is essentially the same, but since we have the formula for y, we can generate as many points on the graph as we like and connect the dots again to generate a smooth-looking curve(!). Use

```
x2 = 1/6 :  1/60 :  5/6
y = 6*x2 + c1*exp(x2) + c2*exp(-x2)
```

(with the c_1, c_2 values you computed in part B). Then

```
plot(x,u,x2,y)
```

plots the approximate solution and exact solution together. What do you conclude about the accuracy of this discretization method?

II. In this problem, you will enter three short MATLAB programs or "scripts" in a text file, read them in, run them, then interpret the results.

A) Using the text editor, create a file containing the MATLAB commands at the top of the next page:

```
a = 1;
u = 1;
b = a + u;
while b ~= a
u = .5*u;
b = a + u;
end
u
```

Call the file `prog1.m`. To read in and execute these commands, in the MATLAB Command Window, just enter the command `prog1` (or right click on the icon for the file in the Current Directory window and select Run). What is this doing? (Hint: Look up "while" in the online help if you don't recognize this.) What does this output tell you about the floating-point number system in MATLAB?

B) Now create a second file `prog2.m` containing the commands

```
a = 1;
while a ~= Inf
a = 10*a
end
```

Run this one as in part A. What is this doing? What does this output tell you about the floating-point number system in MATLAB?

C) Now create a third file `prog3.m` containing the commands

```
a = 1;
while a ~= 0
a = .1*a
end
```

Run this one as in part A. What is this doing? What does this output tell you about the floating-point number system in MATLAB?