MATH2071: LAB #1(a): Preliminaries

1 Introduction

You will find instructions for each lab, including this one, on the web, starting from my home page: `http://www.math.pitt.edu/~sussmanm` . I do not supply copies on paper. Many students find it helpful to print out copies of the lab instructions before the lab session, although it is not necessary. During the lab session, it is convenient to use the online version because you can "copy-and-paste" instructions from the web page directly into Matlab. If you prefer, you will find a version of this lab in Adobe `pdf` format here.

This version of the first lab is intended only for students who did not take Math 2070.

There are two versions of the first lab. This version introduces the Matlab environment and programming language, and presents the general format of the work you need to hand in. If you have already taken Math 2070, this material would be a repetition, so you should take the alternative version of Lab 1(b). Students who complete Lab 1(a) do not have to complete Lab 1(b).

This lab will occupy three lab sessions. The first session will introduce the mechanics of using Matlab here in the lab. There is some reading to be completed before the second session of this lab and you can do that at any computer with web access. The subsequent two sessions of this lab will present exercises in Matlab use.

The discussion that follows assumes that you are basically familiar with browsing the Web. The next few sections will give a brief introduction to Matlab and explain how to use it and those aspects of the environment that will be important to doing the labs.

2 Grading

The labs roughly follow the material presented in lecture, but are independent of the homework and other exercises presented in lecture. Lab grades count as 30% of your course grade.

Attendance is not required (except as described below), but help is most readily available during the lab sessions.

You are encouraged to work together with other students, but you must provide your own diary and summary files (explained further below).

Each lab will be given a grade of A+, A, B, C, D or 0. These grades correspond with percentage grades of 100, 95, 85, 75, 65 and 0. At the end of the semester, your grades will be averaged and then integrated with your grade in lecture. The grading criteria are:

Grade	Value	description
A+	100	All exercises were completed correctly.
A	95	All exercises were attempted and are substantially correct.
B	85	All exercises were attempted but there are some serious errors.
C	75	Substantial portions of some exercises were omitted.
D	65	Little or nothing correct in the submission.
Zero	0	Lab was not submitted.

Some of the labs include extra credit exercises. The percentage values of these extra credit exercises are stated with the exercises themselves. At the end of the semester, the extra credit percentages will be added to the grade percentages and the average computed from the sum, except that averages will not exceed 100. Unused extra credit on one lab will be applied toward *later* labs. Unsubmitted labs will not be eligible for extra credit.

Each lab is due before 11:59 PM the day the subsequent lab begins. Labs submitted after the day the subsequent lab begins will have 1% deducted from the percentage grade for that lab.

If you are unable to complete a lab by its due date, you may take additional time to complete the lab, *provided you attend each lab session until you have completed and submitted the work.* In that case, an additional 2% will be deducted from that lab's grade for the first week it is late (total of 3%), 3% for the second week (total of 6%), 4% for the third week (total of 10%), *etc. If you do not attend each lab session until you submit your work, you will be given a grade of zero for that lab.* Attending lab sessions will offer the opportunity to resolve any questions you might have.

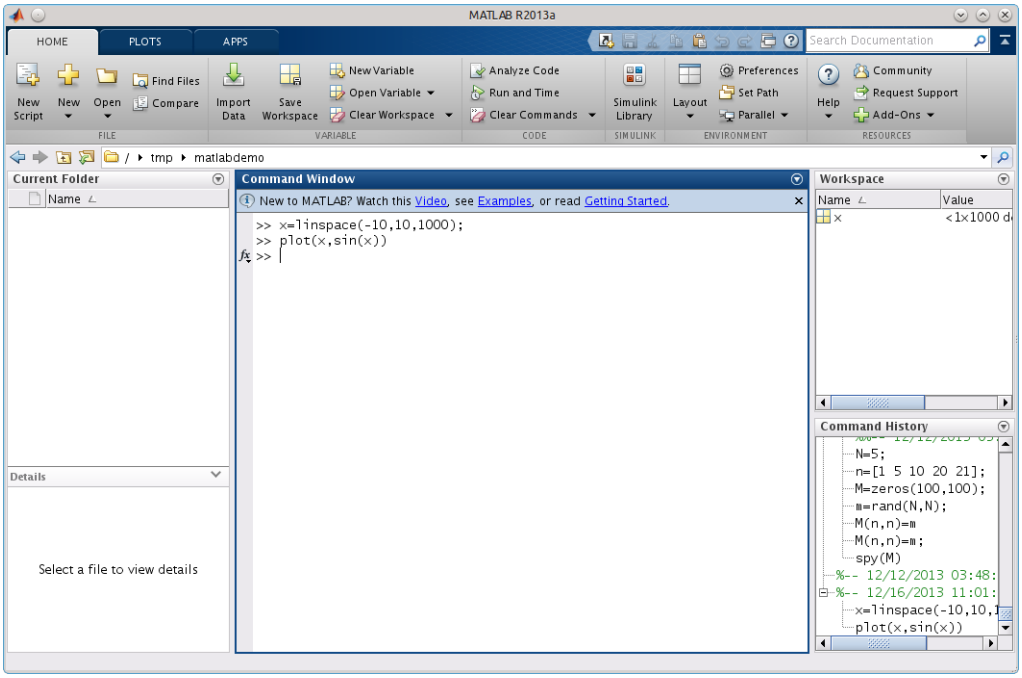
The final due date for labs 1 through 8 is the last day of classes for the semester, and the final due date for labs 9 and 10 will be announced near the end of the semester. Labs that are not submitted on or before their final due date will be given a grade of zero, except under special circumstances.

3 Starting Up Matlab

Matlab is available on the computers in Posvar 1200A, in other computer labs on campus as well as on the Linux computers in the labs in Thackeray hall, such as the seventh floor computer lab opposite the lounge. You can also download Matlab for your personal computer from the university software center, and I recommend you do so.

In this section you will see how to start up of Matlab using the windowing interface. These instructions are the same for Unix, Mac and MS-Windows versions of Matlab. I will also give the command-line equivalents of many of the commands. These command-line equivalents are valid for all versions of Matlab as well as for the Matlab clone named Octave. Generally speaking, anything you can do using a menu can also be done with command-line equivalents. You would use the command-line equivalents when writing scripts and the menus when working interactively.

1. To start up Matlab on the computers in Posvar 1200A, go to the start menu, type `matlab` in the search box and then click on it. A "splash" screen will open and then close by itself, followed by the Matlab desktop window. This window will look something like the following:



The Matlab desktop contains menus for many activities along the top, but we will be using very few of these options. Below the menus is a line containing icons representing "Forward", "Back", "Up to parent folder", and "Open folder", plus a line describing the location of the current folder.

By default, the region below the location of the current folder contains several windowpanes. The pane on the left lists files in the current directory and the contents of one of those files. The pane on the right is divided into a "Workspace" pane containing a list of all variables currently known to Matlab and a "Command History" pane containing copies of your recent commands. The center windowpane is the "Command" windowpane and you will be typing Matlab commands in it. On the left side of this command windowpane there will be a prompt of the form `>>`. Your typed commands go next to this prompt.

2. When working on the computers in Posvar 1200A, I strongly suggest you use a USB drive ("Thumb drive", "Jump drive", "Flash drive", "Memory stick") that you carry with you. Plug it into one of the computer's USB ports. You can use local storage, but it will be wiped out as soon as you log out, so you need to remember to copy your work to your Box account or some other permanent location. When removing your USB drive from the computer, it is best to find the small "USB drive" icon at the bottom of the screen and choose "Safely Remove" to be sure all information has been copied to your drive. (This step is particularly important when using Linux computers!)

3. It is a good idea to organize your work into folders (or directories—they are the same thing). Use the line containing the current folder to navigate to your USB drive (often `e:`). Inside that drive, I suggest that you use a directory named `math2071` for all the work in this lab, and subdirectories `lab01`, `lab02`, ..., `lab10` for each of the labs. You will need to create these directories before you can use them.

   If you wish to use the command line to create and switch directories, after locating your USB drive letter, you can create a directory named `math2071` with the command

```
mkdir math2071
```

and you can make it the current directory with the command

```
cd math2071
```

You can get a listing of the files in the current directory with either the command

```
dir
```

or the command

```
ls
```

Before going on, create the directory `math2071`, with subdirectories `lab01`, `lab02`, ..., `lab10`.

4. The "Command History" windowpane is convenient for recalling what commands you have used recently. In addition, Matlab provides the capability to keep a record of *both* the commands *and* their output. The command to do this is

   ```
   diary diary.txt
   ```

   The name `diary.txt` is actually the name of the file that is created and you can use any name you like. It is a text file, and you should use `.txt` to name it. Before going on, type `diary diary.txt` in the "Command" windowpane. Terminate the command by hitting the "Enter" key. In labs that take up several sessions, using the `diary diary.txt` command in the second session will append to the previous work and will not overwrite it. You must issue the `diary diary.txt` command *each* time you start up Matlab.

   You should also type a comment line that will help you identify your work later. It should start with the comment character, a percent sign `%`, and include the lab number and date. This line will serve as an identifier when you look at the file.

   Note: The diary file contains a complete record of all the commands you enter in Matlab and the responses. You are to retain this *complete* record to send to me for grading. Please do not turn the diary file on and off or edit the diary file to hide your errors. There are times when you write an explanation that is not, in my opinion, correct and I will need the complete diary to establish what really did happen.

5. You can double-click a command in the "Command History" windowpane and it will be executed again. You can also use the mouse to drag a command into the "Command" windowpane and then change it using the arrow keys.

6. You can also recover previous commands by using the up-arrow key and then you can change them using the right and left arrow keys.

7. As will be described below, you will be sending your files to me by email. Since there are sometimes a large number of these files, it is inconvenient to attach them one by one, so I suggest you create a "zip" file. Before you create the "zip" file, exit your diary with the command `diary off`. If you do not, your diary will be incomplete in the "zip" file. You can create the "zip" file from the Matlab command line with the command

   ```
   !zip labfiles.zip *.m *.jpg *.txt
   ```

where "`labfiles.zip`" is the name of the file and can be chosen as you wish. (The exclamation point is necessary to tell Matlab that the command is a "system" command and not a Matlab command.) You then only need to attach this one file to your email. You should do this only once, when you have completed all your work and after you have closed your diary file with the command (`diary off`), or after you have closed Matlab down and then opened it again.

8. After you have completed a lab, exit from Matlab with the command `quit` or use the "X" button along the top to close the window.

4 Using a browser to download files

Some of the labs require that you download files from the web in order to use them. The following exercise illustrates how to download files. The file you will download is a very simple script file.

Exercise 1: If you are not using the online version of this lab, please start it up by starting the browser and finding the online version of this page, beginning from my home page,
`http://www.math.pitt.edu/~sussmanm` . clicking on "Math 2071", scrolling down to the list of labs and clicking on Lab 1.

Right-mouse click on the file demoscript.m to bring up a menu. Choose "Save link as" and a file save box will pop up. Navigate to the directory you made: `math2071` and to its subdirectory `lab01`, and save the file with the name `demoscript.m`. You must use the `.m` extension to tell Matlab that the file contains Matlab commands. Return to the Matlab command window. The file should be visible to Matlab, a fact that you can confirm with the `dir` command or by its presence in the "Current Directory" windowpane. If it is not visible to Matlab, you have probably stored it in a different directory than the one Matlab is looking at. You can find the directory that Matlab is using with the command `pwd` ("print working directory") or in the line immediately above the "Command Window" and "Current Folder" windowpanes in the Matlab window.

Edit the file by typing the command

```
edit demoscript.m
```

or by double-clicking on the file in the "Current Directory" windowpane, or using the "Open" menu. An edit window will show up. Read through the file: the comments make it self-explanitory.

You can tell Matlab to execute (that is, cause the statements in the file to be executed) the file by typing its name, without the "`.m`", on the command line. (There is another method for executing a file that is not so appropriate for this course: you can choose "Run" from the menu on the edit window.)

Do not be confused by the final few statements in the file. They refer to the following exercise.

The following two exercises illustrate the use of the debugging capability of Matlab. Most of the time you will be able to see what is wrong from the Matlab error messages, but sometimes the error is not obvious. In Exercise 2 below, you will see what you might do when you just cannot see why something is wrong.

Exercise 2:

(a) Turn on debugging with the commands

```
dbstop if error
dbstop if naninf
```

or through the Breakpoints menu on the Edit window.

(b) Now, look at `democscript.m` in the edit window. At the bottom there is a statement `%bad=1/(x-1);`. The percent is a comment character, so this statement is not executed. Make this statement active by deleting the percent character and save the changed file.

(c) Execute the script file by typing its name without the `.m` at the command line or by choosing "Run" from the menu on the Edit window.

(d) The division by zero caused an exception and Matlab popped up a window with the offending source line highlighted. You should also note the changed prompt in the command window. **Note:** The window with debugging information in it is the same as the edit window. It may not automatically pop up and you may have to look for it on your desktop.

(e) In the edit window, place the mouse pointer on top of the `x` in the expression `bad=1/(x-1)` and leave it there motionless for a second or so (this is called "hovering"). The value of `x` should be displayed. You can then see why the error occurred.

(f) Look at the prompt in the command window. It has changed to `K>>`. You can do any legal Matlab command at this changed prompt. In particular, you can type the name `x` to get Matlab to print out its value or, more conveniently, you can type the expression `x-1` to find its value. This is an alternative way to see the value of a variable or expression in a debug situation.

(g) You can exit from debugging mode either using the menu in the edit window or with the command

```
dbquit
```

This action will return Matlab to its usual `>>` prompt.

(h) You can turn off debugging features with the command

```
dbclear all
```

or from the debugging menu in the edit window.

Debugging (finding errors in) code you have written is the most time-consuming and least rewarding task in writing code. I am sure you think you will not be making errors, but everyone makes errors and they can be very difficult to find. You can often find your errors based on the line numbers included with Matlab's error messages, but sometimes the error remains a mystery. In that latter case, the Matlab debugger is the most powerful tool you have available.

Another important use of the debugger is for tracing through a piece of code to help you understand how it works. The benefit of the debugger used for tracing is that the effect of each statement is immediatly clear. Careful tracing is the quickest way to learn how code works.

The following exercise illustrates how you can use the debugger to trace execution. It uses the same `democscript.m` file as before.

Exercise 3:

(a) Type the command `clear` at the command line. This will return Matlab to its state just after starting up. No variables will show in the "Workspace" windowpane.

(b) Click on the first executable line (`x=1.3`) of the `democscript.m` file in the edit window. Go to the "Breakpoints" menu and "set" a breakpoint there. A breakpoint is a place where the debugger will always stop execution. When a breakpoint is set, a red dot will be placed next to the line number in the edit window. You can also click on the dash to the left of the line number to set a breakpoint at that line.

(c) Begin executing the file either by choosing "Run" from the debug menu in the edit window, by pressing the "Run" button, or by typing the name `democscript` at the command line. The edit window will show that the script is poised to execute the first line of the file. (That line has not yet been executed.)

(d) Predict, in your mind, what will happen when the line is executed. In this case, the value of x will change to 1.3 and the result "x=1.3" will be printed because there is no semicolon at the end of the line.

(e) Choose "Step" from the Debug menu , press the "Step" button (hovering over a button brings up its description) or issue the command "dbstep" at the command line. This will cause the current line (line 8) to be executed. You will see the variable `x` appear in the Workspace windowpane and `x=1.3000` appear in the command window.

(f) Predict what will happen on the next step. What will the value of the variable `xsquared` become?

(g) Predict what will happen on the next step. What will the value of the variable `p` become? (An estimate is good enough.)

(h) Continue using "Step" and watch the script execute, one line at a time.

(i) If the next line were to call a function, the button next to the "Step" button is the "Step in" button. The "Step" button goes on, using the value of the function while the "Step in" button jumps to the code inside the function and steps there.

(j) Pressing the "Continue" button causes Matlab to continue executing the script until another breakpoint is reached or until the end of the script is reached.

(k) Press the "End Debugging" button to exit from debug mode. Alternatively, use the command `dbquit`.

5 Getting help

It is important to be able to get help when you need it. Matlab provides two help facilities from inside Matlab itself and a third on the web. The easiest way to get help is to use the "Help" menu at the top of the Matlab window. Command-line help is also available from the Matlab prompt by typing "`help` *command*". For example,

```
help diary
```

You will get a short description of how to use the command. You will also get a list of related commands near the bottom of the help description, and you will often find other appropriate commands there. When you write your own Matlab files, you should always include some special comments in the beginning of the file. The comments up to the first executable statement or blank line will be printed out in response to the `help` command. For example, the command

```
help demoscript
```

will give a quick help message from the first three lines of `demoscript.m`. You may notice that the first of these lines is included in the file listing in the "Current Directory" windowpane.

A second way to get help from the command prompt is the following.

```
doc
```

This command brings up a comprehensive help facility, the same one that the Help menu brings up. The content in this help facility is very similar to the one on the web from the URL:
`http://www.mathworks.com/help/matlab/index.html` We will be looking at the help facility on the web later in this lab.

6 Quitting

You exit Matlab by typing `quit` at the command line or by clicking on the "X" at the top of the Matlab window to close the window.

7 Lab summaries

You should complete a report of the results you obtained for each completed lab. This report need not be elaborate. The report consists of at least two files: the complete, original, unmodified, `diary.txt` file(s) plus a summary file. This summary file can be easily created as you do the lab by keeping a text file up in the editor and copying parts of the web page, your commands and output to the file as you work. You can find a sample summary file on my website.

This summary file is very important. It is what I will read first and, if it is well-written and the work is done correctly, I will not need to read anything else. *Never* put incorrect Matlab statements into your summary because it will take me a lot of time to discover you really didn't mean them. I will regard everything in the summary as information you believe is correct and will grade it accordingly. I expect to see

- Brief descriptions of the work you did for each exercise.
- Copies of the main Matlab commands you used for the exercises, along with the numerical results.
- Your description *must* include more detail than merely "yes, I did it and it worked."
- Names of plot files (`.jpg`) corresponding with the different exercises.
- Explanations of anything unusual or interesting, or points of confusion that you were unable to resolve outside lab.
- If you believe I have an error in a lab, please inform me of it. Explain why you think it is an error and, if you like, suggest a correction.

Summarizing your work is important not only for my convenience in grading, but also to help fix in your mind the focus of each exercise.

Equally important, the summary file helps get you into the habit of keeping track of your numerical experiments in some formal manner. When you are doing research, you may be doing hundreds of numerical experiments, and you *must* get into the habit of documenting your work or you will not remember from month to month what each one did. The idea of the summary is that you can easily refresh your memory on exactly what you did to accomplish some particular task.
Here is what I want to see in the summary file:

1. Those parts of the answers to each exercise that I ask for.
2. Explanations of what you did, in full sentences. There should be enough information here to repeat the experiment. Matlab files will help in this documentation.
3. I would like to see a few lines expressing your opinion of the point of each exercise.
4. Easily identified answers to exercises, including numerical values. I do not want to look in your diary file to try to figure out what you did. *Do not* write, "see summary file for details." If you think the details are of interest, copy and paste them into your summary.
5. What was the result of the experiment? Not just the numbers, but what the experiment told you.

Here is what I do NOT want to see in the summary file:

1. Matlab error messages (unless I ask for them).
2. False starts, mistyped commands, etc.
3. Incorrect results that are corrected later.

4. Duplications of anything, unless I explicitly ask for them.

5. Large numbers of printed values, for example, the contents of a vector of length 100. I will not read all these numbers and they end up being like spam.

If you want to know how much detail to include, think of the following scenario. You have completed this course and, a year from now, a friend who is taking the course is having trouble. Your friend comes to you and asks how you did a particular exercise. You have saved your work, so you go look at it. The first place you will look is in your summary to see what you did. If the summary file contains only "Exercise 1.a: complete," you will then have to go re-read the original lab and look for your script files, *etc.* Instead, the summary should describe what you did so you can explain it in general to your friend without referring to other materials. If your friend needs more detail, you can look at the other files you wrote for the lab.

Do not write a summary of the work for the first part of today's lab. Instead, please read the following information about Matlab commands from either the PC here in the lab or from another computer on the web.

8 More on Matlab

The Mathworks, maker of Matlab, includes a short tutorial on using Matlab called *Getting Started*. This tutorial is the first part of more comprehensive Matlab documentation. The "Getting Started" tutorial is also available from the Matlab command prompt with the command `helpdesk` and also from the Help menu, and, if you have your own copy of the Matlab manuals, it comprises the "Getting Started" book.

If you feel you need more explanation of programming techniques, you may find the following reference useful.

> Charles F. Van Loan and K.-Y. Daisy Fan,
> "Insight Through Computing, A MATLAB Introduction to Computational Science and Engineering," SIAM, 2010, ISBN: 978-0-898716-91-7

The "Getting Started" tutorial is an excellent presentation of the basic capabilities of Matlab. In order to have an overview of Matlab, read through the the tutorial. There is only the equivalent of about 10 printed pages of material here, mostly very easy to understand. **Begin the "Getting Started" tutorial now, during your first lab session. Read as much of it as you can now, and complete it at home or from any convenient computer connected to the web.**

If you wish a more comprehensive introduction to Matlab, look at the Matlab documentation or look at the (much shorter) Matlab Primer. Focus on the following sections:

1. Language Fundamentals

2. Mathematics, but just the sections on Elementary Math and Linear Algebra (Matrix operations).

3. Graphics, but just the first section on 2-D and 3-D graphics.

4. Programming Scripts and Functions, but just the first four sections.

If you find the "Getting Started" information too terse or you find you need more detail, there are some video tutorial presentations on the Mathworks web site. I recommend the "Interactive Matlab Tutorial" and the "Computatational Mathematics Tutorial" that are available on the "Academia" portion of the web site. Go to `http://www.mathworks.com`, click on the three horizontal bar icon at the top right, choose "Academia" and then choose "Tutorials" from the "student resources." You may have to log in to your Matlab account, the same one you used to activate your copy of Matlab, or create a new account.

8.1 Matlab fundamentals

This tutorial is quite extensive and may take as much as ten hours to complete. Topics can be chosen individually

- Introduction
- Working with the Matlab User Interface
  - Matlab desktop
  - Saving and Loading Variables
- Variables and Expressions
  - Matlab commands
  - Command History
- Analysis and Visualization with Vectors
  - Introduction
  - Array Operations
    - Arrays
    - Scalar Expansion
  - Mathematical Operations
    - Operators
  - Plotting
    - Plotting Vectors
- Analysis and Visualization with Matrices
  - Introduction
  - Matrix Multiplication
    - Elementwise multiplication
    - Matrix multiplication
  - Function Behavior
  - Plotting Matrices
- Automating Commands with Scripts
  - Introduction
  - Scripts
    - Whale call model
    - Matlab scripts
  - Comments and Help
- Logic and Flow control
  - Introduction
  - Programming
    - Programming introduction

        * Conditional statements
        * For loops

- Writing functions
    - Introduction
    - Matlab functions
        * Matlab functions
        * Creating Functions

8.2 Interactive Computational Mathematics Tutorial

This tutorial is available at `https://www.mathworks.com/academia/student_center/tutorials/computational-math-tu`
It is also a few hours long and covers topics discussed in both 2070 and 2071. It is an excellent supplementary source for these courses. If you plan to pick particular topics, I recommend the following ones.

- Computational Mathematics Tutorial Introduction
- Linear Algebra (most relevant to 2071)
    - Introduction to Linear Algebra
    - Solving Linear Systems
    - Eigenvalue Decomposition
    - Singular Value Decomposition
- Solving Ordinary Differential Equations (most relevant to 2071)
    - Introduction to Solving Ordinary Differential Equations
    - Numerical Solution to ODEs
    - Solving First-Order Systems
    - Matlab ODE Solve Options
- Data Fitting and Working with Nonlinear Equations (most relevant to 2070)
    - Introduction to Data Fitting and Working with Nonlinear Equations
    - Data-Driven Models
    - Solving Nonlinear Equations
    - Solving for Critical Points (extra, but valuable)

The following sections discuss more of the use of Matlab. I will expect a summary of your work for the rest of this lab.

9 Matlab files

The best way to use Matlab is to use its scripting (programming) facility. With sequences of Matlab commands contained in files, it is easy to see what calculations were done to produce a certain result, and it is easy to show that the correct values were used in producing a graph. It is terribly embarrassing to produce a very nice plot that you show to your advisor only to discover later that you cannot reproduce it or anything like it for similar conditions or parameters. When the commands are in clear text files, with

easily read, well-commented code, you have a very good idea of how a particular result was obtained. And you will be able to reproduce it and similar calculations as often as you please.

The Matlab comment character is a percent sign (`%`). That is, lines starting with `%` are not read as Matlab commands and can contain any text. Similarly, any text on a line starting with a `%` can contain textual comments and not Matlab commands. Similarly, a group of lines starting with a line containing only `%{` and ending with a line containing only `%}` will be treated as a comment. It is important to include comments in script and function files to explain what is being accomplished.

Matlab commands are sometimes terminated with a semicolon (`;`) and sometimes not. The difference is that the result of a calculation is printed to the screen when there is no semicolon but no printing is done when there is a semicolon. It is a good idea to put semicolons at the ends of all calculational lines in a function file.

There are three kinds of files that Matlab can use:

1. Script files (or "m-files"),
2. Function files (or "m-files") and
3. Data files.

9.1 Script m-files

A Matlab script m-file is a text file with the extension `.m`, containing comments and Matlab commands. Matlab script files should *always* start off with comments that identify the author, the date, and a brief description of the intent of the calculation that the file performs. Matlab script files are invoked by typing their names without the `.m` at the Matlab command line or by using their names inside another Matlab file. Invoking the script causes the commands in the script to be executed, in order.

9.2 Function m-files

Matlab function files are also text files with the extension `.m`, but the first non-comment line *must* start with the word `function` and be of the form

`function` *output variable(s)* = *function name* `(`*parameters*`)`

For example, the function that computes the sine would start out

```
function y=sin(x)
```

and the name of the file would be `sin.m`. The defining line, starting with the word `function` is called the "signature" of the function. If a function has no input parameters, they, and the parentheses, can be omitted. Similarly, a function need not have output parameters. There can be more than one output parameter, and the syntax for several output parameters is discussed later. The name of the function must be the same as the file name. It is best to have the first line of the function m-file be the signature line, starting with the word "function." The lines following the signature should *always* contain comments with the following information.

1. Repetition of the signature of the function (useful as part of the help message),
2. A brief description of the intent of the calculation the function performs,
3. Brief descriptions of the input and output variables, and,
4. The author's name and date.

Part of the first of these lines is displayed in the "Current directory" windowpane, and the lines themselves comprise the response to the Matlab command `help` *function name*.

The key differences between function and script files are that

- Functions are intended to be used repetitively,
- Functions can accept parameters, and,
- Variables used inside a function are invisible outside the function.

This latter point is important: variables used inside a function (except for output variables) are invisible after the function completes its tasks while variables in script files remain in the workspace.

When I am working on a task, I often start out using script files. As I discover just what tasks are repetitive or when I start to need the same calculation repeated for different parameters, or when I have many intermediate variables that might have the same names as variables in other parts of the calculation, I switch to function files. In these labs, I will specify function file or script file when it is important, and you are free to use what you like when I do not specify.

Because function files are intended to be used multiple times, it is a bad idea to have them print or plot things. Imagine what happens if you have a function that prints just one line of information that you think might be useful, and you put it into a loop that is executed a thousand times. Do you plan to read those lines?

Matlab commands are sometimes terminated with a semicolon (`;`) and sometimes not. The difference is that the result of a calculation is printed to the screen when there is no semicolon but no printing is done when there is a semicolon. It is a good idea to put semicolons at the ends of all calculational lines in a function file.

Matlab also supports data files. The Matlab `save` command will cause every variable in the workspace to be saved in a file called "`matlab.mat`". You can also name the file with the command `save filename` that will put everything into a file named "`filename.mat`". This command has many other options, and you can find more about it using the help facility. The inverse of the `save` command is `load`.

Note: Matlab function names are case-sensitive. This means that the function cos is different from Cos, coS, and COS. File names in Unix and Linux are also case-sensitive, but file names in Microsoft operating systems are not strictly case-sensitive. To avoid confusion for those students who might be using Matlab on a Microsoft system, we will only use lower-case names for Matlab function and script files.

10 Variables and values

Values (with few exceptions) in Matlab are always "floating point" numbers with about sixteen decimal digits of accuracy. When Matlab prints values, however, it will truncate a number to about four digits to the right of the decimal point, or less if appropriate. Values that are integers are usually printed without a decimal point. Remember, however, that when Matlab prints a number, it may *not* be telling you all it knows about that number.

When Matlab prints values, it often uses a notation similar to scientific notation, but written without the exponent. For example, Avogadro's number is $6.022 \cdot 10^{23}$ in usual scientific notation, but Matlab would write this as `6.022e+23`. The `e` denotes 10. Similarly, Matlab would write `1/2048=4.8828e-04`. You can change the number of digits displayed with the `format` command. (See `help format` for details.)

Matlab uses variable names to represent data. A variable name represents a matrix containing complex double-precision data. Of course, if you simply tell Matlab `x=1`, Matlab will understand that you mean a 1×1 matrix and it is smart enough to print `x` out without its decimal and imaginary parts, but make no mistake: they are there. And `x` can just as easily turn into a matrix.

A variable can represent some important value in a program, or it can represent some sort of dummy or temporary value. Important quantities should be given names longer than a few letters, and the names should indicate the meaning of the quantity. For example, if you were using Matlab to generate a matrix containing a table of squares of numbers, you might name the table `tableOfSquares`. (The convention I am using here is that the first part of the variable name should be a noun and it should be lower case. Modifying

words follow with upper case letters separating the words. This rule comes from the officially recommended naming of Java variables.)

Once you have used a variable name, it is bad practice to re-use it to mean something else. It is sometimes necessary to do so, however, and the statement

```
clear variableOne variableTwo
```

should be used to clear the two variables `variableOne` and `variableTwo` before they are re-used. This same command is critical if you re-use a variable name but intend it to have smaller dimensions.

Matlab has a few reserved variable names. You should not use these variables in your m-files. If you do use such variables as `i` or `pi`, they will lose their special meaning until you clear them. Reserved names include

`ans`: The result of the previous calculation.

`computer`: The type of computer you are on.

`eps`: The smallest positive number ϵ that can be represented on the computer and that satisfies the expression $1 + \epsilon > 1$. This value indicates the size of "roundoff." (Note: The name `eps` is a special name, and it means a particular value. We will have occasion to use a variable named `epsilon`. Do not confuse these two names.)

`i, j`: The imaginary unit ($\sqrt{-1}$). In this course you should avoid using `i` as a subscript or loop index.

`inf`: Infinity (∞). This will be the result of dividing 1 by 0.

`NaN`: "Not a Number." This will be the result of dividing 0 by 0, or `inf` by `inf`, multiplying `0` by `inf`, *etc.*

`pi`: π

`realmax, realmin`: The largest and smallest real numbers that can be represented on this computer.

`version`: The version of Matlab you are running. (The `ver` command gives more detailed information.)

Exercise 4: Start up Matlab and use it to answer the following questions. Do not forget to open the diary file by using the command "`diary diary.txt`".

(a) What are the values of the reserved variables `pi`, `eps`, `realmax`, and `realmin`?

(b) Use the "`format long`" command to display `pi` in full precision and "`format short`" (or just "`format`") to return Matlab to its default, short, display.

(c) No matter how it is printed, the internal precision of any variable is always about 15 decimal digits. The value for pi printed in short format is 3.1416. What is `pi-3.1416`? You should see that this value is not zero. You found the value of pi printed using `format long` in the previous part of this exercise. What is the difference between the printed value and pi? This value might not be zero, but it is still much smaller than the value of `pi-3.1416`.

(d) Set the variable `a=1`, the variable `b=1+eps`, the variable `c=2`, and the variable `d=2+eps`. What is the difference in the way that Matlab displays these values?

(e) Do you think the values of `a` and `b` are different? Is the way that Matlab formats these values consistent with your idea of whether they are different or not?

(f) Do you think the values of `c` and `d` are different? Explain your answer.

(g) Will the command `format long` cause all the decimal places in b to be printed, or is there still some missing precision?

(h) Choose a value and set the variable `x` to that value.

(i) What is the square of `x`? Its cube?

(j) Choose an angle θ and set the variable `theta` to its value (a number).

(k) What is $\sin\theta$? $\cos\theta$? Angles can be measured in degrees or radians. Which of these has Matlab used?

(l) Matlab variables can also be given "character" or "string" values. A string is a sequence of letters, numbers, spaces, etc., surrounded by single quotes (`'`). In your own words, what is the difference between the following two expressions?

```
a1='sqrt(4)'
a2=sqrt(4)
```

(m) The Matlab `eval` function is used to EVALuate a string as if it were typed at the command line. If `a1` is the string given above, what is the result of the command `eval(a1)`? Of `a3=6*eval(a1)`?

(n) Use the `save` command to save all your variables. Check your "Current Directory" to see that you have created the file `matlab.mat`.

(o) Use the `clear` command. Check that there are no variables left in the "current workspace" (windowpane is empty).

(p) Restore all the variables with `load` and check that the variables have been restored to the "Current workspace" windowpane.

11 Vectors and matrices

We said that Matlab treats all its variables as though they were matrices. Important subclasses of matrices include row vectors (matrices with a single row and possibly several columns) and column vectors (matrices with a single column and possibly several rows). One important thing to remember is that you don't have to declare the size of your variable; Matlab decides how big the variable is when you try to put a value in it. The easiest way to define a row vector is to list its values inside of square brackets, and separated by spaces or commas:

```
rowVector = [ 0, 1, 3, -6, pi ]
```

The easiest way to define a column vector is to list its values inside of square brackets, separated by semicolons or line breaks.

```
columnVector1 = [ 0; 1; 3; -6; pi ]
columnVector2 = [ 0
                  1
                  9
                  36
                  100 ]
```

(It is not necessary to line the entries up as I have done, but it makes it look nicer.) Note that `rowVector` is **not** equal to `columnVector1` even though each of their components is the same.

Matrices can be written using both commas and semicolons. The matrix

$$\mathcal{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{1}$$

can be generated with the expression

```
A=[1,2,3;4,5,6;7,8,9]
```

or, more clearly, as

```
A=[ 1 2 3
    4 5 6
    7 8 9];
```

Matlab has a special notation for generating a set of equally spaced values, which can be useful for plotting and other tasks. The format is:

```
start : increment : finish
```

or

```
start : finish
```

in which case the increment is understood to be 1. Both of these expressions result in row vectors. So we could define the even values from 10 to 20 by:

```
evens = 10 : 2 : 20
```

Sometimes, you'd prefer to specify the *number* of items in the list, rather than their spacing. In that case, you can use the `linspace` function, which has the form

```
linspace( firstValue, lastValue, numberOfValues )
```

in which case we could generate six even numbers with the command:

```
evens = linspace ( 10, 20, 6 )
```

or fifty evenly-spaced points in the interval `[10,20]` with

```
points = linspace ( 10, 20, 50 )
```

As a general rule, use the colon notation when `firstValue`, `lastValue` and `increment` are integers, or when you would have to do mental arithmetic to get `increment`, and `linspace` otherwise.

Another nice thing about Matlab vector variables is that they are *flexible*. If you decide you want to add another entry to a vector, it's very easy to do so. To add the value 22 to the end of our `evens` vector:

```
evens = [ evens, 22 ]
```

and you could just as easily have inserted a value 8 before the other entries, as well.

Even though the number of elements in a vector can change, Matlab always knows how many there are. You can request this value at any time by using the `numel` function. For instance,

```
numel ( evens )
```

should yield the value 7 (the 6 original values of 10, 12, ... 20, plus the value 22 tacked on later). In the case of matrices with more than one nontrivial dimension, the `numel` function returns the *total* number of entries. the `size` function returns a vector containing two values: the number of rows and the number of columns. To get the number of rows of a variable `v`, use `size(v,1)` and to get the number of columns use `size(v,2)`. For example, since `evens` is a row vector, `size( evens, 1)=1` and `size( evens, 2)=7`, one row and seven columns.

To specify an individual entry of a vector, you need to use index notation, which uses **round** parentheses enclosing the index of an entry. **The first element of an array has index 1** (as in Fortran, but not C or Java). Thus, if I want to alter the third element of `evens`, I could say

```
evens(3) = 2
```

There is special syntax for the final index value. Since `evens` is a vector with length 7, you can write `evens(end)` to mean the same thing as `evens(7)`.

Exercise 5:

(a) Use the `linspace` function to create a row vector called `meshPoints` containing exactly 500 values with values evenly spaced between -1 and 1. Do not print all 500 values!

(b) What expression will yield the value of the 53^{th} element of `meshPoints`? What is this value?

(c) Double-click on the variable `meshPoints` in the "Current workspace" windowpane to view it as a vector and confirm its length is 500.

(d) Use the `numel` function to again confirm the vector has length 500.

(e) Produce a plot of a sinusoid on the interval $[-1, 1]$ using the command

```
plot(meshPoints,sin(2*pi*meshPoints))
```

Please save this plot as a jpeg (.jpg) file and send it along with your summary. You can use the "File→Save as" menu choice from the plot window. Use file type "JPEG image." The command line expression to save a plot as JPEG is

```
print -djpeg plotname.jpg
```

where "`plotname.jpg`" is a name you choose for the file.
Remarks:

- Although these files are typically written with an `.jpg` extension, the print command uses the 4-letter description "jpeg".
- There are many other graphical file formats that Matlab can produce. Please use only "jpeg" format in these labs.

(f) Create a file named `exer5.m`. You can use the Matlab command `edit`, type the commands you used for this exercise into the window and use "Save as" to give it a name, or you can highlight some commands in the history windowpane and use a right mouse click to bring up a menu to save the commands as an m-file. The first lines of the file should be the following:

```
% Lab 1, exercise 5
% A sample script file.
% Your name and the date
```

Follow the header comments with the commands containing exactly the commands you used in the earlier parts of this exercise. Test your script by using `clear` to clear your results and then execute the script from the command line by typing `exer5`.

12 Vector and matrix operations

Matlab provides a large assembly of tools for matrix and vector manipulation. We will investigate a few of these by trying them out.

Exercise 6: Define the following vectors and matrices:

```
rowVec1 = [ -1 -4 -9]
colVec1 = [ 2
            9
            8 ]
mat1 = [ 1  3  5
         7  -9 2
         4  6  8 ]
```

(a) You can multiply vectors by constants. Compute

```
colVec2 = (pi/4) * colVec1
```

(b) The cosine function can be applied to a vector to yield a vector of cosines. Compute

```
colVec2 = cos( colVec2 )
```

Note that the values of `colVec2` have been overwritten. Are these the values you expect?

(c) You can add vectors and multiply by scalars. Compute

```
colVec3 = colVec1 + 2 * colVec2
```

(d) Matlab will not allow you to do illegal operations! Try to compute

```
illegal = colVec1 + rowVec1;
```

Look carefully at the error message. You must recognize from the message what went wrong when you see it in the future.

(e) The Euclidean norm of a matrix or a vector is available using `norm`. Compute

```
norm(colVec3)
```

(f) You can do row-column matrix multiplication. Compute

```
colvec4 = mat1 * colVec1
```

(g) A single quote following a matrix or vector indicates a (Hermitian) transpose.

```
mat1Transpose = mat1'
rowVec2 = colVec3'
```

Warning: The single quote really means the complex-conjugate transpose (or Hermitian adjoint). If you want a true transpose applied to a complex matrix you must use "`.'`" (dot-single quote).

(h) Transposes allow the usual operations. Even if A is a non-symmetric matrix, AA^T is always symmetric, and you might find $\mathbf{u}^T\mathbf{v}$ a useful expression to compute the dot (inner) product $\mathbf{u} \cdot \mathbf{v}$ (although there is a `dot` function in Matlab).

```
mat2 = mat1 * mat1'     % mat2 is a symmetric matrix
rowVec3 = rowVec1 * mat1
dotProduct = colVec3' * colVec1
euclideanNorm = sqrt(colVec2' * colVec2)
```

(i) Matrix operations such as determinant and trace are available, too.

```
determinant = det( mat1 )
traceOfMat1 = trace( mat1 )
```

(j) You can pick certain elements out of a vector, too. Use the following command to find the smallest element in a vector `rowVec1`.

```
min(rowVec1)
```

(k) The `min` and `max` functions work along one dimension at a time. They produce vectors when applied to matrices.

```
max(mat1)
```

(l) You can compose vector and matrix functions. For example, use the following expression to compute the max norm of a vector.

```
max(abs(rowVec1))
```

(m) How would you find the single largest element of a matrix?

(n) As you know, a magic square is a matrix all of whose row sums, column sums and the sums of the two diagonals are the same. (One diagonal of a matrix goes from the top left to the bottom right, the other diagonal goes from top right to bottom left.) Show by direct computation that if the matrix `A` is given by

```
A=magic(100);   % please do not print all 10,000 entries.
```

The matrix `A` has 100 row sums (one for each row), 100 column sums (one for each column) and two diagonal sums. These 202 sums should all be exactly the same, and you **could** verify that they are the same by printing them and "seeing" that they are the same. It is easy to miss small differences among so many numbers, though. **Instead**, verify that `A` is a magic square by constructing the 100 column sums (without printing them) and computing the maximum and minimum values of the column sums. Do the same for the 100 row sums, and compute the two diagonal sums. Check that these six values are the same. If the maximum and minimum values are the same, the flyswatter principle says that all values are the same.
Hints:

- Use the Matlab `min` and `max` functions.
- Recall that `sum` applied to a matrix yields a row vector whose values are the sums of the columns.
- The Matlab function `diag` extracts the diagonal of a matrix, and the composed function `sum(diag(fliplr(A)))` computes the sum of the other diagonal.

(o) Suppose we want a table of integers from 0 to 9, their squares and cubes. We could start with

```
integers = 0 : 9
```

but now we'll get an error when we try to multiply the entries of `integers` by themselves.

```
squareIntegers = integers * integers
```

Realize that Matlab deals with vectors, and the default multiplication operation with vectors is row-by-column multiplication. What we want here is *element-by-element* multiplication, so we need to place a *period* in front of the operator:

```
squareIntegers = integers .* integers
```

Now we can define `cubeIntegers` and `fourthIntegers` in a similar way.

```
cubeIntegers = squareIntegers .* integers
fourthIntegers = squareIntegers .* squareIntegers
```

Finally, we would like to print them out as a table. `integers`, `squareIntegers`, *etc.* are row vectors, so make a matrix whose **columns** consist of these vectors and allow Matlab to print out the whole matrix at once.

```
tableOfPowers=[integers', squareIntegers', cubeIntegers', fourthIntegers']
```

(p) Watch out when you use vectors. The multiplication, division and exponentiation operators all have two possible forms, depending on whether you want to operate on the arrays, or on the elements in the arrays. In all these cases, you need to use the **period** notation to force elementwise operations. Compute the squares of the values in `integers` alternatively using the exponentiation operator as:

```
sqIntegers = integers .^ 2
```

and check that the two calculations agree with the command

```
norm(sqIntegers-squareIntegers)
```

which should result in zero.

Remark: Addition, subtraction, and division or multiplication by a scalar do not require the dot in front of the operator, although you will get the correct result if you use one.

(q) The index notation can also be used to refer to a subset of elements of the array. With the *start:increment:finish* notation, we can refer to a range of indices. Two-dimensional vectors and matrices can be constructed by leaving out some elements of our three-dimensional ones. For example, submatrices an be constructed from `tableOfPowers`. (The `end` function in Matlab means the last value of that dimension.)

```
tableOfCubes = tableOfPowers(:,[1,3])
tableOfOddCubes = tableOfPowers(2:2:end,[1,3])
tableOfEvenFourths = tableOfPowers(1:2:end,1:3:4)
```

Note: `[1:3:4]` is the same as `[1,4]`.

(r) You have already seen the Matlab function `magic(n)`. Use it to construct a 10×10 matrix.

```
A = magic(10)
```

What commands would be needed to generate the four 5×5 matrices in the upper left quarter `AUL`, the upper right quarter `AUR`, the lower left quarter `ALL`, and the lower right quarter `ALR`.

(s) It is possible to construct vectors and matrices from smaller ones in the same way they can be constructed from numbers. Reconstruct the matrix `A` in the previous exercise as

```
B=[AUL AUR
   ALL ALR];
```

and show that `A` and `B` are the same by showing that `norm(A-B)` is zero.

Repeated Warning: Although multiplication of vectors is illegal without the dot, division of vectors is legal! It will be interpreted in terms of the Moore-Penrose pseudoinverse and is almost always **undesirable** in the context of this course! Beware!

13 Flow control

It is critical to be able to ask questions and to perform repetitive calculations in m-files. These topics are examples of "flow control" constructs in programming languages. Matlab provides two basic looping (repetition) constructs: `for` and `while`, and the `if` construct for asking questions. These statements each surround several Matlab statements with `for`, `while` or `if` at the top and `end` at the bottom.

Note: It is an excellent idea to indent the statements between the `for`, `while`, or `if` lines and the `end` line. This indentation strategy makes code immensely more readable. Your m-files will be expected to follow this convention.

	Syntax	Example

`for` loop	`for` *control-variable=start : increment : end*
	Matlab statement ...
	...
	`end`
		`nFactorial=1;`
		`for i=1:n`
		`  nFactorial=nFactorial*i;`
		`end`

`while` loop	*Matlab statement initializing a control variable*
	`while` *logical condition involving the control variable*
	Matlab statement ...
	...
	Matlab statement changing the control variable
	`end`
		`nFactorial=1;`
		`i=1;  % initialize i`
		`while i <= n`
		`  nFactorial=nFactorial*i;`
		`  i=i+1;`
		`end`

a simple `if`	`if` *logical condition*
	Matlab statement ...
	...
	`end`
		`if x ~= 0   % ~ means "not"`
		`  y = 1/x;`
		`end`

a compound `if`	`if` *logical condition*
	Matlab statement ...
	...
	`elseif` *logical condition*
	...
	`else`
	...
	`end`
		`if x ~= 0`
		`  y=1/x;`
		`elseif sign(x) > 0`
		`  y = +inf;`
		`else`
		`  y = -inf;`
		`end`

Note that `elseif` is one word! Using two words `else if` changes the statement into two nested `if` statements with possibly a *very* different meaning, and a different number of `end` statements.

Exercise 7: The "max" or "sup" or "infinity" norm of a vector is given as the maximum of the absolute values of the components of the vector. Suppose $\{v_n\}_{n=1,\ldots,N}$ is a vector in $\mathbb{R}^N$, then the infinity norm is given as

$$\|v\|_\infty = \max_{n=1,\ldots,N} |v_n| \tag{2}$$

If `v` is a Matlab vector, then the Matlab function `numel` gives its length, and the following code will compute the infinity norm. Note how indentation helps make the code understandable. (Matlab already has a `norm` function to compute norms, but this is how it could be done.)

```
% find the infinity norm of a vector v

N=numel(v);
nrm=abs(v(1));
for n=2:N
  if abs(v(n))>nrm
    nrm=abs(v(n));   % largest value up to now
  end
end
nrm    % no semicolon means value is printed
```

(a) Define a vector as

```
v=[ -5 2 0 6 8 -1 -7 -10 -10];
```

(b) Count the number of elements in `v`. Does that agree with the result of the `numel` function?

(c) Use cut-and-paste to put the above code into the Matlab command windowpane and execute it.

(d) What is the first value that `nrm` takes on?

(e) How many times is the statement with the comment "`largest value up to now`" executed?

(f) What are all the values taken by the variable `nrm`?

(g) What is the final value of `nrm`?

14 M-files

If you have to type everything at the command line, you will not get very far. You need some sort of scripting capability to save the trouble of typing, to make editing easier, and to provide a record of what you have done. You also need the capability of making functions or your scripts will become too long to understand. In this section we will consider first a script file and later a function m-file. We will be using graphics in the script file, so you can pick up how graphics can be used in our work.

Exercise 8:

(a) Use cut-and-paste to copy the code given above for the infinity norm into a file named `exer8a.m`. Recall you can get an editor window from the "New script file" button or from the `edit` command in the command windowpane.

(b) Add your name and the date below the introductory comment. You should always place your name and the date in an m-file. Don't forget to save the file.

(c) Redefine the vector

```
v = [-35 -20 38 49 4 -42 -9 0 -44 -34];
```

(d) Execute the script m-file you just created by typing just its name (`exer8a`) without the `.m` extension in the command windowpane. What is the infinity norm of this vector?

(e) The usual Euclidean or 2-norm is defined as

$$\|v\|_2 = \sqrt{\sum_1^N v_n^2} \tag{3}$$

Copy the following Matlab code to compute the 2-norm into a file named `exer8b.m`. Be sure to add your name and the date to the comments.

```
% find the two norm of a vector v
% your name and the date

N=numel(v);
nrm=v(1)^2;
for n=2:N
  nrm = nrm + v(n)^2;
end
nrm=sqrt(nrm)    % no semicolon means value is printed
```

(f) Using the same vector `v`, execute the script `exer8b`. What are the first four values the variable `nrm` takes on? What is its final value?

(g) Look carefully at the mathematical expression (3) and the Matlab code in `exer8b.m`. The way one translates a mathematical summation into Matlab code is to follow the steps:

i. Set the initial value of the sum variable (`nrm` in this case) to zero or to the first term.

ii. Put an expression adding subsequent terms inside a loop. In this case it is of the form `nrm=nrm+`*something*.

Script files are very convenient, but they have drawbacks. For example, if you had two different vectors, `v` and `w`, for which you wanted norms, it would be inconvenient to use `exer8a` or `exer8b`. It would be especially inconvenient if you wanted to get, for example, $\|v\|_2 + 1/\|w\|_\infty$. This inconvenience is avoided by using function m-files. Function m-files define your own functions that can be used just like Matlab functions such as sin(x), *etc.*

Exercise 9:

(a) Copy the file `exer8a.m` to a file named `infinity_norm.m`. (You can use "save as" or cut-and-paste to do this.) Add the following lines to the beginning of the file:

```
function nrm=infinity_norm(v)
% nrm=infinity_norm(v)
% v is a vector
% nrm is its infinity norm

% your name and the date
```

(b) The first line of a function m-file is called the "signature" of the function. The first comment line repeats the signature in order to explain the "usage" of the funciton. Subsequent comments explain the parameters (such as `v`) and the output (such as `norm`) and, if possible, briefly explain the methods used. You should have one line with your name and the date. The function name and the file name *must agree*.

(c) Place a semicolon on the last line of the file so that nothing will normally be printed by the function.

(d) Use the Matlab "help" command:

```
help infinity_norm
```

This command will repeat the first lines of comments (up to a blank line or a line of code) and provides a quick way to refresh your memory of how the function is to be called and what it does.

(e) Invoke the function in the command windowpane by typing

```
infinity_norm(v)
```

(f) Repeat the above steps to define a function named `two_norm.m` from the code in `exer8b.m`. Be sure to put comments in correctly.

(g) Define two vectors

```
a = [ -43 -37 24 27 37 -33 -19 29 43 -49 ];
b = [ -5 -4 -29 -29 30 33 20 31 42 14];
```

and find the value of infinity norm of `a` and the two norm of `b` with the commands

```
aInfinity = infinity_norm(a)
bTwo      = two_norm(b)
```

Note that you no longer need to use the letter `v` to denote the vector, and it is easy to manipulate the values of the norms.

(h) What Matlab expression would yield the value $\|a\|_2 + 1/\|b\|_\infty$? What is this value?

Matlab extends the usual notion of a function that returns a single value by allowing functions to return *several* values. The signature line for such a function that returns three output variables would be

```
function [out1,out2,out3]=func(in)
```

Of course, there could be fewer or more output variables and fewer or more than one input variable. It is important to realize that the expression `[out1,out2,out3]` is *not a vector*, although it looks like one. The reason is that the output variables might not be all of the same type. For example, `out1` might be a number, `out2` might be a matrix, and `out3` might be a string. In the following exercise you will see an example of a function that returns two values.

Exercise 10: Suppose you want to write a function that returns *both* the infinity and two norms of a vector at once. Your code might look like the following.

```
function [normInf,normTwo]=both_norms(v)
% [normInf,normTwo]=both_norms(v)
% returns both the infinity and two norms of the vector v

% your name and the date
normInf = infinity_norm(v);
normTwo = ???
```

(a) Copy the above code template to a file named `both_norms.m` and fill in the line containing question marks (`???`).

(b) What is the result of the following statement, where `b` is the vector from the previous exercise?

```
[nrm,nrm2]=both_norms(b)
```

(c) What is the result of the following statement?

```
both_norms(b)
```

(d) What is the result of the following statement?

```
[~,nrm]=both_norms(b)
```

15 Ordinary differential equations and graphics

In this section you will see how to use plotting to enhance work that is focussed on something else, like solving a differential equation.

The differential equation

$$y' = -y + \sin x$$

with initial condition $y(0) = 0$ has an exact solution $y(x) = .5 * (e^{-x} + \sin x - \cos x)$. It also has an approximate numerical solution defined by Euler's formula (see below for more detail) as

$$y_{k+1} = y_k + h(-y_k + \sin x_k). \tag{4}$$

In some sense, $y(x_{k+1}) \approx y_{k+1}$ We are going to look at how this expression evolves for $x > 0$.

Exercise 11: Copy the following text into a file named `exer11.m` and then answer the questions about the code.

```
% compute the solution of the differential equation
% y'+y=sin(x)
% starting with y=0 at x=0 using Euler's method
STEPSIZE=.5;
NTERMS=30;
```

```
clear x y exactSolution
y(1)=0;
x(1)=0;
exactSolution(1)=0;
for k=1:NTERMS
  x(k+1)=x(k)+STEPSIZE;
  y(k+1)=y(k)+STEPSIZE*(-y(k)+sin(x(k)));
  exactSolution(k+1)=.5*(exp(-x(k+1))+sin(x(k+1))-cos(x(k+1)));
  plot(x,y);  % default line color is blue
  axis([0,16,-1.1,1.1]);
  hold on
  plot(x,exactSolution,'g'); % g for green line
  legend('Euler solution','Exact solution')
  hold off
  disp('Press a key to continue.')
  pause
end
```

It is always good programming practice to define constants symbolically at the beginning of a program and then to use the symbols within the program. Sometimes these special constants are called "magic numbers." By convention, symbolic constants are named using all upper case.

(a) Add your name and the date to the comments at the beginning of the file.

(b) How is the Matlab variable `x` related to the dummy variable x in Equation (4)? (Please use no more than one sentence for the answer.)

(c) How is the Matlab statement that begins `y(k+1)=...` related to the expression in Equation (4)? (Please use no more than one sentence for the answer.)

(d) In your own words, what is the role of the Matlab function `clear` in this function? You can use the Matlab command `help clear` to find out what `clear` does in general.

(e) Use the Matlab help facility to see what the `plot` commands, the two `hold` commands, the `axis` command, the `pause` command, and the `legend` command do.

(f) Execute the script by typing its name `exer11` at the command line. The script displays a plot and waits for you to hit the "enter" key to track evolution of the solution. You do not have to send me these plots.

In general, a first-order ordinary differential equation can be written in the form

$$y' = f(x, y) \tag{5}$$

where $y' = \frac{dy}{dx}$. Such an equation needs an initial condition $y(x_0) = x_0$. Perhaps the simplest method for numerically finding a solution of (5) is to use the "explicit Euler" method wherein a discrete selection of points $x_k = x_0 + h(k-1)$ where h is some fixed step size and $k = 1, 2, 3, \ldots$. Writing the approximate value of $y(x_k)$ as y_k, then Euler's explicit method can be derived by approximating the derivative $(y')_k \approx (y_{k+1} - y_k)/h$ and writing

$$y_{k+1} = y_k + hf(x_k, y_k) \tag{6}$$

This method can be compared with the following expression appearing in `exer11.m`.

```
y(k+1)=y(k)+STEPSIZE*(-y(k)+sin(x(k)));
```

You have now completed all the required work for this lab. There is an extra-credit exercise below. If you wish to gain the extra credit, do that exercise now. If you do not wish to do the extra credit exercise, instructions for sending me your files follow that exercise.

16 Extra credit (8 points)

In this lab, values that you wanted to see were printed by Matlab automatically because the semicolon was left off the end of the lines, or because you used the Matlab `disp` command.

There is an additional command that is used for displaying results and that allows you a great deal of formatting flexibility. It is similar to the "printf" command in C and Java.

Exercise 12: Consider the following simple loop

```
for i=0:16
  disp(['The square root of ',num2str(i^2),' is ',num2str(i)]);
end
```

(a) Execute this loop. Note that the columns of numbers are not aligned, by which is meant the least significant digits are not printed one above the other.

(b) Read the help message about the function `fprintf`. Re-write the loop using `fprintf` so that the numbers that are printed are aligned so their final digits occur on a vertical line on the screen. **Hint:** If the "`fid`" parameter is omitted, then printing will go to the screen by default. Please include a copy of your fprintf command in your summary file.

(c) Replace the square root with the tangent, so that the tangents of the numbers $0, 1, 4, 9, \ldots, 256$ are printed and so the decimal points are aligned. Print at least four digits to the right of the decimal point. Please include a copy of your fprintf command in your summary file.

(d) Similarly, replace the tangent with the exponential, so that the exponentials of the numbers $0, 1, 4, 9, \ldots, 256$ are printed and so the decimal points are aligned. Please include a copy of your fprintf command in your summary file.

17 Sending me your files

When you have finished your work, be sure to send it to me. It should include your summary file, the `diary.txt` file, each of the m-files `exer5.m`, `exer8a.m`, `exer8b.m`, `exer11.m`, `infinity_norm.m`, `two_norm.m`, `both_norms.m` and the plot files (`.jpg` files) you created. Also include your extra credit work if you chose to do it. To create the zip file, the easiest way is to exit Matlab and then:

1. Navigate to and select the folder in which you did your work, or select only the files in that folder that you want to send to me.
2. Right click and choose "Send To"
3. Slide Right and choose "Compressed (zipped) folder"
4. Allow the file or folder to compress
5. You should now see an icon with the same name plus a .zip extension. It may have a zipper on the folder.
6. Rename the file if you'd like.
7. This is the compressed file that you send to me via email. Recall that my email address is **sussmanm@math.pitt.ed**

Last change `$Date: 2016/12/28 22:45:53 $`