MATH 3670 - SCIENTIFIC COMPUTATION I
Fall 2015

Week 1: Starting with MATLAB. Script Files. Arrays and Operations with Arrays

---

Content

- Overview of Course Objectives - Use of MATLAB windows; the Command Window
- Arithmetic operations in MATLAB
- MATLAB as a calculator
- Numeric formatting
- Simple MATLAB built-in functions
- Script files - their creation and execution
- Creating one dimensional arrays (Vectors) and two dimensional arrays (Matrices)
- Handling arrays (addressing, adding and deleting entries, reshaping)
- Operations with arrays (addition, multiplication, array division, element-by-element operations)
- Built-in functions to operate and analyse arrays

• **Suggested Class Work**:
Chapter 1: 7b, 10a, 15, 39; Sample Problem 1-4 page 26
Chapter 2, # 16, Chapter 3, # 11, 28

• **Homework # 1**:
Chapter 1: # 26, 30, **33*, 36***, 40
Chapter 2: # 29, Chapter 3: # 29, 30 (Due Fri. Sept 4, 5pm)

(**Starred*** assignment is mandatory for MATH 3670 students)

---

1. Course Objectives:

I) Introduce what MATLAB is and some of it capabilities:
- As a glorified calculator
- As a computer programming language
- As a software solution environment

II) Introduce some elementary programming concepts:
- Variables (assignments, scope, life spans, etc.)
- Parameters (passing between functions, as return values, etc.)

III) Introduce some elementary programming constructs:
- Variable types and their creation
- Loops
- Conditional statements
- Iterative techniques
- Strategies for deciding how to tackle programming problems

IV) Learn about more advanced computational as needed in the fields of
- numerical linear algebra
- dynamical systems
- calculus in the complex plane
- elements of Fourier analysis, the DFT and FFT method,
- Probabilistic methods (e.g. Monte Carlo Simulations)

---

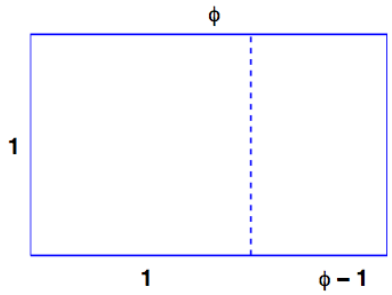
2. Using MATLAB to solve mathematical problems

NOTE: All lines that start with

```
>>
```

are intended to be entered as is by the student in their MATLAB's command window.
Also note: MATLAB output is depicted as this color of green.

We will discuss the usefulness of MATLAB through an example. The golden ratio shows up in many places in mathematics. The golden ratio gets its name from the golden rectangle, shown in the figure below



The golden rectangle has the property that removing a square leaves a smaller rectangle with the same shape (i.e. the ratio of the remaining rectangle's length and width is the same as the original rectangle's). Equating the aspect ratios of the rectangles gives a defining equation for ϕ:

$$\frac{1}{\phi} = \frac{\phi - 1}{1}$$

Multiplying both sides by ϕ produces the polynomial equation

$$\phi^2 - \phi - 1 = 0,$$

which has two real solutions,

$$\phi = (1 \pm \sqrt{5})/2.$$

The positive root is the golden ratio.

```
>> phi = (1+sqrt(5))/2


phi =
    1.6180
```

NOTE: The above line performed the indicated computation and stored the results in a location that is called **phi** due to

```
>> phi =....
```

If, instead the following was entered:

```
>> (1+sqrt(5))/2
```

Then the output would have been:

```
ans =
    1.6180
```

The difference is that this last assignment did not have any explicit variable name assigned, so MATLAB used it's own default variable name **ans**.

```
>> format long
>> phi


phi =
   1.618033988749895
```

This didn't recompute ϕ , it just displayed 15 significant digits instead of 5.

```
>> format bank
>> phi


phi =
          1.62


>> format short
>> phi


phi =
    1.6180
```

Forgot the quadratic formula? MATLAB can calculate the roots of a polynomial, which is represented as a vector of its coefficients, in descending order. The 'vector'

```
>> p=[1 -1 -1]
```

represents the polynomial

$$p(x) = x^2 - x - 1.$$

The roots are computed using the 'roots' function

```
>> r=roots(p)


r =
   -0.61803398874989
    1.61803398874989
```

Note that MATLAB's **roots** function returned a 2 element vector which was assigned a name of **r**, as expected since a quadratic equation will, in general have two solutions.

MATLAB also offers symbolic capabilities of solving algebraic equations. In this case the 'solve' command requires the equation typed in as a string and provides the two desired solutions.

```
>> r = solve('1/x = x-1')


r=
     [ 1/2*5^(1/2)+1/2]
     [ 1/2-1/2*5^(1/2)]
```

The variable r is a vector with two components, the symbolic forms of the two solutions. You can pick off the first component with

```
>> phi = r(1)


phi =
       1/2*5^(1/2)+1/2
```

This expression can be converted to a numerical value in two ways. Most common is the double-precision floating point, which is the principal way that MATLAB represents numbers, with the 'double' function.

```
>> phi = double(phi)


phi =
      1.61803398874989
```

It can also can be evaluated to any number of digits using variable-precision arithmetic with the 'vpa' function, such as using 50 digits

```
>> vpa(phi,50)


phi =
      1.6180339887498948482045868343656381177203091798058
```

Finally, this is yet another way to solve equations such as the golden ratio, using iterations. First, rewrite the equation satisfied by ϕ as

$$\phi^2 = 1 + \phi \qquad \text{or} \quad \phi = \sqrt{1+\phi}.$$

We will seek ϕ as a solution of the equation

$$x = \sqrt{1+x}$$

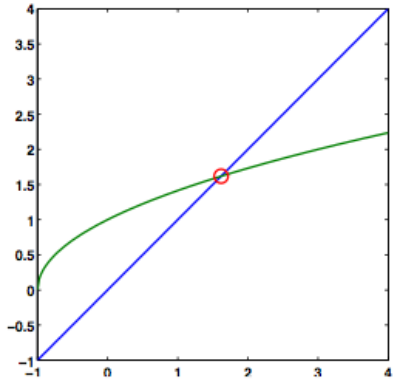
Start with some initial 'guess'.

```
>> x=3;
```

Next, enter this statement

```
>> x=sqrt(1+x);
```

and then repeat this using the up-down arrows on your machine, which recall earlier commands. What do you notice?
After a few iterations the value settles at $x = 1.6180$, or if you use the 'format long' command, to $x = 1.618033988749895$, which is a good approximation for the golden ration ϕ.
This is an illustration of the 'fixed point method' for the function $f(x) = \sqrt{1+x}$, which is depicted in the figure below.



This is the first example of MATLAB graphics. It shows the intersection of the graphs $y = x$ and $y = \sqrt{1+x}$. This figure can be obtained using the following commands:

```
>> x = -1:.01:4;
>> y1 = x; y2 = sqrt(1+x); plot(x,y1,'-',x,y2,'-',phi,phi,'o')
```

The best way to execute several commands at once is to create a so-called SCRIPT file which contains all the commands and which can be saved in a directory.
Let's create a file using the MATLAB editor, to execute the iteration discussed above:

```
x=3
for k = 1:30
   x = sqrt(1 + x)
end
```

Once this text file has been creates using the MATLAB editor, we should save it on the local computer (C: drive) or on the remote Z: drive in case you are on campus and want to access it later.

Each script file can generate a report, via the Publish feature in MATLAB.

One more instance where the golden ration appears unexpectedly: The famous Fibonacci sequence reads

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

and is obtained by iterating the recursion relation

$$F_{n+1} = F_n + F_{n-1}, \quad F_1 = 1,\ F_2 = 1.$$

It turns out that the ratio between consecutive terms of the sequence

$$\frac{1}{1} = 1, \quad \frac{2}{1} = 2, \quad \frac{3}{2} = 1.5, \quad \frac{5}{3} = 1.6666..., \quad \frac{8}{5} = 1.6, \quad \frac{13}{8} = 1.625, \quad \frac{21}{13} = 1.6154$$

does seem to approach the golden ration. This is not a coincidence. Can you tell why?

3. SUMMARY: Things to remember up to this point

• **Display format:**
The 'format' command only modifies the display, such as number of digits , and not the actual value of the variable. See pages 12–13 for the formatting options.

• **Elementary Math built-in functions:**
There are numerous functions that are already built-in MATLAB. Help is a way to learn about them, or the shortcut f_x next to the command window prompt.

• **Defining and operating with variables**
So far we have see how to define scalar variables and assign values to them, in numeric format. Operating with these variables is done via functions, either built in or user defined. Functions can be represented graphically in many different ways, more on this later.

• **Script files**
This is the efficient way to store and execute several commands at once, and they also allow to create reports (using the Publish feature of MATLAB), such as those needed for you to submit HMW. Script files are created by saving a MATLAB editor session and have .m as their file name extension.

IMPORTANT: Executing a script file means that for each line of code within the script file, MATLAB reads then executes it before moving on to the next line. If a line produces an error (evidenced as red error text in the command line window), then MATLAB aborts and the rest of the script file is NOT executed. PLEASE REMEMBER this when you work your homework. If you have good working code following a line that causes an error, this good working code will never get executed. So, make sure that your script file executes without generating any errors (warning messages are OK).

MATLAB started as a matrix calculator. At the core of MATLAB resides the ability to operate with 'arrays', which can be one dimensional arrays (rows or columns), two dimensional arrays (the usual matrices) or multi-dimensional arrays. Matrix (or more generally, array) creation and manipulation is so crucial to many problems that this subject needs to be covered before more interesting problems can be solved. Below are some commands that illustrate this. Chapters 2 and 3 in your textbook demonstrate more of these capabilities.

A note about MATLAB's help. Since there is no **array** command in MATLAB, the software's help for this concept is pretty sparse. However, doing a Google search on **"matlab array"** returns many useful results. Often, Google's result points back to an appropriate place in MATLAB's help.

INTERMISSION

4. Vectors as one-dimensional arrays

- It is very easy to define row-vectors: $\mathrm{x} = [\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_\mathrm{N}]$ or column-vectors: $\mathrm{y} = [\mathrm{y}_1, \mathrm{y}_2, \ldots, \mathrm{y}_\mathrm{N}]^\mathrm{T}$

```
>> x = [ 0, 0.5, 1, 1.5, 2], y = [ 0; 0.5; 1; 1.5; 2]


x =      0    0.5000    1.0000    1.5000    2.0000

y =      0
    0.5000
    1.0000
    1.5000
    2.0000
```

  NOTE that when the elements are separated by a semicolon, a new row is created as was done with the column vector **y**.

- One can access individual elements: $x(j), y(k)$
  NOTE: Even though we have a symbol **x** which is immediately followed by a left parenthesis, this is NOT interpreted as a function call as would be implied by an earlier statement above. MATLAB first sees if the symbol has already been defined by the user and if it has been, the memory location associated with that symbol is analyzed; if this location begins with the ASCII characters **function**, then this syntax is interpreted as a function call, otherwise, this syntax implies an index into an array or matrix so that whatever bit pattern is associated with this address is the value that is being assigned.

```
>>  x(2), y(5)


ans =    0.5000
ans =   2


>> x(6)


???  Index exceeds matrix dimensions.


>> y(0)
```

```
??? Index into matrix is negative or zero.
```

- One can insert or delete entries in a vector, altering its size (length)

```
>> x(8) = 4;
>> x
% increase the dimension of x to 8 and assign 4 to x(8)


x =
         0    0.5000    1.0000    1.5000    2.0000         0         0    4.0000


>>  y = y(2:4);
>> y
% reduce the dimension of y to 3 and assign y(2),y(3),y(4) to a new vector


y =
    0.5000
    1.0000
    1.5000
```

- A very useful operation is the transpose: x',y'

```
>> z1 = y'
>> z2 = y''


z1 =
    0.5000    1.0000    1.5000

z2 =
    0.5000
    1.0000
    1.5000
```

- One can compute the length of a vector (the number of elements in the vector) or its size (the dimension of the vector, viewed as a matrix, taking into account its horizontal or vertical structure)

```
>> length(z1)
>> length(z2)
>> size(z1)
>> size(z2)
```

```
ans =
     3

ans =
     3

ans =
     1     3

ans =
     3                    1
```

- One can initialize equally-spaced vectors in three different ways:

```
>> x = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1]; %\\
>> x = 0:0.1:1; % colon vectorization: first element, step size, last element
>> x=linspace(0,1,11); % first, last element and the number of  points
```

All the commands above generate the same array. Below are a few more examples:

```
>> x1 = 0:0.1:0.25 % the last element is not reached

>> x2 = 0:5 % default step size = 1

>> x3 = 5:-1:0 % the step size can be negative

>> x4 = 0:-1:1 % non-valid operation produces empty vector

>> length(x4) % the empty vector x4 has a zero length


x1 =
         0    0.1000    0.2000

x2 =
     0     1     2     3     4     5

x3 =
     5     4     3     2     1     0

x4 =
   Empty matrix: 1-by-0

ans =
     0
```

```
>> x1  = linspace(0,0.25,5) % the built-in function never mismatches the last elemen

>> x4  = linspace(0,-1,10) % the vectors are always non-empty

>> h1 = x1(2) - x1(1) % the step size can be easily computed (need not to be given)

>> h4 = x4(2) - x4(1)


x1 =
         0    0.0625    0.1250    0.1875    0.2500
x4 =
         0   -0.1111   -0.2222   -0.3333   -0.4444   -0.5556   -0.6667   -0.7778   -
h1 =
    0.0625

h4 =
   -0.1111
```

- One can use implicit indexation:

```
>> k = 1:11;
>> x(k) = 0.1*k;

>> y(1:3:12) = 1;
>> y(2:3:12) = 2;
>> y(3:3:12) = 3;
>> y


y =
     1     2     3     1     2     3     1     2     3     1     2     3
```

- Finally, one can use element-by-element definition of a vector (a for-loop):

```
>> for k = 1:11
>>     x(k) = (k-1)*0.1;
>> end
```

5. Operations with Vectors

- additions and subtractions of equal-sized vectors

```
>> x = [ 0, 1, 2 ];
>> y = [ 0.1, 0, 0.1];
>> z = [ 10, 20 ];
>> x + y
>> x - y


ans =
    0.1000    1.0000    2.1000
ans =
   -0.1000    1.0000    1.9000


x + z


??? Error using ==> +
Matrix dimensions must agree.
```

- additions and multiplications by a scalar

```
>> 0.2 + x
>> 0.2 * x


ans =
    0.2000    1.2000    2.2000
ans =
         0    0.2000    0.4000
```

- element-by-element multiplication and divisions of equal-sized vectors:

  NOTE: This will be a very important part of some of our later work. It is important that students know when element-by-element multiplication (or division or exponentiation) should be used instead of normal matrix multiplication (or division or exponentiation).

  Matrix multiplication and division (using the mathematical definition of such) is performed by using the familiar operators

  ```
  * / \
  ```

However, one can use matrices as place holders for many values so that large numbers of values can be operated on without construction of code containing loops. If one's usage of a matrix is such that the individual elements of one matrix need to be multiplied only to corresponding elements of another matrix, then element by element multiplication (or division) should be employed by prefixing each operator with a dot. This usage of vector math (micro code at the chip level) is far more efficient in computer resources than if one used array indexing to perform the same actions on each individual element.:

```
.* ./ .\
```

Back slash vs. forward slash /:
REMEMBER: matrix multiplication and division are NOT commutative in general!
If matrices **A** and **B** are defined appropriately (consistent dimensions) then:

```
X = A\B is equivalent to inv(A)*B to solve the equation A*X = B
X = A/B is equivalent to A*inv(B) to solve the equation X*A = B
```

Since the mathematical definition of matrix addition and subtraction is element by element, there is no need for a special version of these operations.
NOTE:

```
X = A\B
```

is preferable to inv(A)*B because the latter actually inverts matrix A, which often causes unacceptable round off errors vs. the former uses a type of matrix decomposition prior to multiplication that eliminates this problem.

```
>> x.*y % main advantage of MATLAB (matrix laboratory)
>> x./y % no loops for accessing individual elements are required
>> x.\y % inverse division, the same as y./x


ans =
         0         0    0.2000

Warning: Divide by zero.
ans =
     0   Inf    20

Warning: Divide by zero.
ans =
       Inf         0    0.0500
```

```
x.*z
```

```
??? Error using ==> .*
Matrix dimensions must agree.
```

- element-by-element powers

```
>> x.^3, y.^(0.1)
>> x.^y % the same as x(k)^(y(k)) for any k
>> 3.^x % the same as 3^(x(k)), the output is a vector of the same structure as x
```

```
ans =
     0     1     8
ans =
    0.7943         0    0.7943
ans =
         0    1.0000    1.0718
ans =
     1     3     9
```

- appending vectors

```
>> w = [x, y, z]
>> w = [x'; y'; z']
```

```
w =
         0    1.0000    2.0000    0.1000         0    0.1000   10.0000   20.0000
w =
         0
    1.0000
    2.0000
    0.1000
         0
    0.1000
   10.0000
   20.0000
```

- deleting elements of vectors

```
>> x(2) = []; x
>> w(3:6) = []; w
```

```
x =
     0     2
w =
     0
     1
    10
    20
```

- Strings as vectors of characters

```
>> str1 = 'Jack10';
>> str1(1:4) % the first four characters are displayed
>> length(str1) % the length of character vector "str1"
>> str2 = 'Smithers20';
>> str3 = [str1, str2] % concatenation of two string vectors
>> str4 = [str1(1:4),' ',str2(1:8)]


ans =
Jack

ans =
     6

str3 =
10Smithers20

str4 =
Jack Smithers
```

6. Matrices as 2-dimensional arrays

In linear algebra, a matrix is a rectangular array of elements (numbers). If a matrix A $= [a_{i,j}]$ has m rows and n columns, it has m-by-n elements $a_{i,j}$ where the first index i runs for rows of A and the second index j runs for columns of A. MATLAB matrices are direct and convenient numerical realization of matrices in linear algebra.

```
>> A = [ 0.1 0.2 0.3 0.4 ; -0.1,-0.2,-0.3,-0.4 ; 0 1 0 1 ]
```

```
A =
   0.1000    0.2000    0.3000    0.4000
   -0.1000   -0.2000   -0.3000   -0.4000
     0    1.0000         0    1.0000
```

Notice that rows in MATLAB matrices are separated by a semicolon: ";". The elements of each row can be separated by spaces or commas.

```
>>  A1 = [ 0.1 0.2 0.3 0.4 ; 0 1 0 1 0] ;


??? Error using ==> vertcat
All rows in the bracketed expression must have the same
number of columns.
```

Note that the number of elements in each row must be identical, otherwise one gets an error.

- addressing elements in an array

  The colon sign ":" represents all elements of a row or a column of a MATLAB matrix.

```
>> A(:,1)
>> A(:,4) % columns of A are column-vectors


ans =
        0.1000
       -0.1000
             0

ans =
       0.4000
     -0.4000
    1.0000


>> A(1,:)
>> A(3,:) % rows of A are row-vectors


ans =
        0.1000    0.2000    0.3000    0.4000

ans =
          0     1     0     1
```

```
>> A(1,2)
>> A(3,3) % individual elements of A are scalar variables


 ans =
        0.2000

 ans =
         0
```

7. Operations with Matrices

- The operations introduced for vectors (1 dim arrays), such as element-by-element additions, subtractions, multiplications, divisions, and powers of matrices of the same size are applicable to multi dimensional matrices.
  REMEMBER: the usage of some of MATLAB's built in functions listed below: any time a symbol is followed by a left parenthesis (such as in **size(A)**), the symbol is interpreted as a function call as opposed to being a specification of a variable.

```
>> A = [ 1 2 3 4 ; -1,-2,-3,-4 ];
>> B = [ 4 3 2 1 ; -4,-3,-2,-1 ];
>> C1 = A + B
>> C2 = A.*B
>> C3 = A./B
>> C4 = A.^B


C1 =
     5     5     5     5
    -5    -5    -5    -5

C2 =
     4     6     6     4
     4     6     6     4

C3 =
    0.2500    0.6667    1.5000    4.0000
    0.2500    0.6667    1.5000    4.0000

C4 =
    1.0000    8.0000    9.0000    4.0000
    1.0000   -0.1250    0.1111   -0.2500
```

- size: outputs numbers of rows and columns

```
>> [nRow,nCol] = size(A)


nRow =     2
nCol =     4
```

- transpose: A', outputs a transposed matrix A^T

```
>> B = A', size(B)


B =  1    -1
     2    -2
     3    -3
     4    -4

ans =
     4     2
```

- elementary matrices:

  zeros: outputs a null matrix with all elements being zeros

```
>> zeros(2,6) % a null 2-by-6 matrix
>> zeros(3) % a null 3-by-3 matrix


ans =    0     0     0     0     0     0
         0     0     0     0     0     0

ans =    0     0     0
         0     0     0
         0     0     0
```

- ones: outputs an matrix with all elements being ones

```
>> ones(2,6) % a 2-by-6 matrix of ones
>> ones(3) % a 3-by-3 matrix of ones
```

```
ans =
     1     1     1     1     1     1
     1     1     1     1     1     1

ans =
     1     1     1
     1     1     1
     1     1     1
```

- eye: outputs an identity matrix (a diagonal matrix)

```
>> eye(2,6), eye(3)


ans =
     1     0     0     0     0     0
     0     1     0     0     0     0

ans =
     1     0     0
     0     1     0
     0     0     1
```

- rand, randn: outputs a random matrix with all elements being random numbers

```
>> rand(2,6) % random numbers are uniformly distributed in the interval [0,1]
>> randn(3) % random numbers are normally distributed with mean zero and variance one


ans =
    0.9501    0.6068    0.8913    0.4565    0.8214    0.6154
    0.2311    0.4860    0.7621    0.0185    0.4447    0.7919

ans =
   -0.4326    0.2877    1.1892
   -1.6656   -1.1465   -0.0376
    0.1253    1.1909    0.3273
```

- matrix-vector multiplications

```
>> A = [ 1 2 3 1 ; 0 1 4 2 ; 3 0 2 3 ]
>> x = [ 1; 2; 1; 1; ]
```

```
A =  1     2     3    1
     0     1     4     2
     3     0     2     3

x =   1
      2
      1
      1


>> y1 = A*x  % MATLAB matrix multiplication operator

>> % a row-oriented loop:
>> [n,m] = size(A);
>> for k = 1:n
>>      y2(k) = A(k,:)*x;
>> end
>> y2

% a column-oriented loop
>> y3 = zeros(n,1);
>> for j = 1 : m
>>    y3 = y3 + A(:,j)*x(j);
>> end
>> y3


y1 =  9
      8
      8
y2 =    9     8     8
y3 =   9
       8
       8
```

- matrix-matrix mutiplications:

  If A is a n-by-m matrix, B is a m-by-q matrix, then the matrix multiplication makes sense and $C = A * B$ is a n-by-q matrix with elements: $c_{i,j} = \sum_k a_{i,k} b_{k,j}$. The other product $B * A$ is defined if $n = q$. For a square matrices of the same size, both products are defined but they are not equal in a general case: $A * B \neq B * A$.

```
>> A = [ -2 1 0 ; 1,-2,1 ; 0,1,-2];
>> B = [1 0 1 ; 0 1 0; 1 0 1];

>> C1 = A*B
>> C2 = B*A
```

```
C1 =
    -2     1    -2
     2    -2     2
    -2     1    -2

C2 =
    -2     2    -2
     1    -2     1
    -2     2    -2
```

- array division (the back slash operator)

```
>> A=[4 -2 6; 2 8 2; 6 10 3];

>> b=[8; 4; 0];

>> X=A\b
```

```
X =
  -1.8049
   0.2927
   2.6341
```

yields the same operation as

```
>> X=inv(A)*b
```

```
X =
  -1.8049
   0.2927
   2.6341
```

8. HOMEWORK FORMAT:

Organize all home work assignments as follows:
Put all problems into one .m file similar to below by making the first line identify the class, week number and YOUR NAME. Then, make a new section (demarcated by

%%

) for each new problem

```
%% Your Name, Math 265 WEEK 2 HOMEWORK #2 problems
%% Chapt. 2 #29

< all code for chapt. 2 #29>

%% Chapt. 3 #29

< all code for chapt. 3 #29>

%% Chapt. 3 #30

< all code for chapt. 3 #30>
```

COMMENTS: All lines beginning with a percent sign are a comment. All lines beginning with a double percent (2 percent signs in a row) are treated as a section by the "publish" command.
Before submitting your work, execute the .m file by pressing the little green arrow at the top of your MATLAB session. The output of this execution is what I use to determine if the assignment was done correctly. If you get any error message (red text in your command window), this means that your .m file did not execute completely. It executed up to the point where an error message was generated and then aborted. You must fix the error then re-run. Continue this process until a you get a clean run. After a clean run THEN...

Publish your work by clicking on file then publish. This will cause your .m file to execute again and create an HTML file consisting of all your output. It is THIS OUTPUT that you are to submit. Any output that has errors (such as the above mentioned red text) will not be given full credit.