Introduction to MATLAB

Table of Contents

Preliminary ................................................................. 1
Basics ......................................................................... 1
ICE ........................................................................... 4
Vectors ....................................................................... 5
ICE ........................................................................... 9
Plotting .................................................................... 10
ICE .......................................................................... 12
Matrices ..................................................................... 13
ICE .......................................................................... 17
Matrix/Vector and Matrix/Matrix Operations ......................... 17
ICE .......................................................................... 20
Flow Control ............................................................... 20
ICE .......................................................................... 26
Functions .................................................................... 26
ICE .......................................................................... 29
Variable Scope ............................................................ 29
ICE .......................................................................... 31
File I/O ....................................................................... 32
ICE .......................................................................... 34
Parallel Computing ........................................................ 35
ICE .......................................................................... 38

David Koslicki
Oregon State University
3/20/2015

Preliminary

1. Starting up matlab
2. Layout (command window, workspace, command history, current folder, editor)
3. Using the editor (control flow, saving, executing cells, saving, loading, running)

Basics

MATLAB (short for Matrix Laboratory) is a numerical computing environment / Computer Algebra System and programming language originally released in 1984. MATLAB is one of the premier numerical computing environments and is widely used in academia as well as industry. One of the reasons for its widespread usage is the plethora of built-in functions and libraries/packages for common numerical routines.

While MATLAB is rather expensive, there exists a "clone" called GNU Octave. There are only a few slight syntactical differences between MATLAB and Octave. While Octave is free and open source, MATLAB does have a more robust library/package environment and is generally faster than Octave. Interestingly, Octave is named after Octave Levenspiel, an emeritus chemical engineering professor at Oregon State University.
Let's jump right in to using MATLAB. MATLAB is only on the Windows machines, and should be located under the "Math Software" portion of the start menu. When you first start up, you should notice a few different windows, namely, the **Current Folder**, **Workspace**, **Command Window**, **Command History**, and **Editor** windows. We will first concentrate on the **Command Window**.

You may enter commands at the command line prompt ">>". Let's start with the simplest command possible

1+1

You'll notice that the answer has been printed to your screen. This answer is also stored in the global variable *ans*. This variable will store whatever the previous output to your screen was. Note also that this variable now shows up in your **Workspace** window. This window contains all the variables that have been created in your MATLAB session along with some other information about them (like the value of the variable, how much memory it takes up, etc).

There are a couple different ways to delete a variable. One way is to use

```matlab
clear('ans')
```

Another is to click on the variable *ans* in the **Workspace** window and hit delete (or right mouse click and select delete). All the variables may be cleared by using

```matlab
clear
```

And the command window can be cleared by using

```matlab
clc
```

To quit your matlab session, you can either just close the window, or else type the command `exit`.

There is a built-in help menu for matlab functions that can be accessed by typing `help` and then a search term. For example

```matlab
help lsqnonneg
```

More extensive information can also be obtained by putting your cursor over the function of interest, and then hitting F1. In my opinion, the help documentation is rather poorly done in MATLAB itself, but the internet help documentation is much better. For example, more extensive help information on the *lsqnonneg* function can be found at [http://www.mathworks.com/help/matlab/ref/lsqnonneg.html](http://www.mathworks.com/help/matlab/ref/lsqnonneg.html)

MATLAB has all the standard "calculator" commands including basic arithmetic operations.

```matlab
1+2*(3/4-5/6)^2
```

As usual, the order of operations is

1. Brackets ()
2. Powers ^
3. *./ (left to right)
4. +,- (left to right)

Be careful, the operator \ is very different from !

While it might appear that MATLAB only gives a few significant figures, this is just a style choice on their part. You can view more digits by using the `format` command
format short
1/3
format long
1/3

You can see that more digits are displayed using the long format. We will see later how to precisely control the number of digits to be displayed.

There are a few special numbers/constants in MATLAB. A few of the important ones are

pi
i % square root of negative one
j % square root of negative one
eps % The smallest number MATLAB can represent
NaN % Means Not A Number. Used to handle things like 0/0
Inf % Infinity

Variables

We have already seen one variable (namely, ans). Variables are key components of any programming language and are used to store numbers (or any other kind of data structure). Let's define our first variable

my_sum = 1+2

Notice that the value of this variable has been printed to the screen. If you want to suppress this effect, end the expression with a semi-colon

my_sum = 1+2;

There are a number of variable naming conventions that you will want to follow.

• Pick a name that indicates what is stored in that variable. It is much clearer to use the name TotalCost rather than x.

• Don't use names that are too long or too short. While the variable a is uninformative, the variable Tesing_raster2DCinput2DCalg2evaluatePressure_LIF2_input_Flatten is probably too long.

• Pick a convention for multi-word variable: TotalCost (Camel Case) or total_cost (Snake case)

• Pick a convention for capitalization.

• Don't use a built-in function as a variable name. Basically, take some time to think up your personal naming convention and stick with it.

MATLAB has some other restrictions on variable names. For example, a variable must start with a letter, is case sensitive, cannot include a space, and can be no longer than a fixed maximum length. To find this maximum length, enter the command

namelengthmax

You should also avoid using any of the special characters (as listed above). I.e. Don't use n!, but rather use n_factorial. This can lead to issues when using i as an index, so be careful. For better or for worse, the built in constants can be changed, so be extra careful.

pi
pi=14;
pi

Built in functions
MATLAB has a plethora of built in functions. A non-exhaustive list can be found at http://www.mathworks.com/help/matlab/functionlist.html. Some of the more common functions can be found using

help elfun

which will list most of the elementary mathematical functions. A handy cheat sheet can be found at https://engineering.purdue.edu/~bethel/MATLAB_Commands.pdf.

Script Files

You'll soon get tired of using the command window to enter all your commands, so it's good practice to use the editor instead. The editor can be brought up by typing the command

ead

into the command window. When the window opens, type the following commands into the window (It's ok if you don't know what all the commands mean)

```matlab
x=linspace(0,2*pi,200);
y=sin(x);
plot(x,y)
axis tight
xlabel('x-axis')
ylabel('y-axis')
title('The graph of y=sin(x).')
```

Save the file in your directory as sineplot.m The extension .m tells Matlab that you can run this file as a script. There are a number of different ways to run a script. The first is to make sure that the file is in your directory (using addpath or cd) and then typing

sineplot

Or else, you can use F5, or the editor+run option on the top menu. For quick prototyping, it's convenient to use two percent signs to break your code up into individual cells. The cells can then be evaluated individually by placing your cursor in that cell, and then pressing ctrl+enter. You can abort the evaluation of a command by pressing ctrl+C.

ICE

1. Create a variable Speed_MPH to store a speed in miles per hour (MPH). Convert this to kilometers per hour (KPH) by using the conversion formula $KPH = 1.60934 \times MPH$.

2. Create a variable Temp_F to store a temperature in Fahrenheit (F). Convert this to Celsius (C) by using the conversion formula $C = \frac{5}{9} (F - 32)$.

3. Wind chill factor (WCF) measures how cold it feels when the air temperature is $T$ (in Celsius) and the wind speed is $W$ (in kilometers per hour). The expression is given by $WCF = 13.12 + 0.6215T - 11.37V^{0.16} + 0.3965T \times V^{0.16}$. Calculate the wind chill factor when the temperature is 50F and the wind speed is 20mph.

4. Find a format option that would result in the command $1/2+1/3$ evaluating to 5/6.

5. Find MATLAB expressions for the following $\sqrt{2}, 3^{1.2}, \tan(\pi)$.

6. There is no built-in constant for $e$ (Euler's constant). How can this value be obtained in MATLAB?
7. What would happen if you accidentally assigned the value of 2 to sin (i.e. sin=2)? Could you calculate \( \sin(\pi) \)? How could you fix this problem?

8. Say you have a variable \( \text{iterator} \). Let \( \text{iterator}=7 \). How could you increment the value of this iterator by one unit? Is there only one way to do this?

9. Random number generation is quite useful, especially when writing a program and the data is not yet available. Programmatic random number generation is not truly random, but pseudo-random: they take an input (called a seed) and generate a new number. Given the same random seed the same number will be generated. However, the numbers that are produced are “random” in the sense that each number is (almost) equally likely. Use the function \( \text{rand} \) to generate a random number. Now close out MATLAB, open it up again, enter \( \text{rand} \), and record the number. Repeat this process. Did you notice anything? If you want a truly different number to be generated each time, you need to use a different seed. The command \( \text{rng('shuffle')} \) will use a new seed based on the current time. Open the help menu for \( \text{rng} \) and explore the different kinds of random number generators.

Vectors

As its name suggests, MATLAB is designed to primarily be a matrix-based numerical calculation platform. Hence, most of its algorithms for linear algebra (and matrix-based calculations in general) are highly optimized. Today we will explore how to define and work with vectors and matrices.

There are two kinds of vectors in MATLAB: row vectors and column vectors. Row vectors are lists of numbers separated by either commas or spaces. They have dimensions \( 1 \times n \). Here are a few examples

\[
\begin{align*}
v1 &= \begin{bmatrix} 1 & 3 & \pi \end{bmatrix} \\
v2 &= \begin{bmatrix} 1, 2, 3, 4 \end{bmatrix}
\end{align*}
\]

Column vectors, on the other hand, have dimensions \( n \times 1 \) and are separated by semi-colons

\[
\begin{align*}
v3 &= \begin{bmatrix} 1; 2; 3 \end{bmatrix} \\
v4 &= \begin{bmatrix} i; 2i; 1+i \end{bmatrix}
\end{align*}
\]

It’s very important to keep clear the differences between row vectors and column vectors as a very common MATLAB issue is to accidentally use a row vector where you meant to use a column one (and vice-versa). To determine what kind of vector you have, you can use the \( \text{size()} \) command.

\[
\begin{align*}
\text{size}(v1) \\
\text{size}(v2) \\
\text{size}(v3)
\end{align*}
\]

Additionally, you can use the \( \text{whos} \) command.

\text{whos v1}

The size command lets you store the dimensions of the vector for later usage, while the \( \text{whos} \) command gives you a bit more information.

\[
\begin{align*}
[m, n] &= \text{size}(v1) \\
m & \text{ number of rows} \\
n & \text{ number of columns}
\end{align*}
\]

The command \( \text{length()} \) gives the size of the largest dimensions of a vector (or matrix). Be careful with this unless you are absolutely sure which dimension (the rows or the columns) is going to be larger.

\[
\begin{align*}
v1 &= \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}; \\
v2 &= \begin{bmatrix} 1; 2; 3 \end{bmatrix}; \\
\text{length}(v1)
\end{align*}
\]
length(v2)

Now say we wanted to perform some basic operations on these vectors. For example, say we wanted to add two of the vectors together.

v5=[sqrt(4) 2^2 1];
v1 + v5

Note what happens when we try to do the following

v1 + v2

or

v1 + v3

The error **Matrix dimensions must agree** in this case is an indication that you are trying to add vectors of different dimensions.

Other arithmetic operations can be performed on vectors

v1 - v5
2*v1

However, we have to be careful when using division, for example, note what happens when you enter

[2 4 6]/[2 2 2]

Why didn't we get the answer [1 2 3]? The operation / is a very special MATLAB operator which we will cover later. If you want to perform an element-wise operation on a pair of vectors, use the following notation

[2 4 6]./[2 2 2]

The same is true of the * and ^ operators

[2 4 6].*[2 2 2]
[2 4 5].^[2 2 2]

To be consistent, MATLAB *should* use the operations .+ and .- to designate vector addition and subtraction, but they chose not to for stylistic reasons.

To access individual elements of a vector, round braces are used.

v = [18 -1 1/2 37];
v(1)
v(2)

This can be used to modify individual elements of a vector. For example if you write

v = [1 1 1 1 1 1];

You can change individual elements using assignment

v(2) = 13;
v
v(5) = -1;
v

This permanently changes the corresponding element. Just be sure that you specify the correct number of elements. In this case v(4) is a single element, so while you can use
v(4) = 77;

you cannot do
v(4) = [1 2 3];

MATLAB will automatically make your vector longer if necessary when changing elements in this way
v = [1 2 3];
\texttt{v(4) = 18;}
\texttt{v}

You can also build vectors up from existing ones using concatenation.
\texttt{u = [1 2 3];}
\texttt{v = [4 5 6];}
\texttt{w = [u v]}

Once again, be careful with the dimensions of the vectors.
\texttt{u = [1 2 3];}
\texttt{v = [4; 5; 6];}
\texttt{w = [u v] %Won't work}

To convert a row vector into a column vector (and the other way around too) you can use the \textit{transpose} command. There is a bit of subtlety here as there are two different kinds of transposes: \textit{transpose} and \textit{conjugate transpose}. Transpose simply interchanges the dimensions of a vector and is denoted \texttt{'.}'
\v = [1 2 3]
\v^T = \v.'
\texttt{size(v)}
\texttt{size(v^T)}

The conjugate transpose ' is used to interchange the dimensions of a vector, as well as take the conjugate of any imaginary entries
\v = [1 2 i]
\v^c = \v'^* \texttt{ % Note the i has been replaced with -i}
\texttt{size(v)}
\texttt{size(v^c)}

It is \textit{very} common for programmers to use ' when they actually mean .', but thankfully this is only a concern if you are going to be using imaginary numbers. Just be aware of this.

\textbf{Colon Notation}

To aid in the rapid creation of vectors, MATLAB includes special notation called \textit{colon notation}. It can create vectors, arrays of subscripts, and iterators. The notation is \textit{start:increment:end} with the \textit{increment} part being optional. This is basically like creating a table of numbers.
\texttt{1:10}
\texttt{1:2:10 %Note that the end, 10, isn't included due to the increment size. -10:0}

As mentioned, this can be used to access entire portions of a vector (called: using an array of subscripts).
\texttt{v = [1 -1 2 -2 3 -3 4 -4 5 -5];}
\texttt{v(1:2)}
\texttt{v(1:2:length(v))}
There is a shorthand for the last two commands: if you are using colon notation to access elements of a vector, you can use the \textit{end} command to denote the end of the vector.

\begin{verbatim}
v(1:2:end)
v(2:2:end)
v(end-2:end)
\end{verbatim}

Note that all of MATLAB's vector notation is \textit{1}-based. So the first element of a vector is \textit{1} (as opposed to \textit{0} as in some other programming languages (such as Python)).

The colon notation and transpose can be combined to create column vectors as well.

\begin{verbatim}
(1:.1:2)'
\end{verbatim}

There are two different kinds of subscripting in MATLAB: subscripts and logical indexing. We have already seen subscripts:

\begin{verbatim}
v = [pi exp(1) 1/2 -9 0];
\end{verbatim}

To access the first and third element using subscripts we can use

\begin{verbatim}
v([1 3])
\end{verbatim}

Another way to do this is to use \textit{logical indexing}. That is, treating \textit{1} as \textit{include} and \textit{0} as \textit{exclude} we can access the first and third elements of \textit{v} in the following way too

\begin{verbatim}
v(logical([1 0 1 0 0]))
\end{verbatim}

This can actually be very handy in selecting certain kinds of elements from a vector. For example, say we wanted all the strictly positive elements from \textit{v}

\begin{verbatim}
v>0
\end{verbatim}

This command returns a logical vector indicating which elements are strictly positive. To retrieve these, we can use this to index our vector \textit{v}

\begin{verbatim}
v(v>0)
\end{verbatim}

The command \textit{find} will return the subscripts of the \textit{1}'s in a logical vector

\begin{verbatim}
find(v>0)
\end{verbatim}

\textbf{Vectorization}

One of the most powerful aspects of MATLAB is its ability to \textit{vectorize} calculations. That is, given a vector of numbers, MATLAB can perform the same operation on each one in a very efficient fashion. For example if

\begin{verbatim}
v = [1 2 3];
\end{verbatim}

then

\begin{verbatim}
sin(v)
\end{verbatim}

applies \textit{sin} to each element of \textit{v}. In many cases, MATLAB uses special optimization "tricks" to make this kind of operation incredibly quick. Say we wanted to compute \textit{sin} of the first $10^8$ integers. This could be accomplished by taking \textit{sin} of \textit{1}, then \textit{sin} of \textit{2} etc:

\begin{verbatim}
tic; %starts a "stopwatch" and will tell us how long the calculation takes
\end{verbatim}
for i=1:10^7 %starting at 1, going up to 10^7
    sin(i); %calculate sin(i)
end;
loop_timing = toc %stop the "stopwatch"

However, when we vectorize this, you can notice it is much faster

tic;
sin(1:10^7);
vectorize_timing = toc

Note how many times faster it is

loop_timing / vectorize_timing

It takes practice and experience to recognize when you can vectorize code, but it is typically the number
1 way to speed up your MATLAB code.

Summation

Lastly, the `sum()` command will add up all the elements in a given vector:

```
x=[1 2 3 4];
sum(x)
```

This function is remarkably efficient, and you can easily sum up a few million random numbers in this

```
x=rand(1,10000000);
sum(x);
```

**ICE**

1. How would you use the colon operator to generate the following vector: [9 7 5 3 1]?

2. Use the `length()` command to find the length of the following vectors: 
   `x=1:0.01:5, y=2:0.005:3, z=(100:0.05:200)'`

3. Use Matlab's `sum()` command and vector colon notation to construct the sum of the first 1000 natural
   numbers.

4. Use Matlab's `sum()` command and vector colon notation to find the sum of the fist 1,000 odd positive
   integers.

5. Let `u = [1,2,3]` and `v = [4,5,6]`. Use Matlab to compute `(u + v)'` and `u' + v'`. Compare the
   results. Vary `u` and `v`. What have you observe about these two expressions?

6. Let `v = [1,3,5,7], w = [5,2,5,4]`. Think about the output of each of the following commands;
   write down by hand what you think the answer will be. Next, run the commands in Matlab and check
   your results. Did they match? `v > w`, `v >= w`, `v <= w`, `v < w`, `v == w`, `v ~= w`.

7. The empty vector is denoted `[ ]`. This can be used to delete elements of a vector. Create a vector `v`
   of length 5 and delete the first element by using `v(1)=[]`. What happens when you evaluate `v(1)`?
   What about `v(5)`?

8. Read the documentation for the `primes()` command. Use that command to find the sum of the first 100
   primes. Create a histogram of the prime gaps (differences between successive primes) for all primes
   less than or equal to 10,000,000.
9. Another handy function is `linspace(x,y,n)` which gives a row vector with \( n \) elements linearly (evenly) spaced between \( x \) and \( y \). Using this function, make a row vector of 100 evenly spaced elements between 1 and 0.

10. Using vectorization, estimate the limit \( \lim_{\epsilon \to 0} \frac{\sin(\epsilon)}{\epsilon} \).

11. Generate a list of 10,000 random numbers uniformly between 0 and 1. Approximate the average value of \( \frac{1}{n} \) on \([0,1]\). Does this answer make sense? (Hint: use Calculus).

## Plotting

The plotting capabilities of MATLAB are designed to make their programmatic generation relatively easy. However, this means that sometimes plots take a little fiddling before they come out exactly how you want them to. The MATLAB `plot()` command can be called in a couple of different ways. In particular, `plot(x,y)` creates a 2D line plot with \( x \) specifying the x-coordinates of the data contained in \( y \). For example

```matlab
x = [1 2 3 4];
y = [2 4 6 8];
plot(x,y)
```

Note that MATLAB automatically connects up the data points; this can be changed by using

```matlab
plot(x,y,'.') %The '.' tells MATLAB to treat the data as points
```

Here's an example of plotting a parabola

```matlab
x = linspace(-10,10,100);
y = x.^2;
plot(x,y)
```

If we were to omit the second argument to `plot(x,y)`, then MATLAB would treat the x-axis as \( 1:length(y) \)

```matlab
y=[2 4 6 8];
plot(y)
```

You may have noticed that each time we call `plot()` the previous plot we had is overwritten. Different figures can be created by using the `figure` command

```matlab
figure
plot(-10:.5:10,(-10:.5:10).^3)
figure
plot(0:.01:(2*pi),sin(0:.01:(2*pi)))
```

To combine two or more plots, use

```matlab
figure
x=0:.01:1;
plot(x,2*x+1)
hold on
plot(x,3*x+1)
```

If you want the plot to be different colors, use a single plot command. If you would like to label your axes (always a good idea), you can do this using `xlabel` and `ylabel`. The easiest way to do this is to call these functions immediately after using the `plot()` command.

```matlab
plot(x,2*x+1,x,3*x+1);
```
Let's look at a few more complicated examples of plots.

```matlab
meansQuikr = load('meansQuikr.mat'); % load in the data
meansQuikr = meansQuikr.meansQuikr; % get the correct variable
meansARKQuikr = load('meansARKQuikr.mat');
meansARKQuikr = meansARKQuikr.meansARKQuikr;
figure % create a new figure
plot(meansQuikr(1:5), '-b', 'LineWidth', 2); % dashed blue thicker plot
ylim([0,1]) % set the y-axis limits
hold on
plot(meansARKQuikr(1:5), '--g', 'LineWidth', 2);
ylim([0,1])
legend('Quikr', 'ARKQuikr') % set the x-tick labels
set(gca, 'XTickLabel', {'Phylum', 'Class', 'Order', 'Family', 'Genus'})
set(gca, 'XTick', [1 2 3 4 5 6]) % set the x ticks
xlabel('Taxonomic Rank')
ylabel('Mean VD Error')
ylim([0,1])
```

```matlab
myerror = load('error.mat') % load the data
myerror = myerror.error; % get the proper variable
[AX, H1, H2] = plotyy(1:20, myerror(1:20), 1:20, diff(myerror(1:21)));```
ICE

1. When explicitly specifying the x-coordinates of a plot, it is important to keep in mind the trade-off between speed and accuracy. Say you want to plot the function $\sin(3\pi x)$ for $0 < x < 1$. Use $x = \text{linspace}(0, 1, N)$ and $\text{plot}(x, \sin(3\pi x))$. Compare the results for various values of $N$.

2. Generate a list of the first 100 powers of 2. The following function selects the first digit of a number $x$:

$$\left\lfloor \frac{\log_{10}(x)}{\log_{10}(10)} \right\rfloor$$

Plot a histogram of the first digits of 1,000 powers of 2. Use this to estimate the probability that the first digit of a power of 2 is 1. What is the probability that the first digit is 2? and so on. This phenomena is referred to as Benford’s Law: For most numbers, the probability that the first digit is $d$ is given by $\log_{10} \left( 1 + \frac{1}{d} \right)$. Plot this function and compare this plot to the results we obtained. Interestingly, Benford’s law is periodically used to check for fraud on tax returns.

For the following 6 exercises, set $x = \text{linspace}(a, b, n)$ for the given values of $a$, $b$, and $n$. Calculate $y = f(x)$ for the given function $f(x)$. Plot with $\text{plot}(x, y, s)$ for the given formatting string $s$. Explain the result of the formatting string.

1. $f(x) = \sin(x)$, $a = 0$, $b = 4\pi$, $n = 24$, $s = 'r-'. $
Graph the given function on a domain that shows all the important features of the function (intercepts, extrema, etc.), or on the domain specified. Use enough points so that your plot appears smooth. Label the horizontal and vertical axis with Matlab's `xlabel()` and `ylabel()` commands. Include a `title()`.

1. \( y = x^2 - 2x - 3 \)

2. \( y = x^3 - 3x^2 - 28x + 60 \)

3. \( y = xe^{-x^2} \) \([\text{on } -5, 5] \)

4. \( y = \frac{1}{\sqrt{x^2}} \) \([\text{on } -10, 10] \)

5. \( y = (x^2 - 2x - 3)e^{-x^2} \) \([\text{on } -5, 5] \)

Matrices

Matrices are the bread and butter of MATLAB, and so as you might imagine, there are myriad matrix and matrix-related functions in MATLAB. Let's start with the very basics.

At its most basic, a matrix is a rectangular arrow of numbers with \( m \) rows and \( n \) columns. This is referred to as an \( m \times n \) matrix. There are a few different ways to enter a matrix into MATLAB, but most methods are "row based":

\[
A = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix}
\]

This is a 2x3 sized matrix. This can be verified by using the `size()` command

\[
[m, n] = \text{size}(A)
\]

Note that the `length()` command gives the maximum of \( m \) and \( n \) for the command \([m,n] = \text{size}(A)\).  

Here, \( m \) and \( n \) will store the number of rows and number of columns respectively of \( A \).

Just like with row vectors, the elements of a row of \( A \) can be separated with a comma instead of a whitespace

\[
A = [1, 2, 3; 4, 5, 6]
\]

The colon notation can be used to create matrices as well

\[
B = [1:5; 6:10; 11:2:20]
\]

Just be careful that you do not create a "jagged" matrix. Each row needs to have the same number of columns and conversely as well.
C = [1 2 3; 4 5] %This will lead to an error

Again, similarly to row/column vectors, the ' symbol will transpose a matrix.

A' %This is the conjugate transpose, but all the entries are real

size(A')

Indexing of matrices

Matlab has two different kinds of indexing for matrices: linear indexes and subscripts. Subscript notation is the typical notation most mathematicians are used to. For example, to refer to the element in the second row and third column of the matrix A (that is, $A_{2,3}$), we can use

A(2,3)

Colon notation can again be used to indicate elements of A. Here's how to get the first column

A(:,1)

Here's how to get the second row

A(2,:)

Here's how to get the first square submatrix of A

A(1:2,1:2)

And the second square submatrix of A

A(1:2,2:3)

To understand linear indexing, imagine that you were counting the individual elements of the matrix A column-wise. In this case, you would have the following

A(1,1) %First element
A(1,2) %Second element
A(2,1) %Third element
A(2,2) %Fourth element
A(3,1) %Fifth element (Not Milla Jovovich :) )
A(3,2) %Sixth element

Using linear indexing, we can refer to these very same elements using

A(1)
A(2)
A(3)
A(4)
A(5)
A(6)

Though it doesn't seem like it, linear indexing can actually be quite handy. For example, you can easily get all the elements in a matrix (in essence "flattening it out") by using the command

A(:)

Just make sure you don't ask for an element of a matrix that isn't there
Matrix Assignment

Since we now know how to access individual elements of a matrix, we can easily change them just like we did for vectors.

\[
A = \begin{bmatrix} 1 & 2 & 3; 4 & 5 & 6 \end{bmatrix};
\]
\[
A(1,1) = -7; \quad A(4) = 100;
\]

\[A\]

You can also change entire rows or columns via

\[
A = \begin{bmatrix} 1 & 2 & 3; 4 & 5 & 6 \end{bmatrix};
\]
\[
A(1,:) = [-100 \ -10 \ -1];
\]

\[A\]

Just be careful that the dimensions match between what you are trying to replace in the matrix and what you are replacing it to. I.e.

\[
\text{length}(A(1,:)) \neq \text{length}([-100 \ -10 \ -1])
\]

the sizes match, so it will work. However, we can't do

\[
A(:,1) = [10 \ 100 \ 1000]; \quad \%\text{Won't work}
\]

since

\[
\text{length}(A(:,1)) \neq \text{length}([10 \ 100 \ 1000])
\]

Special Matrices

MATLAB has a bunch of built-in ways to generate a variety of matrices of a given size. Let's run through a few of them

\[
m=3; \%\text{Number of rows}
\]
\[
n=4; \%\text{Number of columns}
\]
\[
A = \text{ones}(m,n) \%\text{A matrix of all ones}
\]
\[
B = \text{zeros}(m,n) \%\text{A matrix of all zeros}
\]

\%
\text{An identity matrix (or more precisely, a matrix with ones down the main diagonal)}
\]
\[
C = \text{eye}(m,n)
\]

Matlab has a few shorthands that can speed the creation of matrices. One of these includes creating a square matrix of dimensions \(n \times n\) when using the following commands

\[
D = \text{eye}(10);
\]
The command `diag` can be used to extract the diagonal of a matrix, as well as used to create a diagonal matrix. For example:

```matlab
E = [1 0 0; 0 2 0; 0 0 3];
```

Then to get the diagonal, we can use.

```matlab
Ediag = diag(E)
```

Now to create a diagonal matrix, say with diagonal entries 10, 100, and 1000, we could use

```matlab
Fdiag = [10 100 1000];
F = diag(Fdiag)
```

Note that the matrix is automatically of square of size `length(Fdiag)`.

**Element-wise matrix operations**

Just like with element-wise vector operations, we can do all the typical element-wise matrix operations.

```matlab
A = rand(2)
B = rand(2)
A + B % Element-wise addition
A - B % Element-wise subtraction
A .* B % Element-wise multiplication
A ./ B % Element-wise division
A .^ B % Element-wise power
```

Note that you can also do the above operations when the second argument to your binary operator is not a matrix, but a scalar.

```matlab
A = rand(2)
B = 5
A + B % Element-wise addition
A - B % Element-wise subtraction
A .* B % Element-wise multiplication
A ./ B % Element-wise division
A .^ B % Element-wise power
```

Again, watch out for dimensions mismatches

```matlab
A = rand(2);
B = rand(3);
A + B % This won't work.
```

Recalling the `sum()` command, you will notice that it operates on the *columns* of a matrix. That is, `sum(A)` will return a vector where each entry is the sum of the corresponding column in `A`.

```matlab
A=[1 2; 3 4];
sum(A)
```

You can change this to work on the rows by using the following command

```matlab
sum(A,2)
```
ICE

1. Create a 5x8 matrix of all zeros and store it in a variable. Now replace the second row in the matrix with all 4's and the first column with the entries [2 4 6 8 10].

2. Create a 100 x 100 matrix of random reals in the range from -10 to 10. Get the sign of every element. What percentage of the elements are positive?

In the following exercises, predict the output of the given command, then validate your response with the appropriate Matlab command. Use $A=magic(5)$.

1. $A > 14$
2. $A <= 12$
3. $(A > 3) \& (A <= 20)$
4. $(A < 5) \& (A >= 21)$
5. $\sim (A <= 6)$
6. $(A > 6) \& \sim (A > 8)$

Matrix/Vector and Matrix/Matrix Operations

(Insert short review of elementary linear algebra operations here)

Matrix/vector and matrix/matrix operations are extremely efficient in MATLAB, and a common programming technique to is reduce whatever problem you have at hand into such an operation. But first, we need to cover how these operations work in MATLAB.

First, we will cover the definition of the product of a matrix with a vector: $A \times x$. This is only defined for a column vector that is of the same length as the number of columns in the matrix. I.e. the following two quantities need to be the same

$$A = \text{rand}(10,12);$$
$$[m,n]=\text{size}(A);$$
$$x = \text{rand}(12,1);$$
$$n == \text{length}(x)$$

We imagine the matrix $A$ as being made up of $m$ row vectors stacked vertically, and then we take the element-wise product of one of these rows with $x$, then sum up this vector. Let's look at an example

$$A = \begin{bmatrix} 4 & 6 & 8 \\ 1 & -1 & 2 \end{bmatrix};$$
$$x = \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix};$$

The first entry of $A \times x$ will be $4*2 + 6*(-1) + 8*1 = 10$. The second entry will be $1*2 + (-1)*(-1) + 2*1 = 5$. Indeed, we have

$A \times x$

Once again, we need to be very careful that the dimensions match up.
Introduction to MATLAB

A = rand(10,12);
x = rand(10,1);
A*x %Won't work

A = rand(10,12);
x = rand(1,12);
A*x %Won't work

Matrix/Matrix products (A*B) are very similar, except that they view the matrix B as a bunch of column vectors sitting side by side. Here, the inner dimensions of A and B need to match

A = [1 2 3; 4 5 6];
B = [1 -1; 2 3; 0 1];
A*B

More on dimensions:

rand(10,11)*rand(11,12) %will work
rand(10,11)*rand(12,12) %Won't work
rand(1,5)*rand(5,1) %Will work

Systems of Linear Equations

Many applied mathematics problems can be reduced to (or at least approximated by) a system of linear equations. A system of linear equations can be expressed in terms of a matrix of coefficients A, a vector of constants b, and a vector of variables x. Then the system

\[ a_1x_1 + a_2x_2 + \ldots + a_nx_n = b_1 \\
\vdots \\
a_mx_1 + a_2x_2 + \ldots + a_nx_n = b_m \]

can be expressed compactly as \( Ax=b \). Every such system either has no solution, exactly one solution, or infinitely many solutions. When A is nonsingular (i.e. you can form it’s inverse \( A^{-1} \)) and square, then the system has a unique solution \( x = A^{-1}b \). While this is theoretically of much value, computationally it is inefficient at best, and at worst can result in quite significant numerical errors (and prohibitively long execution times). However, since solving systems of linear equations is such an important topic, there are myriad different numerical approaches that can be taken in a variety of situations. Thankfully, MATLAB does the thinking for you and will automatically choose with algorithm to use depending on what kind of input you choose. These algorithms are all contained in the innocuous looking \( \backslash \) operator.

Before we discuss what \( \backslash \) does, let’s see it in action.

A = [1 2 3; 4 5 6];
b = [2;8];
x=A\b % This is solving Ax=b for x

We can verify that this is the correct answer by testing

isequal(A*x,b)

Now the backslash operator is more properly called a polyalgorithm since the same notation actually encapsulates many different algorithms. We won’t be going through them all, but here is a flowchart showing how MATLAB chooses the algorithm for \( \backslash \)
ICE

\[ A = \begin{bmatrix} 3 & 3 \\ 2 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 1 \\ 2 & 3 \end{bmatrix}, \quad C = \begin{bmatrix} 3 & 1 \\ 5 & 5 \end{bmatrix} \]

Part 1. Given the matrices \( A, B, C \) use Matlab to verify each of the following properties. Note that 0 represents the zero matrix.

1. \( A + B = B + A \)
2. \( (A + B) + C = A + (B + C) \)
3. \( A + 0 = A \)
4. \( A + (-A) = 0 \)

Part 2. The fact that matrix multiplication is not commutative is a very important point to understand. Use the matrices \( A \) and \( B \) from above to show that none of the following properties is true.

1. \( (ab)^2 = a^2b^2 \)
2. \( (a + b)^2 = a^2 + 2ab + b^2 \)
3. \( (a + b)(a - b) = a^2 - b^2 \)

Part 3.

1. Enter the following coefficient matrix \( A \) and constant vector \( b \) and solve the system \( Ax = b \) using the following three methods: First, calculate the inverse of \( A \) using \( \text{inv}(A) \) and calculate \( x = A^{-1}b \). Next, calculate \( x = \text{A}ackslash b \). Finally, use \( \text{lsqlin()} \).

2. The trace of a matrix is the sum of the diagonal elements of a matrix. Come up with two different ways to compute the trace (one using a built-in MATLAB function, one using \( \text{diag} \) and \( \text{sum} \)).

3. How might we test whether or not a matrix is square?

Flow Control

In this section we will discuss the basic programming flow control constructs available in Matlab. Specifically, we will look at \textbf{if} statements, \textit{switch} statements, \textbf{for} loops, \textbf{while} loops, \textit{any}, \textit{all}, \textbf{break}, etc. These flow control constructs are foundational to programming; it’s amazing what can be accomplished with the proper combination of these!

\textbf{If}

The \textbf{if} statement evaluates a logical expression and executes a block of statements based on whether the logical expression evaluates to \textit{true} (that is, the logical \texttt{1}) or \textit{false} (that is, the logical \texttt{0}). The basic structure is as follows

\begin{verbatim}
if logical_expression
\end{verbatim}
statements
end

If the logical_expression is true, then the statements are evaluated. If it's not true, then the statements are simply skipped. For example

mynum = 4;
if mod(mynum,2) == 0
    fprintf('The number %d is even!\n',mynum)
end

Note what happens when you change mynum to 5.

Else

We can provide an alternative if the logical_expression evaluates to false by using an else command:

if logical_expression
    statements_true
else
    statements_false
end

Here, if logical_expressions is true, then statements_true is evaluated; if logical_expressions is false, then statements_false is evaluated. The advantage of using an if,else construct is that you can be sure that one of your statement blocks is guaranteed to be evaluated. for example:

mynum = 5;
if mod(mynum,2) ==0
    fprintf('The number %d is even!\n',mynum)
else
    fprintf('The number %d is odd!\n',mynum)
end

Elseif

Sometimes you need to add more than one alternative. For this, we have the following construct:

if logical_expression_1
    statements_1
elseif logical_expression_2
    statements_2
else
    statements_3
end

Here, Matlab first evaluates logical_expression_1, if it's true, it evaluates statements_1 and then skips the rest. If logical_expression_1 is false, it moves on to logical_expression_2, if logical_expression_2 is true, it evaluates statements_2 and skips the rest. If logical_expression_2 is false, then it evaluates statements_3. Let's look at an example

mynum = 17;
if mynum < 10
    fprintf('The number %d is less than 10!\n', mynum)
elseif mynum > 20
    fprintf('The number %d is greater than 20!\n', mynum)
else
    fprintf('The number %d is between 10 and 20\n', mynum)
end

Try different values of mynum and see what happens. You can enter as many elseif statements that you wish. Note that it's good programming practice to include an else block to catch any cases you might not have considered. For example

mynum = input('Enter an integer: ');
if  mynum == 1
    fprintf('Your number is equal to 1\n')
elseif  mynum == 2
    fprintf('Your number is equal to 2\n')
elseif  mynum == 3
    fprintf('Your number is equal to 3\n')
else
    fprintf('Your number is not equal to 1, 2, or 3\n')
end

However, this code can be more succinctly represented with a switch expression, which is what we will look at next.

Switch

A switch block conditionally executes one set of statements from several choices. The basic usage is

switch  switch_expression
    case  case_expression
        statements
    case  case_expression
        statements
    otherwise
        statements
end

You can include as many cases as you want. Typically, you use a switch statement when you need to pick one set of code based on the variable value, whereas an if statement is meant for a series of Boolean checks. Here's an example of the above if statement re-written as a switch statement:

mynum = input('Enter an integer: ');
switch  mynum
    case  1
        fprintf('Your number is equal to 1\n')
    case  2
        fprintf('Your number is equal to 2\n')
    case  3
        fprintf('Your number is equal to 3\n')
    otherwise
        fprintf('Your number is not equal to 1, 2, or 3\n')
end

Sometimes a switch statement reads much more clearly than an if statement with a bunch of elseif's. Here's a slightly more complicated switch statement:

a = input('Enter a number: ');
b = input('Enter another number: ');
fprintf('Enter one of the following choices
')
fprintf('1) Add a and b.
')
fprintf('2) Subtract a and b.
')
n = input('Enter your choice: ');
switch n
  case 1
    fprintf('The sum of %.2f and %.2f is %.2f.
', a, b, a+b)
  case 2
    fprintf('The difference of %.2f and %.2f is %.2f.
', a, b, a-b)
  otherwise
    fprintf('%d is an invalid choice, please try again
', n)
end

For loops

A *for* loop is a control structure that is designed to execute a block of statements a predetermined number of times. The general syntax of the *for* loop is:

```matlab
for index=start:increment:finish
  statements
end
```

As an example, let's print out the square of the first 10 natural numbers:

```matlab
for i=1:10
  fprintf('The square of %d is %d
', i, i^2)
end
```

Note that we don't need to use the colon notation for the index of a *for* loop. If *values* is an iterable element, then we can use

```matlab
for index = values
  statements
end
```

For example:

```matlab
vals = randi(100,10,1);
for i = vals
  fprintf('Your number is %d.
', i)
end
```

If you set the values of a *for* loop to a matrix, then Matlab will proceed along the values of the matrix column-wise:

```matlab
A = [1 2 3; 4 5 6; 7 8 9];
for i = A
  fprintf('The value of i is %d
', i)
end
```

*for* loops are extremely powerful, but be careful since Matlab doesn't execute *for* loops in the most efficient way possible. If you can ever vectorize your code (or reduce a loop to a linear algebra operation), then this will be much faster.

```matlab
mysum = 0;
vals = 1:10^7;
```
looptic = tic();
for i = vals
    mysum = mysum + i;
end
fprintf('For loop timing: %f seconds.
', toc(looptic))
vectorizetic = tic();
sum(vals);
fprintf('Vectorized timing: %f seconds.
', toc(vectorizetic))

**While loops**

*while* loops are very similar to *for* loops, but continue evaluating the statements until some expression no longer holds true. The basic syntax is

```matlab
while expression
    statements
end
```

For example

```matlab
i = 1;
N = 10;
mynums = zeros(1,N);
while i<=N
    mynums(i) = i;
    i = i + 1;
end
```

One must be very careful with *while* loops to not get stuck in an infinite loop. The following is an example of an infinite loop

```matlab
i = 1;
N = 10;
mynums = zeros(1,N);
while i<=N
    mynums(i) = i;
end 
%press Ctrl+C to exit this infinite loop.
```

*for* loops are good for when you know in advance the precise number of times the loop should iterate. *while* loops are handy for when you have no predetermined knowledge of how many times you want the loop to execute. Here's a perfect example of when you want a *while* loop instead of a *for* loop (this example comes from the [Collatz Conjecture](https://en.wikipedia.org/wiki/Collatz_conjecture)).

```matlab
n = input('Input a natural number: ');
iter = 0;
myseq = [n];
while n>1
    if mod(n,2) == 0
        n = n/2;
    else
        n = 3*n+1;
    end
    iter = iter + 1;
    myseq(end+1) = n;
end
```
fprintf('It took %d iterations to reach n=1\n',iter)

break

The *break* command terminates the execution of a *for* loop or *while* loop. This happens when the programmer wishes to terminate the loop after a certain state is achieved and pass control to the code that follows the end of the loop. For example, let's say you don't want the above Collatz Conjecture code to run for too long. We can break out of the loop if the number of iterations exceeds some predetermined amount.

```
n = input('Input a natural number: ');
iter = 0;
myseq = [n];
breakflag = 0;
while  n>1
    if  mod(n,2) == 0
        n = n/2;
    else
        n = 3*n+1;
    end
    iter = iter + 1;
    myseq(end+1) = n;
    if  iter >= 100
        breakflag = 1;
        break
    end
end
switch  breakflag
    case  0
        fprintf('It took %d iterations to reach n=1\n',iter)
    case  1
        fprintf('Failed to converge after 100 iterations\n')
end
```

Here, if the number of iterations surpasses 100, the *break* command tells Matlab to leave the *while* loop and proceed to the next set of statements (the *switch* statement in this case).

continue

The *continue* keyword passes control to the next iteration of a *for* or *while* loop, skipping the rest of the body of the *for* or *while* loop on that iteration. For example, consider the following code

```
i=1;
while  i<10
    if  mod(i,2) == 0
        i = i + 1;
        continue
    end
    fprintf('%d\n',i)
i = i + 1;
end
```

Here, if the index *i* is even, then *i* is incremented and the rest of the *while* loop body (that is, the part that prints out the result) is skipped. This is particularly useful if you want to skip over certain cases in your code (for example, don't process prime numbers, invalid input, null entries, NaN's etc.).

Nesting Loops
It's possible to nest one loop inside another! Let's look at an example: A Hilbert matrix is a square matrix with entries being the unit fractions \( H(i, j) = \frac{1}{i+j-1} \). Let's see how we can define our own Hilbert matrix

\[
N = 10; \\
H = zeros(N,N); \\
for \ i=1:N \\
    for \ j=1:N \\
        H(i,j)=1/(i+j-1); \\
    end \\
end
\]

Note that pre-allocating an array saves quite a bit of time in Matlab (this is because otherwise Matlab needs to make an entirely new copy of the array in memory). Try the following: Let \( N \) be a large number and compare the results of the above code to that where the line \( H = zeros(N,N) \) is removed. Which is faster?

**ICE**

1. The number of ways to choose \( k \) objects from a set of \( n \) objects is defined and calculated with the formula \( \binom{n}{k} = \frac{n!}{k!(n-k)!} \). Define a Pascal matrix \( P \) with the formula \( P(i,j) = \binom{i+j-2}{i-1} \). Use this definition to find a Pascal matrix of dimensions 4x4. Use Matlab's `pascal()` command to check your result.

2. Write a for loop that will output the cubes of the first 10 positive integers. Use `fprintf` to output the results, which should include the integer and its cube. Write a second program that uses a while loop to produce an identical result.

3. Write a single program that will count the number of divisors of each of the following integers: 20, 36, 84, and 96. Use `fprintf` to output each result in a form similar to \#The number of divisors of 12 is 6.#

4. Write a program that uses nested loops to produce Pythagorean Triples, positive integers \( a, b \) and \( c \) that satisfy \( a^2 + b^2 = c^2 \). Find all such triples such that \( 1 \leq a, b, c \leq 20 \) and use `fprintf` to produce nicely formatted results.

5. Write a program to perform the following task: Use Matlab to draw a circle of radius 1 centered at the origin and inscribed in a square having vertices (1, 1), (-1, 1), (-1,-1), and (1,-1). The ratio of the area of the circle to the area of the square is \( \pi : 4 \) or \( \frac{\pi}{4} \). Hence, if we were to throw darts at the square in a random fashion, the ratio of darts inside the circle to the number of darts thrown should be approximately equal to \( \frac{\pi}{4} \). Write a for loop that will plot 1000 randomly generated points inside the square. Use Matlab's `rand` command for this task. Each time a random point lands within the unit circle, increment a counter `hits`. When the for loop terminates, use `fprintf` to output the ratio darts that land inside the circle to the number of darts thrown. Calculate the relative error in approximating \( \frac{\pi}{4} \) with this ratio.

**Functions**

You've already encountered function notation in other mathematics courses \( f(x) = 2x^2 - 3 \). Matlab handles the definition of functions in a couple different ways: Anonymous functions and Function M-files.

**Anonymous functions**
The general syntax of an anonymous function is

funhandle = @(arglist) expr

Here expr is the body of the function; it consists of any single valid Matlab expression. The arglist should be a comma separated list of all input variables to be passed to the function. The funhandle is the function handle that stores the anonymous function. You can then evaluate your function using

funhandle(arg1, arg2, ...
   , argN)

For example, we could use

funhandle = @(x) 2*x.^2-3 %be careful to make your function "array smart"
funhandle(10)

You can then use this function just like any other matlab command. For example, let's apply this function to every element of the list 1:10

res = arrayfun(funhandle, 1:10); %equivalently funhandle(1:10)

We can define as many variables as we please

myfunc = @(x,y,z) x.^2+y.^2+z.^2;
myfunc(1,2,3)

You can include pre-defined variables in your function as well

A=1;
B=2;
C=3;
f=@(x,y,z)A*x+B*y+C*z;
f(-1,0,1)

However, if you change the value of A,B or C, then you will need to re-define the function handle

A=10;
f(-1,0,1)
f=@(x,y,z)A*x+B*y+C*z;
f(-1,0,1)

Lastly, you don't actually need an argument for the function to work

t = @(t) datestr(now);
t()

**Function M-files**

Anonymous function should be used very sparingly, and should have relatively small bodies. A function M-file, however, is the "meat and potatoes" of Matlab, allowing you to turn a script into a multi-purpose, reusable, honest-to-goodness program.

Like a script file, you write function M-files in the Matlab editor. The first line of a function M-file must contain the following syntax

    function [out1, out2, ..] = funname(in1,in2,..)
The keyword function must be the first word in the function m-file. This alerts the Matlab interpreter that the file contains a function, not a script. Next comes a comma delimited list of output arguments (surrounding brackets are required if more than one output variable is used), then an equal sign, followed by the function name. The function name funname must be string composed of letters, digits, and underscores. The first character of the name must be a letter. A valid length for the function name varies from machine to machine, but can be determined by the command namelengthmax.

namelengthmax

If you name your function funname, then you must save the file as funname.m. After the function name comes a comma delimited list of input variables, surrounded by required parentheses. Input arguments are passed by value and all variables in the body of the function are #local# to the function (that is, they aren't accessible outside this particular function).

Let's start with a very simple example. Open a new editor window and type

```matlab
function y = myfun(x)
    y = 2*x.^2-3;
end
```

Save this as a file called myfun.m If this file is in your path, you should be able to run it as

```matlab
myfun(10)
```

You are allowed to have more than one output

```matlab
function [area,circumference]=area_and_circumference(r)
    area=pi*r^2;
    circumference=2*pi*r;
end
```

Once again, save this as a file called area_and_circumference.m You can then evaluate it using

```matlab
radius = 12;
[A, C] = area_and_circumference(radius)
```

Documenting your function

The first contiguous block of comment files in a function m-file are displayed to the command window when a user enters help funname, where funname is the name of the function. You terminate this opening comment block with a blank line or an executable statement. After that, any later comments do not appear when you type help funname.

```matlab
function [area,circumference]=area_and_circumference(r)
%AREA_AND_CIRCUMFERENCE finds the area and circumference
% of a circle.
% [A,C]=AREA_AND_CIRCUMFERENCE(R) returns the area A and
% circumference C of a circle of radius R.
area=pi*r^2;
circumference=2*pi*r;
```

You should get in the habit immediately of documenting all of your functions. I guarantee that if you don't, after some time (maybe a few months, maybe a few years) you'll come back to your code, have no idea what it does, and then if there aren't any comments, spend the next few hours/days re-figuring it out.

You can then type
help area_and_circumference

To get information about this function.

Functions with no output

Sometimes you will want to define a function that has no output (for example, one that writes the results to a file instead of passing them back to Matlab). This is easy to implement, simply leave the `out1, out2, ...` part of your function off. For example

```matlab
function write_to_file(input_vector, output_file)
    fid = fopen(output_file, 'w');
    for i=1:length(input_vector)
        fprintf(fid, '%f
', input_vector(i))
    end
    fclose(fid);
end
```

ICE

1. Write an anonymous function to emulate the function $h(u, v) = \cos(u) \cos(v)$. Evaluate the value of $h(1, 2)$.

2. Consider the anonymous function $f(x) = x \cdot \exp(-2\cdot x^2)$; Write a for loop that will plot this family of functions on the interval $[-1, 1]$ for $C$ in $\{-1, -0.8, -0.6, \ldots 1\}$.

3. Write a function M-file that will emulate the function $f(x) = xe^{-C^2x^2}$ an will allow you to pass $C$ as a parameter. Save this to a file and then write a for loop using this function on interval $[-1, 1]$ for $C$ in $\{-1, -0.8, -0.6, \ldots 1\}$.

4. Euclid proved that there are an infinite number of primes. He also believed that there was an infinite number of twin primes, primes that differ by 2 (like 5 and 7 or 11 and 13), but no one has been able to prove this conjecture for the past ~2300 years. Write a function that will produce all twin primes between two inputs, integers a and b.

5. Write a function called `mymax` that will take a column vector of numbers as input and find the largest entry. The routine should spit out two numbers, the largest entry `val` and its position `pos` in the vector. Test the speed of your routine with the following commands:

   ```matlab
   test = rand(1000000,1); tic; [pos, val] = mymax(test); toc.
   ```

   Compare the speed of your maximum routine with Matlab's `tic; [pos, val] = max(test); toc.`

Variable Scope

Now that we know the general structure and use of a function, we’ll now look at the scope of a variable (which is loosely defined as the range of functions that have access to that variable in order to modify or access its value).

Base workspace

Matlab stores variables in parts of memory called workspaces. The base workspace is the part of memory that is used when you are entering commands at the Matlab prompt in the command window. You can clear all variables from your base workspace by entering the command
clear

at the Matlab prompt in the command window.

The command *whos* will list all the variables in the current workspace.

*whos*  

To get more information about a specific variable, we can use

```matlab
M = magic(10);
whos M
```

**Scripts and the base workspace**

When you execute a script file from the Matlab prompt in the command window, the script uses the base workspace. It can access any variables that are present in the base workspace, and any variables created by the script remain in the base workspace when the script finishes. As an example, open the Matlab editor and enter the following lines:

```matlab
x = 1:5
s = sum(x)
p = prod(x)
y = y + 1
```

Save the script as *sumprod.m* and return to the Matlab command window. Type the command

```matlab
y = 1;
```

Now execute the script by typing its name at the command prompt. Now examine the state of the base workspace by typing the *whos* command at the command prompt.

*whos*  

Note that the variables *x*, *s*, and *p* were created in the script and are now present in the base workspace. Note also that the value of *y* was incremented by one. Hence a script both has access to as well as the ability to change the base workspace variables.

You must be **VERY** careful about this, since it is very easy to make a script that will behave very differently depending on what variables are present within your base workspace. It's good practice to either use *clear* before running a script, or, better yet, re-write the script as a function. We'll see why this has an advantage next.

**Function Workspaces**

Function do not use the base workspace. Each function has its own workspace where it stores its variables, and this is kept separate from the base workspace.

For example, say you make the following very simple function, and call it simple.m

```matlab
function simple
v=10;
fprintf('The value is: %d',v)
```

Then running this function using

```matlab
simple
```
will output the message "The value is: 10". However, if you try to check the value of \( v \) in your workspace using

\[
\text{whos } v
\]

nothing will get returned since the variable \( v \) doesn't exist on your base workspace.

One important feature of Matlab is since it cordons off variables for functions, to allow a function to access a variable in your workspace, it has to duplicate the variable (and its associated data) before allowing your function to access it. This is called "passing a value" (rather than "passing a reference". Keep this in mind if your variables are storing data that takes a large amount of memory.

Global variables are one way to allow a function to access variables in the base workspace (though one must be very careful about unintended consequences of this). Globabl variables are declared using

\[
\text{global } v
\]

Then if we assign the value of

\[
\text{v = 0}
\]

and then run the function

\[
\text{simple}
\]

we will see that the value of the variable has indeed changed!

\[
v
\]

ICE

1. Clear the workspace by entering the command `clear all` at the Matlab prompt in the command window. Type \texttt{whos} to examine the base workspace and insure that it is empty. Open the editor, enter the following lines, then save the file as `VariableScope.m`. Execute the cell in cell-enabled mode.

\[
\text{%Exercise #1}\\
\text{clc}\\
P=1000;\\
r=0.06;\\
t=10;\\
I=P*r*t;\\
\text{fprintf('Simple interest gained is %.2f.\n',I)}
\]

Examine the base workspace by entering `whos` at the command window prompt. Explain the output (of the `whos` command, not of the script itself).

2. Clear the workspace by entering the command `clear all` at the Matlab prompt in the command window. Type `whos` to examine the base workspace and insure

\[
\text{%that it is empty. At the command prompt, enter and execute the following}\\
\text{%command.}\\
P=1000; \text{~} r=0.06; \text{~} t=10;\\
\]

Examine the workspace using `whos`. Now add the following cell to the file `VariableScope.m`

\[
\text{clc}
\]
I=P*r*t;
fprintf('Simple interest gained is %.2f \n', I)

State the output of the script. Use who to examine the base workspace variables. Anything new? Explain.

3. Clear the workspace by entering the command clear all at the Matlab prompt in the command window. Type who to examine the base workspace and ensure that it is empty. At the command prompt, enter and execute the following command.

P=1000; r=0.06; t=10;

Examine the workspace with who. Now open the editor, enter the following lines, and save the function as the M-file simpleInterest.m

```matlab
function I=simpleInterest
I=P*r*t;
fprintf('Simple interest gained is %.2f \n', I)
```

Add the following cell to VariableScope.m

```matlab
clc
I=simpleInterest1;
fprintf('Simple interest gained is %.2f \n', I)
```

Evaluate the cell and the state what happens and why.

## File I/O

Direct input of data from the keyboard becomes impractical when the data being analyzed is of any reasonable size (or is being analyzed repeatedly). In these situations, input and output is preferably accomplished via data files. We have already used the commands load and save to load *.mat files, but Matlab is able to interact with much more diverse kinds of data.

When data is written to or read from a file, it is very important that the correct data format is used. The data format is the key to interpreting the contents of a file and must be known in order to correctly interpret the data in an input file. There are a number of different file types we will be primarily concerned with: formatted text files, and binary files.

### Formatted Text Files

First, let's see how we can write text to a data file. Let's say you are recording the temperature on various days of the week. So your data might look like

```matlab
months = {'January', 'February', 'March'};
days = [1, 8, 15, 22, 29];
temps = 10*rand(1, length(days)*length(months)) + 32;
```

Our goal is to output this in a flat text file using the following format: In the first column, list the month, in the second column, list the day, in the third column, list the temperature. So it might look like:

January 1 37.1231
January 8 31.51

etc.

To write a file, we first need to open that file up. This is done using the fopen command:
fid = fopen('/path/to/file/temperatures.txt','w');

This creates a file temperatures.txt and prepares it for writing (erasing any preexisting contents). The variable \texttt{fid} is a file identifier telling Matlab which file has been opened.

Let's write our temperature data to this file:

\begin{verbatim}
for month_index = 1:length(months)
    for day_index = 1:length(days)
        fprintf(fid, '%s	%d	%f
', months{month_index}, ...
                days(day_index), temps((month_index-1)+day_index))
    end
end
\end{verbatim}

If you try to open this file in your OS, it will show as a blank file since Matlab has placed a lock on the file (as it can still write contents to it). You can indicate that you are finished writing to the file by using

\begin{verbatim}
fclose(fid);
\end{verbatim}

We can now reverse the whole procedure by indicating that we want to read the file, and using \texttt{fscanf}

\begin{verbatim}
clear
fid = fopen('/path/to/file/temperatures.txt','r');
Results = textscan(fid,'%s\t%d\t%f\n');
fclose(fid);
months = Results{1};
days = Results{2};
temps = Results{3};
\end{verbatim}

One annoying fact is that Matlab will by default split strings on whitespace characters (so read in the string 'Test String' as {'Test', 'String'}). This can wreck havoc when reading in your data, but can be avoided by using:

\begin{verbatim}
textscan(fid,FormatString,'Delimiter','
')
\end{verbatim}

If you have a matrix written in a text file (so too many columns to make a long format string), you can use \texttt{dlmread} and \texttt{dlmwrite}. Let's write a large matrix to a text file, and then read it back in:

\begin{verbatim}
rand_nums = rand(10^3,10^3);
dlmwrite('rand_nums.csv',rand_nums) \texttt{ %Default delimiter is ','}
\end{verbatim}

Now read it back in

\begin{verbatim}
clear
rand_nums = dlmread('rand_nums.csv');
\end{verbatim}

\textbf{Binary Files}

There are a number of binary file types, and while these are typically not human readable, they have the advantage of being very efficient to read from/write to, as well as typically take less hard drive space. Here, we will look at one such binary format, called HDF5.

HDF5 is a hierarchical data format designed to store \textbf{massive} amounts of numeric data. It was originally designed for use in supercomputing centers, and currently has a very efficient and robust C library implementing its functionalities. We will look at how to use Matlab to call these functions.
You can think of each HDF5 file as its own "hard drive". In fact, you initialize an HDF5 file by giving it a "drive name". Here's an example:

\[
[num\_rows, num\_cols] = \text{size}(\text{rand\_nums});
\]

\[
\text{if exist('test\_file.h5')}
\]
\[
\quad \text{delete('test\_file.h5')}
\]
\[
\text{end}
\]
\[
\text{h5create('test\_file.h5','/drive\_name', [num\_rows num\_cols]);}
\]

This creates an empty HDF5 file of size \(10^3 \times 10^3\). Let's now write a bunch of random numbers to it

\[
\text{h5write('test\_file.h5','/drive\_name', rand\_nums)}
\]

We can get some information about this file by using

\[
\text{info = h5info('test\_file.h5');}
\]
\[
\text{info.Filename}
\]
\[
\text{info.Datasets.Dataspace}
\]

Reading the data back in consists of:

\[
\text{rand\_nums = h5read('test\_file.h5','/drive\_name');}
\]

Try timing \textit{h5read} versus \textit{dlmread} to see the performance advantage of HDF5. Furthermore, HDF5 allows for compression. Let's create another data set inside of test_file.h5 and compress it.

\[
\text{h5create('test\_file.h5','/compressed\_drive', [10^3 10^3], 'Datatype',...}
\]
\[
\quad \text{'int32', 'ChunkSize', [10 10^3], 'Deflate', 7)}
\]

The 'Datatype' is an optional command telling what kind of data will go in there, while 'ChunkSize' tells the HDF5 library how much of the matrix to read/write in any given read/write operation. The 'Deflate' command specifies the level of compression. Let's now put some data in this dataset.

\[
\text{my\_ints = int32(reshape(1:10^6,10^3,10^3));}
\]
\[
\text{h5write('test\_file.h5','/compressed\_drive', my\_ints)}
\]

Note this took longer to write, but if we were to write this to a CSV file, note the differences in file size:

\[
\text{dlmwrite('rand\_ints.csv',my\_ints);}
\]

There are many, many more features of HDF5 files that we won't go into here. One last advantage of using HDF5 files is that most programming languages include ways to easily read/write these kinds of files.

**ICE**

1. Study the available information on Mathworks.com regarding the \texttt{fscanf/scanf} and \texttt{fprintf/fprintf}. What's the meaning of the format string \texttt{%3d\n}?

2. Create a \(10^4 \times 3\) matrix where the rows correspond to \(1:10^4\) and the columns correspond to the functions \(\sin(x), \log(x), \text{and } x^2\). Let the \((i,j)\) entry be the value of the \(j\)th function evaluated on \(i\). Save this matrix as a *.mat file (use \texttt{save}), a comma separated file (use \texttt{dlmwrite}), text file (use \texttt{fprintf}) as an Excel file (use \texttt{xlswrite}), and as a compressed HDF5 file (use \texttt{h5write}). Include these files in the dropbox folder when you complete this homework assignment.

a) Which format took the longest/shortest amount of time to write?
b) Which format used the least/most amount of disk space?

c) Which format took the longest/shortest amount of time to read in?

Parallel Computing

As more CPU cores are squeezed into a single processor (at the time of writing, you could buy a single CPU that has 16 cores on it from Newegg.com), and as the amount of data available to analyze continues to grow (see: Big Data), parallel processing has become an integral part of scientific data analysis. Large computational clusters consisting of thousands of cores are commonly used at research institutions around the world. Here at OSU for example, we have the CGRB which currently has 2,432 CPUs spread over 76 different servers. How can we take advantage of all this processing power? Matlab has a very user-friendly implementation of parallel programming that we will review here (introduced in version 2008a).

Matlabpool

To start a parallel session of matlab, open Matlab as usual. Next, type the following into the command window:

`matlabpool open`

This will automatically spawn independent instances of matlab, one on each physical processor.

Parallel computing is most efficient when the computational job to be accomplished can be divided into as many tasks as processors, the pieces of data sent to the individual processors, the computation performed, and then afterwards recombined. It is less efficient (sometimes horribly so) if the processors need to talk to each other in order to complete the task.

Discuss parallel programming concepts, independence, etc.

Let's look at the most commonly used parallel programming tool in Matlab: `parfor`

`parfor` has exactly the same syntax as the usual `for` loop, but matlab automatically splits the job up to each individual processor. Let's see how this works in action and compare it to the serial version. Let's factor all the integers less than $10^5$.

```matlab
N=10^5;
tic;
for i=1:N
    factor(i);
end
fprintf('Serial version timing: %f seconds\n',toc)
```

```matlab
tic;
parfor i=1:N
    factor(i);
end
fprintf('Parallel version timing: %f seconds\n',toc)
```

To close the worker instances of matlab, use

`matlabpool close`
Parfor variable

To minimize the amount of time spent in the very slow act of communicating between processors or accessing memory, certain types of variables are classified in different ways in Matlab. Let's look at an example

```matlab
a = 0;
z = 0;
c = pi;
r = rand(1,10);
parfor i = 1:10
    a = i; % a is a temporary variable, i is the loop variable
    z = z + i; %z is a reduction variable
    b(i) = r(i); % b(i) is a sliced output variable, r(i)
        % is a sliced input variable
    if i <= c % c is a broadcast variable
        d = 2*a;
    end
end
```

A loop variable serves as a loop index for arrays. Modifying the loop variable is not allowed. A temporary variable is a "local" variable that might be different on each worker. It is any variable that is the target of a direct, non-indexed assignment. A sliced variable is one whose value can be broken up into segments, and this small segments sent to each individual worker. A reduction variable accumulates a value that depends on all the iterations together, but is independent of the iteration order. Matlab CAN parallelize a loop containing a reduction variable. However, Matlab cannot parallelize a loop variable whose value depends on the iteration order.

Here's an example of a loop that cannot be parallelized

```matlab
vals = rand(1,10);
for i=1:10
    vals(i) = vals(max([1,i-1]));
    % Note that vals(i) depends on i-1, so matlab can't split up the task
    % into independant processes.
end
```

Here's an example of a loop with an reduction variable, a variable that depends on all the iterations, but not the order

```matlab
total = 0;
vals = rand(1,10);
parfor i=1:10
    total = total + vals(i);
end
```

Slicing

Slicing is an important concept as it allows Matlab to break an array into smaller pieces and let each worker operate on just these pieces. Unfortunately, it's a bit complicated to define a sliced variable. You must make sure that:

1. The first level of indexing is either parentheses or curly braces

2. Within the first level indexing, the list of indices is the same for all occurrences of a given variable.
3. Within the list of indices for the variable, exactly one index involves the loop variable.

4. In assigning to a sliced variable, the right hand side of the assignment is not deletion (i.e. not assigning to []).

Here’s an example of a sliced variable

```matlab
N = 1000;
A = rand(N);
total = 0;
tic
parfor i=1:N-1
    total = total + sum(A(i,:));
end
toc
```

If you don't slice the variable `A`, then Matlab needs to make an entirely new copy of it in memory, one for each worker.

Let's look now at an example of what happens when you don't slice a variable.

```matlab
tic
N = 10000;
A = rand(N);
total = 0;
parfor i=1:N-1
    total = total + A(i,i+1);
end
toc
```

Note that we are doing less work than before (only summing on a diagonal of A, not the whole thing), yet this takes significantly longer. Open up Windows Task Manager and increase the size of N to see why this is happening. Note in particular the memory usage in both cases.

**SPMD**

SPMD stands for “Single program multiple data” and is useful when you want to do the same program on multiple data sets. Here is a very simple example

```matlab
spmd
% build random matrices in parallel
mat = rand(labindex + 2);
end
for ii=1:length(mat)
    % plot each matrix
    figure
    imagesc(mat{ii});
end
```

The resulting composite object gives the value of `mat` on each processor.

This is particularly handy if you have multiple data files (or one large data set split into multiple files) and want to perform the same analysis on all of them. Here’s an example

```matlab
num_procs = matlabpool('size'); %Get the number of processors
% Create the test data
```
for i=1:num_procs
    data = rand(10^3,1);
    file = sprintf('test_data_%d.csv',i);
    dlmwrite(file,data);
end

% In parallel, compute the mean of each data set. The variable mean will have a different value on each processor
spmd
    file = sprintf('test_data_%d.csv',labindex);
    data = dlmread(file);
    means = mean(data);
end

% Go through and remove the files we just created
for i=1:num_procs
    file = sprintf('test_data_%d.csv',i);
    delete(file);
end

ICE

1. We will test Girko’s circle law, which states that the eigenvalues of a set of random N x N real matrices with entries independent and taken from the standard normal distribution (i.e. using randn) are asymptotically constrained inside a unit circle of radius $\sqrt{N}$ in the complex plane. Create a program that calculates (in parallel using parfor!) the eigenvalues of many (~1000) normally distributed random matrices of size 100x100. Plot the real and imaginary parts of all these eigenvalues on the plane (i.e. use the real parts as x-values and the imaginary parts as y-values), and draw a circle of radius $\sqrt{100}$ around it to verify Girko’s circle law. How long would it have taken if you performed this task in serial instead of parallel?

2. Download the 4 FASTA files located in the ZIP file at http://www.math.oregonstate.edu/~koslickd/TestSequences.zip. Use an spmd program to read the files in (in parallel) using fastaread. Count the number of bases in each sequence (using basecount, note that you can access the results of basecount by using the following syntax: output = basecount(sequence); output.G, this will count the number of Gs). Tally up the total number of Gs that appear in each file. Which file has the most Gs? How much quicker did this task take performing it in parallel rather than in serial?

Acknowledgement

Parts of this text are based on An Introduction to Matlab by David F. Griffiths, as well as on David Arnold and Bruce Wagner’s Matlab Programming.

Published with MATLAB® R2013a