Introduction to MATLAB

Benjamin Hemingway

Bank of Lithuania and Vilnius University

20 November 2018

What is MATLAB?

What is MATLAB?

- MATLAB is a programming language which is used to solve numerical problems such as
  - matrix algebra e.g. OLS
  - simulations e.g. monte carlo
  - solving non-linear equations
  - maximisations/minimisations of functions

Why should economists learn MATLAB?

- Economic models do not always have an analytical solution, must be solved numerically
- Matlab is easy to learn and user friendly, a 'gateway' language
- Economists have designed specific add-ons e.g. Dynare, Uhlig's toolkit
- Extensive resources available online e.g. wikibooks,, Mathworks website

What about Octave?

- MATLAB is commercial software (not free)
- Octave is an open source alternative to MATLAB (free)
- For the most part MATLAB and Octave are the same BUT there are a few differences
- Key differences between MATLAB and Octave are listed here: `https://en.wikibooks.org/wiki/MATLAB_Programming/Differences_between_Octave_and_MATLAB`
- Dynare and Uhlig's toolkit work in both MATLAB and Octave
- The examples in this course will work in both MATLAB and Octave
- Download Octave from `https://www.gnu.org/software/octave`

What to expect from this course

What this course will do

- Introduce you to the basics of MATLAB
- Provide some example codes that aim to explain some of the basics
- Provide some tips on how to structure your code

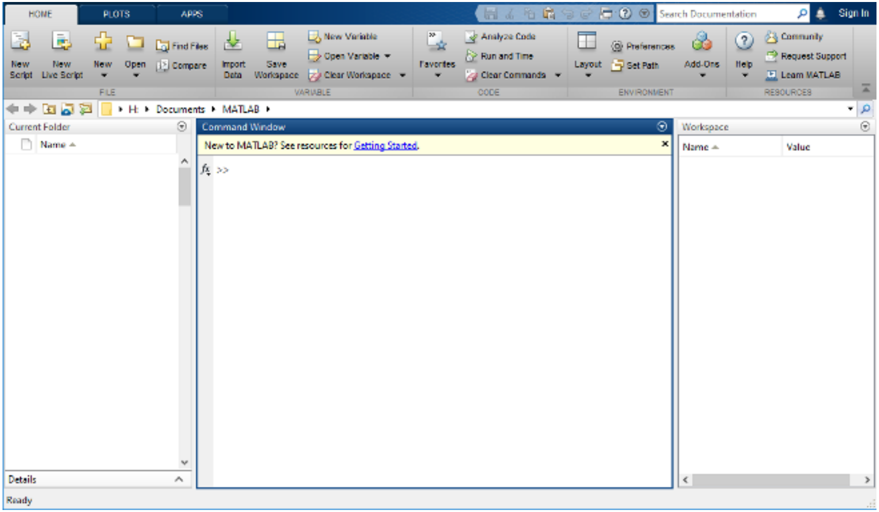
What this course will not do

- List or explain every function of MATLAB (type **help** <function name>)
- Replace a Macroeconomics course
  - No dynamic programming problems
  - No Dynare
- Replace the best way to learn programming: trying things out for yourself!

Online Resources

- The MATLAB wikibook
  `https://en.wikibooks.org/wiki/MATLAB_Programming`
- The MathWorks MATLAB documentation
  `https://www.mathworks.com/help/matlab/index.html`
- Octave documentation `https://octave.org/doc/interpreter/`
- MATLAB answers - a help forum
  `https://www.mathworks.com/matlabcentral/answers/index`

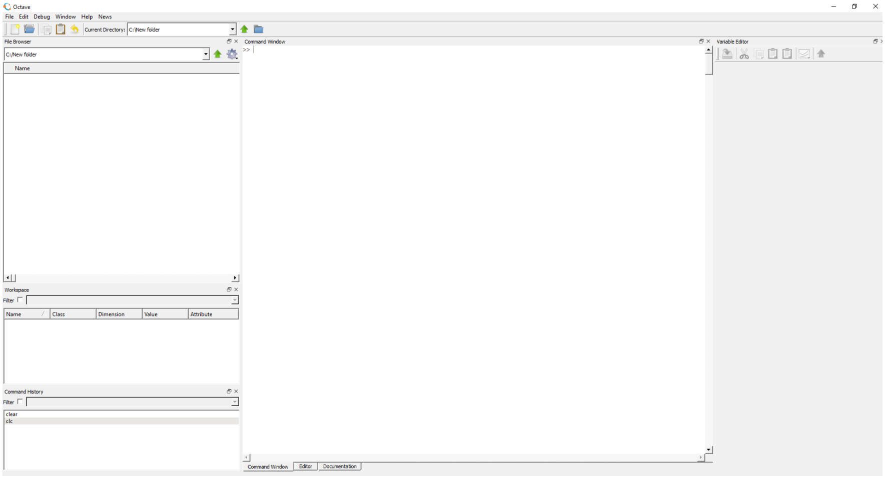
Matlab Workspace



New to MATLAB? See resources for Getting Started.

```
fx >>
```

Octave Workspace



```
>>
```

```
clear
clc
```

Creating Vectors

Row vector size (1,n)

```
A = [1  2  3]
```

Column vector: size (n, 1)

```
A = [1;  2;  3]
```

or

```
A = [1  2  3]'
```

Row vector from 1 to 5 with step size d

```
A = 1:d:5
```

Linearly spaced row vector from 1 and 5 of length n

```
A = linspace(1,5,n)
```

Creating Matrices (1)

Create a matrix

```
A = [1  2; 3  4]
```

2 x 2 matrix of zeros

```
A = zeros(2,2)
```

2 x 2 matrix of ones

```
A = ones(2,2)
```

Creating Matrices (2)

2 x 2 identity matrix

```
A = eye(2,2)
```

2 x 2 uniform random numbers

```
A = rand(2,2)
```

2 x 2 standard normal random numbers

```
A = randn(2,2)
```

Matrix multiplication and division

Matrix multiplication

```
A*B
A^2 % equivalent to A*A
```

Solve linear system Ax=b

```
x = inv(A)*b % when A is square
x = A\b % matrix left division (A can be rectangular)
```

Solve linear system xA=b

```
x = b*inv(A) % when A is square
x = b/A % matrix right division (A can be rectangular)
```

NOTE: It is recommended to use \ or / rather than **inv()** where possible due to better performance

Elementwise multiplication and division

- Sometimes you want to apply an operation to each element of a matrix. This is indicated using e.g. **.*** rather than *****

Elementwise operations

```
A.*B % Multiply each element of A by the corresponding B element
A./B % Divide each element of A by the corresponding B element
A.^2 % Square each element of A
```

```
ones(2,2).*ones(2,2)    % Element by element multiplication
```

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

```
ones(2,2)*ones(2,2) % Matrix multiplication
```

```
ans =
                                        2    2
                                        2    2
```

Some other useful functions

Exponential

```
exp(1.0)
```

Natural Logarithm

```
log(1.0)
```

Logarithm (base 10)

```
log10(1.0) % don't get this confused with log function!
```

Measuring Distance

```
abs(-1.0) % returns the absolute value of the number
norm([1,2,3]) % measures the 2-norm of a vector
```

Some useful values

π

```
a=pi % 3.141592654....
```

$\sqrt{-1}$

```
a=1i % sqrt(-1)
```

Inf

```
a=Inf % infinity
```

NaN

```
a=NaN % not a number e.g. 0/0=NaN
```

Matrix manipulation (1)

Consider the following matrix

$$A = \begin{bmatrix} 3 & 1 & 0 \\ 4 & 2 & 7 \\ 3 & 4 & 1 \end{bmatrix}$$

```
A(2,3)    % Access one element
```

```
ans =

     7
```

```
A(2:3,:) % Access specific rows
```

```
ans =

     4     2     7
     3     4     1
```

```
A(:,1:2) % Access specific columns
```

```
ans =

     3     1
     4     2
     3     4
```

Matrix manipulation (2)

Consider the following matrix

$$A = \begin{bmatrix} 3 & 1 & 0 \\ 4 & 2 & 7 \\ 3 & 4 & 1 \end{bmatrix}$$

```
A([1 3],:) % Remove a row
```

```
ans =
      3   1   0
      3   4   1
```

```
A(:,[1 3]) % Remove a column
```

```
ans =
      3   0
      4   7
      3   1
```

```
A' % Transpose
```

```
ans =
      3   4   3
      1   2   4
      0   7   1
```

Matrix manipulation (3)

Consider the following matrix

$$A = \begin{bmatrix} 3 & 1 & 0 \\ 4 & 2 & 7 \\ 3 & 4 & 1 \end{bmatrix}$$

```
A(:) % Convert to column vector
```

```
ans =
        3
        4
        3
        1
        2
        4
        0
        7
        1
```

```
A(5) % Access 5th element of matrix in column vector form
```

```
ans =
        2
```

Matrix manipulation (4)

You can also change part of the matrix

$$A = \begin{bmatrix} 3 & 1 & 0 \\ 4 & 2 & 7 \\ 3 & 4 & 1 \end{bmatrix}$$

```
A(2,2) = 0 % Change one element
```

```
ans =
        3   1   0
        4   0   7
        3   4   1
```

```
A(:,2) = [0;1;2] % Change a whole column
```

```
ans =
        3   0   0
        4   1   7
        3   2   1
```

Note that if you change an entire column you need to replace it with a column vector of the same size

Finding the minimum (1)

Consider now the following matrix

$$A = \begin{bmatrix} 3 & 1 & 0 \\ 4 & 2 & 7 \\ 2 & 4 & 1 \end{bmatrix}$$

```
min(A) % Returns the minimum along columns
```

```
ans =
                    2    1    0
```

```
min(A,[],2) % Returns the minimum along rows
```

```
ans =
                         0
                         2
                         1
```

```
min(A(:)) % Returns the smallest element of the matrix
```

```
ans =
                         0
```

Note that the **max()** function uses the same syntax and can be used to find the maximum

Finding the minimum (2)

Consider now the following matrix

$$A = \begin{bmatrix} 3 & 1 & 0 \\ 4 & 2 & 7 \\ 2 & 4 & 1 \end{bmatrix}$$

- The **min()** function is one of several functions that can return more than one thing
- It can also return the location of the minimum

```
[minA, Ind] = min(A) % Returns the minimum along columns
```

```
minA =
                2   1   0
Ind =
                3   1   1
```

```
[minA, Ind] = min(A(:)) % Returns the smallest element of the matrix
```

```
minA =
                    0
Ind=
                    7
```

Writing a script

- Working directly in the command window quickly becomes difficult
- A better way is to write all the commands in a file and run it
- MATLAB (and Octave) files have an extension .m
- To open the editor, simply type **edit** into the command window
- You can run an entire script by pressing the **run** button in the editor
- A shortcut for running a highlighted selection of a script is **F9**

A simple example script

```
%{Add a block of comments (description text that won't run as code) %}
% Adds one line of code
 clear % clear any variables in the workspace
 clc % clear any text in the command window
B = 10; % Add a semi-colon to stop output being printed
C = 100 ... % dots allow you to extend one command ...
+ 200; % across two lines
```

Some useful data types

- Most numbers are stored in floating point format either **double** (64bit) or **single** (32bit) precision
  - In almost all cases, we want to use **double** (the default type)
- We can also store numbers as integers. There are several different integers

Class	Range of Values	Conversion Function
Signed 8-bit integer	-2^7 to $2^7 - 1$	int8
Signed 16-bit integer	-2^{15} to $2^{15} - 1$	int16
Signed 32-bit integer	-2^{31} to $2^{31} - 1$	int32
Signed 64-bit integer	-2^{63} to $2^{63} - 1$	int64
Unsigned 8-bit integer	0 to $2^8 - 1$	uint8
Unsigned 16-bit integer	0 to $2^{16} - 1$	uint16
Unsigned 32-bit integer	0 to $2^{32} - 1$	uint32
Unsigned 64-bit integer	0 to $2^{64} - 1$	uint64

- **Logical** type can take two values 0 (false) 1 (true)
- Text is saved using character vectors **char**

```
c = 'Hello World'; % character vector
```

Cells and Structures

- Matrices and vectors can only hold one type of data
- Sometimes we want to combine different types of data into one object
- Cell arrays contain data in cells that you access by numeric indexing

```
C = {'one','two','three';1,2,3}
```

```
C = 2x3 cell array
    {'one'}    {'two'}    {'three'}
    {[  1]}    {[  2]}    {[    3]}
```

- Structure arrays contain data in fields that you access by name

```
patient(1).name = 'John Doe';
patient(1).billing = 127.00;
patient(1).test = [79, 75, 73; 180, 178, 177.5; 220, 210, 205];
patient(2).name = 'Ann Lane';
patient(2).billing = 28.50;
patient(2).test = [68, 70, 68; 118, 118, 119; 172, 170, 169];
patient
```

```
patient = 1x2 struct array with fields:

        name
        billing
        test
```

Creating Output

- It is good practice to place a semi-colon **;** at the end of each line of code
- However, we may still want to provide information both on screen and save output to files

```
disp("Hello World");
```

```
Hello World
```

```
A = [1 2; 3 4];
fprintf('%4.2f %4.1f\n',A); % Print output in two columns
```

```
1.00 3.0
2.00 4.0
```

IMPORTANT! Note the order a matrix is printed, columns first

```
fileSave = fopen('file.txt','w'); % Open new empty text file
fprintf(fileSave,'%s %d\n','char',int8(2)); % Save output
fclose(fileSave); % close the file you opened
```

Loading Input

- As well as saving calculations to files, we can also load data
- Both MATLAB and Octave have several functions for importing data
- One of the simplest to use is **importdata()**

```
filename = 'ols_data.csv'; % let matlab know which file to load
delimiterIn = ','; % tells matlab how data is separated
headerlinesIn = 1; % tells matlab how many rows are headers in the data
data = importdata(filename,delimiterIn,headerlinesIn); % loads data as a
     structure
```

- Missing values will be imported as **NaN**
- The data will be imported as a structure
  - data.data - contains loaded numerical data
  - data.textdata - contains loaded text data
  - data.colheaders - contains column headers

For importing excel data try **xlsread()**

Example 1: OLS

- We will now consider a simple problem that we will solve in MATLAB
- Consider the following OLS model

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \epsilon$$

with $\epsilon \sim N\left(0, \sigma^2\right)$

- With N observations we can rewrite the statistical model as

$$\underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}}_{N\times 1} = \underbrace{\begin{bmatrix} 1 & x_{11} & x_{21} \\ 1 & x_{12} & x_{22} \\ \vdots & \vdots & \vdots \\ 1 & x_{1N} & x_{2N} \end{bmatrix}}_{N\times 3} \underbrace{\begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix}}_{3\times 1} + \underbrace{\begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_N \end{bmatrix}}_{N\times 1}$$

- We will now set $N = 1000$ and $\sigma = 0.1$ and estimate the vector of coefficients
- To do this, we will first 'create' some (random) data then find the OLS estimator

$$\hat{\beta} = \left[\boldsymbol{X}'\boldsymbol{X}\right]^{-1} \boldsymbol{X}'\boldsymbol{y}$$

See **ex1_ols.m** on `https://benhemingway.github.io/teaching`

Loops

- We may want to perform operations multiple times
- Loops allow us to repeat commands simply

```
for a = 10:20
    fprintf('value of a: %d\n', a);  % this prints intergers 10—11
end
```

- Loops are useful for creating sequences or defining vectors/matrices

```
fib = ones(100,1); % good practice to create a vector first
for a = 3:100
        fib(a,1) = fib(a—2,1) + fib(a—1,1); % fibonacci series
end
```

- We can also define a 'nested loops'

```
mat = zeros(100,50); % good practice to create a vector first
for m = 1:100
        for n = 1:50
                mat(m,n) = m+n;
        end
end
```

Note: Best performance if matrices and vectors are 'pre-allocated' outside of a loop

Relational and Logical operators

- Relational operators perform comparisons of every element
- Return a **logical** array of the same size, with 1s (true) and 0s (false)

```
C=A<B;  % A less than B
C=A>B;  % A greater than B
C=A<=B; % A less or equal B
C=A>=B; % A greater or equal B
C=A==B; % A equal B
C=A~=B; % A not equal B
```

- The logical operators combines logical arrays

```
D=(A<B)&(C>B);  % and
D=and(A<B,C>B);  % identical to above
```

```
D=(A<B)|(C>B);  % (inclusive) or
D=or(A<B,C>B);  % identical to above
```

```
D=xor(A<B,C>B);  % exclusive or
```

```
D=~(A<B);  % not
D=not(A<B);  % identical to above
```

Conditional Operators (1)

- The **if** statement evaluates a logical expression and executes a command if true
- The optimal key words **else** and **elseif** allow execution of alternative commands
- Consider the following piecewise equation

$$y = \begin{cases} -1 & x \leqslant -1 \\ x & -1 < x \leqslant 1 \\ x^2 & \text{otherwise} \end{cases}$$

```
if x<=-1 % x less or equal to 1
        y = -1;
elseif x<=1 % x between -1 and 1
        y = x;
else % otherwise
        y = x.^2;
end
```

Conditional Operators (2)

- The **while** loop repeats a group of statements until a condition is met
- Using **while** conditions can be dangerous, if the condition is never met code will run forever!
- A loop can be manually stopped (**CTRL+C** on windows) or you can add a **break**

```
x = 100;
iter = 0;
while x>1e-6 % loop continues until x<=1e-6

        x = x/2; % halves the value of x
        iter += iter+1; % count iterations

        if iter>=100 % if loop takes too long
                break; % this will exit the while loop
        end

end
```

Floating-Point values and zero

- We need to be careful when dealing with non-integer numbers
- The following illustrates a common source of error

```
A = 0.3;
B = 0.1 + 0.1 + 0.1;
A==B
```

ans = logical 0

- Why? Computers cannot store numbers exactly

```
B-A
```

ans = 5.5511e-17

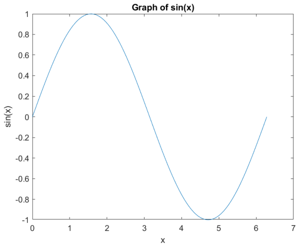
```
d = eps(A); %returns the (+ve) distance from abs(A) to the
    next larger floating-point number of the same
    precision as A
abs(A-B)<=eps(A)
```

ans = logical 1

Plots (1)

- MATLAB can be used to produce graphics. The basic command is **plot()**

```
x = linspace(0,2*pi,100)'; % column vector with 100 points
y = sin(x);
fig = figure; % create a figure
plot(x,y); % plots x and y
xlabel('x'); % labels x-axis
ylabel('sin(x)'); % labels y-axis
title('Graph of sin(x)'); % adds a title
print(fig,'SaveFile','-dpng') % saves graph as SaveFile.png
```

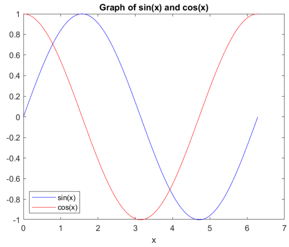


- A bar chart can be created using **bar()**, a 3D plot using **mesh()**

Plots (2)

- MATLAB can be used to produce graphics. The basic command is **plot()**

```
x = linspace(0,2*pi,100)'; % column vector with 100 points
y1 = sin(x);
y2 = cos(x);
fig = figure; % create a figure
plot(x,y1,'-b',x,y2,'-r'); % plots y1 as blue line, y2 as red line
xlabel('x'); % labels x-axis
title('Graph of sin(x) and cos(x)'); % adds a title
legend('sin(x)','cos(x)','location','southwest'); % add a legend
print(fig,'SaveFile2','-dpng') % saves graph as SaveFile2.png
```



Example 2: Monte Carlo Simulation

- A Monte Carlo simulation rely on repeated random sampling to obtain numerical results
- Consider rolling two dice and adding the total together
- If we repeat this enough times we can get a good estimate of the probabilities of each possible total
- To do this in MATLAB we first use **rand()** to generate a set of random variables distributed uniformly between 0 and 1
- Then we can use **loops** and **conditionals** to convert the uniform random variables into the outcomes of

See **ex2_mc.m** on `https://benhemingway.github.io/teaching`

Creating a function file

- MATLAB has a lot of functions, but we may sometimes want to create our own functions
- Functions should be created in a separate file
- To use the function you created, you should ensure MATLAB knows where it is located
- This can be done using the dropdown box or **cd** in the command window

```
cd C:\Path\to\my\files
```

An example function file

```
function u = myfun(c,a)
        % define a function for crra utility
        if a==1
                u = log(c);
        else
                u = (1./1-a)*c.^(1-a);
        end
end
```

Note: You must name the file the same name as the function i.e. **myfun.m**

Anonymous Functions

What is an anonymous function?

- We have learnt how to define a functions in a separate file
- An anonymous function is not stored in a program file, but is associated with a variable
- However, they can contain only a single executable statement

Defining an anonymous function

```
sqr = @(x) x.^2; % Returns the square of the input
```

Using an anyonymous function

```
sqr(5)
```

```
ans =

    25
```

Solving non-linear equations

- We need anonymous functions is to solve a non-linear equation
- Non-linear solvers aim to solve a problem of the type

$$f(x) - b = 0$$

where $f(x)$ is some non-linear function and we want to find x

- MATLAB (and Octave) have the following **local** solvers
  - **fsolve** - in MATLAB requires installation of the Optimization Toolbox
  - **fzero** - tends not to be as reliable as fsolve
- These solvers are local so we need to define a starting point x_0
- The solvers will then attempt to find a solution to the equation near x_0

Using a non-linear solver (fsolve)

```
foo = @(x) x.^2+sin(x)-3; % Define anonymous function
xzero = fsolve(foo,1.0); % solve function around 1.0
```

Minimizing non-linear equations

- We need anonymous functions is to minimize a non-linear equation
- In this case, we are interested in finding x that minimizers a function

$$\min_{x} f(x) - b$$

- MATLAB (and Octave) have **fminsearch** which is a **local** minimizer
- The solvers will then attempt to find a minimum near x_0
- BUT this minimum is not guaranteed to be global

Using fminsearch

```
foo = @(x) x.^2+sin(x)-3; % Define anonymous function
xmin = fminsearch(foo,1.0); % solve for the minimum around
    1.0
```

Tips for non-linear solvers/minimizers

- Read the documentation of your chosen function, there are different options you can set
- **fminsearch** may have found only a local minima not a global minima
- **fsolve** and **fzero** may not find a solution

Setting options

```
options = optimoptions('fsolve','Display','iter'); % Set
    options to return iterative display in fsolve.
```

Getting additional information

```
[xzero,foozero,exitflag] = fsolve(foo,1.0,options);
```

- The **exitflag** is especially useful as it informs you why the solver finished
- Local algorithms may be sensitive to your choice of starting point, experiment!

Example 3: Rosenbrock function

- Consider the following function of x and y

$$f(x, y) = (a - x)^2 + 100(y - x^2)^2$$

  This is the Rosenbrock function

- It has a global minimum at $(x, y) = (a, a^2)$, where $f(x, y) = 0$
- We will set $a = 1$ and find the global minimum
- First we will create a function file **rosenbrock.m**
- Then we will use a 2D grid over (x,y) and search for the minimum value on a grid
- We will use this minimum value as the starting point for **fminsearch()**

See **ex3_funmin.m** on `https://benhemingway.github.io/teaching`

Testing Speed

- It is important to be able to write code that executes quickly
- This is especially true of code that you need to run many times
- To test how long a section of code takes to evaluate we can use **tic** and **toc**

```
tic; % starts the timer
A = rand(1000,1000); % code to be evaulated
toc; % time since last toc
```

```
Elapsed time is 0.016099 seconds.
```

- MATLAB will print the time even following **;** unless it is saved as a variable

```
tic; % starts the timer
A = rand(1000,1000); % code to be evaulated
tt = toc; % no longer printed
fprintf('Time taken by A %f seconds\n',tt); % manual print
```

```
Time taken by A 0.016099 seconds.
```

Vectorizing Code

- MATLAB is optimized for operations involving matrices and vectors
- Thus your code will perform faster if you can replace loops with vector operations
- Optimizing code for MATLAB in this way is called *vectorization*

The following two snippets perform the same operation

```
X = rand(1000,1000);
Y = rand(1000,1000);
tic;
Outv = X.*Y; % Vectorized Code
toc
```

Elapsed time is 0.019561 seconds.

```
tic;
Outl = zeros(1000,1000); % preallocating improves performance
        for i=1:1000
                for j=1:1000
                        Outl(i,j) = X(i,j).*Y(i,j); % loop
        end
                end
toc;
```

Elapsed time is 0.003134 seconds.

Some Frustrating Mistakes to Avoid

- Overlaying commands with variables

```
plot=1:10; % creates a vector called 'plot'
plot(1:10) % the plot function no longer works!
```

```
ans = 1 2 3 4 5 6 7 8 9 10
```

- Loops update the counter variable

```
k = 42;
a = zeros(100,1); % preallocate grid
for k=1:100
        a(k,1) = k;  % update grid in loop
end
fprintf('Value of k=%d\n',k);
```

```
Value of k=100
```

Error Messages

Index exceeds matrix dimensions

- You have entered something like x(i), where x is a vector of length n, and i > n.

```
A = rand(3,3);
A(1,3) % no error
A(2,4) % error
A(5,1) % error
```

Too Many Input/Output Arguments or **Not enough Input/Output Arguments**

- You have called a function with the wrong number of arguments

Undefined Function or Variable

- The variable you are trying to use isn't defined. Often caused by a typo

Subscript Indices Must Be Real Positive Integers or Logicals

- When indexing a matrix, you cannot use floating point values or non positive integers

```
A = rand(3,3);
A(1,3) % no error
A(0,3) % error
A(1.5,3) % error
A(-1,3) % error
```

Thanks for listening!