

Guide for CS 1371

Introduction: The MATLAB Command Interface

MATLAB is basically just a really complicated program that functions as an extremely powerful calculator. As such, it can perform tasks that most calculators cannot, usually in fractions of a second. However, like all computer programs, MATLAB is incredibly stupid and is only ever as useful as your coding capability allows it to be. Never underestimate how irritatingly picky MATLAB can be about small sections of code. The programmers designed some functions to be flexible with input values, but for the most part they didn't bother making it user-friendly.

It is not necessary to actually purchase the MATLAB software to complete this course, although it is probably pretty useful; I do not personally own the software. Many, not all, computers in the library do have it. Additionally, some computer geeks have set up a virtual desktop called Citrix that allows you to access your library desktop from your laptop. Unfortunately, the remote desktop generally runs very slowly and is prone to freezing at frequent intervals. It also likes to give you nonsensical login error messages such as "We are sorry. This desktop has been locked, and only you can use it."

Whenever you run MATLAB in the library or from your remote desktop, it will tell you it's "initializing" and then proceed to take an unreasonable amount of time to generate a pointless message about customizing shortcuts. I have yet to find a method for preventing this. The MATLAB screen is divided into a number of sections with specific names and titles. These are important to learn because they make up a large part of the terminology used in the course.

The Command Window:

This is the large and obvious block in the middle of your screen labeled "Command Window." It can be used to run individual lines of code or display the progress of a script. Although you'd imagine it must play a very important role in writing codes, it is actually pretty much useless. If you use the command window to run multiple lines of code sequentially and realize you made a mistake somewhere, you have to redo the WHOLE process. The command window will not allow you to edit anything previously entered, and once a variable is overwritten the original value cannot be restored.

To enter a command in the command window, simply type it in the little box underneath and press enter. It is possible to enter multiple lines of code such as loops and conditional statements using shift+enter, but I wouldn't suggest it.

You can use the up-arrow to scroll through your most recent commands so you don't have to retype them. If you attempt to do something that MATLAB doesn't like or if there is a typo in your command, MATLAB will make an annoying beeping sound and display a red error message that tells you what the problem is. Learn to fear the error message. At the very bottom-left part of the screen MATLAB will display statuses such as "initializing", "busy", or "ready." If you accidentally run an infinite loop or an extremely long recursive function, you can cancel your last command by pressing ctrl+c. You can enter *clc* to clear whatever is in the command window at the time.

The Workspace:

This is the little box off to the right labeled “Workspace.” Its purpose is to keep track of all the variables and data types you are using. Anytime you save a value to a particular variable or enter any assignment statement, the workspace is automatically updated with the new information. It displays both the name of the variable and its data type (a brief description of data types will be given in section one). If the workspace is becoming too cluttered with useless variables, you can reset it by entering *clear* in the command window.

Command History:

This little box keeps track of all the commands you have entered since the beginning of the MATLAB session. If you accidentally clear the command window and workspace but still need the information from them, you can access it from command history. Otherwise, you should just pretend it doesn’t exist.

The Current Directory:

This box is labeled “Current Folder” and is slightly confusing to become accustomed to. MATLAB can only access files and programs that reside in your current directory; if you attempt to run a function or script from a different folder, MATLAB will give you an error message saying that it doesn’t exist. It would therefore behoove you to create an organized set of folders and sub-folders for CS 1371, with each homework assignment having its own folder. If you need to access a function from another homework, which will inevitably happen for some of the more difficult problems, simply copy/paste it into the current directory’s folder and submit it with your homework problems.

To change the current directory, locate the button on the primary toolbar at the top of the screen and to the right of the current folder display. It should look like an ellipsis. Then just navigate to the folder you wish to select and click “OK.” I suggest saving your entire CS 1371 folder on your GT Prism drive so you can access it from any computer on campus.

Section One: Scripts and the Assignment Operator

I will be writing these sections with the primary focus of assisting in concepts used to complete homework assignments, but all of the basic information is useful on the tests.

The Assignment Operator:

Any line of code that uses a single equals sign (the assignment operator) is called an assignment statement. Assignment is used primarily to set values to variables and works the same way with any data type. All variable names are case-sensitive, so “a” and “A” can be used as separate variables. Be very careful when naming and accessing variables. Variable names are generally short enough to take up little space but long enough to be easily distinguished. The last thing you want to do is have to fix your code when you can’t even figure out what it does.

There are three important restrictions on naming variables:

1. A variable’s name must begin with an uppercase or lowercase letter. The rest of the name may contain any combination of letters and numbers.
2. A variable’s name cannot contain spaces. Obviously, MATLAB would then be unable to distinguish between the variable and the surrounding code.
3. A variable’s name cannot contain any special characters except underscores.
4. A variable should never be given the same name as a built-in MATLAB function (min, max, sort, etc.)

By these rules, the following would be legal variable names: A , a , a1 , a1a , a_a

The following are therefore illegal variable names: 1 , 1a , a: , : , a 1, a?

Assignment statements take the form A = B, where “A” is a variable or collection and “B” is an expression that can be immediately evaluated. MATLAB first evaluates “B” then sets “A” equal to it. Assignment statements do NOT establish relationships between variables; they can only be used to set a variable equal to a known value. Once again, changing the values of other variables used in an assignment statement has no effect on the previously outputted variable.

If no “A” is inputted, the value of “B” will automatically be stored in MATLAB’s temporary variable “ans.” “ans” can be accessed at any time and is always placed in the workspace, but each new non-specific assignment statement overwrites whatever is stored in it. It is generally a bad idea to rely on “ans” to store any values you may be using later.

By default, the result of an assignment statement is immediately displayed on the command window. However, this can become very irritating when you run long scripts or change large collections of values; the command window will quickly become clogged with useless information that slows the execution of your code. To prevent this, you can “suppress” the output of individual lines of code by ending them with a semicolon. The result of a suppressed output is still stored in the workspace but will not be displayed in the command window. With practice, you will learn to subconsciously suppress all outputs as you write. MATLAB will automatically notify you of unsuppressed outputs by discoloring the equal’s sign.

Whenever MATLAB encounters spaces, it automatically ignores them. You can therefore add spaces in lines of code to make them more legible.

Using the Dot with Basic Arithmetic:

The MATLAB notation to add, subtract, multiply, divide, or raise to a power is exactly what you would expect (+, -, *, /, ^). However, MATLAB must have a way of distinguishing between regular arithmetic and special matrix operations. For instance, you can either multiply each element in one matrix by the corresponding element in another or actually perform matrix multiplication using linear algebra. Note: MATLAB uses decimals instead of fractions whenever it encounters a non-integer.

To differentiate between the two, MATLAB utilizes a dot (.) system with the symbols *, /, and ^. To perform element by element multiplication, division, or exponentiation, simply add a single period before the arithmetic symbol (.*, ./, or .^). Linear algebra functions are performed whenever the dot is absent and will cause errors if the dimensions of the matrices do not match properly. Note that the dot has no effect on scalar values (5*5 is the same as 5.*5).

Scripts:

As previously stated, using the command window to solve complex problems is inadvisable because it does not allow you to edit commands you have already run. Consequently, most coding is completed using scripts and the MATLAB function editor. A script is a blank space for inputting blocks of code or functions. When you finish writing a script, you can run the entire thing at once; if an error occurs, you can edit and rerun your code to your heart's content.

To open a new script, click the icon in the upper-left corner that looks like a blank sheet of paper or press ctrl+n. When you finish typing out your code, you can run the script by using the green triangle icon or by pressing F5. To save, go to file→save as... or press ctrl+s. Scripts are automatically saved as .m files, the default file type for MATLAB. Always leave the file type as .m if want to receive credit.

If MATLAB dislikes something you've written on a script, it will inform you by enclosing a variable inside a discolored box or by underlining part of your code in red. This is a very quick way to spot typos, but don't rely on it for everything.

If you plan to test your script frequently, you can dock the script editor inside the command window. This will allow you to view the values of different variables from the workspace without having to minimize the script. To dock the editor, click the small arrow on the left just under the minimize button. The editor can be undocked in the same way by clicking the up-arrow above the minimize button.

If you wish to run a script one line at a time rather than all at once, you can enter debug mode by clicking the gray space to the right of the line numbers on the left side of the screen. A red circle will appear on the debugged line. Any number of lines can be debugged. When run, the script will immediately proceed all the way to the first debugged line, then stop. You can then continue running each individual line by pressing the icon with the curvy blue arrow on top of a sheet of paper labeled "step." The "step in" icon allows you debug any internal functions that your script runs. "Continue" allows the code to run normally until it reaches the next debugged line. While in debug mode, you can display the values of any variables by hovering the mouse over them or by simply viewing their progress

in the workspace. If you cannot locate the source of an error, debugging your script is an excellent way to track the problem.

You can add your own personal comments to any of your code to make it easier for you or other people to understand. Simply precede any comments with a “%” percentage sign. MATLAB completely ignores any green text following a percentage sign when it executes code. To see an example, just look at a hw.m file. To comment a line of code, click on it and press ctrl+r. To uncomment something, press ctrl+t. It is not necessary to actually highlight the code you wish to comment or uncomment. You can earn extra credit points on your homework assignments by commenting your code.

Built-In MATLAB Functions:

The designers of MATLAB included hundreds of built-in functions to assist users in writing code. These functions allow you to perform otherwise complex tasks in a single line of code. To call a function, type its name followed by any inputs in parentheses, separated by commas. I will list any useful functions as well as brief descriptions of them in each section of this guide.

If you wish to perform a relatively common task and are wondering whether MATLAB has a built-in function, it probably does. Go to Help→function browser (shift+F1) and search for the name of the function you are looking for. Function names are generally incredibly obvious, exactly what you would expect them to be. Function names are also case-sensitive.

sin()	Calculates the sine
cos()	Calculates the cosine
sqrt()	Calculate the square root
abs()	Calculates the absolute value
exp()	Calculate e^x

Examples:

Input:

```
5
x = 5
x = 5:
```

Output to Command Window:

```
ans = 5
x = 5
```

Input:

```
x = 5
y = x
x = 6
y
```

Output to Command Window:

```
x = 5
y = 5
x = 6
y = 5
```

Note that changing the value of x had no effect on y.

```
y = X
```

```
Error: undefined variable X
```

Remember that variable names are case-sensitive.

```
x = x + 1
```

```
x = 7
```

Write a script that will calculate the radius and surface area of a sphere given a volume of 15.

```
volume = 15;
radius = (volume.*3./4./pi).^(1./3);
surface_area = 4.*pi.*radius.^2;
```

(The constant π is expressed as "pi" rather than with symbols)

Write a script that uses the linear relationship between the points (3,4) and (-2,5) to find the y-value for $x = 20$.

```
x1 = 3;
y1 = 4;
x2 = -2;
y2 = 5;
slope = (y2 - y1) ./ (x2 - x1);
y_int = y1 - slope .* x1;
y3 = 20 .* slope + y_int;
```

To Complete Homework Assignments:

1. Create a folder in your CS 1371 folder for the homework assignment. Make it easy to find.
2. Go to T-Square→Assignments and click on the homework assignment.
3. Save each of the files under "Additional resources for this assignment" by right-clicking and selecting "save target as..."
4. Open MATLAB and change the directory to the specific homework folder.
5. Open the ABCs.m file and follow the directions. When you are finished, you can check your answers by running the ABCs pretest file. Right click the pretest in the directory and select "Run File." The pretest will display which answers are correct or incorrect on the command window.
6. Open the hw.m file and write your personal information at the top. The descriptions for the drill problems will include all inputs, outputs, and character classes. Always remember to check the character class of your outputs. Don't forget to comment your code.
7. Create a new file in the script editor for each drill problem. When you finish, test them thoroughly. If there are test cases on the hw.m file, you can uncomment them and run the hw file as long as your scripts are saved in the same directory. Alternatively, you can copy the test cases and run them from the command window.
8. After you complete the drill problems, update the information regarding collaboration on the hw.m file. Navigate back to the same assignment page on T-square and submit all files listed in the "Files to submit" list.
9. You can submit as many times as you want before the deadline and will frequently have opportunities to resubmit after you receive your grade. Resubmissions are graded as the average of the two submissions except on test weeks.

Homework assignments are graded automatically by a computer. It will test each of your scripts or functions multiple times with various inputs determined by the TAs. They will use any means possible to make your code run an error and cause you to get a zero on the problem. For this reason, you should never rely solely on the test cases provided with the homework file. Make sure your functions can deal

with inputs such as negative numbers, fractions, and empty vectors or strings. If you can find one output but not another, set the difficult output equal to something pointless like 6 so you can get partial credit. Remember: any code that runs an error results in a zero.

Section Two: Functions

The primary drawback of scripts is their lack of variability; to run a script with different starting values, you must manually change each of the predetermined variables. Additionally, many problems require running the same block of code many times. Using functions allows you to complete such tasks in a more concise, easily understandable, format.

A function is simply a script with variable inputs and outputs. You call the function with whatever inputs you wish to use, it does something, and it returns the final result to whatever called it. Once you have written a function, you don't even have to know exactly what it does—just what it takes in and what it regurgitates.

Writing Functions:

Except for a few lines of code, functions are absolutely identical to scripts. The main difference and, coincidentally, the easiest way to spot a function is the first line, the function header. The header contains all of the basic information concerning the function in the following form:

```
function [output1, output2, output3] = name_of_function (input1, input2)
or
function output1 = name_of_function (input1)
```

When you type the word “function”, MATLAB should recognize it and color it blue. Please remember that spaces do not matter, but commas do. In situations with only one input/output, brackets may be excluded but parentheses are always necessary. A function can have any number of inputs or outputs, even zero. You can use the same variable as both an input and an output as long as you don't need to keep track of its original value. Although you will probably never encounter a function with no inputs, it is quite common for one to have no outputs if its purpose is simply to produce a plot or print something out to the command window. If the function has no outputs, eliminate the equals sign and outputs. Thus:

```
function name_of_function (input1, input2, input3)
```

Most functions are ended with the word “end” to let MATLAB know that the function is complete. Doing so is only completely necessary if you plan on using helper functions, which I will now explain. Although functions are normally called directly from the command window, they can actually call other functions as well, as long as all the files needed are saved in the current directory or added to the path.

If you wish to perform a small task multiple times in a single function, you can write a small helper function to assist you. Just place the helper function after the “end” of the main function. Doing so can save space as well as make your code more readable. Here is an example of a helper function.

```
function [out1, out2] = helperExample (a,b,c)
    out1 = confangleNumber(a);
    out1 = out1 - confangleNumber( a + (b - a).^3);
    out2 = confangleNumber(b + 6.*a);
end
```



```
function num = confangleNumber(num)
    num = abs(ceil(num + (num ./ 2).^2 - round((num + 1)/(num - 1).^2)))
end
```

As previously stated, it is not necessary to have any earthly idea about what a function actually does. When you attempt to save your function, MATLAB will automatically suggest the same name as the one given in the function header. Never name a function anything else.

Because functions are so versatile and easy to grade, this course relies primarily on function-writing for homework assignments and coding problems on tests. If you can't write a function properly, you're probably going to fail the class.

Calling Functions:

Many people become very confused when they actually attempt to use a function they've written. The easiest and most obvious way to remember the format is that you call your own functions in EXACTLY THE SAME WAY as you would call built-in MATLAB functions. Here is a comparison to the built-in `min()` function.

```
[minimum, index] = min(vector);
[distance, speed] = helperExample(length, width, height);
```

Notice that the input and output variable names of the function call above do not match the names given in the function itself. This is because all variables used in a given function are temporary. When you enter the function call above, MATLAB first opens a new command window for the function `helperExample`. It then sets the values of "length", "width", and "height" equal to "a", "b", and "c" respectively (always in the same order given in the function header). When MATLAB reaches the end of the `helperExample` function, it closes the new command window and sets whatever values are stored in "out1" and "out2" equal to "distance" and "speed." However, only those final values will be stored in your original workspace. All other variables are deleted as soon as the function call is complete.

Let me reiterate: MATLAB will return only the output variables specified in the function header; all other variables are considered temporary. Additionally, each function call opens its own workspace, which does not have access to the variables in any other workspace. Therefore, any values used within a function must be given as inputs. If you do not specify output variables in your function call, the first results will automatically be stored in "ans."

Function calls utilize a simple system known as the stack. Basically, the most recent command is executed first in any given situation. In the `helperExample` function, each function call to `confangleNumber` is completed (another command window is opened and closed) before the next line of code is executed. Really, the stack concept is straightforward and easy to remember.

That's pretty much it for functions. There's really not that much to know.

Examples:

Let's just take the scripts from section one and turn them into functions. Remember to make your functions easy to understand by naming variables properly.

```
function [radius, surface_area] = circleStuff (volume)
radius = (volume.*3./4./pi).^(1./3);
surface_area = 4.*pi.*radius.^2;
end
```

```
function y3 = extrapolate (x1, x2, y1, y2)
slope = (y2 - y1) ./ (x2 - x1);
y_int = y1 - slope .* x1;
y3 = 20 .* slope + y_int;
end
```

Section Three: Vectors

So far, we have only considered situations that involve working with single values. However, we may wish to operate on large sets of numbers at the same time. In these situations, we utilize vectors, large collections of scalar quantities. As a matter of fact, even single numbers in MATLAB are interpreted as vectors of length one. Vectors are easy to work with and easy to understand.

Creating Vectors:

To create a vector, simply enclose a list of numbers or variables separated by spaces or commas in straight brackets. Vectors can be of any length, even one or zero. Vectors of length zero, or empty vectors, are usually used to initialize variables that will be filled later. If you want to create a new variable of a non-predetermined length, first initialize it as an empty vector.

You can also create column vectors in much the same way. Simply separate all elements with semicolons to indicate that they should be placed on top of each other. MATLAB indexes and slices column vectors in the same way.

To convert row vectors from row to column or column to row, transpose them by following them with an apostrophe. The transpose function is not limited to transposing vectors; it actually swaps the rows and columns of any one or two-dimensional collection.

```
vec = [1 2 3 4 5];  
vec2 = [6, 7, 8, 9];  
emptyVec = [];  
columnVec = [1; 2; 3; 4; 5]
```

The Colon Operator:

Many vectors are too large to create manually. To generate sets of evenly incremented numbers, MATLAB uses a colon operator, not to be confused the semicolon used to suppress outputs. The colon operator takes in a start value, a step value, and a final value; it generates a vector of numbers ranging from the start value to the stop value and incremented by the step size. Expressions using the colon operator do not have to be enclosed in brackets. All vectors are of type double.

```
colonVec = start:step:stop
```

If a step size is not specified, MATLAB will use the default step of one. Step sizes can be positive or negative. Any of the three inputs can be substituted for a variable stored in the current workspace. Putting stupid expressions into the colon operator almost never produces an error. Here is a list of references for absurd situations.

1. If the step size is equal to zero, regardless of the other two inputs, produces an empty vector.
2. If the start and stop values are equal, returns a vector containing only that number unless the step value is zero.

3. If the start value is less than the stop value but the step size is negative, produces an empty vector.
4. If the start value is greater than the stop value but the step size is positive, produces an empty vector.

Although the start value is always included in the resulting vector, the stop value will only be included if it falls within the exact step size of the value before it. Thus, 0:4:7 produces the vector [0 4], which does not include 7. You must always be careful to set the step value properly if you want the stop value to be included.

Linspace ():

If you absolutely must include both the start and stop values or are interested in generating a vector of a particular length, you can use the `linspace()` function. `Linspace()` takes in a start value, a stop value, and the desired length of the output vector (NOT a step size) and produces a vector of linearly spaced numbers between the start and stop values. The start and stop value are guaranteed to be included, and the space between any two adjacent numbers is always equal. However, you will probably end up with some weird fraction as your step size. If you do not input a desired length, MATLAB will use the default value of 100.

```
linspaceVec = linspace(start, stop, length);
```

Basic Arithmetic with Vectors:

Any basic arithmetic involving a vector and a constant changes the value of all elements in the vector. Remember to use the dot (.) with multiplication and division.

```
vec = [1 3 5 7 9];  
vec = vec + 2  
vec = vec.*2  
vec = [3 5 7 9 11]  
vec = [2 6 10 14 18]
```

Indexing Vectors:

To return a particular range of values from a given vector, you can index into it using parentheses. The item in parentheses, or the range you wish to return from the vector, can be either a double or collection of doubles (another vector) representing the positions in the original vector that you wish to return. By using another vector as the index range, you can return the elements in any order or even the same element multiple times. The length of the vector returned will always be equal to the length of the index.

Because any index that returns multiple values is by definition a vector, indices easily be generated using the colon operator. Doing so allows you to perform many nifty indexing tricks such as reversing a vector or returning only even-numbered positions. If you wish to index up to the last element in a vector, use the keyword “end.” Because MATLAB interprets “end” as a number equal to the length of the vector, you can use subtraction or division to index positions based on the end value. Any attempt to index with a number less than one, a fraction, or a number greater than the length of the

vector you're indexing into will produce an error. You will probably lose a majority of the points on homework assignments from the message "Error: Index exceeds matrix dimensions."

```
vec = [1 3 5 7 9];
vec(3)          ans = 5
vec([1 4 2 2 3]) ans = [1 7 3 3 5]
vec(1:2:5)      ans = [1 5 9]
vec(6)          Error
vec(1:3:end)     ans = [1 7]
vec([1:3, end, end-2]) ans = [1 3 5 9 7]
vec(round(end/2):end) ans = [5 7 9]
```

Slicing Vectors:

The word "slicing" seems to be standard terminology for this course even though it makes no sense to me. Slicing involves replacing some of the elements in one vector with some of the elements in another vector. The most important thing to remember when slicing is to be careful about what you choose to overwrite. To delete elements from a vector, set them equal to empty brackets. Here are the basic forms of slicing:

1. $A(\text{index}) = B(\text{index})$
This is the most common form. MATLAB first evaluates $B(\text{index})$, then replaces $A(\text{index})$ with it. The rest of "A" remains unchanged. Produces an error if $B(\text{index})$ and $A(\text{index})$ are not the same length.
2. $A(\text{index}) = B$
Replaces the elements in $A(\text{index})$ with whatever is stored in B. Once again, $A(\text{index})$ and B must have identical lengths.
3. $A = B(\text{index})$
This is the mistake many students make when slicing. This overwrites the variable A and sets it equal to $B(\text{index})$. All elements in A are replaced, including the ones outside the range of B, so the lengths can be unequal without producing an error.
4. $A(\text{index}) = []$
Setting anything equal to empty brackets deletes it. This can be used to shorten the length of A or delete all of the elements entirely, leaving A as an empty vector.

Contrary to indexing, you can slice elements that are out of bounds of the original vector. If you assign a value to an element that did not originally exist, MATLAB will extend the vector by filling all intermediate positions with zeros. To slice outside the bounds of a vector of unknown length, use addition or multiplication with "end."

```
vec1 = [1 3 5 7 9];
vec2 = [0 2 4 6 8 10];
vec1([1 2 4]) = vec1(3:end)
vec1(1:3) = vec2(end:-1:4)
vec2(2:end) = vec1
vec1 = vec2(1:2:end)
vec2([1 4 5]) = []
vec1(end+2) = 6
vec1 = [5 7 5 9 9]
vec1 = [10 8 6 7 9]
vec2 = [0 1 3 5 7 9]
vec1 = [0 4 8]
vec2 = [2 4 10]
vec1 = [1 3 5 7 9 0 6]
```

The possibilities are endless...

Concatenating Vectors:

MATLAB uses square brackets for just about everything. You can also use them to concatenate vectors, or combine multiple small vectors into one large one. Simply create a vector in the standard manner using vectors as the elements rather than scalar values. To concatenate vectors column-wise, separate the different elements with semicolons.

```
vec1 = [1 5 9];  
vec2 = [6 8];  
bigVec = [vec2, vec1]          bigVec = [6 8 1 5 9]  
colVec = [vec1'; vec2']       colVec = [1; 5; 9; 6; 8]
```

Logical Indexing:

So far, we have only dealt with data of type double, representing real, measurable numerical values. However, another data type frequently used in MATLAB is type logical. Logical data can only take two forms: true or false. In most cases, the number one represents true, while the number zero indicates false; however, MATLAB actually interprets anything not equal to zero as true, including negative numbers and strings.

When using any of the expressions below, keep in mind that the two objects you wish to compare must be dimensionally IDENTICAL. Attempting to compare two vectors of unequal length will produce an error.

A logical expression will generate a collection of logical indicating where the statement is true and where it is false (one for true and zero for false). Logical expressions generally involve comparisons and can take numerous forms:

- ==, equal to
- >, greater than
- <, less than
- >=, greater than or equal to
- <=, less than or equal to
- ~=, not equal to

The tilde (~) is always used to refer to the word “not.” When preceding a logical collection, it changes every true to false and every false to true. Nifty!

```
vec1 = [1 3 5 7 9];  
vec1 == 3          ans = [0 1 0 0 0] (logical)  
vec1 > 3           ans = [0 0 1 1 1]
```

Masking

Indexing with logicals is very different from indexing with doubles. The position numbers of values being indexed are represented by the *values* of double indices and the *position numbers* of logical indices. Thus, a true at position three in a logical collection will index the element in position three of

the collection being indexed. The index may be shorter than the vector being indexed, but not longer. For some reason, logicals can even be used to index other logicals. For some reason, this is known as masking.

```
vec1 = [1 3 5 3 9];
vec1(vec1==3)          ans = [3 3]
vec1(vec1 <= 5)        ans = [5 9]
vec1(true false true)  ans = [1 5]
```

Masking can be used with even more precision using and (&) and or (|). These can be used to combine collections of logicals to form a final index representing all of them. Just remember these seemingly random rules:

1. An “and” statement is only true if all of the elements are true.
2. An “or” statement is only false if all of the elements are false.

When multiple logical collections are combined using & or |, each element in the collections is compared individually to the corresponding elements in the other collections. This strategy can be used to make a single index that matches many different parameters.

```
vec1 = [1 3 5 7 9];
vec1 >1 & vec1 <= 7      ans = [3 5 7]
[true false true] | [false false true]  ans = [1 0 1]
(true | false) & (false & true)        ans = 0
```

Useful Functions:

Vector Stuff:

- min(x) – returns the smallest element in the collection (Note: if the minimum value occurs multiple times, min() and max() will return only the first instance)
- [value, index] = min(x) – returns the smallest element along with its position number
- max(x) – returns the largest element in the collection
- [value, index] = max(x) – returns the largest element along with its position number
- sort(x) – sorts the elements in ascending order
- [newX, index] = sort(x) – returns x in ascending order along with a vector containing the position numbers of the original x to which each value corresponds
- fliplr(x) – flips x left-to-right
- x' – transposes x (row vector \leftrightarrow column vector)
- mod(x,num) – returns the remainder if x is divided by num
- round(x) – rounds x up or down
- ceil(x) – rounds x up to the next-highest integer if x is fractional
- floor(x) – rounds x down to the next-lowest integer if x is fractional
- ones(x,y) – creates an array of ones of dimension x,y (Use 1 for x or y to generate vectors)
- zeros(x,y) – creates an array of zeros of dimension x,y
- true(x,y) – creates an array of logical trues of dimension x,y
- false(x,y) – creates an array of logical falses of dimension x,y
- sum(x) – computes the sum of x

- `prod(x)` – computes the product of `x`
- `rand(x,y)` – generates an array of dimension `x,y` of linearly spaced random numbers between 0 and 1
- `randn(x,y)` – identical to `rand()` except all values are truly random
- `find(logical expression)` – generates a vector of all indices where the logical expression is true

Logical Stuff:

- `all(x)` – true if everything in `x` is true
- `any(x)` – true if anything in `x` is true
- `not(x)` – same as `~x`
- `or(x,y)` – same as `x | y`
- `and(x,y)` – same as `x & y`

Section Four: Arrays

While vectors are merely one-dimensional collections of data stored as rows or columns, arrays are two-dimensional collections stored as grids containing both rows and columns. They operate in **exactly the same way** as vectors except for a few minor differences. Although arrays are supposed to be different from matrices, they're actually not. Arrays can contain doubles, logicals, and various other data types.

While MATLAB is capable of storing data in more dimensions than are readily understandable, this course never involves data collections of more than two dimensions, excluding images.

Creating Arrays:

Arrays are created using square brackets in much the same way as vectors. Simply enter the individual *rows* of the collection separated by semicolons. Yes, the semicolon also has many diverse uses. Arrays must always be rectangular; if at any point MATLAB attempts to create an array that is not rectangular, it will produce an error.

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

```
arr2 = [1 2; 3 4; 5 6]          ans = 1 2
                                     3 4
                                     5 6
```

```
arr3 = [1 2 3; 4 5; 6 7 8]      Error
```

Indexing Arrays:

The items in arrays are organized based on their location as determined by rows and columns. Both rows and columns begin at one and increase as you move down and to the right, respectively. Elements are indexed by their row numbers followed by their column numbers, with a comma in between. To index all rows or all columns, replace the row or column numbers in the index with a colon symbol. You can also use the keyword "end" to access the final element in a particular row or column, or both if it makes you happy.

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

```
arr(2,3)                          ans = 6
arr(2,:)                          ans = [4 5 6]
```

<code>arr(:,2)</code>	<code>ans = [2;5;8]</code>
<code>arr(end,end)</code>	<code>ans = 9</code>

Splitting Arrays into Halves and Quarters:

Don't bother memorizing these. You can easily reproduce them with some practice.

<code>arr(1:round(end/2),:)</code>	Top half
<code>arr(round(end/2):end,:)</code>	Bottom half
<code>arr(:,1:round(end/2))</code>	Left half
<code>arr(:,round(end/2):end)</code>	Right half
<code>arr(round(end/2):end,round(end/2):end)</code>	Bottom-right quarter

MATLAB's Secret Trick:

Although arrays are displayed in the command window as two-dimensional matrices of values, MATLAB actually stores the information as a very long column vector. It numbers the elements by reading down the columns one at a time, meaning that arrays can actually be indexed as if they were vectors. This can be useful to remember because it allows you to reshape arrays into vectors very easily.

	<code>1 2 3</code>
<code>arr = [1 2 3; 4 5 6; 7 8 9]</code>	<code>ans = 4 5 6</code>
	<code>7 8 9</code>
<code>arr(:)</code>	<code>ans = [1;4;7;2;5;8;3;6;9]</code>
<code>arr(4)</code>	<code>ans = 2</code>
<code>arr(8)</code>	<code>ans = 6</code>

Slicing Arrays:

By slicing arrays in creative ways, you can do all sorts of interesting things such as swapping rows or columns, mirroring, deleting elements, and flipping things upside down. Make sure you know and understand the first two, as they are extremely useful.

<code>arr(:,:) = arr(:,end:-1:1)</code>	Mirrors left to right
<code>arr(:,:) = arr(end:-1:1,:)</code>	Flips upside down
<code>arr(2:2:end,:) = arr(end:-2:2,:)</code>	Reverses the even-numbered rows
<code>arr(1:3:end,2:2:end) = []</code>	Deletes every third row and second column

Concatenating Arrays:

Arrays can be concatenated horizontally or vertically just like vectors. Once again, any command that would produce a non-rectangular array results in an error.

<code>[arr1, arr2]</code>	Concatenates horizontally
<code>[arr1; arr2]</code>	Concatenates vertically

Logical Indexing and Masking:

Seriously, it's the *same* thing.

<code>arr = [1 2; 3 4];</code>	<code>ans =</code>	<code>1 2</code>
		<code>3 4</code>
<code>arr > 3</code>	<code>ans =</code>	<code>[0 0; 0 1]</code>
<code>arr(mod(arr,2))==0=7</code>	<code>ans =</code>	<code>[1 7; 3 7]</code>

When you need to determine which elements meet multiple overlapping conditions, you can use and (&) and or (|) to combine multiple sets of logicals.

<code>mod(arr,2)~=0 & arr >= 5</code>	All odd elements greater than or equal to five
<code>arr > 10 arr==7</code>	All elements greater than 10 or equal to 7

Basic Arithmetic:

All arithmetic calculations involving arrays are performed element-by-element. Remember to use the dot with multiplication and division.

<code>arr = [1 2; 3 4]</code>	<code>ans =</code>	<code>1 2</code>
		<code>3 4</code>
<code>arr2 = [5 6; 7 8];</code>	<code>ans =</code>	<code>5 6</code>
		<code>7 8</code>
<code>arr + arr2</code>	<code>ans =</code>	<code>[6 8; 10 12]</code>
<code>arr .* arr2</code>	<code>ans =</code>	<code>[5 12; 21 32]</code>
<code>arr - 1</code>	<code>ans =</code>	<code>[0 1; 2 3]</code>

Transposing and Rotating Arrays:

Just as the apostrophe turns row vectors into column vectors, it also transposes the rows and columns in arrays. The first column becomes the first row, the second column becomes the second row, etc. This is not to be mistaken with rotating an array clockwise or counterclockwise, requires both transposition and indexing.

<code>arr'</code>	<code>ans = [1 4 7; 2 5 8; 3 6 9]</code>
<code>arr(end:-1:1,:)'</code>	Rotates clockwise
<code>arr(:,end:-1:1)'</code>	Rotates counterclockwise
<code>arr(end:-1:1,end:-1:1)</code>	Rotates 180 degrees

Reshape()

This function allows you to change the number of rows and columns in an array without altering its overall size. MATLAB fills the spaces in the new array in the same order as the original, namely reading down the columns one at a time. Thus `arr(4)` will be equal to `new_arr(4)`.

You probably won't be using `reshape()` more than once or twice.

```
new(arr) = reshape(arr, new_rows, new_columns)
```

<code>arr2 = [1 2; 3 4; 5 6]</code>	<code>ans =</code>	<code>1 2</code>
		<code>3 4</code>
		<code>5 6</code>

<code>new_arr = reshape(arr,2,3)</code>	<code>ans =</code>	<code>1 5 4</code>
		<code>3 2 6</code>

Useful Functions:

- `ones(x,y)` – creates an array of ones of dimension x,y (Use 1 for x or y to generate vectors)
- `zeros(x,y)` – creates an array of zeros of dimension x,y
- `true(x,y)` – creates an array of logical trues of dimension x,y
- `false(x,y)` – creates an array of logical falses of dimension x,y
- `magic(x)`—creates a magic square of dimension x
- `[row,col] = size(arr)`—returns the number of rows and columns in the input array

The following functions work differently with arrays than with vectors. Built-in functions generally treat arrays as individual column vectors and express their outputs as row vectors.

- `[value,index] = min(arr)`—returns a row vector of the minimum values in each column. The second output contains the row number of the minimum value in each column
- `[value,index] = max(arr)`—returns a row vector of the maximum values in each column. The second output contains the row number of the maximum value in each column
- `[sorted_arr,index] = sort(arr)`—sorts each column of the array in ascending order from top to bottom. The second output consists of the indices used to sort each column concatenated together as an array
- `sum(arr)`—returns a row vector of the sum of each column in the array
- `mean(arr)`—returns a row vector of the mean of each column in the array

To determine the absolute minimum, maximum, sum, or mean of an array, simply call the respective functions twice.

- `[value, index] = max(max(arr))`—returns the absolute maximum of the array along with its column number
- `[value, index] = min(min(arr))`—returns the absolute minimum of the array along with its column number
- `sum(sum(arr))`—returns the overall sum of the elements in the array
- `mean(mean(arr))`—returns the overall mean of the elements in the array

I know these explanations can be somewhat confusing, so do yourself a favor and play around with the `min()`, `max()`, `mean()`, and `sort()` functions for a few minutes. All four frequently appear on homeworks and test problems.

Section Five: Strings

Vectors and arrays are only useful for organizing sets of doubles and logicals. Now, we will be dealing with a new data type called char (short for character). Using characters allows you to set variables and outputs equal to meaningful words and phrases rather than just numbers.

A string is a vector of numbers that represent the characters and symbols on your keyboard. MATLAB identifies these number-character combinations based on a predetermined set of conversions called an [Ascii Table](#).

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	:	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com

The columns labeled “Dec” contain the ascii codes for the corresponding red characters in the columns labeled “Char.” For example, the ascii code for a capital Z is 90. Note that the code for a lowercase letter is exactly 32 plus its uppercase equivalent. The ascii code for a space is 32.

The most important thing to remember about strings is that they function exactly like vectors in most scenarios. To better understand the functionality of strings, we’ll be looking at a lot of examples.

Creating Strings:

Strings are created by enclosing letters, numbers, and symbols in single quotations marks ("). Unlike with creating vectors, you needn't separate the elements with spaces or commas (remember that spaces and commas now count as elements). You can also create empty strings that function just like empty vectors.

If you have a string saved as a variable in your workspace, it will have a cute little 'ab' symbol next to it. The class() of a string is 'char.' In fact, the class() function always returns an answer of type char.

```
str = 'Happy Birthday';
str2 = '770-253-5796';
emptyString = '';
class(str)           ans = 'char'
class(true)          ans = 'logical'
class('logical')     ans = 'char'
```

Char() and Double():

MATLAB contains two built-in functions for converting between different data types. double() changes a character string into a vector of each character's representative ascii code; char() converts a vector of doubles into the characters they represent. You can therefore create a vector of ascii codes and use char() to change it into a string.

```
str = 'Ishmael';
double(str)           ans = [73 115 104 109 97 101 108]
vec = [77 111 115 101 115];
char(vec)             ans = 'Moses'
```

Basic Arithmetic with Strings:

You can use basic arithmetic to change characters in strings to other characters. From our previous example, we know that adding 32 to a string of uppercase letters will convert them to lowercase. Just remember the golden rule of strings: whenever you add or subtract elements in strings, MATLAB always changes them back to doubles. You will therefore have to use char() to keep your original string format.

```
str = 'SARAI';
str + 32           ans = [115 97 114 97 105]
char(str + 32)     ans = 'sarai'
char(50) + char(50) ans = 100
```

Indexing Strings:

You can index the elements in a string the same way you would with a vector. Just keep in mind that MATLAB now interprets EVERYTHING in the string as a separate element, including spaces and commas. Indexing strings returns the actual characters, not doubles.

```
str = 'Adam and Eve';
str(2)          ans = 'd'
str(5)          ans = ' '
str(3:7)        ans = 'am an'
length(str)     ans = 12
```

Slicing Strings:

If you can slice vectors, you can slice strings. If you slice beyond the bounds of the original string, MATLAB will fill the intermediate spaces with ascii zeros, not 32s.

```
str1 = 'Isaac';
str2 = 'Rebecca';
str1(3:end) = str2(2:4)      str1 = 'Isebe'
str2(1:5) = str1             str2 = 'Isaacca'
str1 = str2(1:end-1)         str1 = 'Rebecc'
str1([1 3]) = []            str1 = 'sac'

str3 = 'Abram';
str3(8) = '0'                ans = 'Abram 0'
double(str3(7))              ans = 0
```

Concatenating Strings:

You only need to worry about concatenating strings horizontally. Technically, they can be concatenated vertically to make freaky string arrays, but doing so is virtually useless. Keep in mind that strings are concatenated just like vectors, so MATLAB will not add spaces to separate words.

You can also concatenate strings with doubles since they're virtually the same thing. MATLAB always retains the string format after concatenation.

```
tstr2 = 'Ham';
str3 = 'Japheth';
[str1 str2 str3]          ans = 'ShemHamJapheth'
[str1 65]                 ans = 'ShemA'
[65 str1]                  ans = 'AShem'
```


Logical Indexing with Strings:

Strings can be used in logical expressions much like vectors. The only difference is that you can now make comparisons between characters and doubles. Remember that comparing vectors of different lengths produces an error. It is impossible with strings of unlike lengths as well; there's a special function for that.

Logical expressions are very useful for determining capital and lowercase letters and for differentiating between letters and symbols. Once again, you can locate elements that fulfill multiple logical conditions using and (&) or or(|).

```
str1 = 'Jacob';
str2 = 'Rachel';
str1=='c'           ans = [0 0 1 0 0]
str2==82           ans = [1 0 0 0]
str1==str2          Error

str>=65 & str<=90    Capital letters
str>=97 & str<=122    Lowercase letters
(str>=65 & str<=90)|(str>=97 & str<=122) All letters
```

Strcmp() and Strcmpi():

The strcmp() function allows you to compare strings of unequal length. This is important because attempting to use logical expressions to do so produces an error. Strcmp() outputs a logical 1 or 0 representing whether the two input strings are exactly equivalent.

Strcmp() is case-sensitive. There is another built-in function, strcmpi(), that performs the same function while ignoring case.

Strcmp() and strcmpi() can also be used to compare a string with a cell array of strings, in which case they output a vector of logicals.

```
str1 = 'Ephraim';
str2 = 'Manasseh';
str3 = 'Ephraim';
str4 = 'manasseh';

strcmp(str1, str2)    ans = 0
strcmp(str1, str3)    ans = 1
strcmp(str2, str4)    ans = 0
strcmpi(str2, str4)   ans = 1
```

Sprintf():

The sprintf() function allows you to create strings containing variables. Consider the following situation: You need to create a string stating how old someone is, but the person's name and age are stored in variables in your workspace. You can either concatenate the string manually or use the sprintf() function.

sprintf() takes in a string containing variable markers followed by the parameters you wish to use. The number of inputs will always be equal to one plus the number of variables in the output string. Variable markers are placed using a percent (%) sign followed by a letter designating the type of variable (double or string, usually). Type %d for a double variable and %s for a string. Remember to list the variable names in the same order you used them; you can use the same variable multiple times in the formatted string, but you still have to list it once for each time it is used.

```
name = 'Methuselah';
age = 969;
[name ' is ' age ' years old']      ans = 'Methuselah is 969 years old'
sprintf('%s is %d years old, name, age)  ans = 'Methuselah is 969 years old'
sprintf('%d %d %d %d, age, age, age, age)  ans = '969 969 969 969'
```

Strtok():

The strtok() function is used to split (tokenize) an input string into two parts based on the location of a specific character called the delimiter. Basically, you choose what character to search for (the delimiter) and MATLAB will divide the input string into two output strings according to the following rules:

1. MATLAB locates the first non-delimiter character and deletes all delimiters preceding it.
2. MATLAB locates the next delimiter after the character in (1). The first output is every character from the remaining string up to the character before the delimiter. The first output can NEVER contain the delimiter.
3. The second output is everything else, including the delimiter found in (2).
4. If the input is an empty string, both outputs will be empty strings.
5. If the string does not contain any delimiters, the first output is the entire string, and the second output is an empty string.
6. If the string contains only the delimiter, both outputs will be empty strings.

Strtok() is one of the most important functions to know for MATLAB. It is used heavily in both string manipulation and file input/output.

```
str = 'eeeeAbel Cain';
[A,B] = strtok(str,'e')      A = 'Ab'
                              B = 'el Cain'
```

```
[C,D] = strtok(B,' ')
[E,F] = strtok(D,' ')

C = 'el'
D = ' Cain'
E = 'Cain'
F = ''
```

Lower() and Upper():

The lower() and upper() functions take in a string and convert all letters to lowercase or uppercase letters, respectively. Because they ignore special characters in strings, they are generally very useful.

Solving Madlib Problems:

For some reason, Madlib problems are very popular in string-related drill problems. The difficulty arises from the concept of replacing part of a string with another string of unequal length. Consider the following Madlib sentence:

```
str = 'I can do all things through @ who strengthens me.'
```

A typical problem may ask you to replace the '@' symbol with the word 'Christ'. Unfortunately, simply slicing letters into the original string will overwrite the end of the sentence rather than insert the word in place (just think of the same problem using vectors instead of strings). You must therefore use either strtok() or manual concatenation to accomplish the insertion.

Remember that the find() function returns a vector of all true positions, not merely the first one.

```
index = find(str=='@');
index = index(1);
str = [str(1:index-1), 'Christ', str(index+1:end)];
OR
[A,B] = strtok(str, '@');
str = [A, 'Christ', B(2:end)];
```

Section Six: Conditionals

Conditional statements allow you to determine whether or not to run a certain block of code based on some non-predetermined information. Basically, it's what you wish you could have been using for all the previous homework assignments. Conditionals come in two forms—if and switch statements, but knowing switch statements is more useful than absolutely necessary.

If Statements:

All if statements are coded in the following basic format:

```
if <logical expression>
    <code block>
end
```

MATLAB will automatically turn the words “if” and “end” blue to designate them as markers pertaining to the code block in between them. MATLAB first evaluates the logical expression to the right of “if.” If the statement is true, the code block between “if” and “end” is run as if the conditional statement did not exist. Otherwise, MATLAB jumps from “if” to “end” and ignores the code in between.

More complicated conditionals can be created by adding the keywords “elseif” and “else.” Elseif statements function like extra if expressions (logical1 was false, but maybe logical2 is true). You can have as many elseif statements as you want but each must contain its own corresponding code block. If you find that multiple statements contain the same code block, you should consider combining them into one logical statement using or (|).

An else statement, if you use one, must be the final statement in your overall conditional. It functions like an elseif statement that is automatically evaluated as true. However, an else statement will by definition only be run if all preceding conditional statements are evaluated as false.

The order of conditional statements is extremely important because MATLAB will run only the code block corresponding to the FIRST true logical expression. Thus, it will not evaluate any expressions following the first true one and will skip to “end” instead.

Finally, EVERY conditional statement (if or switch) must have a corresponding “end.” Failing to include enough “end”s will cause an error. MATLAB automatically aligns any “end” you type with the nearest designated conditional statement.

```
If <logical expression1>
    <code block1>
elseif <logical expression2>
    <code block2>
```

```
elseif <logical expression3>
    <code block3>
else
    <code block4>
end
```

Nested Conditionals:

Believe it or not, you can actually place conditionals INSIDE other conditionals ☺. Sometimes, nesting conditionals is the only way to solve a particular problem. More frequently, however, you can use it to make your code more legible, pretty, and/or cool. Consider the following example:

Write a function that takes in a vector and checks to see if all its elements are perfect squares between 6 and 23 that are not divisible by four. The function should output a logical true or false.

```
function out= Swordfish(vec)
    out = false;
    if all(mod(sqrt(vec),1)==0)
        if all(vec>6)
            if all(vec<23)
                if all(mod(vec,4)~=0)
                    out = true;
                end
            end
        end
    end
end
end
```

As you can see, this code is much easier to understand than its one-line equivalent. Notice that a common strategy is to arbitrarily initialize the output variable as one possibility and change it depending on the result of the if statements; doing so ensure that your output variable will always be defined, thereby reducing the chance of your code producing an error.

Switch Statements:

Switch statements are if statements that determine the output based on the value stored in a defined variable. The terminology is slightly different for no necessary reason. “case” now replaces “elseif,” and “otherwise” replaces “else.” Once again, you can have as many cases as you want. The first line always reads “switch” followed by the name of the variable you are “switching” over.

```
switch <variable>
    case <possible value>
        <code block>
    case <possible value>
        <code block>
    otherwise
        <code block>
end
```

For instance, if you have a string stored in the variable “str,” you can use a switch statement to output whether it is four, five, or six.

```
switch num
    case 'four'
        out = 4;
    case 'five'
        out = 5;
    case 'six'
        out = 6;
    otherwise
        out = 'Pharaoh';
end
```

Seriously, folks, conditionals are the easiest thing in the world to write. If you encounter a difficult problem involving conditionals, the actual difficulty will come from the other elements and not from forming the conditionals themselves.



Section Seven: Iteration

Iteration is a fancy MATLAB word for loops, which is a fancy computing word for running a single code block multiple times. Loops are reasonably difficult to understand because, unlike many other coding tools, they are not intuitive. Whereas you would tend to focus on the whole problem to find all solutions at once, MATLAB can look at every possible solution individually.

Consider a problem asking you to solve a fourth-order polynomial equation for all real zeros. One approach would be to algebraically or graphically find the solutions (all at once); another approach would be to try plug numbers into the equation until you find all four solutions. Loops allow you to implement the latter method, typically called a “brute force” approach.

There are two types of loops, for loops and while loops. Technically, for loops are unnecessary because all for loops can be rewritten as while loops. Paradoxically, students almost never resort to using while loops because they are slightly more difficult to understand.

For Loops:

All for loops are written in the following basic format:

```
for i = <vector>
    <code block>
end
```

When MATLAB reaches the loop, it will automatically set *i* equal to the first value in the vector, henceforth called the iteration parameter. When it reaches the end of the loop, namely the word “end”, MATLAB returns to the first line and changes the value of *i* to the next value in the vector. This process continues until there are no values remaining for *i* to assume, at which point MATLAB terminates the loop and reads the rest of your code.

Determining the first line is generally the most complicated part of writing a loop. The parameter you set determines how many times the loop will run. Thus, the primary drawback of using for loops is that you must be able to predetermine how many times to iterate. You will generally use the variable *i* as an index that changes for each iteration. There are two basic strategies:

1. Set *i* equal to a range of values using the colon operator.
for *i* = 1:10
for *i* = 1:length(variable)
2. Set *i* equal to a predetermined or inputted vector. People usually forget that the vector can be ANYTHING, not just a range of values with step size 1. MATLAB will still iterate a number of times equal to the length of the vector, but *i* will automatically assume values from the

vector itself rather than indices.
for i = vec

I always use the variable i when implementing for loops because that's what my TA used. However, use whatever variable system makes sense to you. The best way to learn loops is by looking at examples.

Write a function `sumVec` that takes in a vector and outputs the sum of its elements using iteration.

```
function out = sumVec (vec)
    out = 0;
    for i = 1:length(vec)
        out = out + vec(i)
    end
end
```

However, this solution has an unnecessary complication. I deliberately ignored the more sensible approach of simply setting i equal to the vector itself.

```
function out = sumVec (vec)
    out = 0;
    for i = vec
        out = out + i
    end
end
```

Write a function `compareVec` that takes two vectors and outputs a vector of all elements that appear in both vectors in the same position with the same value.

Any problem that asks you to build an output vector inside a loop will have a solution in the following format:

```
out = [];
for i = <vector>
    if <logical expression>
        out(end+1) = <whatever your output should contain>
    end
end
```

The idea is to initialize the output as an empty vector OUTSIDE the loop and build it element-by-element INSIDE the loop. By indexing the vector at position `end+1`, we continually increase the size of the output vector by adding elements onto the end. A simple assignment statement (`out = i`) would overwrite the current output variable every time the loop is run. Returning to the `compareVec` problem:


```

function out = compareVec (vec1, vec2)
    out = [];
    L = length(vec1);
    if length(vec2) < length(vec1)
        L = length(vec2);
    end
    % We must first determine the length of the shorter vector so we know how
    %many times to iterate.
    %Further iteration will result in indexing out-of-bounds.
    for i = 1:L
        if vec1(i)==vec2(i)
            out(end+1) = vec1(i);
        %We could also tack on values to the end by concatenating the current vector
        %with the new value.
        %out = [out, vec1(i)];
    end
end
end

```

Loops placed inside other loops are called nested loops. Nesting loops allows you to iterate over multiple parameters simultaneously. As a final example, we will consider a case that cannot be solved with only a single loop.

Write a function `compStr` that takes in two strings of equal length and returns the largest set of consecutive letters that appear in both strings at the same position, disregarding case.

```
compStr('henceforth', 'heraldrth') = 'rth'
```

```

function out = compStr(str1,str2)
    out = '';
    %Once again we're building an output element-by-element, so we have to
    initialize it OUTSIDE the loop.
    for i = 1:length(str1)
        for j = i:length(str2)
            %If j is less than i, MATLAB will waste time comparing nonexistent ranges
            like str(4:3). Since str1 and str2
            %must be of equal length, we could have used the length of str1 instead.
            if strcmpi(str1(i:j), str2(i:j)) && j-i+1 > length(out)
                %We compare each range of values from str1 with the equivalent range from
                str2. Unless the match is
                %longer than the ones we've already found, it is discounted.
                out = str1(i:j)
            end
        end
    end
end

```

```
    end
end
```

While Loops:

All while loops look like this:

```
while <logical expression>
    <code block>
end
```

MATLAB first evaluates the logical expression and runs the loop until the expression becomes false. Thus, you must have something inside the loop that changes the outcome of the logical expression. Otherwise, MATLAB will enter an infinite loop that can only be terminated using control+c.

Although there is no iteration variable associated with while loops, all for loops can be rewritten as while loops using a counter variable inside the loop. The idea is to increase the counter's value by one each time the loop is run. Let's rewrite the sumVec function as a while loop.

```
function out = sumVec (vec)
    num = 1;
    out = 0;
    while num <= length(vec)
%Don't forget to include the equals sign!
        out = out + vec(num);
        num = num + 1;
%This line moves the while loop toward the terminating condition.
    end
end
```

However, while loops are usually used for loops in which the number of iterations is impossible to evaluate beforehand.

Write a function Gideon that takes in a number. If the number is even, divide it by two; if it is odd, multiply it by three and subtract one. The function should repeat this process until the number equals one (this always happens) and outputs the number of iterations required.

```
function out = Gideon (num)
    out = 1;
    while num ~= 1
        if mod(num,2)==0
            num = num/2;
        else
```

```
        num = num*3 - 1;
    end
    out = out + 1;
end
end
```

In more colloquial terms, loops are “dead useful.” You can solve almost any problem with a well-implemented for loop. Make sure you understand loops, because almost every problem from now on will involve them.

Section Eight: Cell Arrays

Cell arrays are another data type MATLAB utilizes to store information. Despite being intimidating for most people, cell arrays are easy to work with as long as you UNDERSTAND what they are. Seriously, you don't want to end up guessing when cell arrays appear on exams.

Cell arrays are a unique data type in that they are capable of storing data of multiple other types. Each element in a cell array is a cell, and a cell can contain anything (doubles, logicals, strings, structures, or even other cells). My shorthand for a cell array is ca.

Creating Cell Arrays:

Cell arrays are collections enclosed in curly brackets. The only difference is that elements can now be of any data type. This is the easy part.

```
ca = { [1 2 3], 'Judah', [true true false] };
```

Indexing Cell Arrays:

The complicated thing concerning cell arrays is that they can be indexed in true different ways, using parentheses () or curly brackets {}. Indexing with parentheses returns the actual element (in a cell array, a cell), whereas indexing with curly brackets returns the CONTENTS of the element. In other words, curly brackets open a cell to display what it contains; only cells can be indexed with curly brackets.

To return specific components of the contents of a cell, use curly brackets to open the cell followed by parentheses to index its contents. When cells are nested inside each other, you must use multiple sets of curly brackets to reach whatever is ultimately inside. The rule, which is well worth memorizing, is that you can only index with parentheses once, and the parentheses must come last in a set of indices; this makes sense if you pause to think about it.

```
ca = {[true false], 'Reuben', {[1 2 3], 'Gad'}};
class(ca)                ans = 'cell'
class(ca(1))             ans = 'cell'
class(ca{1})             ans = 'logical'
ca(3)                    ans = {1x2 cell}
ca{2}                    ans = 'Reuben'
ca{2}(2)                 ans = 'e'
ca{3}{1}(3)              ans = 3
```

That's pretty much it for cell arrays. Remember that you still use square brackets, not curly brackets, when concatenating and deleting cell arrays. Most of my examples will involve intricate combinations of cell arrays and loops.

When using cell arrays and especially iteration and conditionals, it is generally necessary to keep track of data types. The functions `isdouble()`, `islogical()`, `ischar()`, and `iscell()` return a logical true or false of whether the input is of the specified data type. The function `isempty()` returns true if the input is an empty string, vector, cell array, or structure array.

Write a function `Manasseh` that takes in a cell array containing vectors of equal length and meshes them together into one long output vector. The first element in the output vector should be the first element of the first vector, followed by the first element of the second vector, and so on.

```
Manasseh([1 2 3] [4 5 6] [7 8 9]) ans = [1 4 7 2 5 8 3 6 9]
```

```
function out = Manasseh (ca)
%Perhaps the most intuitive way to solve this problem would be to create a
simple test case and focus %on the identity of the output in relation to the
inputted cell array. For the cell array {[1 2] [3 4] [5 6]},
%the output is [ca{1}(1), ca{2}(1), ca{3}(1), ca{1}(2), ca{2}(2), ca{3}(2)].
It now becomes apparent that the
%first index starts at one and approaches the number of vectors, and the
second index starts at one and
%approaches the length of the vectors.
    out = [];
%The solution is deceptively simple but requires two simultaneous for loops.
    for i = 1:length(ca{1})
%The outer loop runs once for every element in the vectors (the vectors are
all of equal length).
        for j = 1:length(ca)
%The inner loop runs once for each vector in the cell array.
            out(end+1) = ca{j}(i);
        end
    end
end
```

Write a function `Mephibosheth` that takes in a string and divides it into individual words. The function should output a cell array containing one word per cell.

```
Mephibosheth('Jacob I have loved') ans = {'Jacob' 'I' 'have' 'loved'}
```

```
function out = Mephibosheth (str)
    out = {};
```

```
    while ~isempty(str)
%Iterate until there is no string remaining.
        [word, str] = strtok(str, ' ')
%Reassigning the value of str moves us toward the condition that terminates
the loop.
        out{end+1} = word;
    end
%You'll be using this code to break down text from notepad files in a few
weeks.
end
```

Section Nine: High-Level File Input/Output



File input/output, hereafter referred to as file IO, is a process by which MATLAB can read documents from other from other computer programs, manipulate them, and save them as new documents in your current directory. High-level file IO specifically involves dealing with highly organized data in notepad and Microsoft Excel. We will be working with three types of files: csv files, dlm files, and excel spreadsheets.

The primary difficulty with file IO involves conversions between different data types. If you find yourself using loops to accomplish simple conversions or comparisons, you're probably doing something wrong. There are a few easy tricks that will make these problems much, much easier.

Reading and Writing CSV Files:

CSV stand for comma-separated values. Basically, a CSV file is a Notepad .txt document containing only an array of numbers separated by commas. Any spaces and empty lines in the file are completely ignored. If the document contains anything other than doubles (negative numbers and fractions are okay) and commas, it is not a CSV file. There are two functions you need to know.

```
arr = csvread(filename)
```

The function `csvread()` takes in the filename (always a string) of a .txt CSV file in your current directory, removes the commas, and outputs the resulting array of doubles. If the array is not rectangular, it fills any unused space with zeros.

For example, `csvread` will turn the following file into the resultant array:

1,2,3
4,5,6
7

 \rightarrow

1	2	3
4	5	6
7	0	0

```
csvwrite(filename, array)
```

The other significant function is `csvwrite()`, which does exactly the opposite. It takes in an array of doubles and saves the corresponding CSV document as a file in your current directory. Please note that `csvwrite()`, and any writing function for that matter, does not have any output; you should not suppress it with a semicolon or set it equal to anything.

Reading and Writing DLM Files:

A dlm file is exactly the same as a csv file except that the doubles can be separated by any delimiter rather than just commas. Delimiters can be characters such as exclamation points, letters, or even numbers. Remember that the delimiter is always considered a string even if it is a number. Once again, there are two functions you need to know:

```
arr = dlmread(filename, delimiter)
```

Unlike with csv files, MATLAB does not automatically know what character is being used as the delimiter in a text file; you will therefore need to specify the delimiter in string format as a second input. If you don't feel like including the second input, MATLAB will attempt to guess the delimiter for you; it likes to choose things like commas and spaces. You can also write delimited files just like csv files; the only difference is that you must now specify the delimiter you want MATLAB to use.

```
dlmwrite(filename, array, delimiter)
```

Reading and Writing Excel Spreadsheets:

Performing tasks on excel spreadsheets is easily the most useful application of high level file IO, which is why most homework and test problems focus on this topic specifically.

```
[num, text, raw] = xlsread(filename)
```

The function to read in excel documents has three outputs instead of one. Although the names you assign them do not matter, the standard convention in CS 1371 is to use the names num, text, and raw. Obviously, MATLAB will always produce the same outputs in the same order, so knowing the order of the outputs is very important.

The first output, num, finds all cells in the spreadsheet that contain numbers and turns them into an array of doubles, much like csvread(). MATLAB finds the smallest rectangle of cells that includes all numbers in the spreadsheet and forms them into an array. If any cells in the rectangle contain non-numbers or are empty, MATLAB pads the empty space with NaN, which stands for "Not a Number."

The second output, text, finds all the words (non-numbers) in the spreadsheet and formats them into a cell array of strings in the same way, creating the smallest rectangle possible. Empty cells and cells containing numbers are padded with empty strings.

The third output, raw, takes the entire spreadsheet and turns it into a cell array. Empty spaces are padded with cells containing NaN.

The most important things to remember are the respective data types of the three outputs. Num is an array of doubles, whereas text and raw are both cell arrays. When performing numerical calculations, it is often easier to use num; the drawback is that num is not necessarily the same size as the spreadsheet itself. Raw is by definition the same size but carries the drawback of being a cell array. When solving problems, choose whichever output seems easiest for you.

Given the spreadsheet “people.xls”

	A	B	C
1	Name	Age	Major
2	Kevin	19	ChBE
3	Lucy	20	Management
4	Brandon	20	ME
5			

```
[num, text, raw] = xlsread('people.xls')
```

```
num =
    19
    20
    20
```

```
text =
    'Name'    'Age'    'Major'
    'Kevin'   ''       'ChBE'
    'Lucy'    ''       'Management'
    'Brandon' ''       'ME'
```

```
raw =
    'Name'    'Age'    'Major'
    'Kevin'   [19]     'ChBE'
    'Lucy'    [20]     'Management'
    'Brandon' [20]     'ME'
```

You can also write arrays and cell arrays into excel spreadsheets. The function name is pretty obvious.

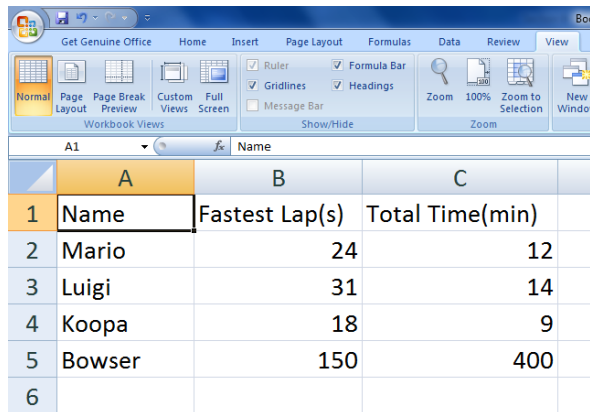
```
xlswrite(filename, arr)
```

And that’s basically it for high level file IO. I’ll quickly recap the six functions before moving onto example problems.

```
arr = csvread(filename)
csvwrite(filename, arr)
arr = dlmread(filename, delimiter)
dlmwrite(filename, arr, delimiter)
[num, text, raw] = xlsread(filename)
xlswrite(filename, arr)
```

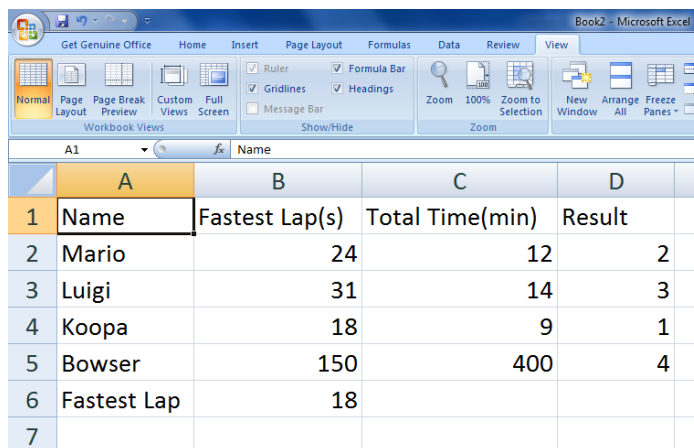
Write a function `Haggai` that takes in the name of an excel document containing information about a footrace in the following format. The document may have any number of rows but will always contain

three columns with the following column headers:



	A	B	C
1	Name	Fastest Lap(s)	Total Time(min)
2	Mario	24	12
3	Luigi	31	14
4	Koopa	18	9
5	Bowser	150	400
6			

The function should read in the spreadsheet and add an additional column called “Result” as the fourth column. This column contains the final position of each competitor based on his total time. The function should also add an additional row to the end of the spreadsheet. The first column in this row will be the string “Fastest Lap”, and the second column will contain the overall fastest lap in the race. The function should write the resultant cell array to an excel file. The filename should be the input filename with “_edited” appended to the end. The final output will look like:



	A	B	C	D
1	Name	Fastest Lap(s)	Total Time(min)	Result
2	Mario	24	12	2
3	Luigi	31	14	3
4	Koopa	18	9	1
5	Bowser	150	400	4
6	Fastest Lap	18		
7				

```
function Haggai(fn)
    [num, ~, raw] = xlsread(fn);
    % Reading in the file is usually the first step.
    totals = num(:,end);
    %For the first step, we're only interested in the total time of each
    %competitor. This is a race, so lower
    % total times will receive lower position numbers. To achieve this, we'll use
    %the second output of the
    %sort() function, which is the indices used to sort the input.
    [~, rank] = sort(totals);
    %Unfortunately, a simple sorting method doesn't quite work with ranking
    %people, so we have to call the
    %sort function AGAIN on the indices. I'll overwrite both variables from the
    %previous function since we %won't be using them anymore.
```

```

    [~, rank] = sort(rank);
%Now we have a column vector called index that contains the final rank of
%each competitor. To
%concatenate this onto the raw spreadsheet data, we have to convert it to
%type cell.
%index = {index} will put the entire vector inside a SINGLE cell, which is
%unfortunately not what we want.
%We need to create a vector of cells, each containing one number from our
%index vector. Luckily,
%there's a MATLAB function that does just that.
    rank = num2cell(rank);
%Now we need to concatenate rank onto the string 'Result'. Since we're
%concatenating cells, both
%elements must be of type cell. We therefore enclose the string in curly
%brackets before concatenating.
    column = [{'Result'}; rank];
%We're finished with the last column, so we can add it onto the raw
%spreadsheet data.
    raw = [raw, column];
%Now we just have to determine the fastest lap and perform the same
%algorithm. Remember that we're
%adding a row instead of a column, so horizontal and vertical concatenations
%will be reversed.
    fastest_lap = min(num(:,1));
%We can't concatenate this time because the dimensions are not consistent, so
%we'll use MATLAB's
%indexing out of bounds trick. MATLAB will pad the remaining spaces with
%empty cells.
    raw(end+1,1) = {'Fastest Lap'};
    raw(end,2) = {fastest_lap};
%Now all we have to worry about is writing the array to the correct filename.
%Simply appending
%"_edited" onto the end will produce something like "race.xls_edited", which
%is not a valid filename.
%The ".xls" part must come last. There are probably a dozen ways to do this.
%Here's an elegant one.
    fn2 = sprintf('%s_edited.xls', strtok(fn, '.'));
%You can also use find() to locate the period.
    xlswrite(fn2, raw)
end

```

Useful Functions:

- num2cell(arr)—converts an array of doubles into a cell array where each cell contains one double
- cell2mat(ca)—converts an array of cells containing doubles into an array of doubles
- [sort, index] = sort(vec)—always important for sorting

Section 10: Low-Level File Input/Output

Obviously, not all data are organized into csv files, dlm files, and excel spreadsheets. In particular, MATLAB is useful for manipulating text documents in notepad, hereafter called low-level file IO. Because low-level file IO involves looping through documents one line at a time, it is much more confusing than high-level file IO. I will outline a basic procedure for any coding problems you may encounter.

1. Use `fopen()` to read in the .txt file. You may need to use `fopen()` to create an output file as well.
2. Use `fgetl()` or `fgets()` to read in the first line of the document.
3. Create a while loop that runs as long as the line is of type char.
 - a. You may wish to use `strtok()` and another while loop to pull out each individual word.
 - b. Use `fgetl()` or `fgets()` to read in the next line of the document.
 - c. Somewhere in the while loops, use conditionals to implement whatever changes you need.
 - d. Depending on your loop, you may need to use a loop-and-a-half.
4. Once you have completed your output, `fclose()` all documents.

Just like most MATLAB concepts, understanding the functions and concepts is easy; the difficult part is using the correct loops and conditionals in the correct places. For this reason, tracing problems can also be fairly annoying.

Fopen()

The first step to any problem is to actually open the file you're interested in. You can assume that all files are .txt documents written in notepad.

```
fh = fopen(filename, permission)
```

Once again, the filename is always a string with .txt at the end. The output for `fopen()` is called a file handle (please take note of my abbreviation "fh"). A good pictorial representation of a file handle is an arrow that points to a particular line in the document; the file handle is initialized at the first line of the document and moves down one line at a time until it reaches the end. The file handle is always a double. Although you can't directly do anything with the file handle itself, it becomes the input for other file IO functions that are useful.

The second input for `fopen()` is a permission statement, a string representing what you're allowing MATLAB to do with the file. For instance, MATLAB cannot alter previously existing documents unless you give it a "write" permission. The three most common permissions are 'r', 'w', and 'a' (read, write, and append).

The read permission allows you to read in a file but not change it, basically for extracting data to perform other functions on. Write allows you to overwrite an existing document (useless) or create an

entirely new blank document. If you specify the write permission and the filename does not exist in your current directory, MATLAB will automatically create a blank notepad file along with its respective file handle. The append permission allows you to add more information onto the end of an existing file. Fortunately, you'll probably never be appending anything in this class, so don't worry about it.

There is one interesting error you may encounter when using `fopen()`. If a call to the file handle later in your script produces an error like "Error using `fgetl`: invalid file identifier", `fopen()` probably failed to open the input file. This tends to happen only when using the write permission to create a new file and is caused by some sort of MATLAB permission error. MATLAB will continue reading your code but will set the file handle equal to -1 to indicate a failure in opening the file. If you encounter this error, change your current directory to something else and then change it back; this will reset MATLAB's permissions.

```
fh = fopen(fn, 'r')
fh = fopen(fn, 'w')
fh = fopen(fn, 'a')
```

Fgetl() and Fgets()

As I stated earlier, the file handle is actually relatively useless; you need a different function to move through the document and take out each line. The `fgetl()` and `fgets()` function return the line the file handle is currently pointing to, although in slightly different ways. Please note my shorthand "line" to represent a line of text.

```
line = fgetl(fh)
line = fgets(fh)
```

These two functions output the current line (whichever one the file handle is pointing to) as a string. Each time you call either of these functions, the file handle moves down to the next line in the document, so you can actually move through the entire document simply by running the same line of code over and over.

Because of this fact, a simple while loop could easily pull out every line from a particular document. When the file handle has moved down far enough that there are no new lines remaining in the document, `fgetl()` and `fgets()` output a -1 instead of a line. Because this -1 is a double rather than a string, you can set your while loop to run as long as the current line is of type `char`.

```
while ischar(line)
    <code block>
end
```

There is only one noticeable difference between `fgetl()` and `fgets()`. Each line of text in a document ends with a new line character: `\n`. This symbol tells computer programs to skip to the next

line on the page before continuing. Although the new line character is usually invisible (it's not really text), it is inherently there.

`Fgetl()` removes the new line character from the output and returns only the relevant text as a character string. `Fgets()` returns the entire line including the new line character. `Fgetl()` is more useful if you are only concerned with interpreting data in a notepad file; `fgets()` is more useful when you are copying data to a different file because it automatically copies new line characters as well.

Fprintf()

The `fprintf()` function allows you to print text to the command screen or a notepad file. The word print may be confusing if you are unused to it. To see an example of printed text, type a line of code without suppressing the output; the result of your code will be "printed" in the command window. `Fprintf()` does not have an output.

```
fprintf(filename, str, var1, var2,...)
```

Like `sprintf()`, `fprintf()` can be used to print formatted strings, or strings containing variables. Simply replace the doubles and strings in the formatted string with `%d` and `%s`, respectively. If you don't specify the file handle of the document you wish to print to, MATLAB will print to the command window. To utilize tabs and new lines, print `'\t'` and `'\n'`, respectively.

In the following examples, we're printing to a notepad file called "Rahab.txt".

<code>fh = fopen('Rahab.txt', 'w')</code>	
<code>fprintf(fh, '%s', 'Mordecai')</code>	prints 'Mordecai'
<code>fprintf(fh, '%s\n', 'Mordecai')</code>	prints 'Mordecai' and new line character
<code>fprintf(fh, '\n')</code>	prints new line character

Strtok Loops

File IO coding problems frequently entail analyzing every individual word of a document. This task is most easily accomplished using `strtok()` and a while loop, with a space as the delimiter. The loop should continue until the remainder of the string, the second output of `strtok()`, no longer contains anything.

```
line = 'Jeroboam'
while ~isempty(line)
    [word, line] = strtok(line, ' ')
    <code block>
end
```

Initializing “line” is perhaps the easier method to understand. The other option is to actually run `strtok()` or `fgetl()` before entering the respective while loop. Note also that overwriting “line” instead of using a different variable name allows us to reduce our number of `strtok()` calls by one.

Loop-and-a-Half:

Depending on how you structure your while loops, it may be necessary to implement a loop-and-a-half at some point. Certain problems can occur if the terminating condition has been met while you still have data left to use. In, in our above example with `strtok()`, we decided to have the code block ABOVE the `strtok` call, MATLAB would pull out the final word of the line (making “rest” empty) and fail to reenter the loop. Adding a simple conditional statement fixes this problem.

```
line = 'Jeroboam'
while ~isempty(line)
    <code block>
    [word, line] = strtok(line)
    if isempty(line)
        <code block>
    end
end
end
```

Fclose()

The `fclose()` function closes documents you’ve opened. Having a bunch of extraneous documents open will tend to slow down MATLAB and is considered sloppy (that means you lose points). `Fclose()` does not have an output.

fclose(fh)

`Fclose()` is easy to use and even to forget about. Remember, `fclose()` is always called on the file handle, not the filename; this is a very common mistake. Let’s work an example problem to see how all these elements combine.

Write a function `Elisha` that takes in the filename of a `.txt` document. The function should output the number of words in the document (anything separated by spaces counts as a word). The input file contains a space after the last word of each line. `Elisha` should create a new file with these extraneous spaces removed. The filename of this new file should be the original filename followed by “_edited”.

```
function num = Elisha(fn)
```

```

fh = fopen(fn, 'r');
fn2 = [strtok(fn, '.') '_edited.txt'];
fh2 = fopen(fn2, 'w');
num = 0;
%We now have both files open and two respective file handles. We have also
%initialized our output.
line = fgetl(fh);
while ischar(line)
    while ~all(line==32)
%Since we're also eliminating spaces, we use a slightly different terminating
%condition.
        [word, line] = strtok(line);
        num = num + 1;
        if all(line==32)
            fprintf(fh2, word);
        else
            fprintf(fh2, [word 32]);
        end
    end
    line = fgetl(fh);
    if ischar(line)
        fprintf(fh2, '\n');
    end
end
end
end

```


Section 11: Structure Arrays



For most people, structures constitute the most difficult MATLAB concept. Structures and structure arrays are merely another useful method for organizing information. Much like cell arrays, structure arrays can be of any length and are capable of storing every data type, including other structures.

A structure is basically a list of information stored in different headings, termed fields. Each field is a string capable of storing one piece of information (double, logical, string, cell, structure, etc.) per structure. Fieldnames are case-sensitive, just like everything else.

Creating Structures Manually:

There are two methods for creating structures. The first is to assign each datum manually using the dot (.) operator. Structure assignments require three inputs: the name of the structure, a fieldname, and something to assign to it, organized in the following fashion using the dot.

```
structure.field = something
```

If the structure does not already exist, MATLAB automatically creates one and stores it in your current workspace. If the field does not already exist, MATLAB automatically adds it to the respective structure below any fields already present. Otherwise, the structure is updated to include whatever new information you're storing in it.

My shorthand for structures is "sa", standing for "structure array."

```
sa.name = 'Kevin';  
sa.age = 19;  
sa=  
name:'Kevin'  
age:19
```

Individual structures can be concatenated in rows and columns just like arrays of doubles (homework and test problems never seem to move beyond structure vectors, fortunately). To create a second structure under the same variable name, simply add its position number in parentheses before the dot.

```
sa(x).field = something
```

Just like with vectors, you use the dot assignment to index out of bounds. MATLAB will automatically fill any in-between space with unassigned fields and structures with empty vectors.

Similarly, adding an additional field to one structure will also add it to every other one, with an empty vector for any unassigned fields.

```
sa(2).name = 'Micah';
sa(1) =
Name: 'Kevin'
Age: 19
sa(2) =
Name: 'Micah'
Age: []
```

You will frequently encounter situations where the fieldname you wish to create or access is not given but stored as a string in one of your variables. In that case, you can enclose the variable referring to the fieldname in parentheses; MATLAB interprets items in parentheses as variable names rather than direct fieldnames.

```
Joseph = 'Age';
sa.Joseph = 19
sa =
Joseph: 19
sa2.(Joseph) = 19
sa2 =
Age: 19
```

Creating structures manually is easy to remember and implement, so there obviously must be a more complicated method used to torment students on tests. That method is called the `struct()` function.

Struct():

Entire structure arrays can be created in a single line of code via the `struct()` function. It takes in any number of input fieldnames and cell arrays of data and creates the corresponding structure.

```
sa = struct('field1', {...}, 'field2', {...}, ...)
```

The number of structures in the resultant array is always equal to the length of the input cell arrays. The first structure will contain all of the fieldnames (the odd-numbered inputs) and the first element of each cell array (the even-numbered inputs). The second structure will contain all fieldnames and the second elements of each cell, and so forth.

```
sa = struct('Name', {'Kevin', 'Micah'}, 'Age', {19, 20})
sa(1) =
Name: 'Kevin'
Age: 19
sa(2) =
Name: 'Micah'
Age: 20
```

If one of the even-numbered inputs is replaced with a double, a string, a structure, or a cell containing only one value, MATLAB will place that element in every structure of the resultant array. The number of structures in the array is still determined by the length of the other cell arrays in the function call.

```

sa = struct('Name', {'Kevin', 'Micah', 'Joash', 'Abel'}, 'Age', 19)
sa(1) =          sa(2) =          sa(3) =          sa(4) =
Name:'Kevin'     Name:'Micah'     Name:'Joash'    Name:'Abel'
Age:19           Age:19           Age:19          Age:19

sa = struct('Name', {'Kevin'}, 'Age', {19, 20})
sa(1) =          sa(2) =
Name:'Kevin'     Age:19
Name:'Kevin'     Age:20

```

The `struct()` function is like the Force—you can put faith in your blasters and try to survive without it, but eventually some Jedi will show up and hack you to pieces.

Indexing Structure Arrays:

The code to pull individual elements out of structure arrays is simply the reverse of the dot (.) assignment used to create them.

```
something = sa(x).field
```

This code is only useful for pulling out individual elements from a particular structure. Fortunately, there are two other ways to index structure arrays. You can index an entire structure by leaving out the filename.

```
something = sa(x)
```

In this case, the output of the index is another structure array. Obviously, you could also index a range of structure arrays using the colon operator. Perhaps the most useful method for indexing structure arrays is to access every element in a particular field; once again, you can use parentheses for fieldnames stored as variables.

```

something = sa.field
something = sa.(field)

```

Unfortunately, indexing an entire field produces multiple outputs that MATLAB does not automatically store. It saves each value in the temporary variable “ans”, effectively overwriting all but the final element. You will therefore need to enclose the variable you’re creating in square or curly brackets to create vectors and cell arrays representing a fieldname.

```

sa = struct('Name', {'Matthew', 'Mark', 'Luke', 'John'}, 'Age', {19, 20, 21, 22});
Names = sa.Name          Names = 'John'
{Names} = sa.Name        Names = {'Matthew', 'Mark', 'Luke',
'John'}
[Ages] = sa.Age           Ages = [19, 20, 21, 22]

```

And now we’ll take a look at the bunch ‘o’ functions associated with structure arrays.

Setfield()

Manipulating elements in structure arrays is much more complicated than in vectors or cell arrays. Fieldnames serve as rather fixed entities, and there are no little tricks involving the colon operator or empty brackets available.

Setfield() takes in a structure array, the name of a field in the structure array, and a new fieldname (fieldnames are ALWAYS strings) and changes the name of the pre-existing field to the new string. You'll probably use this function once and then forget it exists.

```
sa = setfield(sa, field, new_field)
```

Rmfield()

Alternatively, you may actually want to delete a field entirely from a structure array. Once again, the only way to accomplish this is to use the built-in function. Rmfield() takes in a structure array and a field and deletes the field from the structure array.

A common mistake is to forget to set rmfield() and setfield() equal to anything; if you don't, MATLAB will store the updated structure array in the temporary "ans" variable, which is not what you want. Always remember to overwrite the structure array by putting it on BOTH sides of the equals sign.

```
sa = rmfield(sa, field)
```

Getfield()

Getfield() does the same thing as indexing a structure array by fieldname—it takes in the structure and the field and returns the corresponding element in the structure. Getfield() is useless.

```
something = getfield(sa(x), field)
```

Fieldnames()

Fieldnames() takes in a structure array and outputs a cell array containing the names of all fields in the structures. A common homework problem ploy is to not preallocate the fieldnames in the input structure array, in which case you would use fieldnames() along with indexing indirectly (with parentheses) to access the contents.

```
ca = fieldnames(sa)
```

Isfield()

Is a certain string one of the fieldnames in a structure? Isfield() returns a logical true or false.

```
logical = isfield(sa, field)
```

Istruct()

Works just like ischar(), is numeric(), and iscell(). Can you guess what it does?

```
logical = isstruct(something)
```

That's pretty much it for structure arrays, believe it or not. Just memorize all the little rules and functions because structure problems love to pop up on the final exam. I just have one theoretical discussion to add that may be useful.

Whenever you have a collection of data such as a vector, array, cell array, or structure array and you wish to pull out one element at a time (probably to perform some kind of logical or conditional statement) you have to use a series of nested for loops. The number of for loops necessary will always be equal to the number of dimensions in the data collection (one for vectors, two for array, etc.). The important thing to remember about structure arrays is that the set of fieldnames adds an extra dimension, so a one-dimensional structure array acts like a two-dimensional object.

Given a one-dimensional structure array with unknown fieldnames, how would you index each element individually?

```
fields = fieldnames(sa);  
for i = 1:length(sa)  
    for j = 1:length(ca)  
        x = sa(i).(fields{j});  
    end  
end
```

Second, remember that you can use strcmp() to compare a string to a cell array of strings to find matches. Fieldnames() outputs a cell array of strings.

Section 12: Recursion



A function can actually call itself in the same way it would call another function. When a function contains a call to itself, it is called a recursive function (the act of a function calling itself is called recursion). A recursive call causes MATLAB to open another command window called a stack frame on top of the one your function is running. Each successive call opens yet another stack frame, and the process continues until no more recursive calls are made. MATLAB then evaluates the each individual stack frame starting with the MOST RECENT ONE.

Think of a stack of papers on a table. If you set your calculus, physics, and chemistry homework on a desk and then pick them up one at a time, you'll finish your chemistry homework first because it ends up on top. This will all seem rather ethereal until we look at an example.

Unfortunately, having your function call itself imposes a few very specific limitations to avoid infinite loops. Here are three "pillars" of recursion.

1. The function calls itself.
2. The function contains a terminating condition.
3. Successive calls move the function toward the terminating condition.

And here is the basic "template" you'll be using with recursion.

```
if <terminating condition>
    <code block>
elseif <maybe another terminating condition, ad infinitum>
    <code block>
else
    <recursive call>
```

So what on earth is a terminating condition? The most severe limitation of recursion is that, if left unchecked, the function will call itself infinitely, opening an infinite number of stack frames and eventually crashing your computer. MATLAB has a built-in safeguard against this possibility; it will give you a warning and ask for your permission before opening too many stack frames at once. Never give it permission to do so.

To prevent this tragedy, you simply change the inputs of each recursive call (make them variable) in such a way that they approach a known value. Eventually, the function will reach this "terminating condition" and immediately resolve all waiting stack frames. Let's look at the famous fibonacci example to see how this works.

A fibonacci sequence is a series of numbers where each element is equal to the sum of the previous two elements. The first two numbers are arbitrarily chosen as zero and one. 0,1,1,2,3,5,8,13,21

and so forth. We want to write a recursive function that takes in a number and returns the fibonacci value at that position.

```
function out = fib (num)
    if num==1
        out = 0;
    elseif num==2
        out = 1;
    %These are our two terminating conditions. Obviously, the sequence cannot
    %contain any elements before position one.
    else
        out = fib(num-1) + fib(num-2);
    %This line constitutes the recursive call. MATLAB evaluates each recursive
    %call completely before continuing finishing the code. For higher numbers,
    %each call will open many stack frames that quickly become compounded like
    %branches of a tree. Each successive call reduces the value of the input
    %until it is equal to one or two (the terminating condition).
    end
end
```

If you attempt to use this function to evaluate a moderately high number like 50, you will find that MATLAB takes all eternity to return the answer—it would almost be faster to evaluate the sequence by hand. Why? The enormous branches of recursive calls actually end up evaluating the same numbers over and over again. Since MATLAB doesn't inherently save the value for fib(20), it has to reevaluate it each time it is prompted to do so.

Thus, the other major limitation of recursion is that it becomes pointlessly redundant for all but the simplest tasks. Why, then, would you use recursion for a problem such as this? You wouldn't.

The only reason you learn recursion in this class is so you can be tested on it. On rare occasions, recursion is the only way to efficiently solve a problem, but for something like this you will always be better off using iteration instead.

As a final note and warning, consider what would happen if anything but a positive integer were inputted into the above function. It would run forever.

The fibonacci function involved approaching the terminating condition by reducing the value of the input. Let's look at an example involving a different type of terminating condition.

Write a function recursiveSum that takes in a vector and returns the sum of its elements using recursion.

```
function out = recursiveSum(vec)
    if length(vec)==1
        out = vec;
    else
        out = vec(1) + recursiveSum(vec(2:end));
    end
end
```

Once again, we only used recursion because the direction told us to. Any sane person would have simply used the `sum()` function. This time, we approached the terminating condition by reducing the length of our input but leaving the values unchanged.

Finally, some problems may ask you to keep track of the number of recursive used to obtain the solution.

Write a function `recursiveLength` that takes in a vector and returns its length using recursion.

```
function out = recursiveLength(vec)
    if length(vec)==0
        out = 0;
    elseif length(vec)==1
        out = 1;
    else
        out = 1 + recursiveLength(vec(1:end-1));
    end
end
```

The fact that using the `length()` function was necessary is a testament to the uselessness of recursion.

Perhaps the only task for which recursion is absolutely necessary involves the variability of cells and structures. The variable could potentially contain any number of cells and structures contained within other cells and structures. Thus, we would use recursion to “unwrap” the contents of the variable. Let’s take cells as an example.

Write a function `unwrapCell` that takes in a one-dimensional cell array and locates every non-cell element. The function should print the value of all non-cell elements to the command window, separated by spaces. The cells will not contain structures but may contain other cells.

```
function unwrapCell(ca)
    for i = 1:length(ca)
        if iscell(ca(i))
            unwrapCell(ca(i))
        elseif ~isempty(ca(i))
            fprintf(ca(i))
        end
    end
end
```

Finally, the TAs may for some reason decide to assign a drill problem that is unsolvable the way it is written. Remember that the ONLY way to approach a terminating condition during recursive calls is to vary the inputs of the function. If you need to change a variable that is not an input to the function in order to approach the terminating condition, the problem is unsolvable. Simply write a helper function for your function that uses more inputs than the original.



Section 13: Plotting

Compared to the concepts we've been dealing with recently, plotting is both fun and easy. While MATLAB has the capability to produce almost any conceivable kind of graph or chart, CS 1371 seems to be concerned only with basic two-dimensional graphs and three-dimensional surfaces. The course schedule sometimes presents these as two separate sections, but for clarity we will keep it as one guide.

Basic plotting presents no difficult or theoretical concepts. All you have to do is memorize a new slew of functions and then use the knowledge you already have to determine the appropriate inputs to produce the desired graph.

Recall from the discussion on functions that some functions can be run without specifying any output. Plotting problems frequently make use of this nuance by requiring you to produce a plot rather than return an output. Therefore, you need to remember this function header template:

```
function name_of_function (input1, input2, input3)
```

Plot()

Strangely enough, MATLAB's plotting function is actually called `plot()`. `Plot()` takes in a vector of x-data, a vector of y-data, and an optional character string; it produces (not outputs) a two-dimensional xy plot of the input data. Each point is automatically connected to the previous one with a straight line.

MATLAB's default plot utilizes a solid blue line. However, you can change this by adding a character string as a third input to the plot function. The first character specifies a color, the next character specifies a plot symbol, and the remainder of the string specifies a line type (solid line, dashed line, etc.). Here is a chart of relevant symbols for you to peruse:

b	blue	.	point	-	solid
g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	--	dashed
m	magenta	*	star		
y	yellow	s	square		
k	black	d	diamond		
w	white				

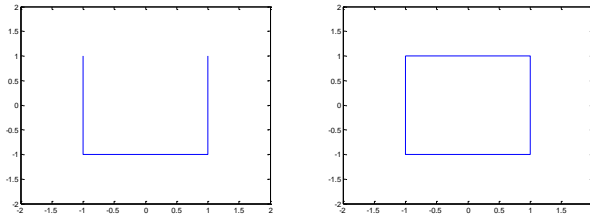
```
plot(x,y,'b*')
plot(x,y,'k-.' )
plot(x,y,'wd--')
```

When creating closed shapes, remember to repeat the first set of coordinates; otherwise, MATLAB will not know to draw a final line back to it. For example,

```
plot([-1, -1, 1, 1], [1, -1, -1, 1])
```

plots only three sides of the unit square—the top is not connected. We would need to add a fifth coordinate identical to the first.

```
plot([-1, -1, 1, 1, -1], [1, -1, -1, 1, 1])
```

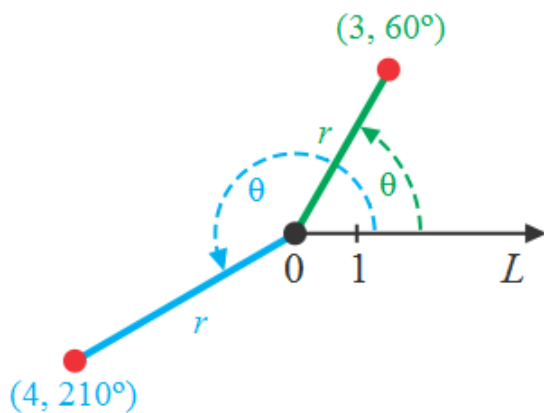


Polar Coordinates:

Plotting with polar coordinates is a technique used to draw circle-like shapes. Drawing a circle by plotting individual x-y coordinates would be difficult at best, so we turn 'x' and 'y' into functions of 'r' and 'theta' (radius and angle) using the following conversions:

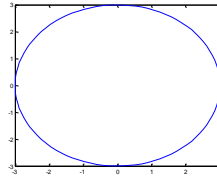
```
x = r.*cos(theta)
y = r.*sin(theta)
```

By simply adjusting the radius and angle from the origin, we can “sweep out” any circle-like area on the xy plane. Here’s a picture from Wikipedia.



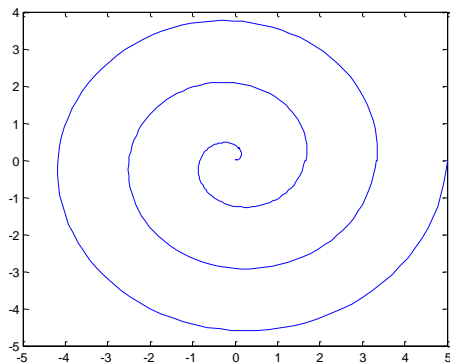
So, to plot an ordinary circle of radius three, we would set 'r' equal to three and allow 'theta' to range from 0 to 2π . `linspace()` is particularly useful here for generating a bunch of points, namely 100.

```
r = 3;
theta = linspace(0,2*pi);
plot(r.*cos(theta), r.*sin(theta))
```



By setting 'r' equal to a range of values as well, we can make interesting two-dimensional spiral shapes. Here we'll let 'theta' repeat three times to ensure multiple revolutions.

```
r = linspace(0,5,300);
theta = linspace(0, 2*pi);
theta = [theta theta theta];
plot(r.*cos(theta), r.*sin(theta))
```



When plotting two-dimensional shapes, just be aware of the geometry involved and use polar coordinates when necessary.

Figure Functions:

There are numerous functions MATLAB uses to adjust axes and label figures. Just know them, and please remember that many of the inputs MUST BE STRINGS.

Plot labeling

```
title(string) %puts a title on the plot
xlabel(string) %labels the x-axis
ylabel(string) %labels the y-axis
zlabel(string) %yes, we will be using the z-axis as well
legend(string) %you probably won't ever use this
```

Axis adjusting

```
axis square %makes the current axis box square in size
axis tight %makes the smallest possible axes to display the plot
axis equal %makes the axes of equal length. There are other axis options, but
they don't matter.
grid on %turns on the grid on your current plot
```

CLF:

In the same way that 'clc' clears the command window, 'clf' clears the current figure.

Hold on

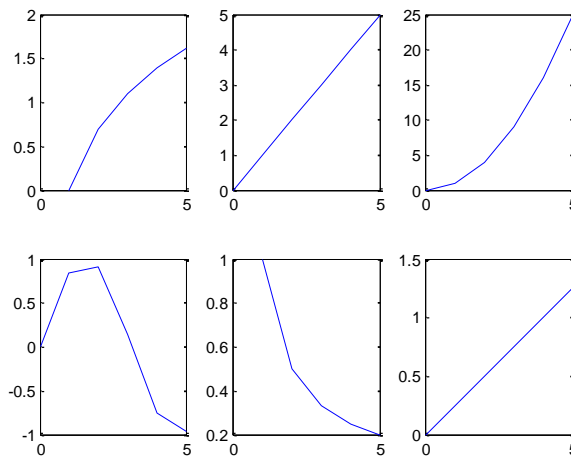
Yes, this is actually a function. Unfortunately, MATLAB automatically overwrites your current plot whenever you try to plot something else. To prevent this, you must toggle a command called 'hold'. 'Hold on' allows you to add more plots to the same figure, while 'hold off' resets MATLAB's default behavior. This is EASY to forget on tests.

Subplot()

If you want to place multiple individual plots in the same figure, you can use subplot() to organize them into a grid. The inputs are the number of rows in the resulting figure, the number of columns in the resulting figure, and the individual plot you wish to modify.

Unlike when indexing arrays, MATLAB reads individual plots from left to right, like a book. As an example, we'll make six plots in one 2x3 figure. Subplot() must be called once for every new plot.

```
x = 0:5;
subplot(2,3,1)
plot(x,log(x))
subplot(2,3,2)
plot(x,x)
subplot(2,3,3)
plot(x,x.^2)
subplot(2,3,4)
plot(x,sin(x))
subplot(2,3,5)
plot(x,1./x)
subplot(2,3,6)
plot(x,x./4)
```



Notice the order of the subplots:

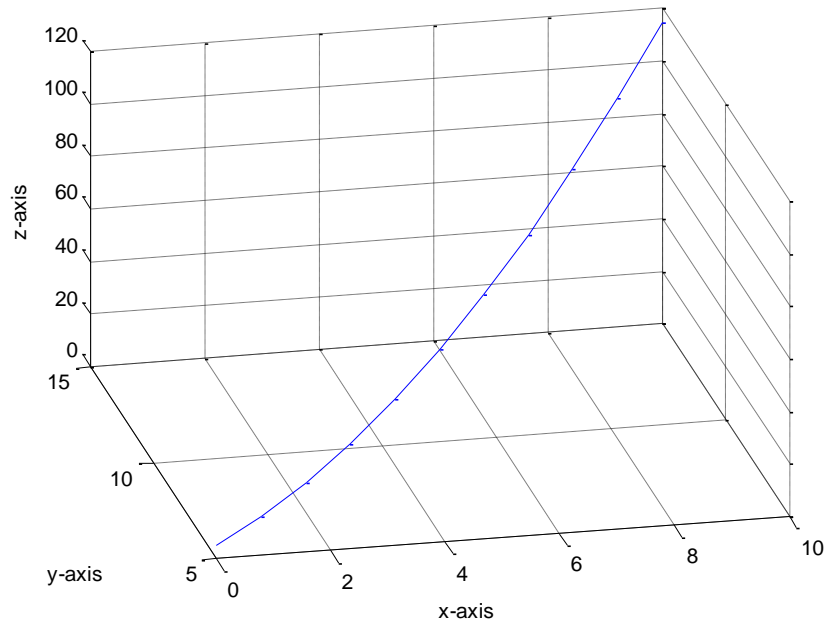
```
1 2 3
4 5 6
```

```
subplot(rows, columns, current_plot)
```

Plot3()

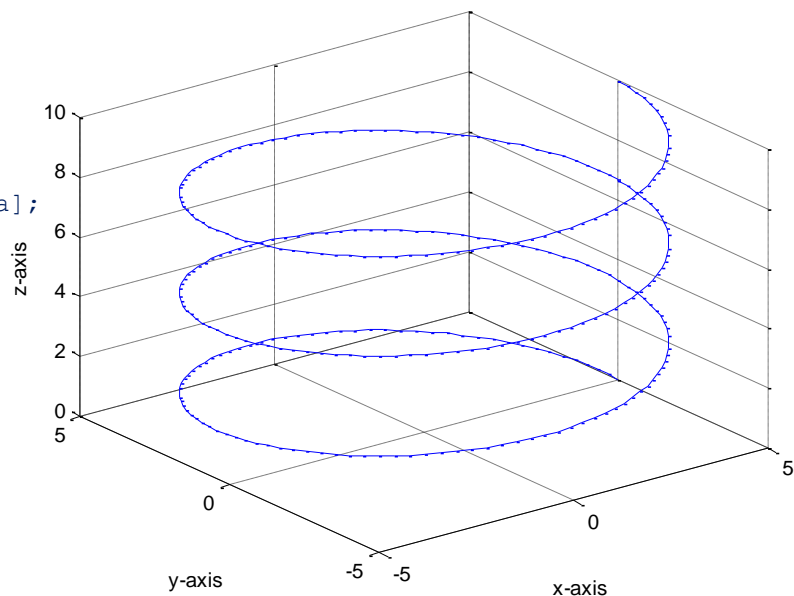
Plot3() works exactly the same way as plot(), except that it takes in three vectors and produces a three-dimensional plot. The only difference here is that we add a dimension 'z' that is a function of 'x' and 'y'. If you haven't worked with this before, don't worry—it's actually pretty simple. Just note that plot3() produces line plots, not surfaces (there are other functions for that). Let's draw some pointless three-dimensional curve:

```
x = 0:10;  
y = 5:15;  
z = x.^2 + y;  
plot3(x,y,z)  
grid on  
xlabel('x-axis')  
ylabel('y-axis')  
zlabel('z-axis')
```



Three-dimensional plotting will also work with polar coordinates. Just specify a 'z' value to correspond to the other coordinates. You can plot three-dimensional spirals by using 'x' and 'y' values for a circle and letting 'z' range slowly upward. We'll allow 'theta' to repeat again to obtain a cooler-looking spiral.

```
r = 5;  
theta = linspace(0,2*pi);  
theta = [theta theta theta];  
x = r.*cos(theta);  
y = r.*sin(theta);  
z = linspace(0,10,300);  
plot3(x,y,z)  
grid on  
xlabel('x-axis')  
ylabel('y-axis')  
zlabel('z-axis')
```



Surf() and Mesh()

One of the most powerful assets of MATLAB is its ability to produce surface plots from matrix operations. A surface plot resembles a three-dimensional line plot in that it uses 'z' as a function of 'x' and 'y'. The difference is that 'x' and 'y' are not fixed to any particular range of values; for every combination of an 'x' value and a 'y' value, there is a corresponding 'z' coordinate.

Obviously, to use plot3() with vector inputs would be nauseatingly complicated if not impossible. Luckily, MATLAB has a few built-in functions that produce surface plots using arrays as inputs; the two important functions are surf() and mesh(), which are very similar.

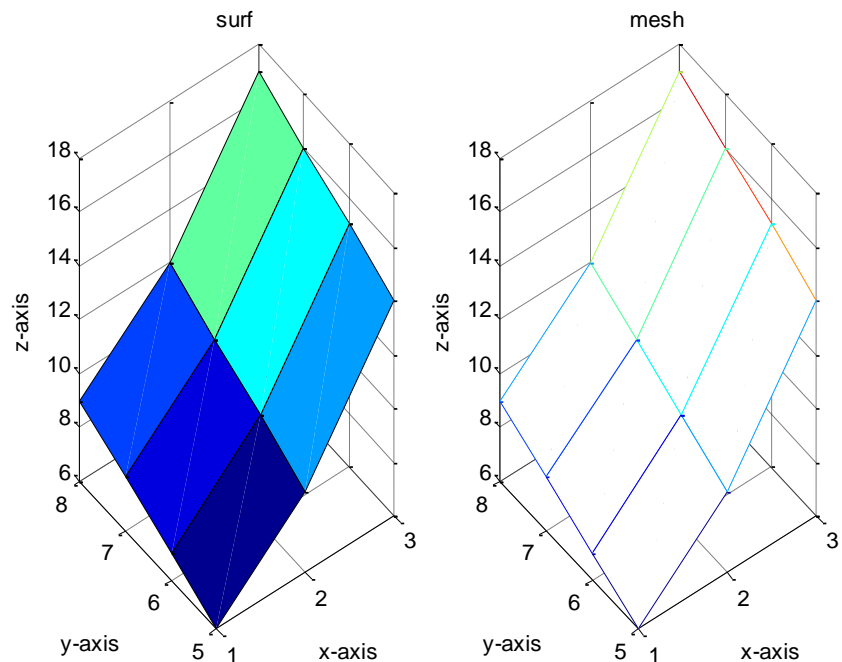
Here we introduce a new notation for function inputs: when we use matrices rather than vectors for plotting inputs, 'x' becomes 'xx', 'y' becomes 'yy', and 'z' becomes 'zz'. This notation can be useful for distinguishing between vectors and arrays produced by meshgrid().

Surf() and mesh() take in three arrays ('xx', 'yy', and 'zz') of EQUAL size and match each set of corresponding elements to produce a point. For instance, the very first element in each array becomes a single point with a unique 'xyz' value, as does each other position in the arrays. MATLAB plots the resulting surface and adds color based on the 'z' value of each point. It also interpolates to produce a reasonably smooth plot regardless of the inputs.

Surf() produces a pretty-looking plot with all interior points colored in, while mesh() uses only the original points and produces a sort of wire-frame of what the surface should actually look like. There are many other variations on surf() and mesh() but we're not really concerned with them for the purposes of this class.

```
xx = [1 2 3; 1 2 3; 1 2 3; 1 2 3];  
yy = [5 5 5; 6 6 6; 7 7 7; 8 8 8];
```

```
zz = xx.^2 + yy;  
subplot(1,2,1)  
surf(xx,yy,zz)  
title('surf')  
subplot(1,2,2)  
mesh(xx,yy,zz)  
xlabel('x-axis')  
ylabel('y-axis')  
zlabel('z-axis')
```



Meshgrid()

Next to such infamous functions as `strtok()` and `fprintf()`, `meshgrid()` is one of the most confusing functions in CS 1371. The obvious question in the previous example is how to generate arrays like 'xx' and 'yy' from 'x' and 'y' vectors. This is certainly possible but would require some complicated loops and a lot of unnecessary work. `Meshgrid()` takes in two vectors and outputs those arrays.

The vectors I used to create the arrays in the previous example were `x = 1:3` and `y = 5:8`. We can immediately see that the resulting arrays (both of the SAME size) were of dimension 4x3. The four comes from the length of the second input, while the three is the length of the first—very tricky.

`Mesgrid()` has two outputs. The first array consists of the first vector repeated along the rows once for each element in the second vector. The second array consists of the second vector repeated along the columns once for each element in the first vector. This is just something you'll have to memorize.

```
x = 1:3;
y = 5:8;
[xx,yy] = meshgrid(x,y)
    1  2  3
xx = 1  2  3
    1  2  3
    1  2  3

    5  5  5
yy = 6  6  6
    7  7  7
    8  8  8
```

Obviously, we don't need to use `meshgrid()` to create a 'zz' array since 'z' will always be a function of 'x' and 'y'.

Shading:

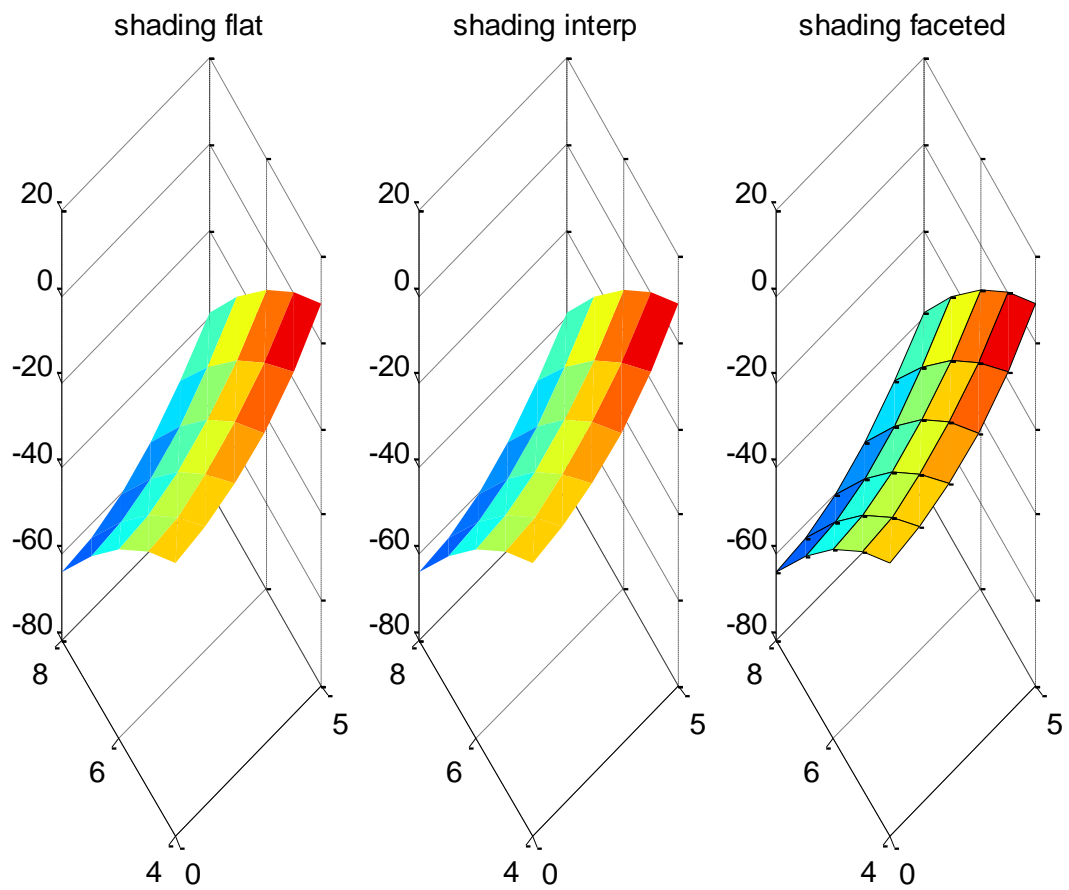
We can also adjust the manner in which MATLAB shades surace plots to make them look prettier. Shading, like 'hold on', does not require any parentheses.

```
shading flat %shades all pieces as one solid color determined by the enpoint
values
shading interp %uses linear interpolation to estimate the values of interior
points, producing a smooth shading
shading faceted %flat shading with black mesh lines to divide regions. This
is the default shading.
```

```

[xx, yy] = meshgrid(0:5,4:8);
zz = xx.^2 - yy.^2;
subplot(1,3,1)
surf(xx,yy,zz)
shading flat
title('shading flat')
subplot(1,3,2)
surf(xx,yy,zz)
shading interp
title('shading interp')
subplot(1,3,3)
surf(xx,yy,zz)
shading faceted
title('shading faceted')

```



Colormap:

Finally, we can adjust the colors MATLAB uses for shading surfaces. I will list the options below, but there is no particular need to remember them.

```
colormap hsv
colormap hot
colormap gray
colormap bone
colormap copper
colormap pink
colormap white
colormap flag
colormap lines
colormap colorcube
colormap vga
colormap jet
colormap prism
colormap cool
colormap autumn
colormap spring
colormap winter
colormap summer
```

Sorting Points:

One of CS 1371's favorite test questions involves plotting points that aren't necessary in ascending order. Remember that MATLAB only plots points in the order they are inputted. Unfortunately, calling the `sort()` function on the 'x' and 'y' values separately will mix up the points to be plotted because 'y' is a function of 'x' (not necessarily ascending). The secret is to use the second output of the `sort()` function to sort each vector in the same manner.

```
[x, index] = sort(x);
y = y(index);
```

Section 14: Bodies of Rotation

People despise bodies of rotation because, unlike regular plotting, it is not intuitive unless you REALLY understand what you're doing. We can rotate two-dimensional plots either around the origin by a fixed angle 'theta' or around an axis, creating an interesting-looking surface. The latter is actually the same "bodies of rotation" example as is used in calculus I to study integration.

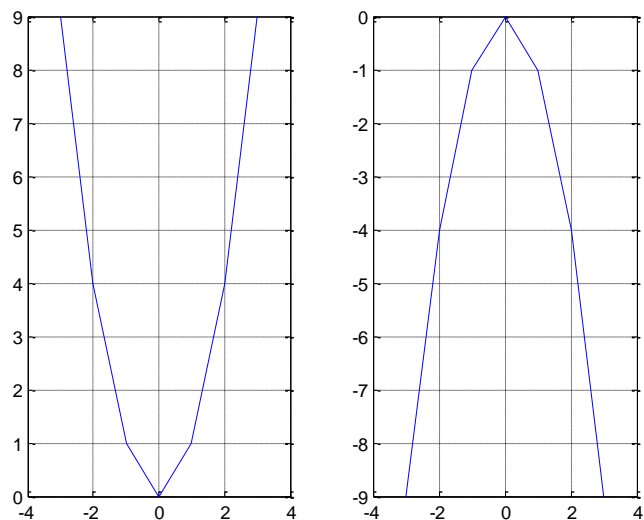
Rotation Around the Origin:

To rotate a two-dimensional plot around the origin by a fixed angle, simply matrix multiply the coordinates by the two-dimensional rotation matrix (the rotation matrix MUST be on the left). The coordinates themselves must be organized in a 2xn matrix, with a row of 'x' data on top of a row of 'y' data.

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} xdata \\ ydata \end{bmatrix} = \begin{bmatrix} rotated\ xdata \\ rotated\ ydata \end{bmatrix}$$

Note that we should NOT use the dot (.) to multiply the matrices, which would only result in an error anyway. After the multiplication step, simply plot the 'x' and 'y' data by indexing the rows of the resulting matrix. I like to call the rotation matrix 'R'. Note that 'theta' must be in radians.

```
theta = pi;
x = -3:3;
y = x.^2;
subplot(1,2,1)
plot(x,y)
grid on
R = [cos(theta) -sin(theta);...
     sin(theta)  cos(theta)];
mat = R * [x;y];
subplot(1,2,2)
newx = mat(1,:);
newy = mat(2,:);
plot(newx, newy)
grid on
```

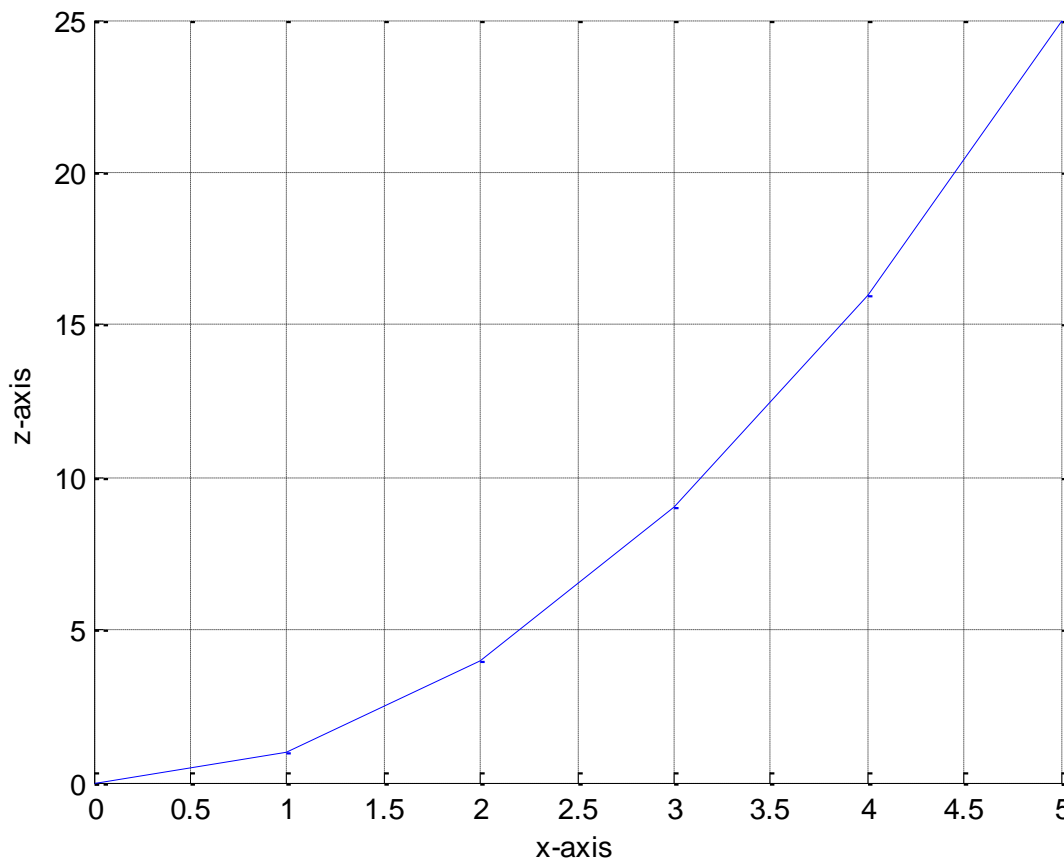


This method only works for a counterclockwise rotation given in radians. To convert from degrees to radians, multiply by $\pi/180$. To rotate clockwise, multiply by -1.

Rotation Around an Axis:

This is much more complicated than rotation about the origin because it turns a two-dimensional plot into a three-dimensional body, requiring us to specify another dimension.

Here I must introduce more bizarre CS 1371 notation. Rotation examples are usually done with z vs. x data, which actually makes more sense for creating three-dimensional bodies. The ' zx ' plane is generally depicted as directly in front of you and perpendicular to your line of sight (your computer screen), with the ' y ' axis stretching ahead of you in space (behind your computer screen). Rotating around the ' y ' axis would now be rotation about the origin, so we only need to worry about rotation around the ' x ' and ' z ' axes.



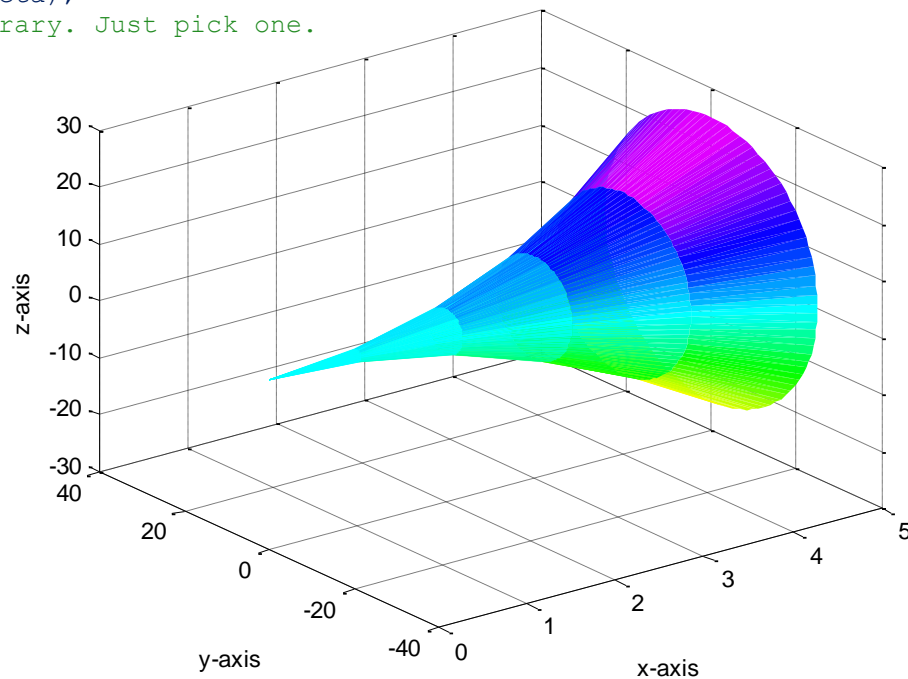
From my experience, problems are usually given with ' u vs. v ' data rather than ' x vs. z ' data "to avoid confusion." They're the same thing. To create a three-dimensional rotation body, simply follow (or memorize) this list of steps.

1. Make ' θ ' using `linspace()`.
2. Use `meshgrid()` to convert ' u ' and ' θ ' to ' uu ' and ' $t\theta$ '.
3. Use `meshgrid()` to convert ' v ' and ' θ ' to ' vv ' and ' $t\theta$ '.
4. Decide which variable (' uu ' or ' vv ') is always equal to the radius and set it to ' rr '.
5. Set the variable (' xx ', ' yy ', or ' zz ') of the rotation axis equal to ' uu ' or ' vv '.

6. Set the other two variables equal to 'rr.*cos(ttheta)' and 'rr.*sin(ttheta)'.
7. Use surf() or mesh() to plot the resulting body.

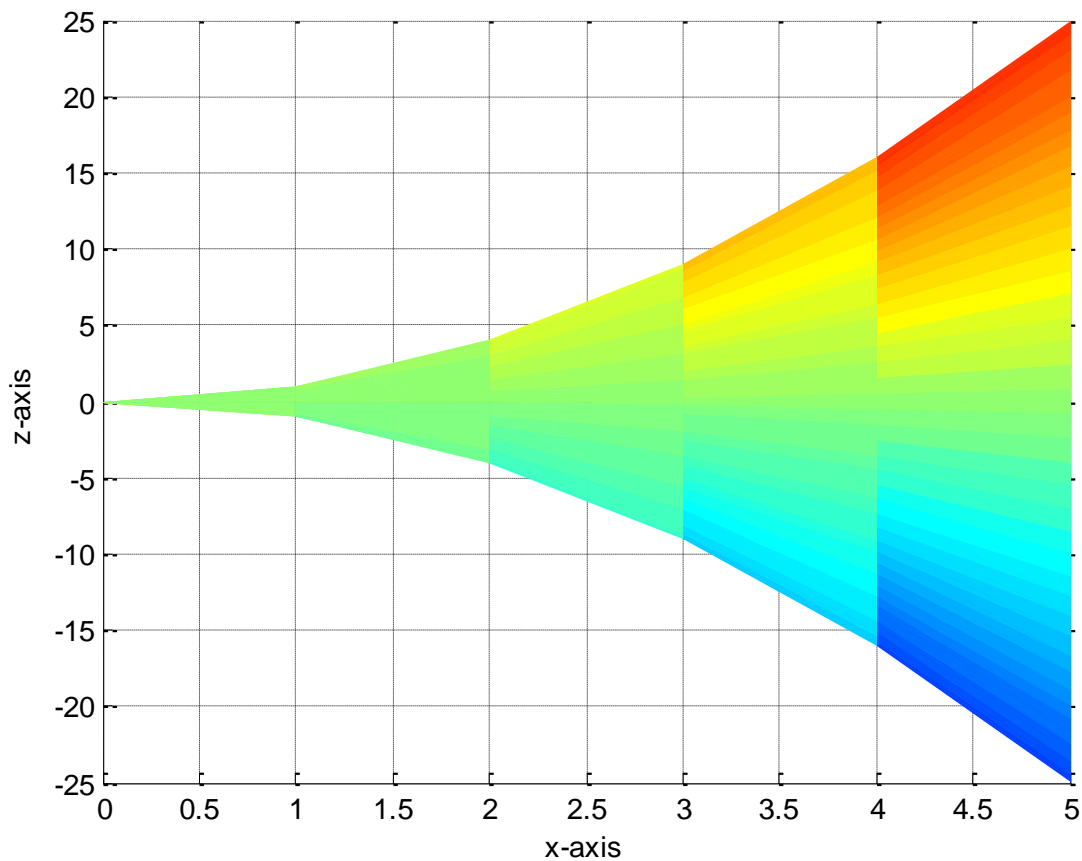
That list probably made absolutely no sense, so let's work through an example. We want to plot the rotation of the above 'xz' data around the x-axis.

```
u = 0:5;
v = u.^2;
%These will be given in the problem statement.
theta = linspace(0,2*pi);
%This ensures a complete rotation.
[uu, ttheta] = meshgrid(u,theta);
%Step one.
[vv, ttheta] = meshgrid(v,theta);
%Step two. ttheta is exactly the same, so don't worry about overwriting it.
rr = vv;
%Visualize the final figure. If we take a circular slice parallel to the
%y-axis, the radius of that circle is always equal to the value of 'v' at
%that point.
xx = uu;
%The 'x' value doesn't change, since we're rotating around the x-axis.
yy = rr.*cos(ttheta);
zz = rr.*sin(ttheta);
%These are arbitrary. Just pick one.
surf(xx,yy,zz)
shading interp
xlabel('x-axis')
ylabel('y-axis')
zlabel('z-axis')
```



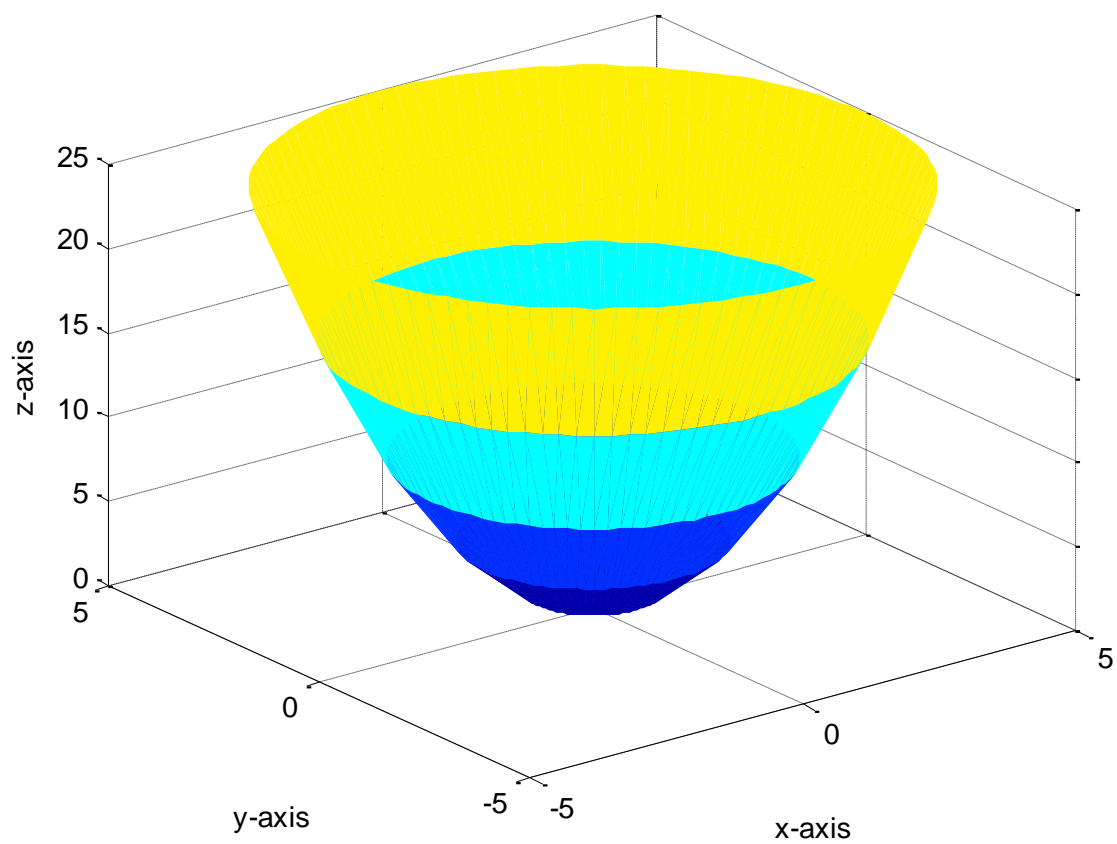
If we view the figure in the 'xz' plane, we see that the original graph has been mirrored along the x-axis, exactly as expected.

```
view(0,0)
```

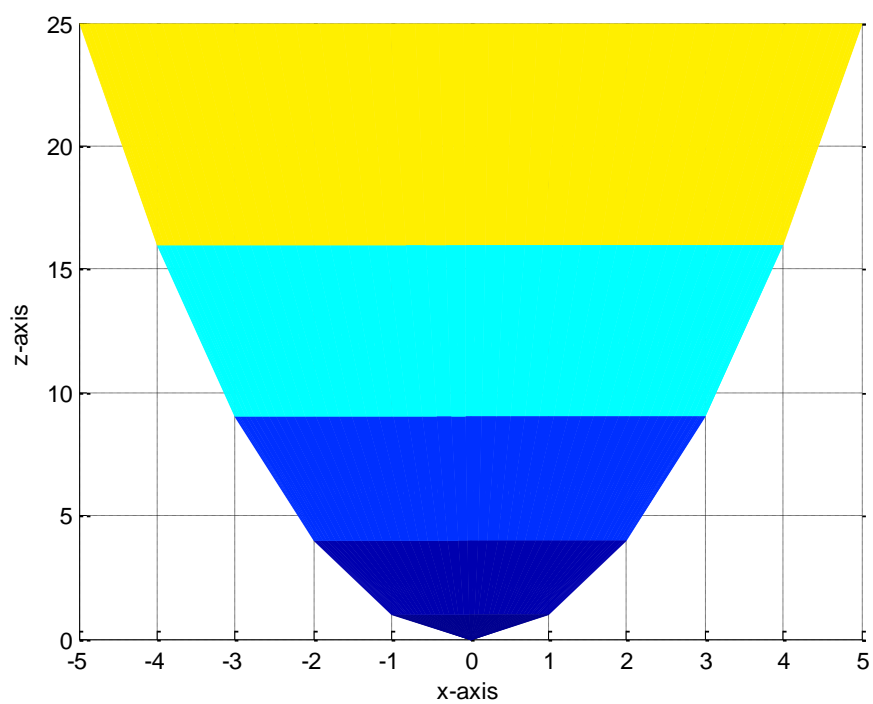


Now we'll rotate the same plot around the z-axis.

```
u = 0:5;
v = u.^2;
theta = linspace(0,2*pi);
[uu, ttheta] = meshgrid(u,theta);
[vv, ttheta] = meshgrid(v,theta);
rr = uu;
%The imaginary circular slice now lies in the 'xy' plane. The
%radius(distance from the z-axis) is equal to 'u'.
zz = vv;
%Once again, the axis of rotation become fixed. This time, the distance
%along the z-axis(up and down) is equal to the original 'v'.
xx = rr.*cos(ttheta);
yy = rr.*sin(ttheta);
%Very repetitive.
surf(xx,yy,zz)
shading interp
xlabel('x-axis')
ylabel('y-axis')
zlabel('z-axis')
```

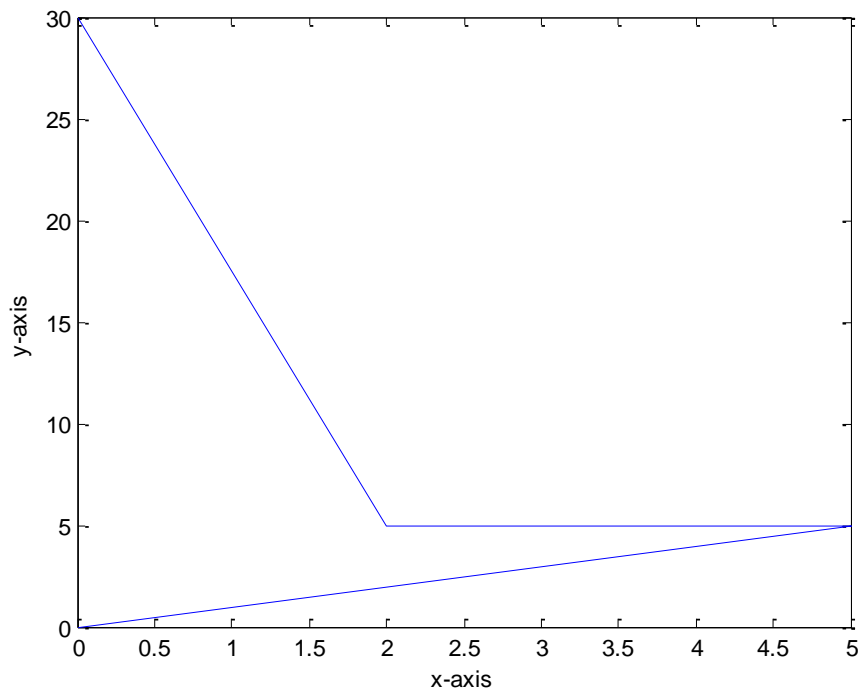


```
view(0,0)
```



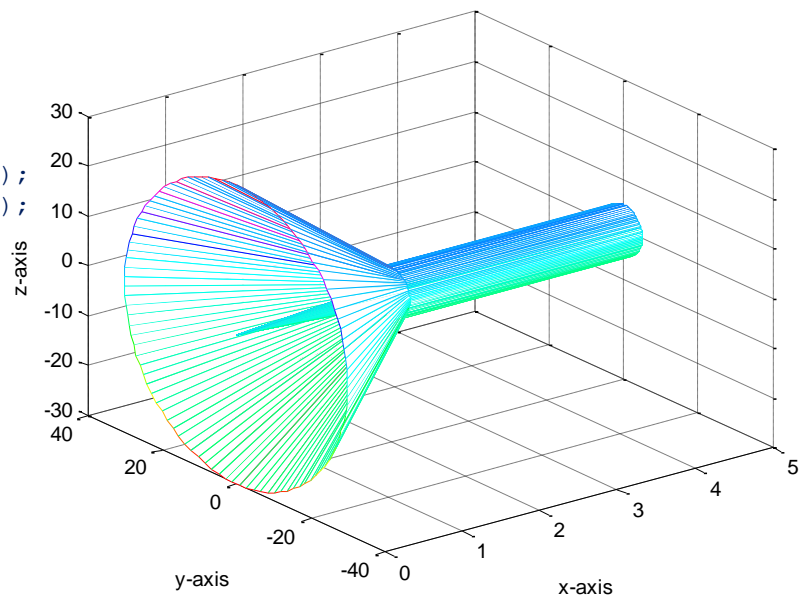
Just for practice, we can rotate some 'xy' data.

```
u = [0, 5, 2, 0];  
v = [0, 5, 5, 30];  
plot(u,v)  
xlabel('x-axis')  
ylabel('y-axis')
```



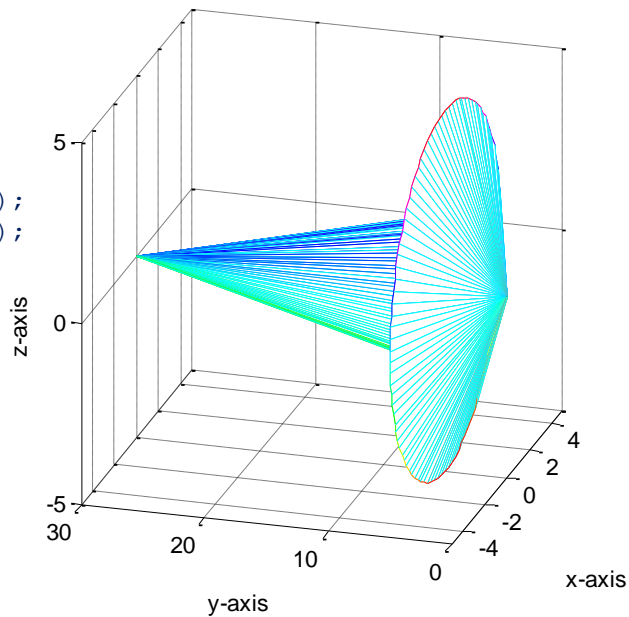
Around x-axis:

```
u = [0, 5, 2, 0];  
v = [0, 5, 5, 30];  
theta = linspace(0, 2*pi);  
[uu, ttheta] = meshgrid(u, theta);  
[vv, ttheta] = meshgrid(v, theta);  
rr = vv;  
xx = uu;  
zz = rr.*cos(ttheta);  
yy = rr.*sin(ttheta);  
mesh(xx, yy, zz)  
shading interp  
xlabel('x-axis')  
ylabel('y-axis')  
zlabel('z-axis')
```



Around y-axis

```
u = [0, 5, 2, 0];  
v = [0, 5, 5, 30];  
theta = linspace(0, 2*pi);  
[uu, ttheta] = meshgrid(u, theta);  
[vv, ttheta] = meshgrid(v, theta);  
rr = uu;  
yy = vv;  
xx = rr.*cos(ttheta);  
zz = rr.*sin(ttheta);  
mesh(xx, yy, zz)  
shading interp  
xlabel('x-axis')  
ylabel('y-axis')  
zlabel('z-axis')
```



Consider these examples carefully. Rotation will seem incredibly complicated until you understand the thought process.

Three-Dimensional Rotation Matrices:

Finally, you can rotate a plot by a fixed angle 'theta' around the 'x', 'y', or 'z' axes by multiplying the appropriate three-dimensional rotation matrix. This method works just like two-dimensional rotation about the origin. There is probably no need to remember these since they rarely appear on homework or test problems.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix}$$
$$R_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix}$$
$$R_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Section 15: Images



MATLAB is capable of reading in and manipulating images much like Notepad or Excel files from file IO, although I can't understand why anyone would want to use MATLAB to play with images. Fortunately, there are only a few important functions and concepts to learn, so this is one of the easiest topics in the course.

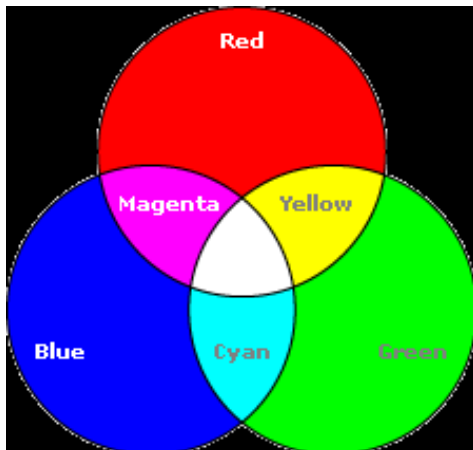
Image arrays are HUGE. Never forget to suppress your outputs.

Background on Images:

Computers store and interpret images as enormous arrays of elements called pixels. Each pixel has three values associated with it: red, green, and blue. The computer interprets different combinations of these three numbers as different colors; each individual pixel becomes one solid color, and large collections of these colors become images.

The pixel values are based on mixing the three primary colors of light: red, green, and blue. All colors can be made from different combinations of those three colors of light. A pixel with all zeros would represent the absence of light, the color black, while a pixel with the highest possible values for all three would be pure white.

MATLAB stores images in $M \times N \times 3$ three-dimensional arrays, with the third dimension representing the three primary colors. The image itself is an array of red values on top of an array of green values on top of an array of blue values.



Uint8():

Because the addition of light must be bounded by a finite limit (the color white), images are stored in a special new data type called uint8, which stands for unsigned 8-bit integer. A thorough discussion of this definition would be pointless here, so I will cover the important points. Uint8 numbers function like regular doubles except for three differences.

1. All numbers lower than the minimum value of 0 are automatically set to 0.
2. All numbers above the maximum value of 255 are automatically set to 255.
3. All decimals are automatically rounded up or down to the nearest whole number.

Basically, uint8 is a data type that only allows for whole numbers ranging from 0 to 255. This makes sense in terms of images because fractional pixel values would be meaningless, and black (0) and white (255) must have finite values.

Luckily, the function that converts other data types to uint8 is very easy to remember. It's called uint8().

```
arr = uint8(arr)
```

In most image problems, we will first convert the image array to doubles because they can be more easily manipulated. Usually, the final step will be to convert all the data back to type uint8.

```
uint8(300)           ans = 255;
uint8(-5)            ans = 0;
uint8(3.5)           ans = 4;
class(uint8(5))      ans = 'uint8'
```

Imread():

MATLAB has a built-in function to read images called imread(). Imread() takes in a string representing an image file in the current directory and outputs an image array of type uint8. Image arrays are just like regular arrays with a third dimension. The number of rows and columns will vary between images, but there will always be three layers: red, green, and blue, in that order. My shorthand for an image array is 'im'.

```
im = imread(filename)
```

In the same way that two-dimensional arrays are indexed by rows followed by columns, three-dimensional arrays are indexed by rows, then columns, then layers. You can create three-dimensional arrays either by manually creating each layer or by using the cat() function, explained below.

```
A = magic(3);
im(:, :, 1) = A;
im(:, :, 2) = A;
im(:, :, 3) = A;
```

Alternatively, it is easy to divide images into layers using the same method.

```
im = imread(filename)
red = im(:,:,1);
green = im(:,:,2);
blue = im(:,:,3);
```

Cat():

There is also a magical MATLAB function that concatenates equivalently sized arrays along the third dimension, much like using square brackets. The inputs should be the number 3 (for three-dimensional concatenation) followed by all the arrays you wish to concatenate.

```
A = magic(3);
im = cat(3,A,A,A);
```

Imwrite():

Imwrite() takes in a three-dimensional uint8 array and saves the corresponding image to the current directory. You must also specify the format in which you wish to save the image (jpg, bmp, etc.) as a string.

```
imwrite(arr,filename,format)
imwrite(A,'hello.jpg','jpg')
imwrite(A,'greetings.bmp','bmp')
```

If the input array has only one layer instead of three, imwrite() will create a grayscale image with the same array as all three layers.

Image() and Imshow():

Image() and imshow() can be used to display image arrays in the current figure window. The main difference is that image() causes MATLAB to do strange things to the image before displaying, such as adding axis marks to indicate pixel numbers, while imshow() just displays the image. If the image is too big to fit on the screen, imshow() will automatically scale it down to a more manageable size. When in doubt, use imshow().

```
image(im)
imshow(im)
```

Salem:

Image problems involve manipulating image arrays using various algorithms to produce the desired results. None of these concepts should be any different from what we could already perform with arrays of doubles. For all of the following examples, I will be using an image of my cat to demonstrate the results. His name is Salem, just like the talking cat from *Sabrina, the Teenage Witch*.



```
im = imread('salem.jpg');
```

Size():

We return now to a brief recapitulation of the `size()` function used to find the number of rows and columns in arrays. When used regularly on a three-dimensional array, `size()` will rather stupidly decide to make up for the extra dimension by tripling the number of columns, which will probably throw you off if you're not watching for it. ALWAYS call `size()` with three outputs when dealing with images, even though you already know there are three layers.

```
[row, col, layer] = size(im);
```

If it makes you feel better, just replace 'layer' with a tilde so it won't be saved in your workspace as a variable.

```
[row, col, ~] = size(im);
```

Resizing Images:

It is often desirable to make an image larger or smaller without significantly changing its aggregate appearance. Unfortunately, adding or deleting random elements from the image array would either add or delete information from the image itself.

The solution is to use indexing tricks to either repeat or delete evenly spaced rows and columns. To make an image twice as large, we copy every row and column an extra time; to halve the size, we delete every other row and column. The easiest way to accomplish this is by using the `linspace()` function to create a range of the desired number of indices and then rounding the indices to the nearest whole number. If we index the original image array at this new array of position values, MATLAB will automatically repeat or delete information—the use of `linspace()` ensures that this information is evenly spaced and thus difficult to notice.

Here we simply set 'x' and 'y' equal to the desired effect on rows and columns, respectively. To halve the rows and double the columns, use `x=0.5` and `y=2`.

```
[row, col, layer] = size(im);  
row_index = round(linspace(1, row, row.*x));  
col_index = round(linspace(1, col, col.*y));  
im = im(row_index, col_index, :);  
imshow(im)
```



Once again, remember to index rows, columns, AND, all three layers or `imshow()` will produce a grayscale image based only on a single layer.

Grayscale Images:

If a certain pixel has the same intensity value for red, green, and blue, MATLAB interprets the color as gray; smaller numbers generate darker shades of gray, as we would expect. To convert a colorful image to grayscale (shades of gray varying according to the intensity of the individual pixels), just take the average value of the three layers of the image array. That average value is then repeated three times to create identical red, green, and blue layers.

Remember that taking averages does not work very well with data type `uint8()` because it automatically rounds fractions and caps numbers at 255. Some course administrators will tell you that you can avoid this issue by dividing through by 3 first when taking averages, but this method could also

potentially fail if MATLAB decides to round a number in an unexpected direction. Take my advice and convert to double and back to uint8—it will probably save you some headache in the long run.

```
im = double(im);  
gray = (im(:,:,1) + im(:,:,2) + im(:,:,3))/3;  
im = cat(3, gray, gray, gray);  
im = uint8(im);  
imshow(im)
```



Taking the Negative of Images:

Taking the negative is a fancy phrase for reversing the intensity of every pixel—white becomes black, and black becomes white. Just subtract the entire image array from 255.

```
negative = 255 - im;
```

Swapping Layers:

Creative indexing also gives us the ability to change the order of the color layers in a three-dimensional array. Remember waaay back from vectors that an range of indices is simply a vector of position numbers. Thus, instead of using a colon (:) to access all layers of an image, we can use a vector to swap them instead. Red=1, Green=2, and Blue=3. Let's swap red and green.

```
im = im(:, :, [2 1 3]);  
imshow(im)
```



As you can see, the table now looks purple.

Dividing Images:

Dividing images into halves or quadrants is easy but tedious. This just requires indexing and heavy use of the `round()`, `floor()`, or `ceil()` functions to avoid decimals. I always use `round()`.

<code>im(1:round(end/2), :, :);</code>	Top half
<code>im(round(end/2)+1:end, :, :);</code>	Bottom half
<code>im(:, 1:round(end/2), :);</code>	Left half
<code>im(:, round(end/2)+1:end, :);</code>	Right half
<code>im(1:round(end/2), 1:round(end/2), :);</code>	Top right corner
<code>im(1:round(end/2), round(end/2)+1:end, :);</code>	Top right corner
<code>im(round(end/2)+1:end, 1:round(end/2), :);</code>	Bottom left corner
<code>im(round(end/2)+1:end, round(end/2)+1:end, :);</code>	Bottom right corner

Transposing Images and `permute()`:

For some unknown reason, the designers of MATLAB decided that using the apostrophe (`'`) to transpose three-dimensional arrays should cause an error. You can either divide the image into three layers and transpose them individually or use the `permute()` function.

`permute()` takes in an array and a vector indicating the order you wish to rearrange the dimensions. The natural order for a three-dimensional array is `[1 2 3]`, or rows, columns, layers. To swap rows and columns, use `[2 1 3]` instead.

Once again, remember that transposing is fundamentally different from rotating. Notice that the lizard is now in front of Salem's RIGHT paw.

```
red = im(:, :, 1);
green = im(:, :, 2);
blue = im(:, :, 3);
trans_im = cat(3, red', green', blue');
imshow(trans_im)
```

OR

```
trans_im = permute(im, [2 1 3]);
```



Rotating Images:

We can also use the same indexing tricks from the section on arrays to rotate images 90 degrees clockwise or counterclockwise.

Remember that, in the first two cases, we transpose the array and then reverse either the rows or columns. To rotate 180 degrees, transposition is unnecessary.

```
im = permute(im,[2 1 3]);          Rotates counterclockwise
im(end:-1:1, :, :)

im = permute(im,[2 1 3]);          Rotates clockwise
im(:, end:-1:1, :)

im(end:-1:1, end:-1:1, :)          Rotates 180 degrees
```

To convert larger or smaller angles to their 90 degree equivalents, either add or subtract 360 inside a while loop or take the absolute value of the modulus from 360. The latter method returns 0, 90, 180, or 270. Thanks go to Carmie Cuda for this idea.

```
angle = abs(mod(angle,360));
```

For all other concepts such as concatenation, simultaneously changing pixel values (masking), and basic manipulation ideas, refer to the section on arrays. Remember the golden rule of images: they work just like regular arrays.

Section 16: Numerical Methods

Numerical methods is a fancy term for various problems that would normally be cumbersome but that MATLAB can solve easily using “brute force” methods. Although the actual scope of numerical methods encompasses very fascinating topics such as root finding, linear regression, and solutions to partial differential equations, none of these are covered in CS 1371.

Instead, this section focuses on a few basic topics like solving linear systems, curve fitting, numerical integration, and linear interpolation. Most problems can be solved easily using the appropriate built-in MATLAB function.

Solving Linear Systems:

A system of many linear equations (no exponents greater than one) can be solved using basic matrix division. Here I will not attempt to explain the finer points of linear algebra; the only important equation is $A\mathbf{x} = \mathbf{b}$, where ‘A’ is a matrix of the coefficients for the linear equations, ‘ \mathbf{x} ’ is a column vector of x-values (the solutions to the linear equations), and ‘ \mathbf{b} ’ is a column vector of the right-hand sides of all the equations. Thus, the system

$$x_1 + 2x_2 = 3$$

$$4x_1 - 7x_2 = -3$$

can be transformed into the equivalent linear algebra equation

$$\begin{bmatrix} 1 & 2 \\ 4 & -7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ -3 \end{bmatrix}$$

Although this particular system would be easy to solve by hand, MATLAB can solve much more complex systems using its amazing capability to invert matrices. There are three ways to solve any system of the form $A\mathbf{x} = \mathbf{b}$.

1. Multiply the inverse of ‘A’ times ‘ \mathbf{b} ’ using the `inv()` function. Remember that you must ALWAYS multiply on the left-hand side.
2. Multiply the inverse of ‘A’ times ‘ \mathbf{b} ’ using the MATLAB power operator.
3. Use matrix division to employ Gaussian elimination and back-substitution without finding the inverse itself. This is by far the most efficient method. MATLAB uses the backslash ‘\’ to indicate matrix division.

```
A = [1 2; 4 -7];  
b = [3; -3];  
x = inv(A)*b;  
x = A^-1*b;  
x = A\b;
```

Problems statements will often give the equations in some strange manner in an attempt to confuse you. Make sure your two matrices are set up exactly as in the example above.

Curve Fitting:

MATLAB has one important function for fitting a polynomial curve to a collection of data points. Polyfit() takes in a vector of x-values, a vector of corresponding y-values, and the degree of the polynomial you wish to form. It outputs the coefficients of said polynomial.

The third input may require some discussion. The number you input will be equal to the highest-order exponent in the resulting polynomial (2 for quadratic, 3 for cubic, etc.). However, any polynomial can have a constant coefficient as well as a coefficient for every power of the independent variable, so inputting a three will actually give you a vector of length four. [A B C D] represents $Ax^3 + Bx^2 + Cx + D$.

We'll use some contrived data as an example:

```
x = 0:5;
y = x.^3 - x.^2 + x - 1;
coeff = polyfit(x,y,3)           coeff = [1 -1 1 -1]
```

MATLAB can usually make reasonable fits for a variety of different highest order coefficient choices. Mathematically, the highest order possible to produce a unique fit is one less than the length of the data—a higher order fit would not have a sufficient number of points to go through. As an example, consider a simple system of two points. You could feasibly plot an infinite number of parabolas through those two points, but only one line.

Polyval()

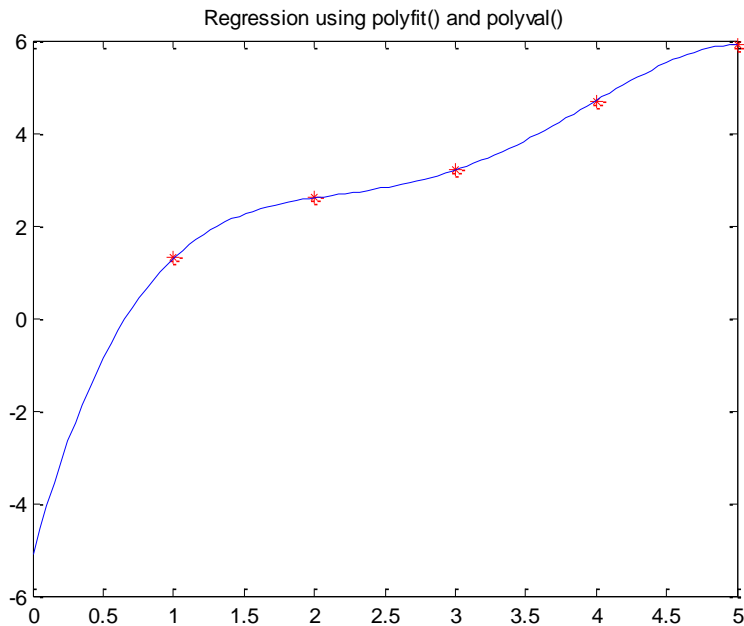
Obviously, a vector of coefficients is not very useful if you want to evaluate the function at other points. Fortunately, we have the polyval() function to fix that problem.

Polyval() takes in a vector of coefficients representing a polynomial equation and a vector of x-values at which to evaluate the function. It outputs the associated y-values based on the input function.

Using polyfit() and polyval() can be used in conjunction to convert a collection of disconnected points into a function and evaluate the function at more points to produce a smooth plot.

```
x = 1:5;
y = [1.3 2.6 3.2 4.7 5.9];
plot(x,y,'r*')
coeff = polyfit(x,y,4);
xi = linspace(0,5);
yi = polyval(coeff,xi);
hold on
```

```
plot(xi,yi)
title('Regression using polyfit() and polyval()')
```

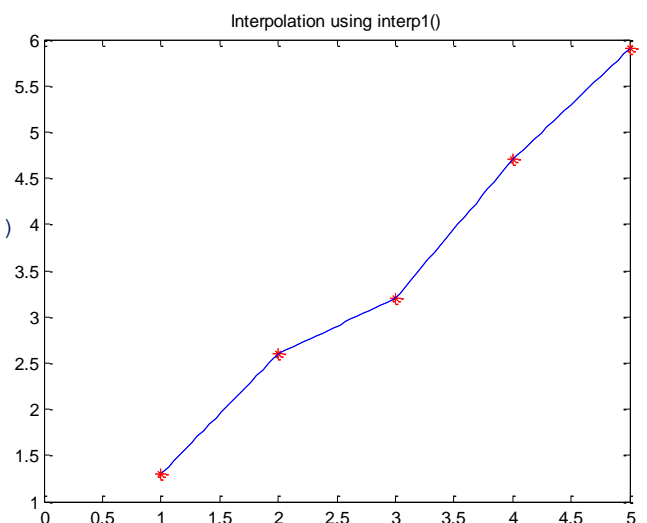


Interp1():

Don't forget the numerical '1' at the end of this function name. Interp() is a completely different function that is never used in CS 1371. Interpolation allows you to estimate the values at points in between individual data points by connecting them with straight lines. Linear interpolation does NOT produce smooth curves based on data trends; it merely connects each set of two points with a straight line.

The inputs for interp1 are a vector of x-values, a vector of y-values, and a vector of new x-values at which you wish to interpolate. Interp1() outputs the estimated y-values for those new x-values.

```
x = 1:5;
y = [1.3 2.6 3.2 4.7 5.9];
plot(x,y,'r*')
hold on
xi = linspace(0,5);
yi = interp1(x,y,xi);
plot(xi,yi)
title('Interpolation using interp1()')
```



Interp1() can also be used for extrapolation. Extrapolation is the process of guessing the y-value of a data point outside the initial data range. As such, extrapolation is often very untrustworthy because there is no way of accurately estimating trends outside of a given data range. MATLAB appears to have changed this particular function in its newest addition, so extrapolation now requires five inputs.

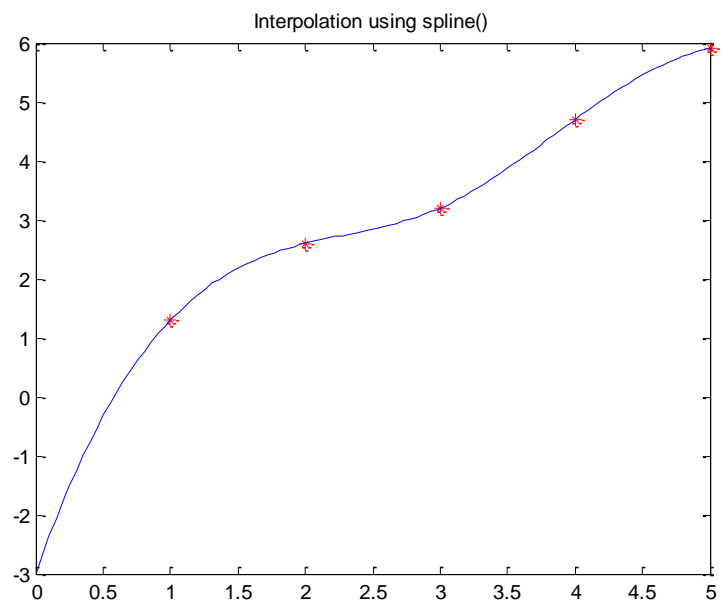
```
yi = interp1(x,y,xi,'linear','extrap');
```

Don't use extrapolation.

Spline():

Spline() is an odd function that is confusing for most people, but it is fortunately relatively unimportant. Spline() is used to achieve nonlinear interpolation based on piecewise cubic functions. Basically, it takes every set of three points and uses some sort of special MATLAB trick to connect them with a cubic function. Spline interpolations always go through every point.

```
x = 1:5;
y = [1.3 2.6 3.2 4.7 5.9];
plot(x,y,'r*')
hold on
xi = linspace(0,5);
yi = spline(x,y,xi);
plot(xi,yi)
title('Interpolation using spline()')
```



It looks a lot like the polyval() plot.

Differentiation and Integration of Polynomials:

Because the derivatives and integrals of polynomials are based only on a simple, easy formula, we can use MATLAB to produce the coefficients very quickly. The secret is to create a vector of exponent values, like [3 2 1 0] for a cubic function and use it to multiply or divide the original coefficients.

Remember the basic formulas:

$$\frac{d(x^n)}{dx} = nx^{n-1}$$

$$\int x^n dx = \frac{x^{n+1}}{n+1} + C$$

So for differentiation we multiply each coefficient by its corresponding exponent and delete the last one (remember that the derivative of a constant is zero). We integrate by dividing each coefficient by its corresponding exponent plus one and adding an arbitrary constant (usually zero) to the end. Consider the function $f(x) = 5x^3 + 3x^2 - 2x + 1$.

```
coeff = [5 3 -2 1];
der = coeff.*(length(coeff)-1:-1:0);
der(end) = [];
int = coeff ./ (length(coeff):-1:1);
int(end+1) = 0;
```

Diff():

The `diff()` function takes in a vector and outputs the differences between each set of two elements in the vector. The length of the output vector will always be one less than the length of the original vector. `Diff()` is not very useful but can be used with 'xy' data to estimate the derivative dy/dx at multiple points.

```
x = [1 2 5 8 13];
y = [-1 5 2 -10 5];
diff(x)                                ans = [1 3 3 5]
diff(y) ./ diff(x)                     ans = [6 -1 -4 3]
```

Cumsum():

Another function with relatively little practical use, `cumsum()` takes in a vector and, for each element, outputs the sum of that element and all previous elements. The output vector is of the same length as the input vector.

```
x = [1 2 5 8 13];
cumsum(x)                                ans = [1 3 8 16 29]
```

Trapz() and Cumtrapz():

Trapz() provides an estimate of the numerical integral of a set of points (the area beneath the curve) by using the trapezoidal approximation from calculus. If you don't remember the details about the trapezoidal approximation, don't worry—it's not very important. Trapz() takes in ONLY ONE vector, the set of y-values, and seems to assume even spacing between them. The output is a single number, the estimation of the definite integral.

Cumtrapz() works much like cumsum() in that it outputs the sum of the current trapezoid along with all previous ones for each element in the vector. The output will also be of the same length as the input vector. The first element of the output is always zero, and the last element is always equals the total approximation given by the trapz() function.

```
y = [1 2 5 8 13];  
trapz(y)           ans = 22  
cumtrapz(y)        ans = [0 1.5 5 11.5 22]
```

That's pretty much it for numerical methods; most of these functions have very little practical value and will probably be ignored on homework and test problems. Most numerical methods problems are relatively straightforward. Always remember how to use the linspace() function.

Useful Functions:

- polyfit(x,y,order)—outputs the coefficients of the best-fit polynomial with highest exponent equal to 'order'
- polyval(coeff,xi)—outputs the y-values of the function given by the coefficients in 'coeff' at all points in 'xi'
- interp1(x,y,xi)—uses linear interpolation to approximate the y-values at 'xi'
- spline(x,y,xi)—uses spline (strange cubic) interpolation to approximate the y-values at 'xi'
- diff(vec)—calculates the difference between every pair of elements in 'vec'
- cumsum(vec)—outputs the sum of each element plus all previous ones
- trapz(y)—estimates the numerical integral of the y-values using trapezoidal approximation
- cumtrapz(y)—outputs the sum of each trapezoid plus all previous ones, using trapezoidal approximation

Section 17: Sound

Sound is the last homework topic and is usually somewhat difficult for people to grasp conceptually, although the actual MATLAB coding is fairly straightforward. MATLAB can be used to read in and manipulate sound, much like images.

Almost all coding with sound involves executing a specific series of commands to perform a task. Many of these algorithms are difficult to understand, by which I mean that I don't understand them. You should be fine if you just memorize them.

Wavread():

Wavread() reads in sound files from your current directory. The input is the filename, a string. The first output is the sound itself stored in a **COLUMN** vector. This vector contains the amplitude values (samples) for the sound in type double—higher numbers represent louder sounds. The second output is the sampling frequency, the rate at which the amplitude vector is read by MATLAB in samples per second. If you prefer musical terminology, think of sampling frequency as the tempo of the sound. Please notice my sound notation before it confuses you later, and never forget to suppress this function.

```
[data, fs] = wavread(fn);
```

Basic Algorithms:



I will give a brief explanation for each of these; if you forget the code, you can probably figure it out logically.

- Number of samples—every element in the amplitude vector represents one sample.
`num = length(data);`
- Δt , the time per sample—if the sampling frequency gives us samples per second, its inverse must be seconds per sample.
`dt = 1/fs;`
- The duration of the sound—now that we have 'dt', we can multiply seconds per sample times the total number of samples to calculate the total duration in seconds.
`dur = dt*num;`
- The maximum possible sampling frequency for the amplitude vector—there's some weird concept called the Nyquist Theorem that I probably understood at one point. It says that the maximum frequency is equal to half of the given sampling frequency.
`f_nyquist = fs/2;`

- The sound played in reverse—this is just basic indexing of vectors.
`reverse = data(end:-1:1);`
- The fast fourier transform of the sound—this is another idea that I don't understand. Luckily, the TAs usually post a guide for the `fft()` function in MATLAB, which you can read if you have the time before finals. Remember to take the absolute value, multiply by two, divide by the length of the data, and delete the second half of the result.
`FFT = 2*abs(fft(data))/length(data);`
`FFT = FFT(1:round(end/2));`

Changing the Pitch of a Sound:

There are two main methods to make a sound higher or lower. Remember that this is fundamentally different from changing the volume, which we would accomplish by simply adding to or subtracting from the amplitude vector.

First, we can change the sampling frequency. Doubling the frequency raises the pitch by one octave, while halving the frequency decreases the pitch by one octave. Unfortunately, doing so will affect the total duration of the sound, which is not always desirable.

```
fs = fs*2*n; %where n is the number of octaves by which the pitch is raised
or lowered
```

Second, we can use the same method we used with images to index the sound vector at certain intervals to artificially affect the rate at which samples are chosen. This, too, will by definition increase or decrease the duration of the sound.

```
data = data(round(linspace(1,num,num/2^(n/12)))); %where n is the number of
half-steps by which the pitch is raised or lowered.
```

Because these methods affect the total duration of the sound, problems will generally ask you to truncate or zero-fill the amplitude vector to decrease or increase the final duration, respectively. Thus, to maintain the same number of samples after transposition:

```
function data = transpose_sound(fn,n)

    [data, fs] = wavread(fn);

    num = length(data);

    dur = num/fs;

    data = data(round(linspace(1,num,num/2^(n/12))));

    new_dur = length(data)/fs;
```



```

if new_dur > dur
    data = data(1:num);

elseif new_dur < dur
    data(num) = 0;

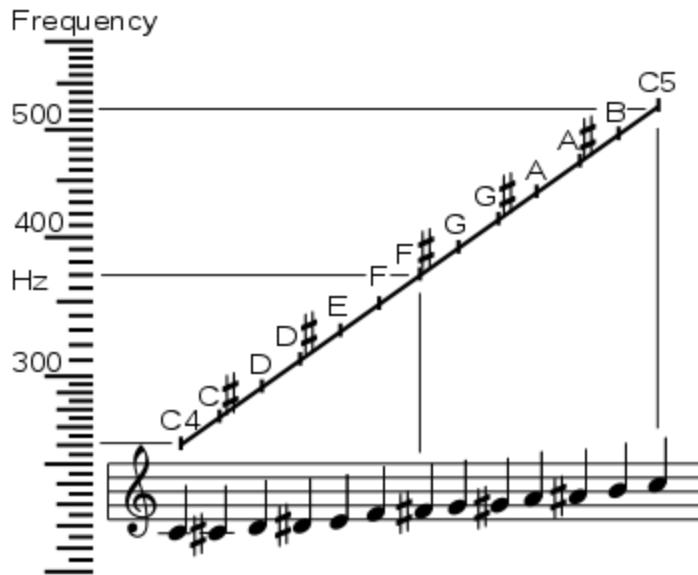
end

end

```

Basic Musical Jargon:

- Pitch—the frequency of the sound. Higher pitches sound “higher.”
- Note—a sound produced by a single pitch. The seven natural notes are A,B,C,D,E,F, and G.
- Octave—a series of eight notes, starting and ending with the same one. Octaves on the piano are labeled with numbers, with C4 being the C closest to the center of the piano. A C5 is one octave higher than a C4, so an example of an octave would be C4 to C5.
- Whole step—the difference in pitch between two natural notes, such as D and E, is defined as a whole step.
- Half-step—some notes fall in between whole steps. These are called accidentals, and are denoted by sharps (#) and flats (b). The difference in pitch between a natural note and the nearest sharp or flat is defined as a half-step. Thus, there are two half-steps in a whole step. A C# is one half-step higher than a C, while a Cb is one half-step lower than a C. Musically, the half-step between any two natural notes can be denoted by either the flat or the sharp—they are the same note. Thus, Eb and D# are termed *isotonic*, and are exactly the same pitch. The only exceptions to this rule are B#, Cb, E#, and Fb; a B# is isotonic with a C, while a Cb is isotonic with a B. Basically, there is no half-step between C and D or between E and F. Here’s another great picture from Wikipedia to illustrate this concept.



- Chord—a group of notes played simultaneously that blend to produce a more interesting pitch.

Generating Chords:

Supposedly, you can generate chords in MATLAB by simply adding together the amplitude vectors for the different notes. Unfortunately, the examples given in this course usually involve a series of irritating beeping noises that the Tas term a “chord.” Real chords are actually pleasing to the ears.

Plotting FFT vs. Normalized Frequency:

Once again, I don’t know what FFT is. For this problem, we use the FFT algorithm described above to generate the ydata for the plot. The xdata will be a series a values ranging from 0 to the nyquist frequency, also described above. We use linspace() to generate a vector of xdata equal in length to the ydata.

Finally, problems will often ask you to identify the principal frequency of a sound, which is simply the frequency with the highest FFT value.

```
function Plot(data)

    y = 2*abs(fft(data))/length(data);

    y = y(round((1:end/2)));

    f_nyquist = fs/2;

    x = linspace(0,f_nyquist,length(y));

    plot(x,y)
```

```
hold on

[Max,index] = max(y);

f_principal = x(index);

plot(f_principal, Max, 'r*')

end
```

Wavwrite():

The opposite of wavread. Wavwrite() takes in an amplitude vector, a sampling frequency, and a filename and creates the appropriate .wav file. Wavwrite() is the only writing function in which the filename comes LAST, not first. Try to remember that for the final, as it is very tricky.

```
wavwrite(data,fs,fn)
```

There isn't really too much to know for sound. Keep in mind that the sound plotting problem with the fft() function is a VERY popular coding problem on the final exam. I have seen it pop up for two consecutive semesters now.

Section 18: Sorting



Sorting constitutes one or two days of lecture at the end of the semester, one pointless day of recitation, and a single multiple choice question on the final exam. While sorting appears to take place instantly in MATLAB with the `sort()` function, it is actually a complex iterative process that can be accomplished in many different ways. Sorting in CS 1371 involves understanding and learning to recognize four of these sorting methods—you will NOT have to code any of these methods.

Insertion Sort:

The simplest and most obvious sorting method, insertion sort is also one of the least efficient. Insertion sort pulls out one number at a time from the original vector and builds the sorted vector element-by-element. The actual algorithm requires MATLAB to iterate through the vector being built in order to determine the proper location of the current element.

1. Create an empty output vector.
2. Remove an element from the unsorted vector.
3. Iteratively determine the element's location in the output vector.
4. Place the element in the output vector.
5. Repeat steps 2 through 4.

<code>vec = [3 1 7 2]</code>	<code>out = []</code>
<code>vec = [1 7 2]</code>	<code>out = [3]</code>
<code>vec = [7 2]</code>	<code>out = [1 3]</code>
<code>vec = [2]</code>	<code>out = [1 3 7]</code>
<code>vec = []</code>	<code>out = [1 2 3 7]</code>

```
function out = insertion_sort(vec)
    if isempty(vec) || length(vec)==1
        out = vec;
        return
    end
    out(1) = vec(1);
    vec(1) = [];
    while ~isempty(vec)
        num = vec(1);
        vec(1) = [];
        for i = 1:length(out)
            if num <= out(i)
                out = [out(1:i-1), num, out(i:end)];
                break
            end
        end
        if i==length(out)
            out = [out num];
        end
    end
end
```

Bubble Sort:

Bubble sort is similar to insertion sort in that it requires step-by-step iteration through the entire starting vector. The difference is that bubble sort produces the output by repeatedly swapping side-by-side elements in the starting vector instead of creating a new one.

1. Take the first element in the unsorted vector.
2. If it is greater than the element on its right, swap it with the element on its right.
3. Repeat step 2 until the element is no longer greater than the element on its right.
4. Repeat steps 1 through 3 until the vector is sorted.

This one is tough to visualize, which is once again why we have Wikipedia:

First Pass:

(5 1 4 2 8) \rightarrow (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps them.
(1 5 4 2 8) \rightarrow (1 4 5 2 8), Swap since $5 > 4$
(1 4 5 2 8) \rightarrow (1 4 2 5 8), Swap since $5 > 2$
(1 4 2 5 8) \rightarrow (1 4 2 5 8), Now, since these elements are already in order ($8 > 5$), algorithm does not swap them.

Second Pass:

(1 4 2 5 8) \rightarrow (1 4 2 5 8)
(1 4 2 5 8) \rightarrow (1 2 4 5 8), Swap since $4 > 2$
(1 2 4 5 8) \rightarrow (1 2 4 5 8)
(1 2 4 5 8) \rightarrow (1 2 4 5 8)

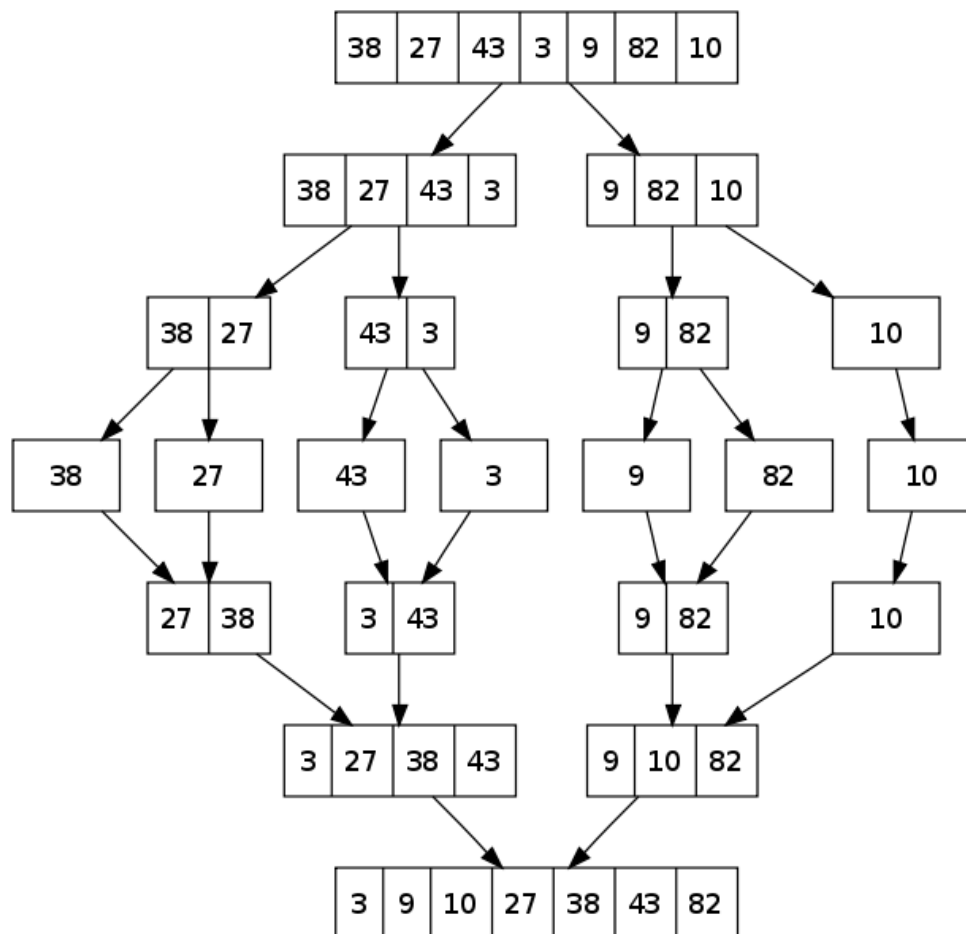
```
function vec = bubble_sort(vec)
    if isempty(vec) || length(vec)==1
        return
    end
    for i = 1:length(vec)-1
        for j = 2:length(vec)
            if vec(j-1) > vec(j)
                vec = [vec(1:j-2), vec(j), vec(j-1), vec(j+1:end)];
            end
        end
    end
end
```

Merge Sort:

Merge sort is a recursive method based on the principle that combining two smaller, presorted vectors is faster than sorting the entire thing manually. Merge sort arbitrarily breaks the unsorted vector into smaller pieces, sorts the pieces, and then merges them back together step by step.

1. Divide the vector into halves.
2. Continue dividing the halves into smaller halves until all pieces have length 1.
3. Simultaneously combine and sort each pair of elements.
4. Repeat step 3 with each resulting vector until the entire list is rebuilt.

The principle of merging vectors is that, if the two vectors are already sorted, you only have to compare the first element in each vector, thus saving considerable time by eliminating excessive iteration.



```

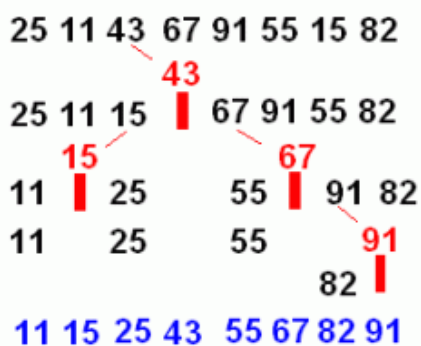
function out = merge_sort(vec)
    if isempty(vec) || length(vec)==1
        out = vec;
    else
        left = merge_sort(vec(1:round(end/2)));
        right = merge_sort(vec(round(end/2)+1:end));
        out = merge(left,right);
    end
end

function out = merge(left,right)
    out = [];
    while ~isempty(left) || ~isempty(right)
        if ~isempty(left) && ~isempty(right)
            if left(1) <= right(1)
                out = [out left(1)];
                left(1) = [];
            else
                out = [out right(1)];
                right(1) = [];
            end
        elseif isempty(left)
            out = [out right];
            right = [];
        else
            out = [out left];
            left = [];
        end
    end
end

```

Quicksort:

Quicksort is another recursive algorithm based on the idea of combining presorted vectors. It is somewhat more efficient than merge sort, however, in that it sorts the elements while breaking them apart rather than afterward. The quicksort algorithm selects a random element from the unsorted vector, called the pivot, and separates the remaining elements according to whether they are greater than or less than the pivot. The result of this separation is that the pivot automatically ends up in the correct position, so recursively creating more pivots eventually sorts the vector.



```

function out = quicksort(vec)
    if isempty(vec) || length(vec)==1
        out = vec;
        return
    end
    less = [];
    greater = [];
    pivot = vec(1);
    for i = length(vec):-1:2
        if vec(i) < pivot
            less = [less vec(i)];
        else
            greater = [greater vec(i)];
        end
        vec(i) = [];
    end
    out = [quicksort(less) pivot quicksort(greater)];
end

```

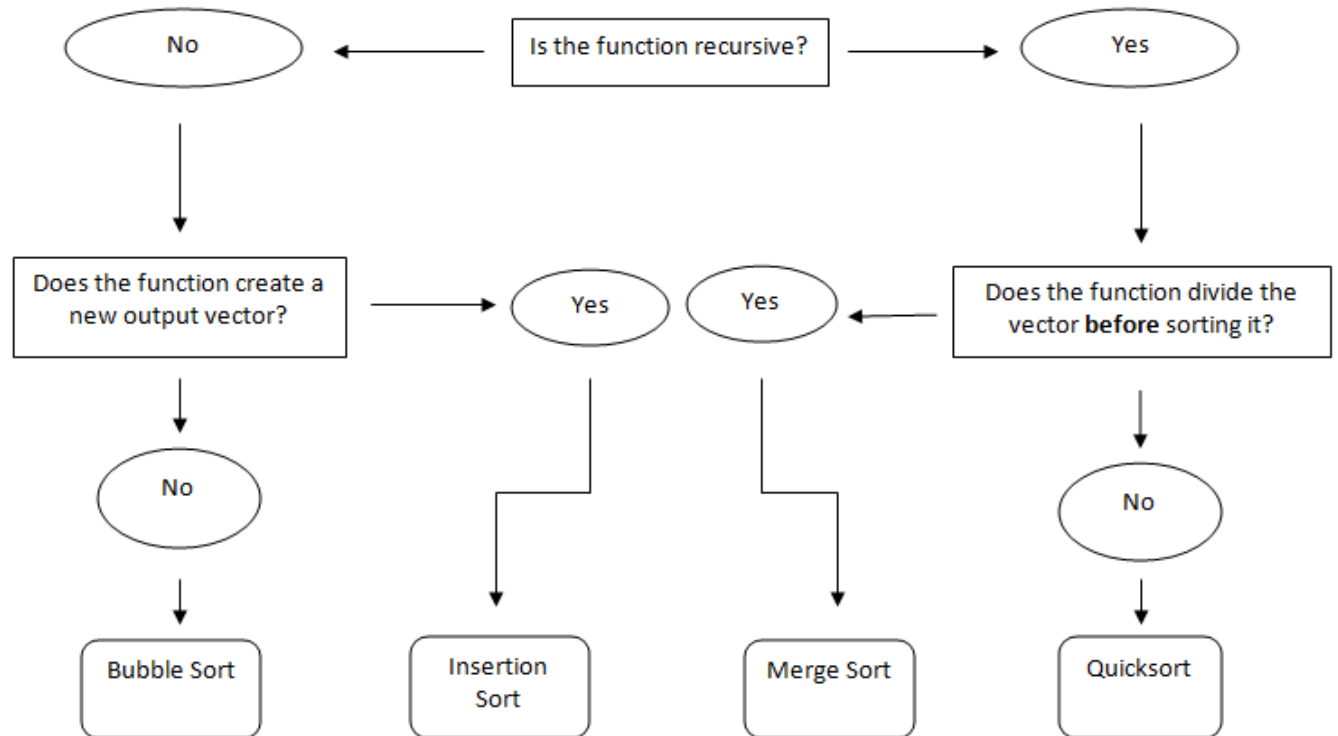
Big O Values:

Each sorting method is assigned a “big O” value based on its overall efficiency. While they are all capable of sorting small lists of numbers virtually instantaneously, constantly iterating through databanks of millions of numbers incurs significant drawbacks for less efficient sorting methods. The first two methods, insertion sort and bubble sort, are considered $O(n^2)$, meaning that the time necessary to sort the vector increases quadratically with the length of the vector.

The other two algorithms, quicksort and merge sort, have values of $O(n \log n)$, meaning that the increase in time is logarithmically related to the length of the vector. Thus, quicksort and merge sort are significantly faster when sorting longer lists, for the reasons explained above.

Identifying the Sorting Method Used:

Most sorting problems involve analyzing a piece of code like the ones above and identifying which of the four sorting methods is being used. Just note the differences and follow this easy flowchart.



A problem might try to trick you by using the merge sort algorithm but naming one of the variables 'pivot', or other such tricks. Just follow the general flowchart and you should be fine.

I abstained from commenting any of the above sorting codes in hopes that they will appear clearer as a whole. Most of the ideas involved are fairly straightforward, and if you need some fun coding practice in iteration or recursion before the final, these algorithms are highly instructive.

Section 19: Conclusion

Thank you for reading my guide for CS 1371. It has been a great pleasure to work with so many fascinating students while completing this project, and I have garnered numerous friendships in the process. I also appreciate the many thanks I have received both personally and through email.

Regarding payment, please don't offer me anything in return for this guide—consider it my free gift to you. I only ask that you consider how much potential you have to make a difference in the lives of others; helping people is as simple as being perceptive enough to notice a problem and caring enough to correct it.

There are most likely countless small coding mistakes and other such errors in this document. If you notice any specific ones, please let me know. You may also inform me of any additions, revisions, or clarifications that may be beneficial for future readers of this study guide. My email address is kmcarr@ymail.com.

Feel free to distribute these guides to whichever students may find them useful. I have tried my best to generate my own practice problems and avoid giving answers to real homework or test problems, so this guide should be considered an honest source of material for the course. No one has my permission to sell this document or to remove my name from it.

I am planning to edit and revise the study guide next semester, most likely by adding a separate example section for each topic as well as three practice exams and a practice final. If you would like the final revised copy, let me know at the end of next semester.

The Final Exam:

The CS 1371 final is comprehensive and is generally worth 35% of your overall grade. There will be twenty to thirty multiple choice tracing problems and three or four coding problems, if I remember correctly. You should find the final significantly easier than any of the individual tests because the skills you accumulate throughout the course build on each other.

For studying I suggest reworking all three of your previous exams as well as any homework problems that were relatively straightforward but gave you trouble conceptually. Topics to focus on are file IO, structures, recursion, images, and sound.

Aggregate Useful Functions List

Data Type Conversion:

- `double(x)`—converts x to type double
- `char(x)`—converts x to type char
- `uint8(x)`—converts x to type uint8
- `class(x)`—returns the data type of x as a string

Data Type Determination:

- `isnumeric(x)`—determines if x is of type double
- `ischar(x)`—determines if x is of type char
- `iscell(x)`—determines if x is of type cell
- `isstruct(x)`—determines if x is a structure
- `islogical(x)`—determines if x is of type logical
- `isempty(x)`—determines if x is an empty vector, string, etc.

Vectors and Arrays:

- `min(x)` – returns the smallest element in the collection (Note: if the minimum value occurs multiple times, `min()` and `max()` will return only the first instance)
- `[value, index] = min(x)` – returns the smallest element along with its position number
- `max(x)` – returns the largest element in the collection
- `[value, index] = max(x)` – returns the largest element along with its position number
- `sort(x)` – sorts the elements in ascending order
- `[newX, index] = sort(x)` – returns x in ascending order along with a vector containing the position numbers of the original x to which each value corresponds
- `fliplr(x)` – flips x left-to-right
- `flipud(x)`—flips x top-to-bottom
- `x'` – transposes x (row vector \leftrightarrow column vector)
- `mod(x,num)` – returns the remainder if x is divided by num
- `round(x)` – rounds x up or down
- `ceil(x)` – rounds x up to the next-highest integer if x is fractional
- `floor(x)` – rounds x down to the next-lowest integer if x is fractional
- `ones(x,y)` – creates an array of ones of dimension x,y (Use 1 for x or y to generate vectors)
- `zeros(x,y)` – creates an array of zeros of dimension x,y
- `true(x,y)` – creates an array of logical trues of dimension x,y
- `false(x,y)` – creates an array of logical falses of dimension x,y
- `sum(x)` – computes the sum of x
- `prod(x)` – computes the product of x
- `find(logical expression)` – generates a vector of all indices where the logical expression is true
- `linspace(a,b,num)`—creates an evenly spaced vector of length num ranging from a to b

Logical Expressions:

- `all(x)` – true if everything in x is true
- `any(x)` – true if anything in x is true
- `not(x)` – same as `~x`
- `or(x,y)` – same as `x | y`
- `and(x,y)` – same as `x & y`

Strings:

- `num2str(x)`—converts a double to its numerical equivalent as a string
- `str2num(x)`—converts a string to its numerical equivalent as a double
- `strcmp(x,y)`—determines if two strings are exactly identical
- `strcmpi(x,y)`—determines if two string are identical, ignoring case
- `sprintf(str,var1,var2,...)`—creates a string using variable strings or doubles
- `strtok(x,delimiter)`—breaks up string x based on the location of the delimiter
- `lower(x)`—converts x to lowercase letters
- `upper(x)`—converts x to uppercase letters

Cell Arrays:

- `num2cell(x)`—places each element in x in an individual cell in a cell array
- `cell2mat(x)`—converts cell array x to its equivalent double or string array

File IO:

- `csvread(fn)`—stores csv file x as an array
- `csvwrite(fn,x)`—saves x as a csv file
- `dlmread(fn,delimiter)`—stores dlm file x as an array
- `dlmwrite(fn,x,delimiter)`—saves x as a dlm file using the input delimiter
- `[num,txt,row] = xlsread(fn)`—reads in an Excel spreadsheet
- `xlswrite(fn,x)`—stores x as an Excel spreadsheet
- `fopen(fn,permission)`—opens the file and returns the file handle
- `fgetl(fh)`—returns the next line in the file
- `fgets(fh)`—returns the next line in the file, including the next line character (`\n`)
- `fprintf(fh,str,var1,var2,...)`—prints the formatted string to the file or the command window
- `fclose(fh)`—closes the designated file

Structure Arrays:

- `struct(field1,{...},field2,{...},...)`—creates the input structure
- `setfield(sa,field,new_field)`—changes a fieldname in a structure
- `rmfield(sa,field)`—removes a field from a structure
- `getfield(sa,field)`—returns the value of the field in the structure
- `fieldnames(sa)`—returns a cell array of the fieldnames in the structure
- `isfield(sa,field)`—determines if the fieldname exists in the structure

Plotting:

- `plot(x,y,parameters)`—plots two-dimensional data
- `axis`—specifies various type of axes (tight, equal, square, etc.)
- `title(str)`—creates the plot title
- `x/y/zlabel(string)`—labels the x,y, or z axis
- `clf`—clears the current figure
- `hold on`—prevents the current plot from being overwritten
- `subplot(row,col,plot)`—creates grids of different plots
- `plot3(x,y,z)`—plots three-dimensional data
- `surf/mesh(xx,yy,zz)`—creates surface plots
- `[xx,yy] = meshgrid(x,y)`—“meshes” two vectors into two arrays
- `shading`—sets the shading for the current plot (flat, faceted, interp)
- `colormap`—sets the color scheme used for the current plot

Images:

- `imread(fn)`—stores an image as a uint8 array
- `cat(dimension,var1,var2,...)`—concatenates the inputs along the specified dimension
- `imwrite(fn,im,type)`—stores the uint8 array as an image
- `image/imshow(im)`—displays the image
- `saalem`—my cat

Numerical Methods:

- `polyfit(x,y,order)`—outputs the coefficients of the best-fit polynomial with highest exponent equal to ‘order’
- `polyval(coeff,xi)`—outputs the y-values of the function given by the coefficients in ‘coeff’ at all points in ‘xi’
- `interp1(x,y,xi)`—uses linear interpolation to approximate the y-values at ‘xi’
- `spline(x,y,xi)`—uses spline (strange cubic) interpolation to approximate the y-values at ‘xi’
- `diff(vec)`—calculates the difference between every pair of elements in ‘vec’
- `cumsum(vec)`—outputs the sum of each element plus all previous ones
- `trapz(y)`—estimates the numerical integral of the y-values using trapezoidal approximation
- `cumtrapz(y)`—outputs the sum of each trapezoid plus all previous ones, using trapezoidal approximation

Sound:

- `[data,fs] = wavread(fn)`—returns the amplitude vector and sampling frequency of the sound
- `wavwrite(data,fs,fn)`—saves the data as a sound file with the specified sampling frequency

FIN