<u>Vectors</u>

- A vector – the primary way MATLAB stores homogenous data (data of one type – for example, all numbers, all letters, etc)
- Creating a vector
    - Direct entry
        - A = [1 2 3 4 5]
    - Range-specification (“colon operator”)
        - A = 1:5 OR A = 1:1:5 (both the same)
        - (When the “1” in the middle of 1:1:5 is omitted, the step between each number is understood to be one)
        - B = 1:2:7 = [1 3 5 7]
    - Using the function linspace
        - A = linspace(1,5,5) = [1 2 3 4 5]
        - First number is the first element in the vector, second number is the last element in the vector, third number is the total number of elements in the vector
- Accessing/Manipulating a vector
    - Indexing – the primary method of accessing/removing/changing elements inside of a vector
        - Create vector A: A = [1 2 3 4 5]
        - To access the third element of A and assign it to a variable called B, do the following:
            - B = A(3)
            - Now, B is equal to the number 3 (which is the third element of A)
        - To change the third element of A to the number 8, do the following:
            - A(3) = 8
            - Now, A = [1 2 8 4 5]
        - What if we wanted to make the 8th element of A the number 4? If A only has 5 elements, will we receive an error? No – MATLAB will compensate, filling in zeros for elements 6 through 7:
            - A(8) = 4
            - Now, A = [1 2 8 4 5 0 0 4]
        - To remove an item from a vector, set the vector indexed at the element in question equal to empty brackets ([]):
            - A(5) = [] (delete the fifth element of A)
            - Now, A = [1 2 8 4 0 0 4]
            - Notice how the length of A decreased by one when we deleted an element (kind of obvious, but this has important later)
        - All of these indexing techniques apply to ranges as well as single numbers; for example, to make the 1st five elements of A equal to 3:
            - A(1:5) = 3
            - Now, A = [3 3 3 3 3 0 0 4]
        - To delete the first five elements of A:
            - A(1:5) = []
            - Now, A = [0 0 4]
    - Concatenating Vectors

- To “Concatenate” two vectors means to create a new vector by using elements of the old vectors. For example,
    - Create vector A: A = [1 2 3 4 5]
    - Create vector B: B = [6 7 8 9 10]
    - Now, create vector C where C vectors A and B concatenated together:
        - C = [A, B]
        - Now, C = [1 2 3 4 5 6 7 8 9 10]
- This technique can also be combined with indexing to make a new vector with only certain elements of the first two vectors. For example, To create a new vector C from vectors A and B that consists of the first three elements of A and the last three elements of B:
    - C = [A(1:3), B(end-2:end)]
    - Now, C = [1 2 3 8 9 10]
        - Note: the ‘end’ operator is simply is another way of indexing from the last element of a vector; we can also specify indices using mathematical operations such as + and -. So, “end-2” would be read by MATLAB as the last element of B minus two more elements. Thus, if B has five elements, then “end-2” would be 5-2, or the third element (which in this case is the number 8). So another way of reading B(end-2:end) would be B(3:5), or the vector [8 9 10]

- Numeric/Logical Operators
    - Operations on vectors include:
        - + (add)
        - - (subtract)
            - Scalars can be added and subtracted from vectors, but a vector can only be subtracted from a vector if they are both the same length
        - * (multiply)
        - / (divide)
        - ^ (exponent – DOES NOT WORK WITH VECTORS, ONLY SCALARS)
            - Vectors can be multiplied by scalars, divided by scalars, or raised to a certain scalar power (using .^ and NOT ^). HOWEVER, when multiplying/dividing/raising vectors to powers using other vectors, both vectors must be the same length and you must specify element-by-element (and NOT matrix) operations using the following operators:
                - .* (vector multiplied by a vector of equal length – element by element)
                - ./ (vector divided by a vector of equal length – element by element)
                - .^ (vector raised to a power by another vector, element by element – also vector raised to a scalar power)
    - Logical Operators:
        - & “and” (compare elements in two vectors)
        - && double “and” (compare single elements)
        - | “or” (compare elements in two vectors)

- || double "or" (compare single elements)
- <   (less than)
- >   (greater than)
- <=   (less than or equal to)
- >=   (greater than or equal to)
- ==   (equal to)
- ~=   (NOT equal to)
- Logical operators compare vectors element by element and return a vector of logical values that tell you where the comparison between A and B is true. The vector of logical values is displayed by MATLAB as a vector of 1's and 0's, but don't forget that this is just MATLAB's way of saying true and false. For example
    - Create vector A: A = [1 2 4 2 1]
    - Create vector B: B = [2 3 4 2 3]
    - The operations:
        - A == B
            - Returns [0 0 1 1 0] or, equivalently, [false false true true false]
        - A < B
            - Returns [1 1 0 0 1] or, equivalently, [true true false false true]
        - A <= B & A ~= B
            - Returns [1 1 0 0 1]
            - This is really the same as saying A < B
        - A > B | A == B
            - Returns [0 0 1 1 0]
            - This is really the same as saying A >= B
- To create a vector of logical values, just use true and false:
    - A = [true true false true false]
- Indexing with logical values is another way to index. This looks a little confusing at first, but is really quite simple when you think about it. If indexing a vector B with a logical vector A, you are really comparing each logical value in A to its corresponding element in B and keeping only the values in B that correspond to true, while throwing out the false values. For example:
    - Create vector A: A = [true false true true false]
    - Create vector B: B = [1 2 3 4 5]
    - C = B(A)
        - Basically, you are making a new vector C that has every value in B that is true
        - C = [1 3 4]
        - The first element of B (the number 1) corresponds to the first element of A, which is true, so we keep that. The second element of B (the number 2) corresponds to the second element of A, which is false, so we throw that out. Do this over and over until you run out of

elements, and you have your answer. This (as well as other slicing techniques) only works with vectors of equal length

Loops/Iteration

The script that I emailed everybody after the last recitation just about covers this topic…in case someone did not get that email, I attached it again. Take a look at it if you need to review, as these are two of the most important things you will need to know for this class

Functions

- Encapsulation: A function is just a 'wrapped up' script; kind of like a black box. Stuff goes in and different stuff comes out. The function has its own workspace and creates its own variables, so you never see what is going on inside (unless, of course, you intentionally un-suppress the output or print variables throughout the course of its execution)
- Procedural Abstraction: A function can solve a general problem with different data (or different inputs) rather than be written for a single input (so, a function built to sum the elements in a vector will work for any vector, while a regular script will only work for one specific vector, such as vec = [4 2 3 1 9])
- The general format of a function (single output and input):
    - function output = function_name(input)
- The general format of a function (multiple outputs and multiple inputs):
    - function [output1, output2, output3, etc] = function_name(input1, input2, input3, etc)
- When calling multiple outputs from a function that provides multiple outputs, be sure that you use brackets around the variables!
    - [maximum_value, where_its_at] = max(vector)
    - The max function gives you both the maximum value in a vector and where that value occurs; however, you have to specify two outputs if you want the index of the maximum value as well as the value itself (use brackets!)

Strings

- To create a string, use single quotes: A = 'this is a string'
- Strings are stored as vector of letters, so anything you can do with a vector, you can do with a string (this includes logical comparisons, indexing, concatenation, slicing, and yes, addition/subtraction/multiplication/division!)
- MATLAB stores letters as numbers, or ASCII code. Look up the ASCII table online to see what numbers correspond to what symbols
- Each type of data has a "character class" assigned to it. In MATLAB, the classes you will need to know are:

char	(a string)
double	(a number)
logical	(a logical value, like true)
cell	(a cell array)

- MATLAB also has functions to convert a variable of one class to another:

- The function “uint8” takes in a string and converts each character to its ASCII numerical equivalent. For example:
    - Create the string str: str = ‘a string’
    - str = uint8(str)
    - Now, str is a vector of ASCII values equivalent to its original character values: str = [97 32 115 116 114 105 110 103]
    - The variable has effectively had its class changed from “char” to “double”
- The function “double” effectively does the same thing as “uint8”, but uses 64 bits (instead of 8 bits) to store the data
- Values for uint8 (or “unsigned 8-bit integer”) can range from 0 to 255
- The function “char” takes in a vector and converts each number to its ASCII character equivalent. For example:
    - The string str = [97 32 115 116 114 105 110 103]
    - str = char(str)
    - Now, str is once again the string ‘a string’
    - The variable has effectively had its class changed back from “double” to “char”

- When a mathematical operation is preformed on a string, such as addition or subtraction, the ASCII value representing the string is what is affected by the operation; so, if you add the number one to a string, every character in the string will have the number one added to its ASCII value and a vector of new ASCII values will be returned. This vector can then be converted to characters using the “char” function. For example**:**
    - Create the string str: str = ‘a string’
    - str = str + 3
    - Now, str = [100 35 118 119 117 108 113 106]. Use the “char” function on str:
    - str = char(str)
    - Now, the value of str is the string ‘d#vwulqj’. Kind of nonsensical, but you get the idea. This applies to any mathematical operations that are valid with vectors
- The “ischar” and “isspace” functions return true values if the input is a character or a space, respectively
- The function “int2str” converts a number to its string representation. For example:
    - int2str(2) = ‘2’ (notice how the input is of type double – a number – and the output is of type char – a string)
    - num2str does the same thing, but has a second input which is simply the number of decimal places you would like to have in the output
- The function “fprintf” prints a string with dynamic (changing or adjustable values) to the screen. For example:
    - Create variable N: N = 4
    - fprintf(‘The variable N is equal to the number %d\n’, N)
    - This is what you should see printed to the screen: ‘The variable N is equal to the number 4’
    - The %d indicates where you want the number display; if N was a string, you would use %c. %f is for real values, and %g is for general numbers
    - The symbol “\n” simply means “new line”. All this does is move your cursor down a to a new line after the string is printed. Similarly, “\t” is just a tab.
- You can have multiple dynamic elements in the string. For example:
    - Create variable N: N = 12

    - Create variable str: str = ‘a string’
    - fprintf(‘The variable N is equal to the number %d and the variable str is equal to the string “%c”\n’, N, str)
    - Notice the how the second input, N, corresponds to the first symbol, %d, and the third input, str, corresponds to the second symbol, %c
    - This is what should be printed to the screen: ‘The variable N is equal to the number 12 and the variable str is equal to the string “a string”’
- The function “sprintf” does the same thing as “fprintf”, except it assigns the string to a variable. For example:
    - Create variable N: N = 21
    - A = sprintf(‘The variable N is equal to the number %d\n’, N)
    - Now we have created a variable A which is the string ‘The variable N is equal to the number 21’

<u>Cell Arrays</u>

- Cell arrays are the primary way that MATLAB stores non-homogenous collections of data (the other way is structures, but you can have a structure inside of a cell array). If this sounds confusing, just remember the bag analogy: a cell array is like a giant bag full of smaller bags. Inside of these bags are the individual data items. To get to those items, you have to open both bags – not just one
- Creating/indexing cell arrays
    - Cell arrays can be created using direct entry, like vectors:
        - A = {[1 2 3], ‘stuff’, true}
    - A cell array of empty cells can also be created using the “cell” function
        - A = cell(1,4)
        - Now, A is an empty cell array with one row and four columns: A = {[], [], [], []}
    - Cell arrays can be indexed like vectors. HOWEVER, you must be careful how you do this…the curly brackets {} are used to directly access the data in a particular cell, while the regular parentheses () are used to access just a particular cell. For example:
        - Create the cell array A: A = {[1 2 3], ‘stuff’, true}
        - A(2) = {‘stuff’}
        - A{2} = ‘stuff’
        - Notice how one is a cell with the string ‘stuff’ inside, and the other is just the string ‘stuff’
    - If you want to access a particular character or number in a string of vector or number inside of a cell array, you have to first access the string or vector itself using {} and THEN access the character or number you want using (). For example:
        - Create the cell array A: A = {[1 2 3], ‘stuff’, true}
        - A{1} = [1 2 3]
        - A{1}(2) = 2
        - A{2} = ‘stuff’
        - A{2}(4) = ‘f’
    - Cell arrays can also be sliced and concatenated like vectors; just use the [] brackets when you want to concatenate two cell arrays together. For example:
        - Create the cell array A: A = {[1 2 3], ‘stuff’, true}

- Create the cell array B: B = {false, [3 4 5 6], ‘morestuff’}
- C = [A B]
- Now, C = {[1 2 3], ‘stuff’, true, false, [3 4 5 6], ‘morestuff’}
- Or, we can create a new cell C with the first two elements of A and the last two elements of B:
    - C = [A(1:2), B(end-1:end)]
    - Now, C = {[1 2 3], ‘stuff’, [3 4 5 6], ‘morestuff’}
    - Notice how we used the () brackets when deciding what parts of the cell arrays to concatenate…for concatenation, we want to work with the individual cells themselves and NOT what they contain
- Items from cell arrays are deleted exactly like items from vectors are deleted: using the () brackets:
    - Create the cell array B: B = {false, [3 4 5 6], ‘morestuff’}
    - B(3) = []
    - Now, B = {false, [3 4 5 6]}

Arrays

- Arrays are just multidimensional vectors: whereas before we only had a vector of columns, now we have an array of rows AND columns
- An array must be “rectangular”; every row has to have the same number of columns and every column as to have the same number of rows. This is very important when you are creating arrays by concatenation.
- Creating an array
    - An array can be created similarly to a vector, but using the semicolon to separate rows. For example,
        - A = [1 2 3; 3 4 5; 5 6 7]
        - This creates an array with three rows and three columns. The first row is equal to [1 2 3], the second row is [3 4 5], and the third row is [5 6 7]
    - Arrays can be indexed like vectors too, but now instead of using a single number to locate an element we use two numbers – first the row, and then the column. For example, using the array A that we just created:
        - A(3,2) returns the number 6
    - We can pull “chunks” from an array just like we can from a vector using the colon operator in the indexing. For example, again using the array A:
        - A(1:3,1) returns the column vector equal to [1; 3; 5]. This basically reads as, “Rows 1 through 3 of A, column 1 of A”
        - A(1:3,3) returns the column vector equal to [3; 5; 7]
        - A(1,2:end) returns the row vector equal to [2 3]. This is simply “first row of A, second to end column of A”, or, more simply, “return all elements in the first row of A that are also in the second to last columns of A”
        - A(end:-1:1, :) returns the array A with all of the rows in reversed
        - A(:, end:-1:1) returns the array A with all of the columns reversed
        - A(end:-1:1, end:-1:1) returns the array A with all of the columns and rows reversed

- When assigning values to an array, if you index outside bounds of that array MATLAB will compensate by filling in the space with zeros, just like with vectors. Using the previously created array A, try typing A(4,8) = 21 and see what happens
- To remove values from an array, simply set A indexed at (r,c) equal to []. For example, using the previously created array A:
    - A(1,:) = []
    - Now, the entire first row of A has been deleted
    - A(:,1) = []
    - Now, the entire first column of A has been deleted
    - A(2:3, :) = []
    - Now, both the second and third row of A has been deleted
- Unlike a vector, you cannot delete individual values from an array; you must either delete an entire row or an entire column (at the very least) or you will get an error
- Arrays can be concatenated just like vectors; however always remember these two things:
    - When concatenating multiple arrays VERTICALLY (one on top of the other), make sure they have the SAME NUMBER OF COLUMNS
    - When concatenating multiple arrays HORIZAONTALLY (side-by-side), make sure they have the SAME NUMBER OF ROWS
    - Really, it's kind of intuitive if you think about it

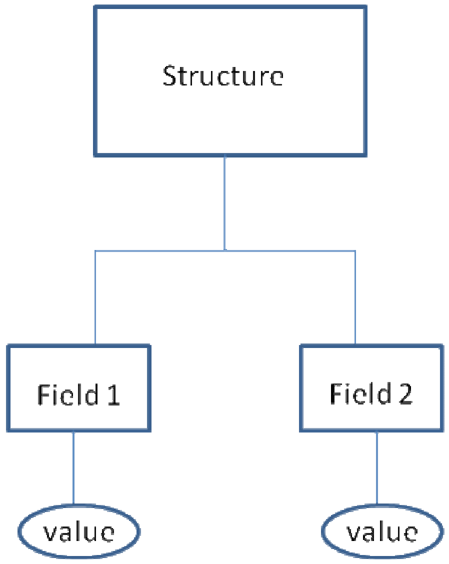
Structures and Structure Arrays

- A structures is MATLAB's second way (apart from cell arrays) to store non-homogenous data (i.e., data that is not all of the same class)
- Layout of a structure

The Structure Name

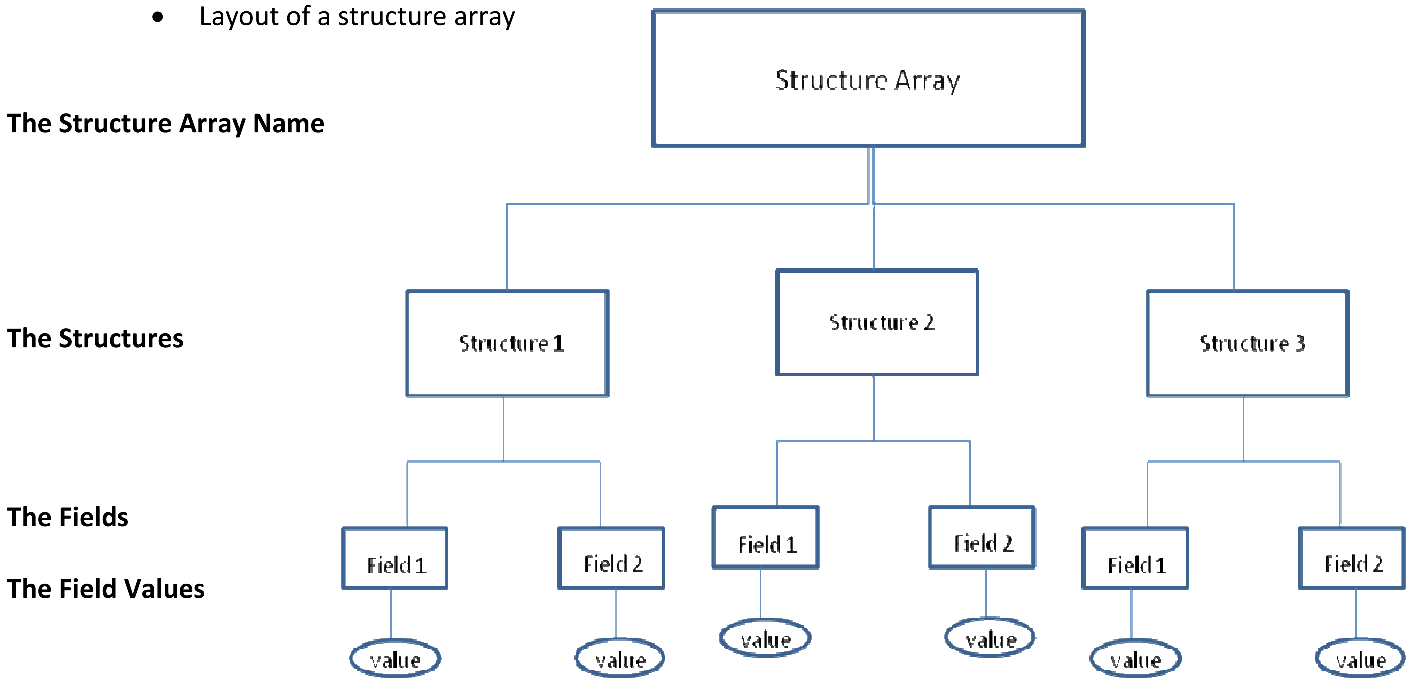
The Fields

The Values in the Fields



- Creating a Structure
    - Manual Creation
        - A structure can be created manually by directly entering the structure name, followed by a dot operator (.) followed by the field name, then an equals sign and the field value. For example:
            - Structure.field_one = ‘Field Value’
            - This creates a structure named “Structure” with a single field named “field_one” and a value at that field equal to the string “Field Value”
        - To add another field to a structure, just use the same procedure used to create the structure:
            - Structure.field_two = [12 34 89]
            - Now the structure “Structure” has another field named “field_two” with the value equal to the vector [12 34 89]
        - The field value can be anything – a logical, a string, a vector, a scalar, an array, another structure, a structure array, a cell array – you get the idea
    - Using the “struct” function
        - A structure can also be created using the built-in “struct” function. This function has the general format

            structure = struct(‘fieldname1’, value, ’fieldname2, value, etc)
            - Structure = struct(‘field_one’, ‘Field Value’, ‘field_two’, [12 34 89])
            - This creates a structure named “Structure” equal to the same structure that was created previously using the dot operator
- The Structure Array
    - A structure array is just – you guessed it – an array of structures. It can have as many structures inside of it as your heart desires, but always remember this VERY IMPORTANT THING:
        - **All structures in a structure array MUST HAVE THE SAME FIELDS IN THE SAME ORDER**
        - Otherwise, you will get an error and on a homework assignment or test, zero points
    - Layout of a structure array



- Creating a Structure Array
    - Structure arrays can be created just like regular structures, but by applying the principles of indexing and concatenation (like we used with vectors, strings, cell arrays, etc). A structure array can be indexed by number, just like a vector. For example:
        - s.field_one = ‘The Answer is…’
        - Creates a structure named “s” with a single field “field_one” with a value equal to the string ‘The Answer is…’
        - s(2).field_one = 42
        - Adds a second structure to s – now s is a structure array with two structures, both with one field named “field_one”
            - To index the structure array s, use the parenthesis just like you would index a vector
            - For example, typing s(1) would return the first structure in the structure array s
        - Now, what happens if we type s(2).field_two = ‘value’? Remember, all structures in a structure array must have THE SAME FIELDS IN THE SAME ORDER (in case you missed it, this is important). By adding another field to s(2) we automatically add that same field to all other structures in the structure array s. However, the since we did not assign a specific value for that field to those other structures, they take on the default value equal to empty brackets. For example:
            - s(2).field_two = ‘What an anti-climax’
            - Now, when we access s(1) we find the second field “field_two” has also been added to s(1) and s(1).field_two = []
        - This principle can be extended to almost any length. For example, working with our structure array s with the fields “field_one” and “field_two”, we can add a new structure thusly:
            - s(3).new_field = ‘A new field with a new value’
            - Ok, so we just added a third structure to our array that contains neither of the fields in that array. MATLAB will compensate for our own incompetence and add the field “new_field” to the first two structure in s, give it the value [], and add the “field_one” and “field_two” to the newly created third structure and give them the both the value []
            - s is now a 1x3 structure array with the fields “field_one”, “field_two” and “new_field”
    - Structure arrays can also be created using the “struct” function, but when using the “struct” function be sure that you give an equal number of values to every field. For example:
        - new_s = struct(‘field_one’, {‘one’, true, [1 2 3]}, ‘field_two’, {4, ‘thirteen’, false})
        - This creates a new 1x3 structure array named new_s with the fields “field_one” and “field_two”
        - However, if we try the following, we get an error:

        - bad_s = struct(‘field_one’, {‘one’, true}, ‘field_two’, {3, 4, 5})
        - Notice how the number of elements for “field_one” is different than for “field_two”. MATLAB does not compensate for this error
    - Multiple structures and structure arrays can be concatenated together to form a new structure array (just like vectors) provided that they ALL HAVE THE SAME FIELDS IN THE SAME ORDER
- Structure related functions
    - rmfield
        - The “rmfield” function has two inputs: first the structure array and then the field you want to remove from that structure array. For example:
            - s = rmfield(s, ‘field_one’)
            - Now, the field “field_one” has been removed from the structure s
            - NOTE THE FOLLOWING:
                - rmfield(s, ‘field_one’)
                - The code above does ABSOLUTELY NOTHING PRACTICAL. You did not assign an output to the function, so no data is saved. You must set something equal to “rmfield” in order for it do anything useful
    - fieldnames
        - The “fieldnames” function returns a vertical cell array with all of the fieldnames in the input structure. For example:
            - new_s = struct(‘field_one’, {1, 2, 3}, ‘field_two’, {3, [3 2 1], true})
            - names = fieldnames(s)
            - Now, “names” is a vertical cell array with two values ‘field_one’ and ‘field_two’
    - isfield
        - The function “isfield” has two inputs: the structure array and a field (as a string). It returns a true if the field name you put in is a field in the structure, and a false otherwise.
    - setfield
        - This function takes in a structure array, a field name (as a string) and a value that you want that field to have. For example:
            - new_s = struct(‘field_one’, {1, 2, 3}, ‘field_two’, {3, [3 2 1], true})
            - new_s = setfield(new_s(1), ‘field_one’, ‘new_value’)
            - Now the first structure in new_s has a value of ‘new_value’ in the field “field_one”
                - This is the same as doing the following:
                - new_s(1).field_one = ‘new_value’
        - Again, this function will do ABSOLUTELY NOTHING PRACTICAL if you do not assign it an output
    - getfield
        - This is similar to setfield, except that instead of giving a new value to a field in a structure, it just returns the current value. For example:
            - new_s = struct(‘field_one’, {1, 2, 3}, ‘field_two’, {3, [3 2 1], true})
            - value = getfield(new_s(1), ‘field_one’)
            - Now, the variable “value” is equal to the number 1

            - This is the same as doing the following:
            - value = new_s(1).field_one
        - One more time for emphasis: this function will do ABSOLUTELY NOTHING PRACTICAL if you do not assign it an output

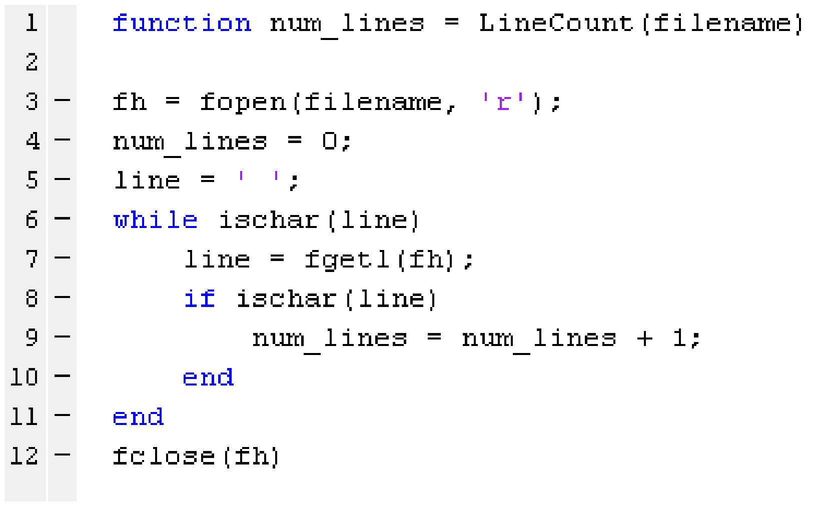
File I/O

- MATLAB can act as the medium from which data is gathered from and written to different kinds of files. This is basically all that file input/output is.
- File I/O is divided into two different types: upper level and lower level. Upper level file I/O is for dealing with files that have some kind of predictable format (such as excel files or delimited files) while lower level file I/O, as far as it concerns us here, is for dealing with simple text files.
    - Upper level
        - Several functions are associated with reading data and writing data for upper level file I/O – each one is tailored to deal with a specific type of file
            - xlsread
                - xlsread is the function for reading data from an excel file. It takes in the filename as a string (for example, 'exceldata.xls') and returns three things. In order, they are: the numeric data (as a matrix), the text data (as a cell array), and all of the data (as a cell array). Often you will see these outputs abbreviated as num, txt, and raw, but really they are variables so they can be named anything – just remember the order they are called in the function and you should have no problems if they try to trick you on the test
                - The numeric data is returned as the smallest possible array (so every row has the same number of columns and every column has the same number of rows). So, if there is a cell with text right in the middle of the numeric data array, then that cell will be filled with the value NaN, which basically stands for 'Not a number'; NaN is of type double and is basically just a placeholder.
                - The text data is also returned as the smallest possible cell array, and if there is numeric data somewhere inside of it, it returns a cell with an empty string character where the numeric data was.
                - Finally, if there are empty cells in the raw data, a cell with the NaN value is returned in the place of the empty cell
            - xlswrite
                - xlswrite is the function for writing data to an excel file. It has two primary inputs: the filename (a string) and the cell array of raw data
            - dlmread
                - dlmread is the function for reading delimited files – the data in the delimited files should be NUMBERS, not characters!
                - The inputs to dlmread are: the filename to be read and (optional) your delimiter (input it in quotes as a string). If you do not give a delimiting character to the function, MATLAB will

            use regularly repeating characters in the file as the delimiter. dlmread returns an array of numeric data from the file
    - dlmwrite
        - dlmwrite is the function for writing an array of numeric data to a delimited file. It has two primary inputs: the filename (as a string) that you will be writing the data to and the array of data. The third optional input is the delimiter itself; if you do not specify the delimiter, the default will be a comma
    - csvread
        - csvread is very similar to dlmread, except it deals with files where the NUMERIC data is separated by commas (csvread/write only work for files with numeric data, just like dlmread/write). It has one input – the filename as a string – and one output, which is an array of data
    - csvwrite
        - csvwrite takes in the new filename and the array of data and writes the data to a new file, automatically separating the numbers with commas
- Lower level
    - Lower level file I/O is a little bit different than upper level – this is for files without a predictable format, and is considerably more “messy” than upper level
    - I like to break the process of dealing with these files into several steps: 1. Open the file, 2. Get the data, 3. Manipulate the data, 4. Close the file
        - Step one is opening the file. We do this with the fopen function
            - fopen
                - fopen has two inputs: the filename (as a string) and the permission. There are three permissions: the read permission (‘r’), which is simply for gathering data from the file, the write permission (‘w’) for creating a new file or overwriting an old file, and the append permission (‘a’) for adding data to the end of an existing file.
                - fopen returns the file handle (or file identifier) corresponding to the file that it opens; this is simply a single number, and that number is just the computer’s way of “naming” or identifying that file.
                - If you open several files at once, then make sure you give your file handle variables different names! (for example, fh1 = fopen(‘file1.txt’,’r’),
                  fh2 = fopen(‘file2.txt’, ‘w’)
        - Step two is gathering the data from the file. For lower level file I/O, MATLAB sees all of the data as a string, and the data is read in ‘lines’ of text. There are two functions to do this: fgetl and fgets
            - fgetl/fgets
                - fgetl and fgets do basically the same thing – they both take in a file handle number and return the next line of text in that file (for example, if you call it once it will

return the first line of text, if you call it a second time it will return the second line of text, etc). However, fgetl returns the line of text WITHOUT the end line character, and fgets returns the line of text WITH the end line character. Usually we are not concerned with the end line character, so we just use fgetl.

- The file MUST be opened in **<u>read mode only</u>** to use the fgetl and fgets functions!
- Once you try to use fgetl and you reach the end of the file, the function returns the number -1. There is obvious 'while' loop potential here. Below is an example of a function designed to count the number of lines in a file

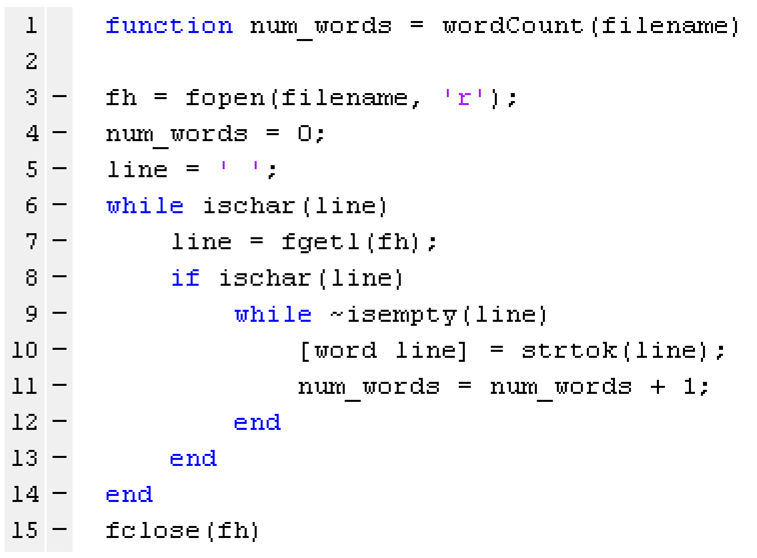


```
1   function num_lines = LineCount(filename)
2
3 - fh = fopen(filename, 'r');
4 - num_lines = 0;
5 - line = ' ';
6 - while ischar(line)
7 -     line = fgetl(fh);
8 -     if ischar(line)
9 -         num_lines = num_lines + 1;
10 -    end
11 - end
12 - fclose(fh)
```

Line 3: Open the file with (read mode)
Line 4: Initialize the count to 0
Line 5: Set line to a character – any character will do, since we just want to enter the loop
Line 7: Get the next line
Line 8: Check to see if the line is a character
Line 9: If it is, add one to the line count

Line 12: Don't forget to close the file!

- Step three would be to manipulate the data. Now that we know how to get the individual lines, we can parse these lines into 'tokens' using the strtok function
    - strtok
        - strtok has two inputs – the string you want to parse into a token, and the delimiter that separates that token from the rest of the string (if that input is not given, it is assumed to be a space).
        - It has two outputs: the token that is "chopped" off of the string, and the rest of the string (including the delimiter)
        - If you try to tokenize a string where there are no delimiters, then the token will be the string, and the rest of the string will simply be an empty string (since there is nothing left after the token is removed)
        - There is obvious loop potential here, too. Below is an example of a function that counts the number of *words* in a text file; notice that the only real difference between this function and the last one is the extra while loop



```
 1     function num_words = wordCount(filename)
 2
 3 -   fh = fopen(filename, 'r');
 4 -   num_words = 0;
 5 -   line = ' ';
 6 -   while ischar(line)
 7 -       line = fgetl(fh);
 8 -       if ischar(line)
 9 -           while ~isempty(line)
10 -               [word line] = strtok(line);
11 -               num_words = num_words + 1;
12 -           end
13 -       end
14 -   end
15 -   fclose(fh)
```

Line 7: Get the next line of text from the file
Line 8: Check to see if the line is still a character
Line 9: If it is, begin the next loop
Line 10: Chop off the next token from the line
Line 11: Add one to the number of words

Line 15: Close the file

- "Manipulating the data" can also refer to writing the data to a new file. If you want to write lines of text to a file, use the fprintf function
    - fprintf
        - fprintf has several inputs: the first input is the file handle that corresponds to the file you are writing text to, the second is a string, and the third is the value that corresponds to any placeholders (such as '%s' for a string, '%g' for a general number value) in the string
        - Example: If you wanted to print a string of text to a file whose file handle was stored in the variable fh, and that string was stored in the variable 'str', then you would write the following code: fprintf(fh,'%s',str)
- The last step is just closing the file; we do this with the fclose function, which takes in the file handle corresponding to the file we want to close

Recursion

- The whole idea behind recursion is simply a function calling itself over and over, its input being adjusted each time until a 'terminating condition' is met, at which point each function starts doing its work and spitting out an answer in the opposite order in which they were called.
- The three pillars of recursion (A GUARENTEED TEST QUESTION):
    - **A recursive function must have a terminating condition**

- **A recursive function must move towards that terminating condition (that is, every time it calls itself its input must be adjusted so that it gets closer to the terminating condition)**
- **A recursive function must call a copy (or clone) of itself**

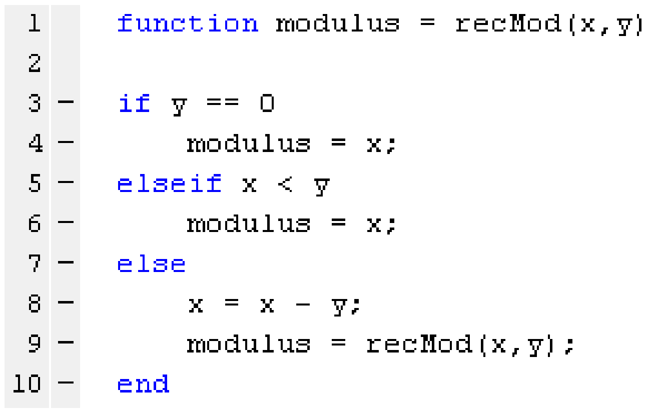
- Usually the first thing you see in a recursive function is that mostly everything is contained within a big conditional statement – the first if statement will, of course, be your terminating condition (if the output meets some logical criteria, then set the output equal to this)
- So, before writing a recursive function, ask yourself this: What is my terminating condition?
- Once you know your terminating condition, you can then decide how you want to design your recursive call so that the new input is always moving towards that condition – this, of course, is up to you and the nature of the question.
- A good technique for designing a recursive function to solve a problem is to solve the problem by hand in a repeated fashion first – then look for the pattern and apply it
- Finally, the best way to learn recursion is by example, so I have provided several examples here:

Recursion example 1: Using recursion to find the modulus of a number after division.

Let's start out with a specific example by hand, and see if we can find the pattern. Take the numbers 13 and 4. What is the modulus of 13 and 4?

First divide 13 by 4 to get 3 with remainder 1.
Next, subtract 4 from 13 to get 9.
Now, divide 9 by 4 to get 2 with remainder 1.
Next, subtract 4 from 9 to get 5.
Now, divide 5 by 4 to get 1 with remainder 1.
Next, subtract 4 from 5 to get 1.
We no longer get a whole number if we divide 1 by 4, so 1 must be the modulus of 13 and 4.

Examining this pattern, we find a terminating condition – when the first number (which is 1 in our final case) is less than the second number (which is 4, and never changes) the modulus is just the first number. Also, remember that the modulus of a number and 0 is just that number – this gives us two terminating conditions.
Now, how do we adjust the function's input when we call it again so it moves towards this terminating condition? Well, we just subtract the second number from the first number, and that result is our new first input (the second never changes, as you can see
Here is the code for a function that calculates the modulus of two numbers recursively:



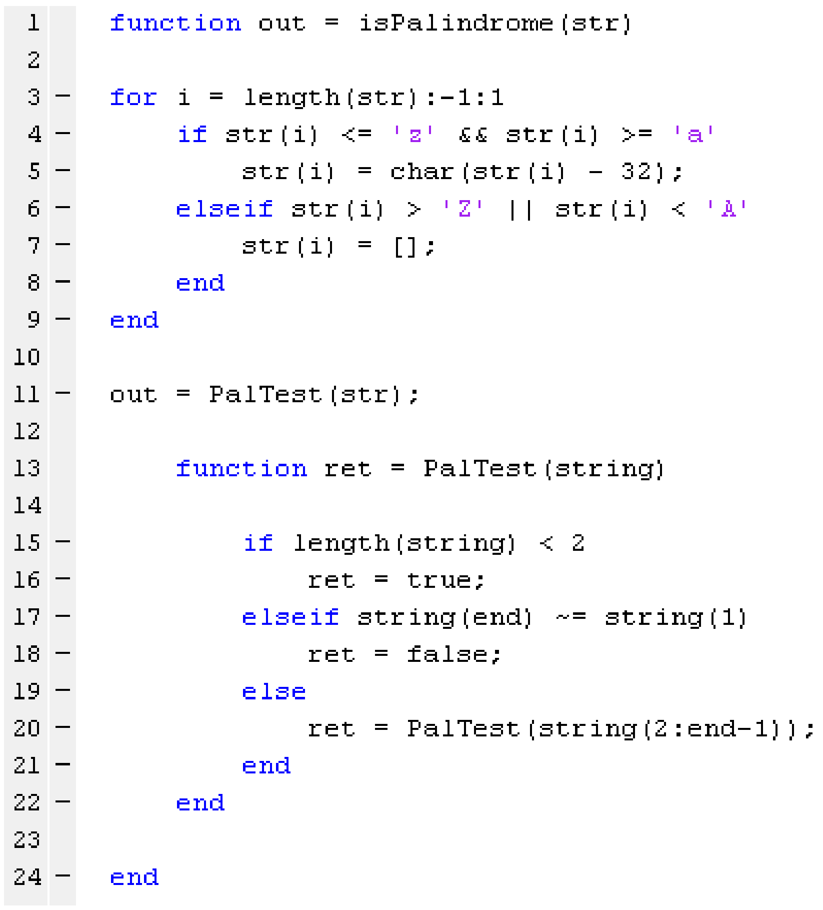
```
1      function modulus = recMod(x,y)
2
3  -   if y == 0
4  -       modulus = x;
5  -   elseif x < y
6  -       modulus = x;
7  -   else
8  -       x = x - y;
9  -       modulus = recMod(x,y);
10 -   end
```

Line 3: The first terminating condition

Line 5: The second terminating condition

Line 8 – 9: If the terminating conditions are not met, we must call the function again and ADJUST ITS INPUT so that it every time it is called, it gets a little closer to the terminating condition. In this case, we inferred from our pattern that we must subtract the second number from the first to change the first input

- Wrapper/Helper functions
    - A wrapper function is just a way to clean up your data – inside of a wrapper function, you write a new function (the helper function) that does all of the actual work. This function is invisible from the outside – the actual function name is that of the wrapper function (it's "wrapped around" the helper)
    - So, the wrapper function takes in the inputs and tests them to see if they are bad/good/whatever. If they are/are not, then it adjusts them and fixes them. Finally, it passes these inputs to a helper function which does all of the calculations – the helper function is the one that spits out the actual answer (but it passes it to the wrapper function in the process)
    - Here is an example of a recursive function that determines whether or not a string is a palindrome:



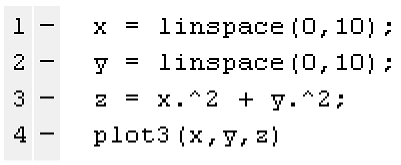
```
1      function out = isPalindrome(str)
2
3  -   for i = length(str):-1:1
4  -       if str(i) <= 'z' && str(i) >= 'a'
5  -           str(i) = char(str(i) - 32);
6  -       elseif str(i) > 'Z' || str(i) < 'A'
7  -           str(i) = [];
8  -       end
9  -   end
10
11 -   out = PalTest(str);
12
13         function ret = PalTest(string)
14
15 -           if length(string) < 2
16 -               ret = true;
17 -           elseif string(end) ~= string(1)
18 -               ret = false;
19 -           else
20 -               ret = PalTest(string(2:end-1));
21 -           end
22 -       end
23
24 -   end
```

Line 1: The name of the function; this is actually the *wrapper* function
Line 3: The clean-up process; this loop and the following conditionals convert all letters to a single case and remove everything from the string that is not a letter

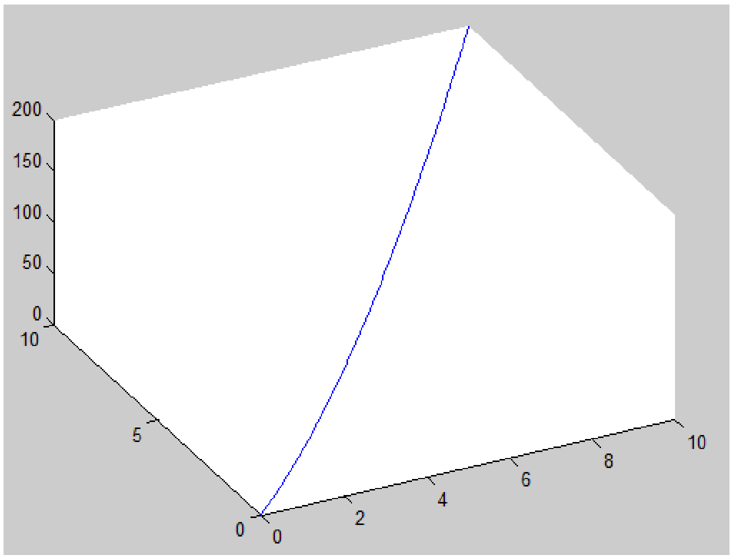
Line 11: Now the wrapper assigns its output and passes its modified input (str after it has been "fixed") to a hidden, helper function
Lines 13 – 22: This is the hidden helper function that will determine whether or not str is a palindrome using recursion

Plotting

- 2D plotting is pretty straightforward: you use the plot function which takes in two variables (two vectors x and y) and a third parameter which defines the color and style of the line or points. **BOTH X AND Y HAVE TO BE THE SAME LENGTH!!!!!!**. This is obviously very important
- You will usually see the linspace function used to define x, and then y will be (typically, though it doesn't have to be) a function of x, like y = x.^2 (notice the "dot" operator before the exponentiation, indicating vector element-by-element exponentiation)
- The only thing that could possibly be tricky about this is when dealing with parametric plots. For a parametric plot (such as a circle) you would define a single, independent variable (such as theta), and then you would plot both x and y as functions of theta (so, for a circle, x would be the cosine of theta and y would be the sine of theta)
- You can plot multiple data sets on the same plot, either by calling them all in one plot command (i.e., plot(x,y,x1,y1,x2,y2…)) or by using the hold on function
- There are a plethora of functions associated with plot such as axis, legend, title, xlabel, ylabel, etc, and I would advise that you LEARN THEM. Type "help plot" for a succinct list, or just check the book reference.
- 3D Plotting is a little more in-depth; remember, every plot in MATLAB is actually a 3D plot, but all the values for z when you use the regular, boring plot function are automatically set to 0, resulting in a plot that is in only the x-y plane. If we wanted to plot the same thing but in the y-z plane, we would use plot3 and set all of the x values equal to 0, and similarly, if we wanted to plot a function in the x-z plane we would set all of the y values equal to 0.
    - The real power behind plotting in three dimensions is the plotting of surfaces – basically functions of two variables (for those of you who are calculus-inclined)
    - So, if I were to graph the function $z = x^2 + y^2$, first I would have to define x and y along some domain; next, I would define z. Finally, we will use the plot3 function to plot x, y, and z. Below you can see the code used to accomplish this and the resulting plot:



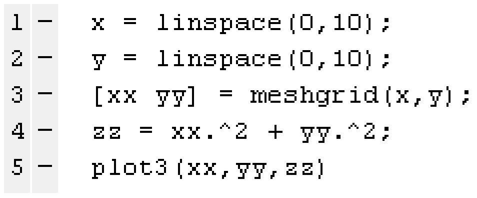
```
x = linspace(0,10);
y = linspace(0,10);
z = x.^2 + y.^2;
plot3(x,y,z)
```



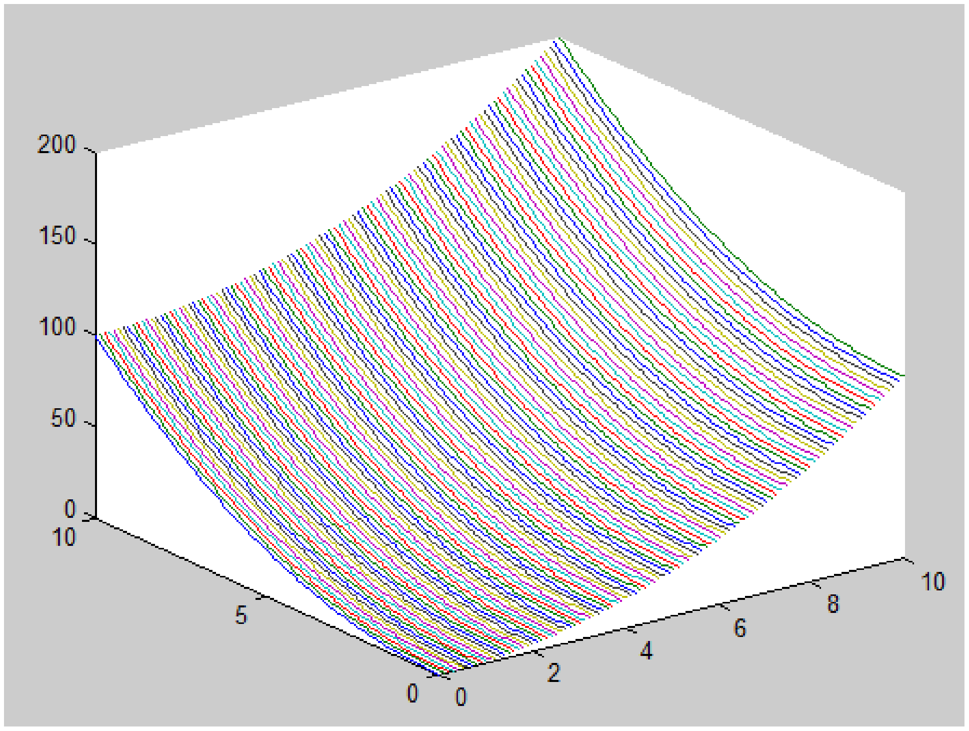
- What went wrong? This looks like a regular curve! That is, in fact, because it is just a curve – but not, dear friends, a *regular* curve. It is a single curve on the surface $z = x^2 + y^2$. Why did MATLAB plot a single curve and not the whole surface? Well, this requires a look at what MATLAB does when it plots the vectors x, y, and z.
    - We know that a single curve along a surface represents but one possible combination of x and y. Thus, it comes to reason that we only plotted one possible combination of x and y (since we got just one curve). When MATLAB plots, it reads through the vectors one element at a time and matches up that particular element with the corresponding element in the other vectors. For example, lets' have vector x = [1 2 3 4], vector y = [1 2 3 4], and vector z = x.^2 + y.^2, or z = [2 8 18 32]. When we type plot3(x,y,z), our points are going to be (1,1,2), (2,2,8), (3,3,18), and (4,4,32). See the issue?
    - How do we resolve this? Well, it would appear that we need to trick MATLAB into reading every possible combination of x and y and calculating the corresponding z. We can do this manually by creating two matrices that, when read through element-by-element, will give us every possible combination of x and y, or we can use the meshgrid function and it will do that for us.
    - The meshgrid function (typically) has two inputs – two vector sets of points – and for those two inputs will return two other values: two new matrices (each corresponding to the vector set of points inputted) that, when read through column by column, will give us every possible combination of all of the points in our first input with all of the points in our second input. For example, let's say x = [1 2 3 4 5] and y = [1 2 3 4 5]. If I say [xx yy] = meshgrid(x,y), xx and yy will equal what you see below:

```
xx =

     1     2     3     4     5
     1     2     3     4     5
     1     2     3     4     5
     1     2     3     4     5
     1     2     3     4     5


yy =

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

    - Notice how the in xx, the x vectors are arranged one on top of the other, and in yy, the y vectors are arranged side-by-side. When we read down the columns, one element at a time, we will eventually read through every possible combination of x and y. Now we can define z (or, conventionally, you may see this variable named “zz”) as x.^2 + y.^2 and use plot 3 again. Observe the difference below:



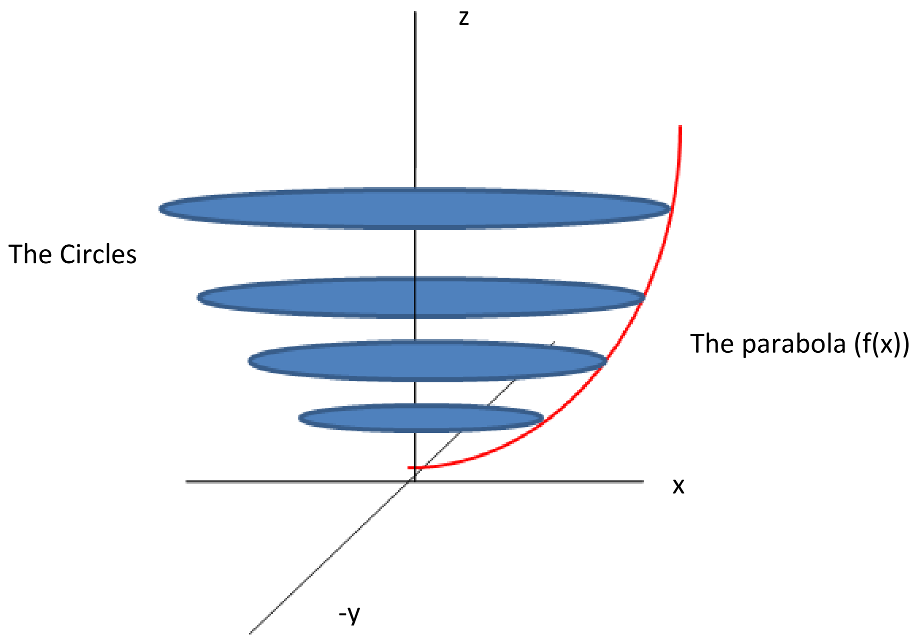
```
x = linspace(0,10);
y = linspace(0,10);
[xx yy] = meshgrid(x,y);
zz = xx.^2 + yy.^2;
plot3(xx,yy,zz)
```



- Hey, it worked! What a surprise. But one question remains: how many curves are there? To answer this question and dispel the suspense, there are exactly 100 curves…count them yourself if you doubt me, though I wouldn’t advise it, as they are rather tiny. How did I come to this conclusion? Well, there are exactly 100 elements in x and y – which equals 100 curves to this surface, or 100 possible sets of combinations to yield our curve zz. If I defined x as x = linspace(1,10,1000) then there would be 1000 curves, and so on and so forth
- Say you wanted a more aesthetically pleasing surface. Well, just use the surf or mesh function in place of plot3, and you will get a surface with filling in between the curves (in the case of surf) or a nice chicken-wire pattern connecting every point (in the case of mesh). The smoothness of your surface will, of course, be directly proportional to the number of curves you have
- There are several functions associated with surf that you need to know. The first (and one of the more important) is the shading function. This function has three possible inputs: interp, smooth, and faceted. Faceted is the default value, but interp will give you a nice smooth color change, while smooth will give you just one color. Know other functions such as colormap as well

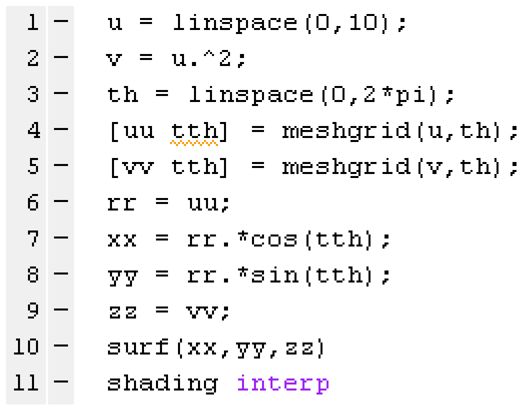
Bodies of Rotation

It helps to have a visual when dealing with bodies of rotation. Below is a picture of half of a parabola rotated around the z-axis, with some notes on how we can think about it so we can reproduce it in MATLAB:



The resulting bowl shape can be thought of a as just a bunch of circles stacked on top of each other – now all we have to do to get this bowl is plot all of these circles. How do we plot a circle lying in a plane in 3-D space? Parametrically, (which is the easiest way) we define an independent variable theta that ranges from 0 to 2π, and our x coordinate becomes the radius of the circle times cos(theta) and our y coordinate becomes the radius of the circle times sin(theta). Our z coordinate can just be thought of as how “high” our circle is off of the “ground”, or, in the above case, how far away the center of the circle is from the origin (which is also the vertex of our parabola). So, for the case of the rotation about the z-axis, as we move along the x-axis the radius of our circles will always be equal to our x-axis value, and the “height” of our circles will always be equal to the value of our function at that point. Since the circle is in the x-y plane, our x and y coordinates will be r*cos(theta) and r*sin(theta), respectively.

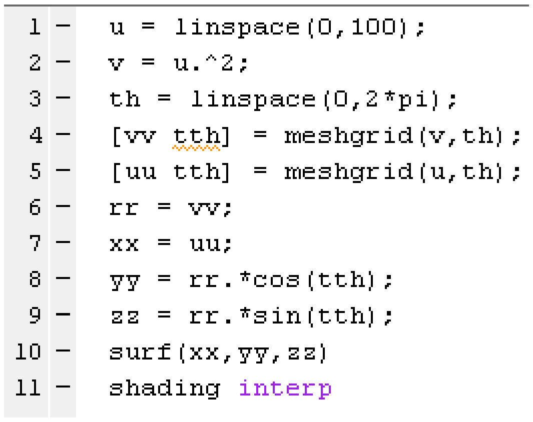
Below is the code used to rotate a function in the x-z plane about the z axis:



```
 1 -   u = linspace(0,10);
 2 -   v = u.^2;
 3 -   th = linspace(0,2*pi);
 4 -   [uu tth] = meshgrid(u,th);
 5 -   [vv tth] = meshgrid(v,th);
 6 -   rr = uu;
 7 -   xx = rr.*cos(tth);
 8 -   yy = rr.*sin(tth);
 9 -   zz = vv;
10 -   surf(xx,yy,zz)
11 -   shading interp
```

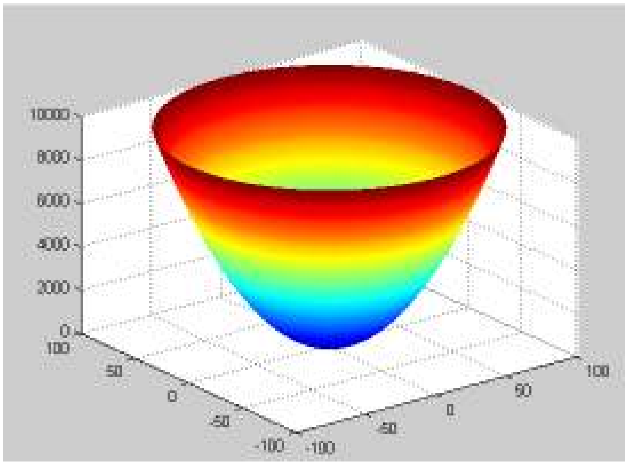
Line 1: Define the independent variable u
Line 2: Define the dependent variable v (this is a simple parabola)
Line 3: Define the independent variable theta
Line 4-5: Find every possible combination of u, v and theta
Line 6: The radius of the circles will be the x (or u) coordinate
Line 7-8: The circles are in the x-y plane
Line 9: The z coordinate is defined by the function v
Line 10: Make the graph using surf
Line 11: Set the shading to interpolated, giving a smooth appearance

For rotation about the x axis, we just rethink our rr, xx, yy, and zz values. Since the circles will now be in the y-z plane, the yy and zz coordinates will now be rr.*cos(tth) and rr.*sin(tth). The radius of the circle will now be equal to the function itself (or vv), and our x coordinates will be unchanged, or equal to our independent variable uu. Below is the code used to rotate the same parabola about the x-axis:

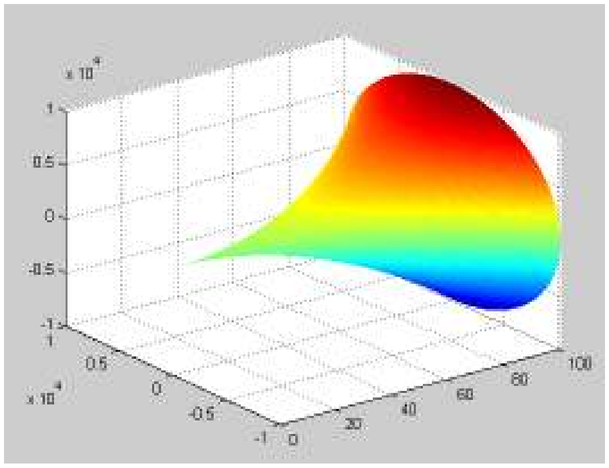


```
u = linspace(0,100);
v = u.^2;
th = linspace(0,2*pi);
[vv tth] = meshgrid(v,th);
[uu tth] = meshgrid(u,th);
rr = vv;
xx = uu;
yy = rr.*cos(tth);
zz = rr.*sin(tth);
surf(xx,yy,zz)
shading interp
```

Below are the resulting plots:



The z-axis



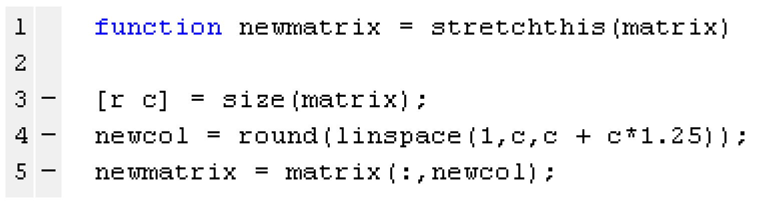
The x-axis

Images

- An image in matlab is just an enormous, three-dimensional array of numbers, where each number represents a single pixel
- Each pixel has a red, green, and blue color value – the values themselves are of type uint8 (unsigned eight-bit integers) and can have values ranging from 0 – 255
    - The lower the value for a color layer, the less of that color the pixel has. So, a pixel that had values of 0, 0, and 255 for red, green, and blue respectively would be a completely red pixel
    - Remember, the values themselves are measures of intensity – 0 means no color at all, while 255 means all color. So, a pixel with all 0 values will be black (no intensity for

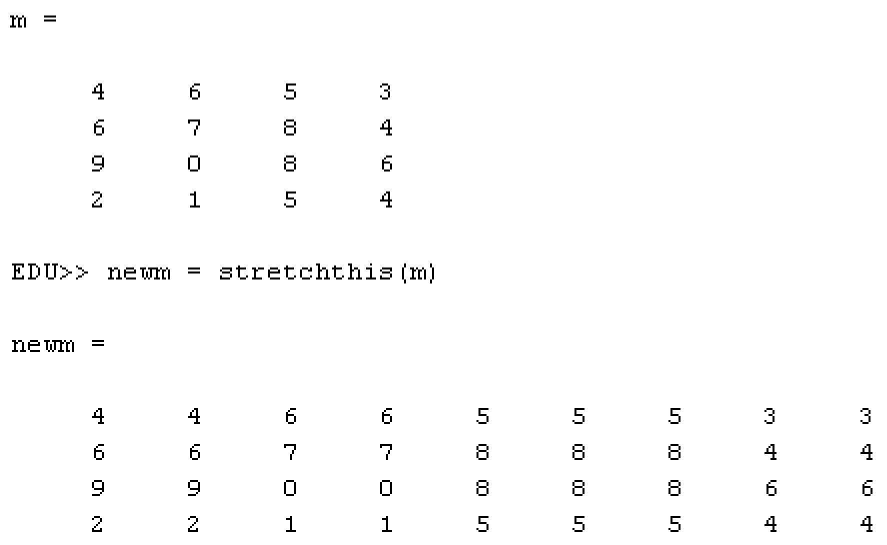
every color) and a pixel with 255 for all colors will be white (maximum intensity for all colors)

- Reading image data
    - To read an image into matlab, use the imread function
    - Two inputs: the filename (a string) and the format (a string) (such as ‘jpg’, ‘tiff’, etc); the second input is optional
    - Single Output: A three-dimensional matrix with row and column dimensions equal to the number of pixels in the image and three color layers: red (1) green (2) and blue (3)
- Displaying image data
    - To display an image, two options are available: the image function or the imshow function
    - The image function takes in the raw image data (NOT the filename) and will stretch or shrink the image to fit the size of a normal plot figure
    - The imshow function also takes in the raw image data, but will display the picture at its actual size
- Writing image data to a new file
    - The function imwrite takes in the image data, the name of the new file, and the file extension (i.e. ‘jpg’, ‘tiff’, etc)
- Accessing and Manipulating image data
    - Since images are just glorified arrays, everything you can do with an array you can do with an image – and all the same rules apply. This includes indexing, slicing, concatenating, etc
    - Stretching/Shrinking an image
        - Images can be indexed by vectors, just like arrays, and through this special type of indexing they can either be stretched or shrunk
        - The linspace function is used to create the indexing vector; it should range from one to the number of rows or columns, and the total number of elements should be equal to however many new rows or columns you want your image to have
        - For a visual example of what will happen to your image when we change its dimensions using this technique, consider the function ‘stretch this’ below that takes in the given matrix and stretches it horizontally by 125% its original length:



```
function newmatrix = stretchthis(matrix)

[r c] = size(matrix);
newcol = round(linspace(1,c,c + c*1.25));
newmatrix = matrix(:,newcol);
```

First we find the size of the matrix
Next, we make our new vector index – the starting number will be the first column (1), the ending will be the last (the total number of columns) and our new number of columns will be our old plus the old times 1.25 (since we are increasing the length by 125%). Finally, round the vector (to get all whole numbers) and use it to index the old matrix. Below are the results:



```
m =

     4     6     5     3
     6     7     8     4
     9     0     8     6
     2     1     5     4

EDU>> newm = stretchthis(m)

newm =

     4     4     6     6     5     5     5     3     3
     6     6     7     7     8     8     8     4     4
     9     9     0     0     8     8     8     6     6
     2     2     1     1     5     5     5     4     4
```

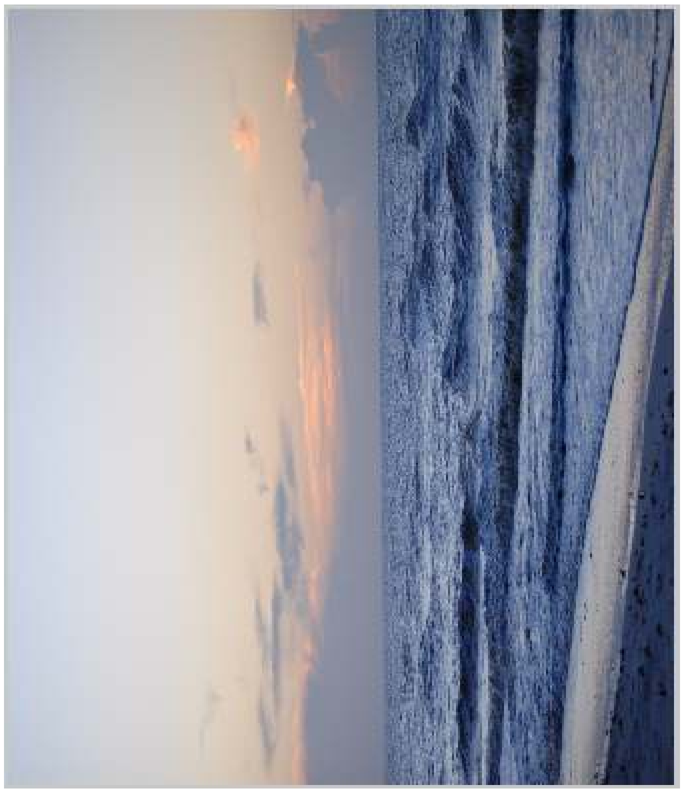
    - Notice how the columns are duplicated across. This indicates that when an image is stretched in this matter, pixels will be duplicated and thus the image will undergo some degree of distortion. The same function can be used to stretch an image by 125% its original size, but with the following change: the third line ([r c] = size(matrix)) MUST BE CHANGED to [r c d] = size(matrix). Notice the third output to the size function; since an image array has three dimensions, three size outputs must be called, otherwise matlab will multiply the number of columns by the number of layers, and we will get an incorrect number of columns for c.
- Slicing and flipping an image
    - Since an image is just an array, indexing that array we can 'cut' the image into pieces. For example, if we have some image matrix "im" and we want just the top right quarter of the image, we could index the image thusly: newim = im(1:round(end/2), 1:round(end/2),:) (we round the indexes just in case the image has an odd number of rows or columns)
    - To flip an image upside down, just invert the rows: newim = im(end:-1:1,:,:)
    - To flip an image left to right, invert the columns: newim = im(:,end:-1:1,:)
- Rotating an image
    - To rotate an image 90^o, the image must first be 'transposed' – we turn the rows of the image matrix into columns and the columns into rows. This is accomplished using the ' operator, or the "transpose" function. The image matrix must be transposed one layer at a time – you can't do all layers at once. Below is an image, followed by the code executed to modify it, and lastly the modified image. Notice the effect transposing the image matrix has on the image:

The original image

The code used to transpose it, one layer at a time

```
EDU>> im = imread('Cape_Sunset.jpg');
EDU>> newim(:,:,1) = im(:,:,1)';
EDU>> newim(:,:,2) = im(:,:,2)';
EDU>> newim(:,:,3) = im(:,:,3)';
EDU>> imshow(newim)
```

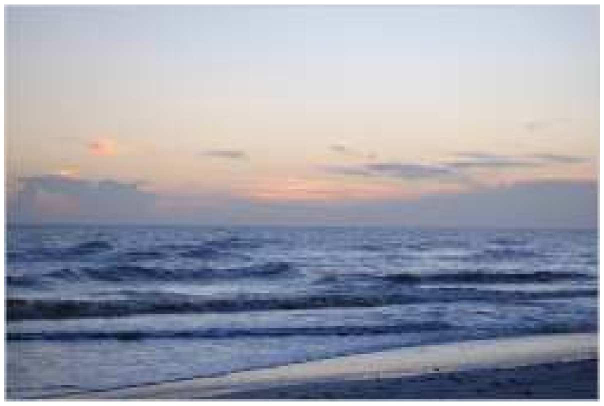
The Transposed Image



Now to get the rotated image, all we have to do is flip the rows for a counter-clockwise rotation and flips the columns for a clockwise rotation
Note: MATLAB's built in rot90 function will also accomplish this same thing: it takes in the image data and the number of times you want it rotated 90^o counter-clockwise

- Changing colors
    - Manipulating colors in an image is simply a matter of indexing. Here are three examples of a way to manipulate the color layers in an image:

The original image:

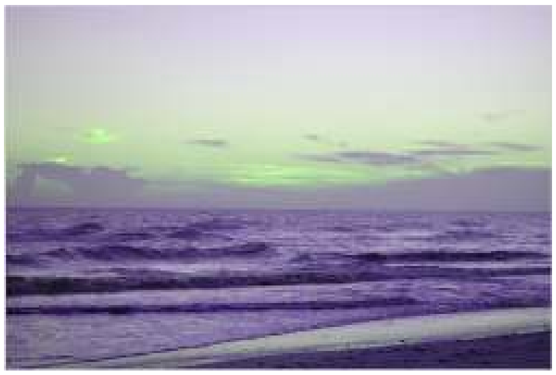


Reverse the red and green layer intensities:

```
EDU>> newim = im(:,:,[2 1 3]);
```

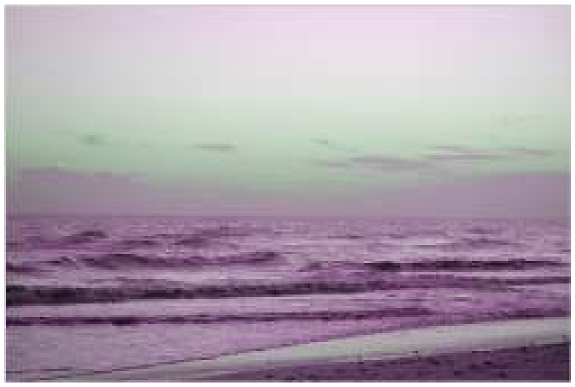
Alternatively, first setting newim = im,

```
EDU>> newim(:,:,1) = im(:,:,2);
EDU>> newim(:,:,2) = im(:,:,1);
```



Give the red layer the same intensity as the blue layer:

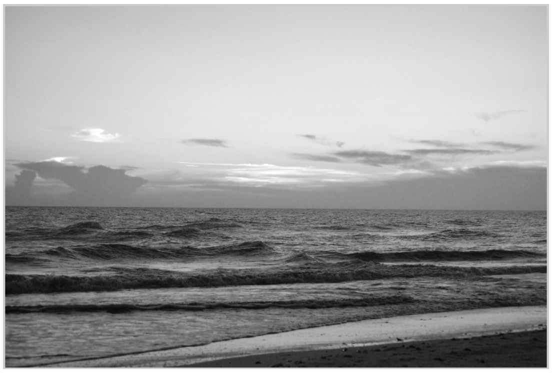
```
EDU>> newim = im(:,:,[3 2 3]);
```



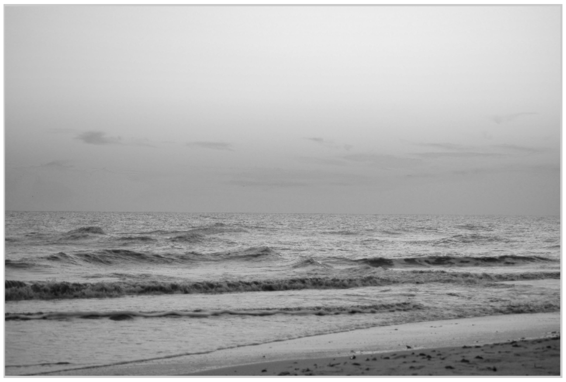
To understand grayscale, remember that when we mix equal amounts of red, blue, and green we get black. Thus, since the pixel numbers for each color layer (0 – 255) are a measure of intensity, then when they are all the same, depending on their intensity, they will give us a shade of black, ranging from completely black to completely white. So a pixel with values 45, 45, 45 for red, green, and blue will give us a very dark shade of black, while a pixel with values 213, 213, 213 for red, green, and blue will give us a much lighter shade of black (closer to white). And grey of course, is just a shade of black.

So, we have two ways to convert a color to grayscale: make every color the same intensity as one particular layer, or make every color the same intensity by averaging all the color layer values and applying them to each layer. Here is the image converted to grayscale using the first method, first with the red layer and then with the blue layer:

```
EDU>> newim = im(:,:,[1 1 1]);
```



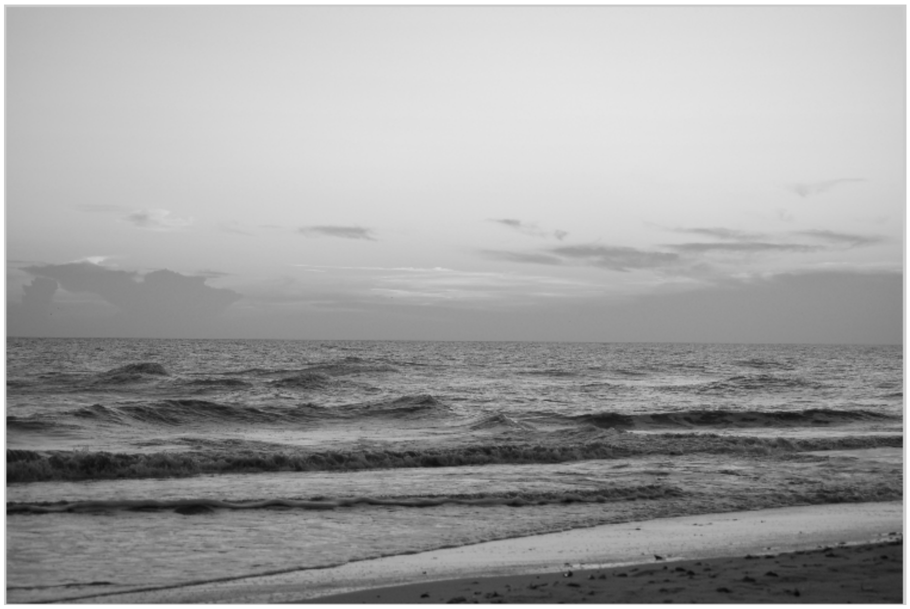
```
EDU>> newim = im(:,:,[3 3 3]);
```



Notice how the images are different – because they are based on a single color intensity, they will be darker or lighter in some spots depending on how intense the color is at that location. However, we can get a consistent grey image by taking the average of each color layer and applying that to all layers of the new image, as shown below:

```
EDU>> newim = uint8((double(im(:,:,1))+double(im(:,:,2))+double(im(:,:,3)))./3);
EDU>> newim(:,:,2) = newim(:,:,1);
EDU>> newim(:,:,3) = newim(:,:,1);
```

The new grayscale image



An additional operation on images is the creation of a negative – for this, all we do is subtract the old image from 255. This reverses the intensity of every color value, giving us the negative of the image.

Remember, you MUST convert the image data to the type 'double' before you perform any mathematical operations on it, and then you MUST convert it back to type 'uint8' once you finish!

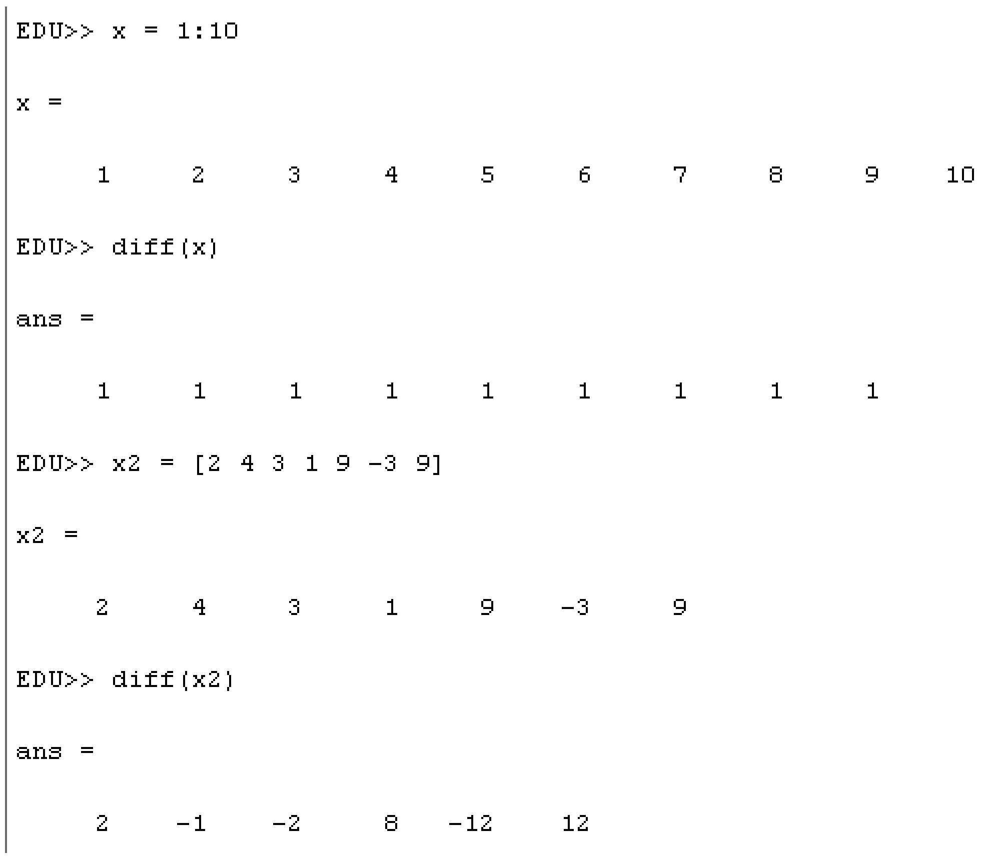
Numerical Methods

- Numerical methods is mostly just an exercise in function memorization, and can be divided four categories: Interpolation and curve fitting (which go together, so to speak) and differentiation and integration (which also go together)
- Interpolation
    - The problem that interpolation addresses is that of finding unknown values that lie between two data points – note that I said BETWEEN the data points and not OUTSIDE. Thus, interpolation SHOULD NOT BE USED TO EXTRAPOLATE, because that is not how it is designed to function
    - Linear interpolation
        - The function 'interp1' (note that is the number one and not the letter L) takes in the old x data points and the old y data points, and then it takes in the new x points that you want to find corresponding y points for and then finds those new y data points. The new x points MUST lie inside of the old x points (but besides that, there are no restrictions). For every new x value that you give interp1 that does not lie within your range of old x values, MATLAB will return NaN for the corresponding y
        - There is a fourth optional input that can modify the behavior of the function; it is a string, and some examples are as follows: 'linear' (which is the default, and tells the function to perform linear interpolation) and 'spline' (which tells the function to perform cubic-piecewise interpolation)
    - Cubic spline interpolation
        - The MATLAB function 'spline' takes in old x data points, old y data points, and then the new x data points that you want to find the corresponding y data points for and returns those new y data points. Unlike the interp1 function (or the interp1 function with 'spline' as the forth input) spline will allow you to extrapolate – however, it is not advised, because chances are the patterns between your data points do not hold outside that range
- Curve fitting
    - The difference between interpolation and curve fitting is, primarily, when you interpolate the curve passes through every point, and when you fit a curve it is possible that the curve could not pass through a single point – however, that curve best approximates some pattern that the data follows, and you can use it to extrapolate outside of your range of data points
    - Fitting a curve
        - There are two functions used to fit a curve: the first gives you the coefficients to your function (in effect giving you the function – and thus the curve – itself) and the second gives you your new y values for this function. The functions, in

order of usage, are: polyfit (which gives the coefficients) and polyval (which gives the new y-values). If you find yourself getting the names mixed up,just remember: poly*fit* fits a curve, and poly*val* gives you the new y-values

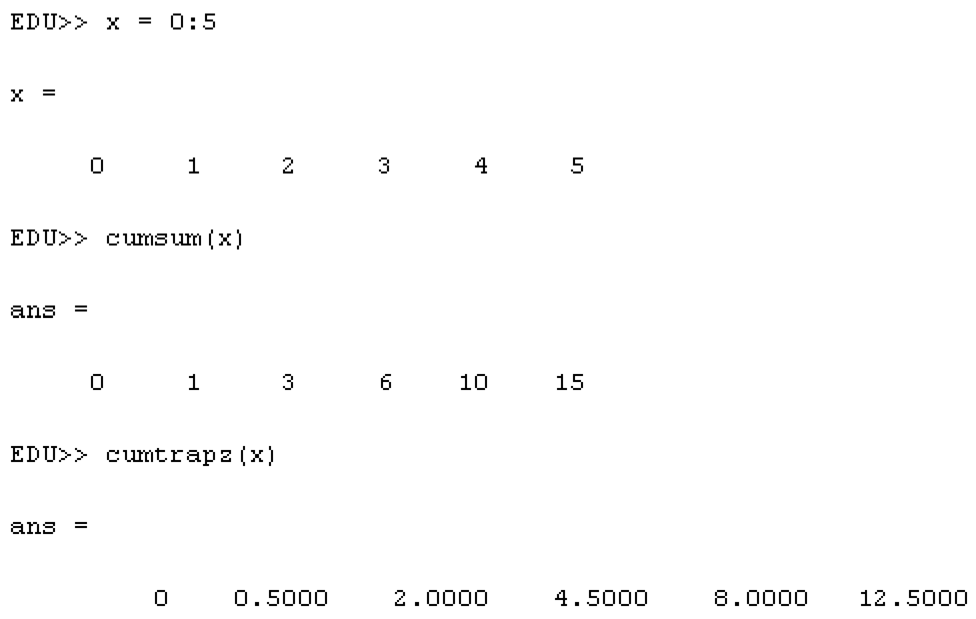
- Polyfit
    - Inputs: old x data, old y data, and some number that represents the largest possible exponent in your function
    - For the calculation to work, the number you input minus one must be LESS than the length of x. The function will still work, but its output will not tell us much
    - Output: vector of coefficients representing the function itself, in order of decreasing power. For example, [0 9 -3 2 1] represents the function $0x^4 + 9x^3 - 3x^2 + 2x + 1$
- Polyval
    - Inputs: the coefficient vector from polyfit and the new x values you want to find y values for
    - Outputs: the new y values

- Differentiation
    - The diff function in MATLAB provides an indirect method of finding the derivative of a function by providing us a vector of numbers that represents the differential – an approximation of the derivative
    - Here is an example of the diff function at work:



```
EDU>> x = 1:10

x =

     1     2     3     4     5     6     7     8     9    10

EDU>> diff(x)

ans =

     1     1     1     1     1     1     1     1     1

EDU>> x2 = [2 4 3 1 9 -3 9]

x2 =

     2     4     3     1     9    -3     9

EDU>> diff(x2)

ans =

     2    -1    -2     8   -12    12
```

    - Notice how the resulting vector is ONE SHORTER than the original. This is important when it comes to plotting derivatives, because when using the plot function both vectors have to be the same length
    - To obtain the derivative of a function with respect to y (or dx/dy) the following code is used: derivative = diff(x)./diff(y), where x is the set of x values and y is the set of y vales of the function. NOTICE THE "." IN FRONT OF THE DIVIDE SLASH
    - Now we can plot this derivate using the plot function thusly: plot(x(2:end), derivative). X must be reduced by one so that both vectors have the same length
- Integration
    - We have two functions to help give us approximations of integrals: cumsum(y) and cumtrapz(x,y); cumsum adds consecutively all of the y elements, while cumtrapz gives you an approximation of the integral of y with respect to x. Below is an example of both functions at work:



```
EDU>> x = 0:5

x =

     0     1     2     3     4     5

EDU>> cumsum(x)

ans =

     0     1     3     6    10    15

EDU>> cumtrapz(x)

ans =

         0    0.5000    2.0000    4.5000    8.0000   12.5000
```

    - Notice how cumsum gives us the simple cumulative sum of the vector, while cumtrapz does not – it uses the trapezoidial rule to approximate the integral (and with only one input, it uses unit x spacing)
    - To graph the integral, we would simply plot the old x values followed by the cumsum or cumtrapz of y times the difference between each x value (so basically, x followed by ∫f(x)dx)

Sorting

- Sorting is pretty straight forward: you just need to memorize the four algorithms, their Big-O's, and how they sort. Listed below are the four algorithms and aspects of each:
- Insertion Sort
    - Insertion sort works by taking an empty vector and 'inserting' the next number from the old vector into place in the new one
    - BigO: O(N^2)

    - Defining characteristic: The initialization of an empty vector, iteration
    - Example of Insertion sort at work:

Original vector: [21 8 1 12 -1]

New vector:
[21]
[8 21]
[1 8 21]
[1 8 12 21]
[-1 1 8 12 21]

- Bubble Sort
    - Bubble sort works by 'bubbling' a single number down the row until the vector is sorted: at the end of each step, the largest or next-largest number should be at the end of the sorted vector
    - BigO: O(N^2)
    - Defining characteristic: The swapping of two elements in a vector, usually by creation of a temporary variable that holds the number to be swapped
    - Example of bubble sort at work:

Original vector: [21 8 1 12 -1]

New vector:
[8 1 12 -1 21] (the 21 moves all the way to the end)
[1 8 -1 12 21] (the 8 moves until it encounters a larger number – 12 – and then the 12 moves until it too encounters a larger number – the 21)
[1 -1 8 12 21]
[-1 1 8 12 21]

- Merge Sort
    - Merge sort works by breaking the vector down into halves until all that is left is a bunch of vectors of length one – since they only have one element, they are already sorted (duh). The vectors are then 'merged' back together by comparing the first element of each and placing the lowest into the merged vector, until there are not elements left in either
    - BigO: O(Nlog(N))
    - Defining characteristic: RECURSION – look for the breaking up of the vector into halves
    - Example:

Original vector: [21 8 1 12 -1]

New Vector:
[21 8 1] [12 -1] (break into halves)
[21 8] [1] [12] [-1]
[21] [8] [1] [12] [-1]
[8 21] [1] [12] [-1] (put it back together the same way you took it apart)
[1 8 21] [-1 12]

```
[-1     1      8      12     21]    (the sorted vector)
```

- Quick Sort
  - Quick sort works through the selection of a pivot point – for our purpose, this will be the first point. The pivot point is then placed in the middle and everything less than the pivot is placed on the left and everything greater to the right (in the order that it is found in the vector). This process repeats until there are no pivots left – or until the vector is broken down into a bunch of pivots, that by their nature, are already sorted. The vector is then put back together.
  - BigO: O(Nlog(N)); however, for selection of a bad pivot it can be O(N^2)
  - Defining characteristic: RECURSION – and the selection of a pivot
  - Example: (see next page)

```
Original Vector: [21    8      1      12     -1]

New Vector:
[8      1      12     -1]    [21]   (21 is selected as the pivot)
[1      -1]    [8]    [12]     |    (8 is selected as the pivot)
[-1]    [1]     |              |    (1 is selected as the pivot)
[-1     1]      |              |    (the sorted vector is put back together)
[-1     1      8      12]      |
[-1     1      8      12     21]
```

Sound

- MATLAB gives several built in functions to read in sound data: wavread, which takes in a .wav filename, and auread, which takes in a .au filename. The both output a column vector of amplitudes ranging from -1 to 1 and a sampling frequency, which is just the number of sound samples played per second (in Hz)
- To play a sound, the sound function is used, which takes in the sound data and the playback frequency
- Sound attributes
  - Two basic sound attributes are: the time between samples (which is just the inverse of the sampling frequency, or 1/Fs) and the maximum time that the sound plays, which is the number of samples (or the length of the sound vector) times time between samples
- Adjusting frequency
  - There are two ways to adjust the frequency of a sound: you can either simply input a new frequency using the sound function (not so good) or you can index the original sound using a new frequency vector (a better option)
  - To understand the second option, think back how we made images longer or shorter depending on how we indexed them using the linspace function – we'll be doing the same thing here, only with sound. When we make the sound shorter, basically "compress" the sound waves, making them smaller and causing them to play in a shorter time period (and thus increasing the frequency). When we make the sound

longer, we make the sound waves longer and cause them to play over a longer time period, decreasing the frequency (the frequency is inversely related to the wavelength, remember)

- An example of doubling the frequency of a sound:

```
[note Fs] = wavread('soundfile.wav')
N = length(note)
Newnote = note(round(linspace(1,N,Fs*2)))
```

- To adjust a note further, we can think of adjusting the frequency in terms of music – doubling the frequency would be raising the note by an octave. There are twelve 'half steps' in an octave, so we could write each frequency multiplier in terms of 2 raised to some fraction of 12. The spectrum of half steps is shown below:

$2^{0/12}*Fs$ $2^{1/12}*Fs$ $2^{2/12}*Fs$ $2^{3/12}*Fs$ … $2^{12/12}*Fs$

- The first is obviously the base frequency – 0 out of 12 half steps. The second is the multiplier used to raise the sound by one half step (1 out of 12 half steps), the third is the multiplier used to raise the sound by two half steps (2 out of 12 half steps), and the last is simply used to raise the sound 12 half steps, or one full octave. To lower the sound by any number of half steps, just raise it to the negative power
- Manipulating the sound vector
  - Since sound is just a vector, we can reverse it, concatenate it, etc just like we could any other vector. These operations are relatively straightforward
- The FFt
  - The fast Fourier transform, or fft, is used to analyze the frequency domain of the sound – MATLAB's built in fft function takes in a sound and outputs the real and imaginary parts of the spectrum. Normally we are only concerned with the real part of the spectrum, so taking the absolute value of the fft of a sound will give us all the relevant frequency data we need
  - Once we have the frequency data, we can use the max function to find the value and location of the maximum frequency of the sound. We can then use the index location obtained from max to find the actual value of the principle frequency. Simply create a vector of frequencies ranging from 0 to the sampling frequencies and in steps of fs/N (where fs is the sampling frequency and N is the number of samples, so our steps are just in units of 1/seconds). Next use the index value obtained from the max function to index this vector, giving you the principle frequency (the frequency with the most energy, as found using the max function on our fft function output)

<u>Graph Searching and Queues</u>

- You can think of a queue as a line – the first person in the line is the first person so receive service (or whatever is at the end of the line), and people enter the line at the back. This is different from a stack, where the first thing to enter the stack is the last thing to leave.
- A graph can be thought of as a bunch of nodes connected by different paths, or lines – the paths can be either weighted (have a value or "cost" associated with them) or not. We use the act of

"in-queuing" and "de-queuing" paths to find the most efficient way to traverse graphs, or, in other words, to get from one point on a graph to another. There are two algorithms you will need to know to traverse graphs: the Breadth First Search and Dijkstra's Algorithm

- Breadth First Search
    - Breadth First Search looks for strictly the shortest path from A to B
    - The steps in the algorithm:
        - Start by in-queuing and then de-queuing the starting node
        - In-queue all paths from the next node in your de-queued list to every next possible node that has not already been visited (for example, if the next node in your de-queued list has a final node of J, and J connects to K,L,and M, then you would in-queue the paths J-K, J-L, and J-M)
        - De-queue the next path in your in-queue list; if the end-node on that path is already in your de-queued list, do not place that path in the list (ignore it since you've already got another path that gets you to that node)
        - Repeat until you de-queue a path that whose final node is the one you are trying to get to
- Dijkstra's Algorithm
    - Instead of looking for the shortest path from A to B, Dijkstra's Algorithm picks the path that costs the least
    - The steps in the algorithm:
        - Initialize by in-queuing and then de-queuing the starting node (with a cost of 0)
        - In-queue all paths from the next node in your de-queued list to every next possible node that has not already been visited (be sure to list their costs)
        - De-queue the next minimum path in your in-queue list. If the end node in that path has already been reached by some other path already in your de-queued list, then ignore the path (i.e., cross it out from your in-queued list but don't put it in your de-queued list)
        - Repeat until you de-queue a path whose final node is the one you're trying to get too – this is guaranteed to be the shortest path, assuming you did the above steps correctly