A SHORT INTRODUCTION TO MATLAB

ANDY W. H. KHONG

Email: andykhong@ntu.edu.sg
Web: www.ntu.edu.sg/home/andykhong
Office: S2.2-B2-01
Tel: 6790 6008

1. Introduction

2. Vector and Matrices

- 3. Complex numbers

- 4. Plotting

- 5. Functions and scripts

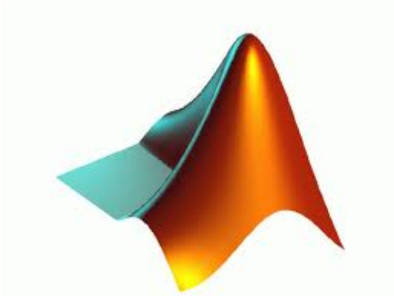
- 6. Sine/Cosine Plots
  - 6.1 Sampling- definition
  - 6.2 Discrete-time signal from continuous-time signal
  - 6.3 Plotting a discrete cosine wave

- 7. Applications
  - 7.1 Acoustic signal processing (spectrogram)
  - 7.2 Image signal processing

CHAPTER I

Introduction

- MatLab is an acronym for matrix laboratory and is owned by MathWorks.
- It was conceptualized in University of New Mexico and Stanford University before being commercialized via MathWorks in 1984.
- The latest version is MatLab R2012b.
- MatLab is a programming software to
  - aid visualization of mathematical functions
  - aid algorithmic development
  - compute complex functions
  - and many more



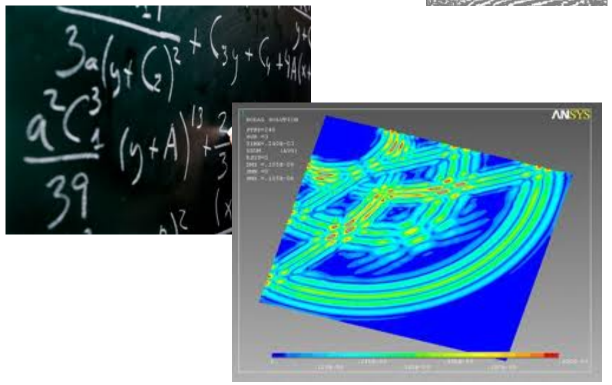
- A typical algorithm development cycle involves the following

Setting the goal (aim)

- Acoustic source localization
- Face detection
- Fingerprint authentication
- Footstep detection

Problem formulation

- Signal processing tools
- Mathematical formulation
- Wave propagation
- Contrast detection



Real-time implementation

- PC-based (C/C++)
- DSP implementation (TI, Analog Devices)
- Android based (Arduino)
- Field programmable gate array (FPGA)

Risky

- A typical algorithm development cycle involves the following

Setting the goal (aim)

↓

Problem formulation

↓

Simulate

↓

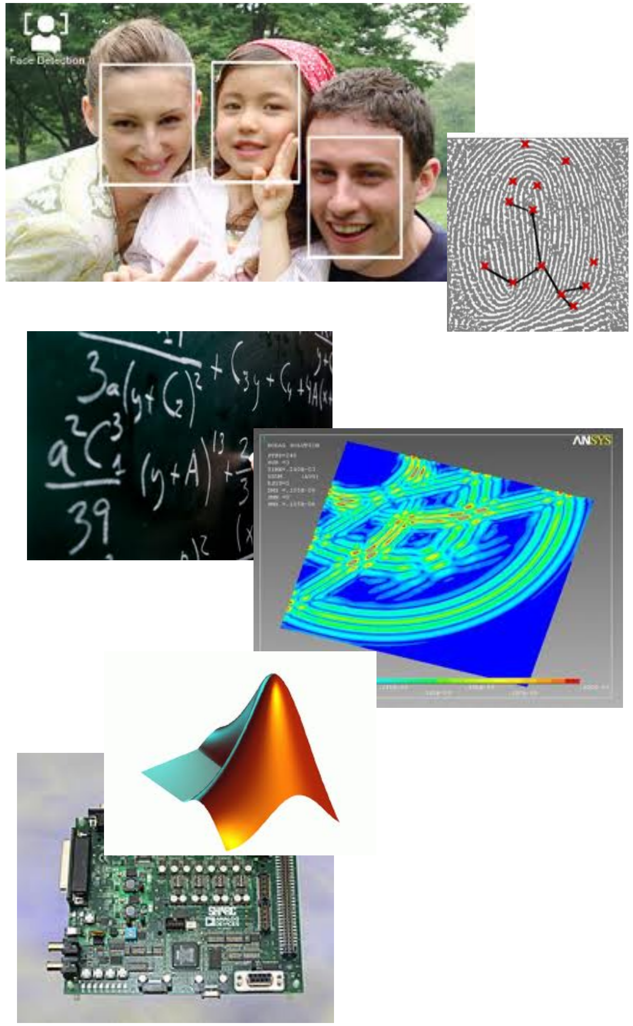
Real-time implementation

Risky

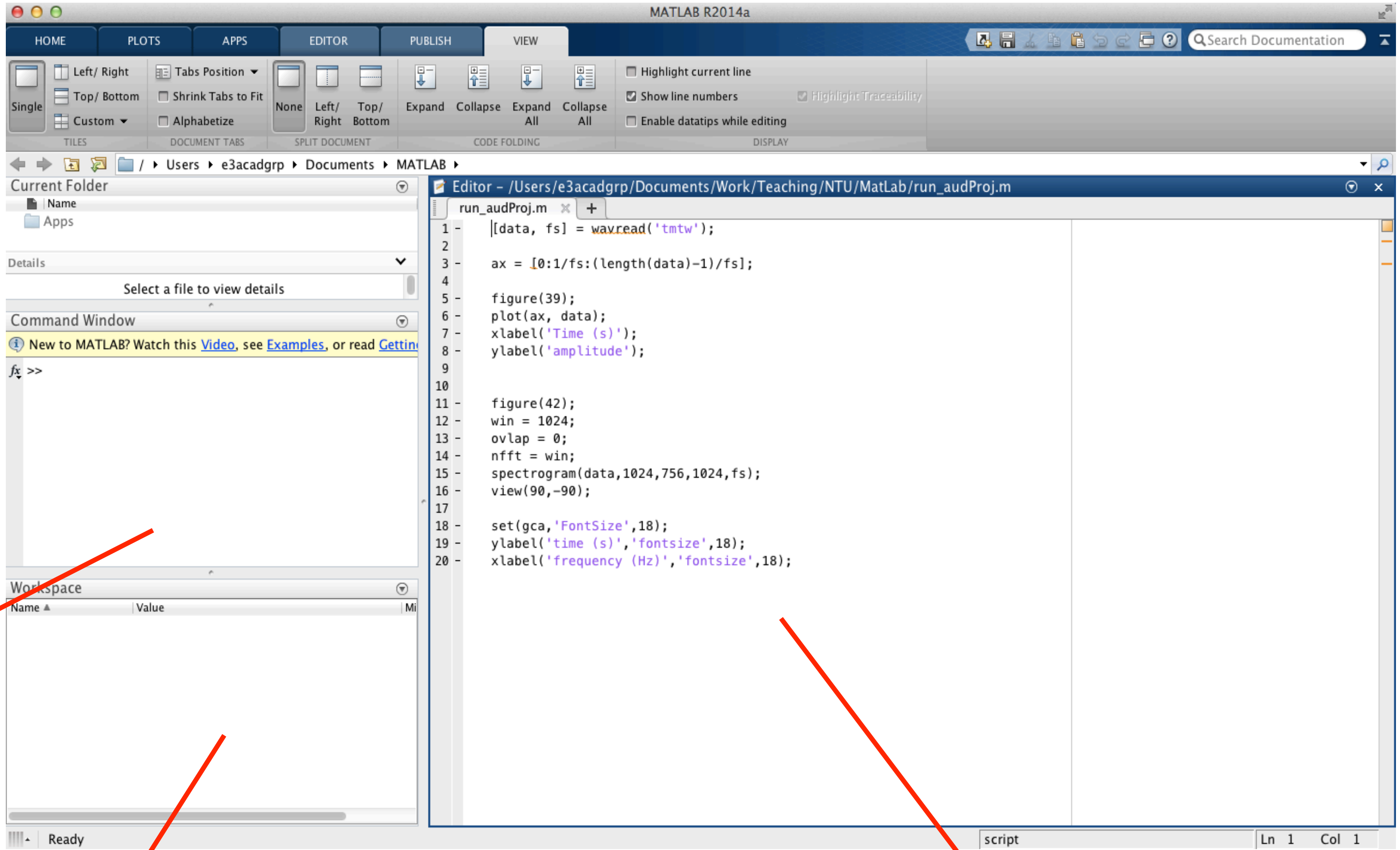
- Acoustic source localization
- Face detection
- Fingerprint authentication
- Footstep detection

- Signal processing tools
- Mathematical formulation
- Wave propagation
- Contrast detection

- MatLab
- PC-based (C/C++)
- DSP implementation (TI, Analog Devices)
- Android based (Arduino)
- Field programmable gate array (FPGA)



NANYANG TECHNOLOGICAL UNIVERSITY



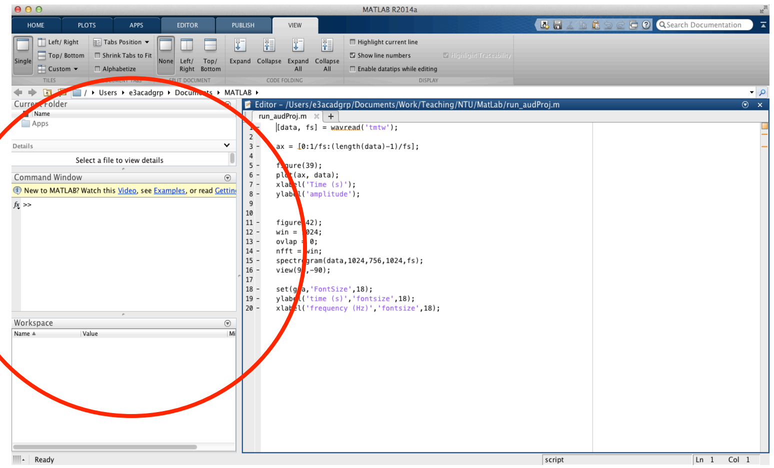
Command window

Workspace

Command window

- The command window is where a lot of researchers will do their programming.

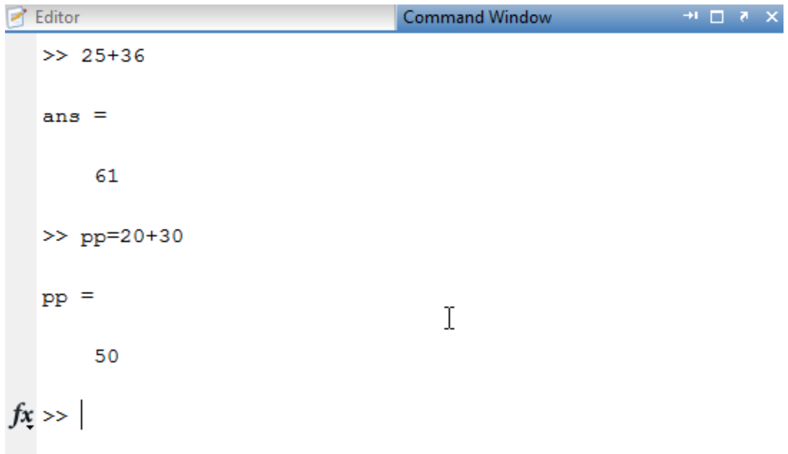
- It offers a fast and easy way to compute equations just like an ordinary calculator.



- Try the following: >> 25+36

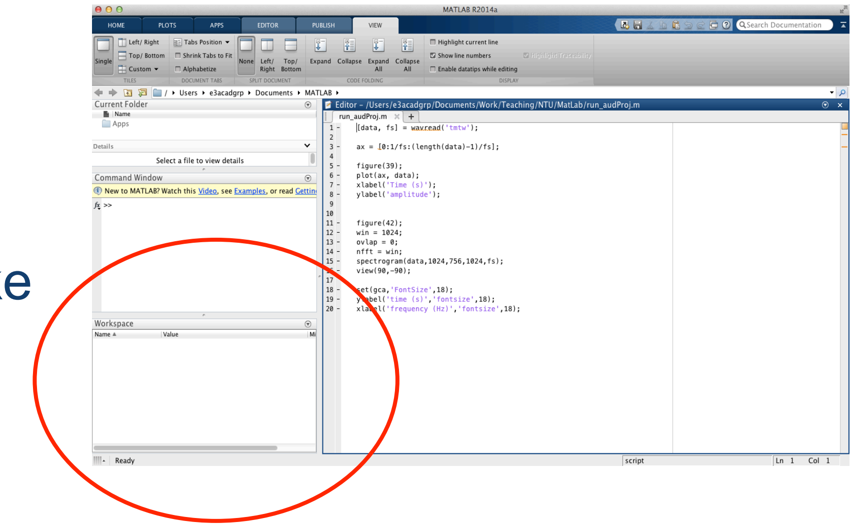
- And you will get: ans =
  61

- Try also the following:
  >> pp = 20+30

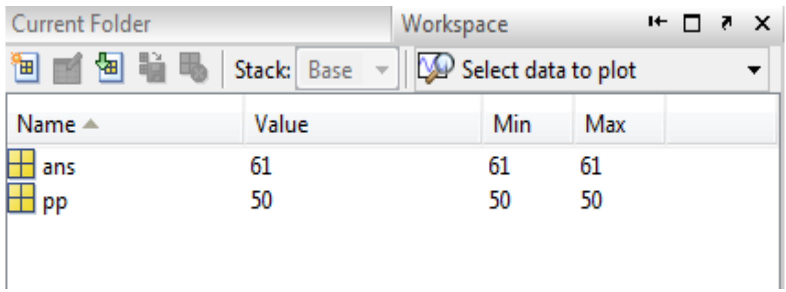


- Very often, in a research project, we define a lot of variables (> 30).

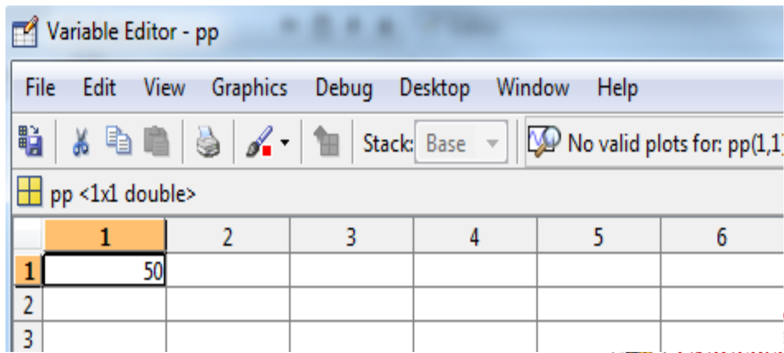
- The workspace is where we would like to keep track of variables and their values.



- It is a place where researchers know:
  - what has been defined
  - the maximum and minimum values



- Double-clicking the variables will allow you to see the variables in a form similar to excel spreadsheet.



1.5 Command history

- Very often, we enter a lot of commands and it is sometimes useful to keep track of them.

- To repeat some computations without re-typing them, simply hit the “up” arrow key in the command window

- Another way to repeat any previous commands is
  - to place the cursor in the command window
  - hit the “up” or “down” arrow keys to cycle through previous commands

- MatLab has documented a comprehensive set of help files.

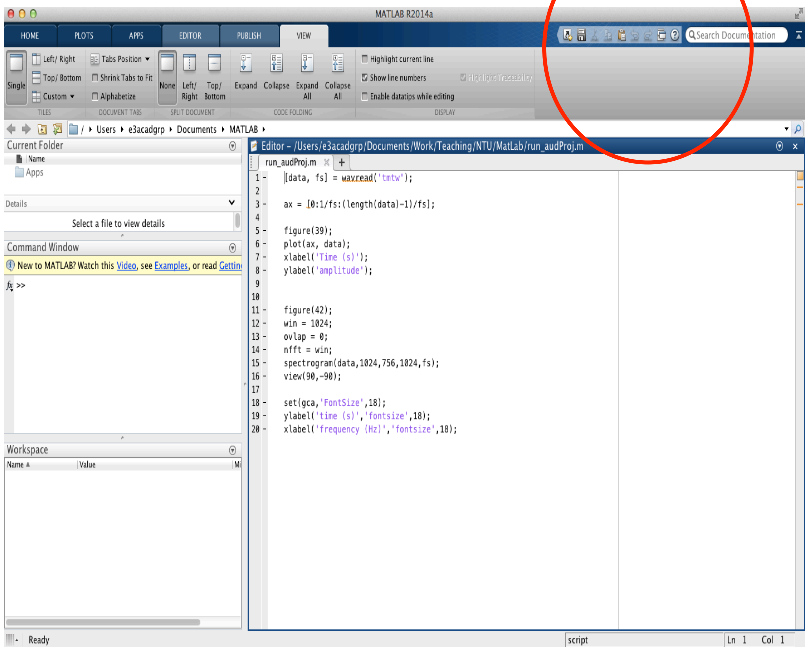
- These files can be accessed from the “Help” menu.

- Try accessing it via

  Help → Product Help

- You may search different functions by typing keywords into the search box.

- Try typing “mean” into the search box



Clearing memory and command window

- For large projects, one often have to declare lots of variables.

- Some variables may hold many numbers and if the program is not using them, it may be wise to free up the memory by deleting these variables.

- To delete a particular variable, say the variable "pp", use

  >> clear pp

  You will notice the variable "pp" disappearing from the workspace.

- To delete all variables, i.e., to clear all memory simply type

  >> clear

- To clear the command window, use

  >> clc

CHAPTER 2

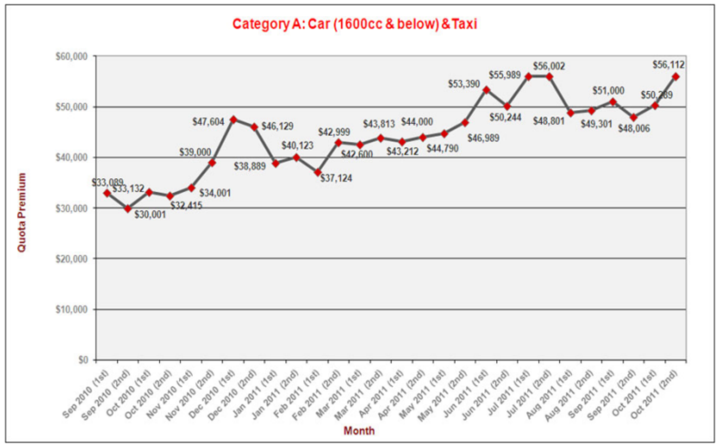
Vectors and Matrices

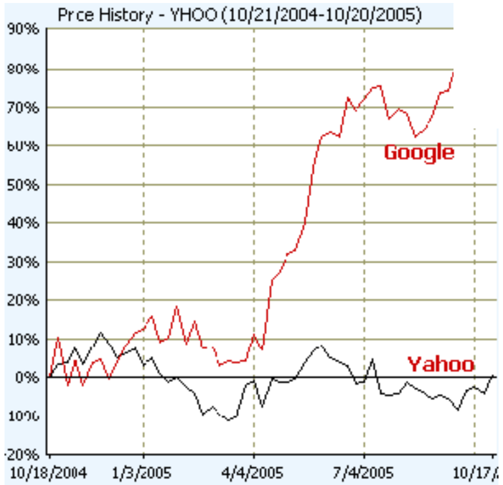
- Vectors are arrays that store a series of numbers

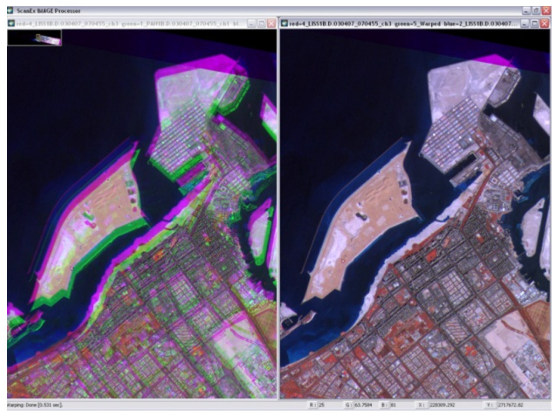
- Vectors can be classified into row and column vectors

$$\begin{bmatrix} 1 & 40 & 2 & 200 & 12 \end{bmatrix} \qquad \begin{bmatrix} 2.4 \\ 3.2 \\ 3 \end{bmatrix}$$

- Many real-world signals can be expressed in the form of vectors/ matrices







- To define a row vector use

  >> rowVecA = [1 4 2]

$$\mathrm{rowVecA} = \begin{bmatrix} 1 & 4 & 2 \end{bmatrix}$$

- The semi-colon “;” is used to concatenate numbers to the next row. Useful to form a column vector:

  >> colVecB = [2; 1; 3]

$$\mathrm{colVecB} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}$$

- The above can be extended to form a matrix.
  - First define the row
  - To define the next row, use the semi-colon
  - Remember to make sure each row has the same number of elements

  >> matC = [1 3 5; 3 2 6; 1 1 3]

$$\mathrm{matC} = \begin{bmatrix} 1 & 3 & 5 \\ 3 & 2 & 6 \\ 1 & 1 & 3 \end{bmatrix}$$

- Sometimes its clumsy to list down all elements manually if the numbers exhibits certain characteristics.

- We can use the colon “**:**” operator which is the same as counting from a number “to” another number (in steps of 1).

- Example: To generate a vector called “num” containing numbers 30 to 50, we use

```
>> num = [30:50]
```

$$\text{num} = \begin{bmatrix} 30 & 31 & \dots & 50 \end{bmatrix}$$

- We can use two colons if we want to count in steps other than 1.

- Example: To generate a vector of even numbers from 30 to 50

```
>> eveNum = [30:2:50]
```

$$\text{eveNum} = \begin{bmatrix} 30 & 32 & 34 & \dots & 50 \end{bmatrix}$$

- To reference an element within a vector/matrix, we use the elemental position of the variable.

- Therefore, to reference an element, we use the format

  >> variableName(rowIndex,columnIndex)

- To reference the 2nd element of the vector "rowVecA", defined in Section 2.1, we use

  >> rowVecA(2)

$$\text{rowVecA} = \begin{bmatrix} 1 & 4 & 2 \end{bmatrix}$$

- To reference the 2nd row, 3rd column of the matrix "matC", defined in Section 2.1, we use

  >> matC(2,3)

$$\text{matC} = \begin{bmatrix} 1 & 3 & 5 \\ 3 & 2 & 6 \\ 1 & 1 & 3 \end{bmatrix}$$

- We can also use the colon operator ":" to reference a range of elements.

- Therefore, to reference the 2nd to 3rd element of "rowVecA", we use

```
>> rowVecA(2:3)
```

$$\text{rowVecA} = \begin{bmatrix} 1 & 4 & 2 \end{bmatrix}$$

- To reference the last element of the vector, we can use the keyword "end"

```
>> rowVecA(end)
```

$$\text{rowVecA} = \begin{bmatrix} 1 & 4 & 2 \end{bmatrix}$$

- To determine the length of the vector, we can use the keyword "length"

```
>> lenVecA = length(rowVecA)
```

- To transpose a matrix, use the “prime” key, located on the immediate left of the “Enter” key.

- Transpose of the column vector “colVecB” will form a row vector:

  >> transpVecB = colVecB‘

$$\mathrm{colVecB} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}$$

$$\mathrm{transpVecB} = \begin{bmatrix} 2 & 1 & 3 \end{bmatrix}$$

$$\mathrm{matC} = \begin{bmatrix} 1 & 3 & 5 \\ 3 & 2 & 6 \\ 1 & 1 & 3 \end{bmatrix}$$

- To transpose a matrix, we use the same prime notation

  >> transpmatC = matC‘

$$\mathrm{transpmatC} = \begin{bmatrix} 1 & 3 & 1 \\ 3 & 2 & 1 \\ 5 & 6 & 3 \end{bmatrix}$$

- Unlike scalar multiplication, matrix/vector multiplication can only be performed when we take the dimension into account.

- In general,

$$\mathbf{A}_{M\times N}\times\mathbf{B}_{N\times P}=\mathbf{C}_{M\times P}$$

- Example

```
>> A = [1  3  4; 2  4  7]
>> B = [5; 6; 1]
>> C = A*B
```

$$\mathbf{A}_{2\times 3}=\begin{bmatrix}1 & 3 & 4\\ 2 & 4 & 7\end{bmatrix}\qquad \mathbf{B}_{3\times 1}=\begin{bmatrix}5\\ 6\\ 1\end{bmatrix}$$

$$\begin{aligned}\mathbf{C}_{2\times 1} &= \begin{bmatrix}1 & 3 & 4\\ 2 & 4 & 7\end{bmatrix}\begin{bmatrix}5\\ 6\\ 1\end{bmatrix}\\ &= \begin{bmatrix}(1\times 5)+(3\times 6)+(4\times 1)\\ (2\times 5)+(4\times 6)+(7\times 1)\end{bmatrix}\\ &= \begin{bmatrix}27\\ 41\end{bmatrix}\end{aligned}$$

- Would the following work?

```
>> D=B*A
```

- It is possible to perform element-by-element multiplication using the dot-multiplication notation, i.e., “ **.*** ”

- For element-by-element operation, make sure they are of the same dimensions.

$$\mathbf{A}_{2\times 3} = \begin{bmatrix} 1 & 3 & 4 \\ 2 & 4 & 7 \end{bmatrix} \qquad \mathbf{D}_{2\times 3} = \begin{bmatrix} 2 & 8 & 1 \\ 1 & 1 & 5 \end{bmatrix}$$

- Example

```
>> A = [1  3  4; 2  4  7]
>> D = [2  8  1; 1  1  5]
>> E = A.*D
```

$$\mathbf{E}_{2\times 3} = \begin{bmatrix} (1\times 2) & (3\times 8) & (4\times 1) \\ (2\times 1) & (4\times 1) & (7\times 5) \end{bmatrix} = \begin{bmatrix} 2 & 24 & 4 \\ 2 & 4 & 35 \end{bmatrix}$$

- Are the following valid?

```
>> E = A*D
>> F = A*D'
```

- MatLab offers an excellent tool for solving simultaneous equations.

- Consider the following example:

$$\begin{aligned} 3x + 4y - 2z &= 6 \\ 4x - 6y + 2z &= 1 \\ 2x + y + 0.2z &= 2 \end{aligned}$$

- To find the unknown variables x, y and z, we re-write in matrix form

$$\begin{bmatrix} 3 & 4 & -2 \\ 4 & -6 & 2 \\ 2 & 1 & 0.2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 6 \\ 1 \\ 2 \end{bmatrix}$$

$$\mathbf{Aq} = \mathbf{p}$$

- To find the unknown, i.e., elements in the vector $\mathbf{q}$, we only need to use the following

$$\mathbf{q} = \mathbf{A}^{-1}\mathbf{p}$$

- So how do we solve it in MatLab?

$$\begin{bmatrix} 3 & 4 & -2 \\ 4 & -6 & 2 \\ 2 & 1 & 0.2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 6 \\ 1 \\ 2 \end{bmatrix}$$

$$\mathbf{A}\mathbf{q} = \mathbf{p}$$

$$\mathbf{q} = \mathbf{A}^{-1}\mathbf{p}$$

- Define all variables. Compute the unknown by calculating the inverse of a matrix using "inv( )"

- Consider the following example:

$$\begin{aligned} 3x + 4y &= 6 \\ 6x + 8y &= 12 \end{aligned}$$

- To find the unknown variables x, y and $z,$ we re-write in matrix form

$$\begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 6 \\ 12 \end{bmatrix}$$

$$\mathbf{A}\mathbf{q} = \mathbf{p}$$

- To find the unknown, i.e., elements in the vector $\mathbf{q},$ we only need to use the following

$$\mathbf{q} = \mathbf{A}^{-1}\mathbf{p}$$

- In MatLab

- The above generates the result

$$\mathbf{q} = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \text{inf} \\ \text{inf} \end{bmatrix}$$

with the warning message

Warning: Matrix is singular to working precision.

- The above implies that there are no solutions for $x,$ and y

- This can be verified by the high condition number of the matrix $\mathbf{A}$

```
>> cond(A)
```

- A high conditional number of $\mathbf{A}$ implies that it is non-invertible.

- An invertible $\mathbf{A}$ has a low condition number of 1.

CHAPTER 3

Complex Numbers

- In addition to real numbers, MatLab also supports complex numbers.

- This is achieve via the variables “i” and “j” (if they haven’t been defined). These variables have already been pre-defined as complex numbers in MatLab

```
>> i
>> j
```

$$i = 0 + 1i$$

$$j = 0 + 1j$$

- We can define complex numbers using

```
>> val = 3+2j
```

$$\text{val} = 3 + 2j$$

- An array of complex numbers can then be defined using

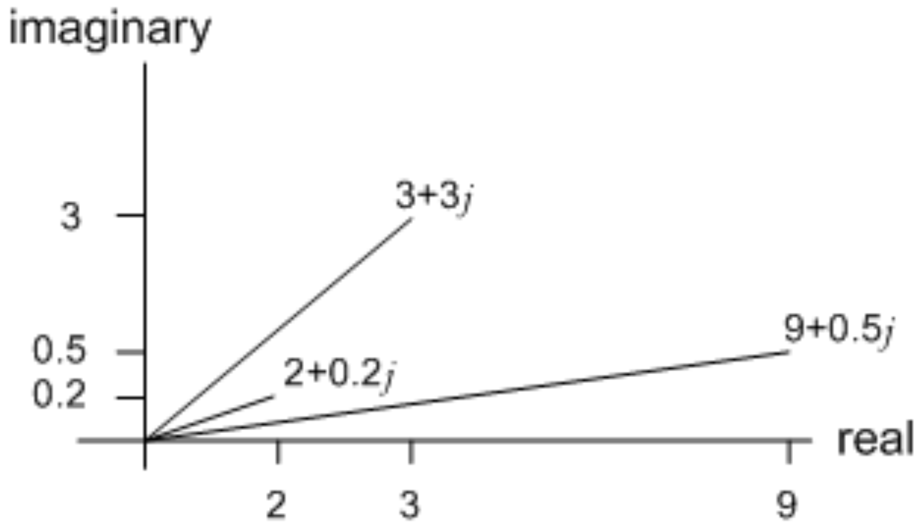
```
>> cplAry = [2; 3; 9]+ j*[0.2; 3; 0.5]
```

or

```
>>  cplAry = [2+0.2j; 3+3j; 9+0.5j]
```

$$\text{cplAry} = \begin{bmatrix} 2 \\ 3 \\ 9 \end{bmatrix} + j \begin{bmatrix} 0.2 \\ 3 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 2 + 0.2j \\ 3 + 3j \\ 9 + 0.5j \end{bmatrix}$$

- Any complex numbers can be characterized by its magnitude and phase



$$\text{cplAry} = \begin{bmatrix} 2+0.2j \\ 3+3j \\ 9+0.5j \end{bmatrix}$$

- To compute the magnitude use "abs( )"

```
>> absAry = abs(cplAry)
```

$$\text{cplAry} = \begin{bmatrix} 2+0.2j \\ 3+3j \\ 9+0.5j \end{bmatrix}$$

$$\text{absAry} = \begin{bmatrix} \sqrt{2^2+0.2^2} \\ \sqrt{3^2+3^2} \\ \sqrt{9^2+0.5^2} \end{bmatrix} = \begin{bmatrix} 2.01 \\ 4.24 \\ 9.01 \end{bmatrix}$$

- The phase can be computed via "phase( )"

```
>> phAry = phase(cplAry)
```

$$\text{phAry} = \begin{bmatrix} \tan^{-1}(0.2/2) \\ \tan^{-1}(3/3) \\ \tan^{-1}(0.5/9) \end{bmatrix} = \begin{bmatrix} 0.0997 \\ 0.7854 \\ 0.0555 \end{bmatrix}$$

- Note that since, by default, MatLab computes angles in radians, angles in degrees can be computed easily using

```
>> phAry = phase(cplAry).*180/pi
```

$$\begin{aligned} \text{phAry} &= \begin{bmatrix} \tan^{-1}(0.2/2) \\ \tan^{-1}(3/3) \\ \tan^{-1}(0.5/9) \end{bmatrix} \times 180/\pi \\ &= \begin{bmatrix} 5.71 \\ 45.00 \\ 3.18 \end{bmatrix} \end{aligned}$$

- To extract the real parts within an array, simply use “real( )”

  >> rlAry = real(cplAry)

$$\text{cplAry} = \begin{bmatrix} 2+0.2j \\ 3+3j \\ 9+0.5j \end{bmatrix}$$

$$\text{rlAry} = \begin{bmatrix} 2 \\ 3 \\ 9 \end{bmatrix}$$

- To extract the real part of a particular element, you may apply Section 2.2. Therefore, to extract the real part of the 3rd element in the variable “cplAry”, use

  >> real(cplAry(3))

- To extract the imaginary parts within an array, use “imag( )”

  >> imAry = imag(cplAry)

$$\text{imAry} = \begin{bmatrix} 0.2 \\ 3 \\ 0.5 \end{bmatrix}$$

CHAPTER 4

Plotting

- Compared to C/C++, MatLab offers an excellent and simple way to visualize functions.

- Plotting is done via the “plot( )” function and specifying the abscissa and ordinate values.

- The plot function has two arguments (inputs)
  - a vector containing abscissa values
  - a vector containing ordinate values
  - plot (abscissaValues, ordinateValues)

- Note that the number of elements in both vectors must be the same.

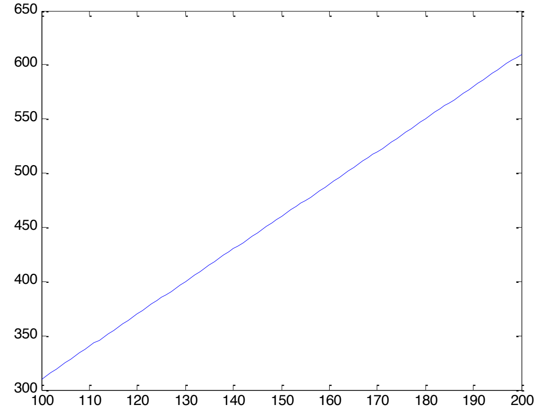
- Example: To plot the function of $y = 3x + 10$ within the range of $x = 100...200$
  - Create a new figure

    >> figure(20)
  - Specify a vector containing the abscissa (counting from 100 to 200)

    >> x = [100:200];
  - Compute the ordinate values

    >> y = 3*x+10;
  - Plot the function

    >> plot(x,y)



- The number "20" after the "figure" keyword is to identify which figure MatLab should plot the data. If there is no figure 20, it will create a new one.

- The semi-colon ";" at the end of the 2^{nd} and 3^{rd} command is there to prevent MatLab from listing down the elements of the vector.

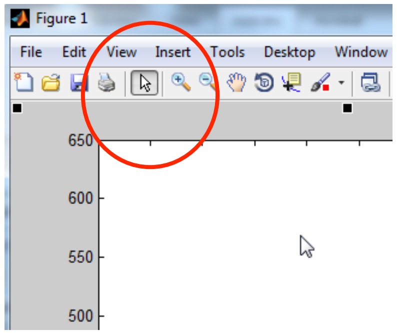
- It is very useful particularly if you do not want to display long vectors.

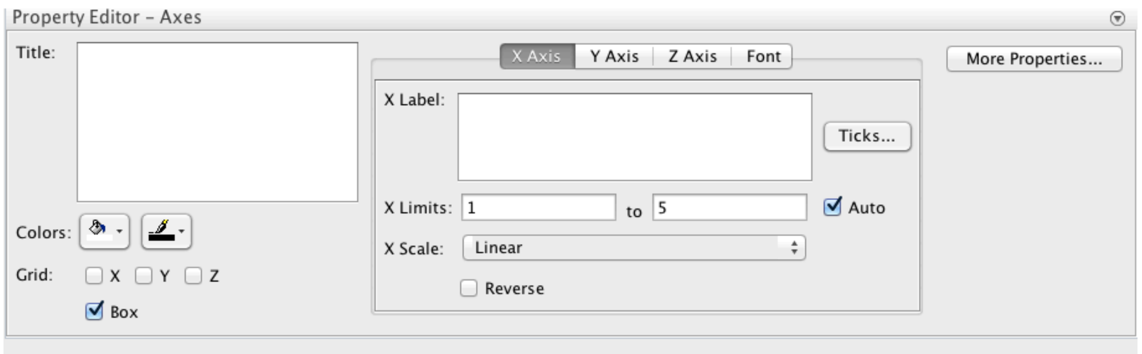
- As engineers its very important to label the axes of the plots.

- The programming way to label the axes is to employ

```
>> xlabel('x values')
>> ylabel('y values')
```

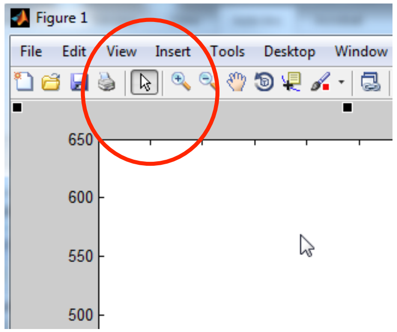
- One can also use the graphical approach.
  - Click on the arrow icon in the figure you have plotted
  - Double-click on the white space of your plot
  - Use the property-editor at the bottom of your plot to change/insert information.

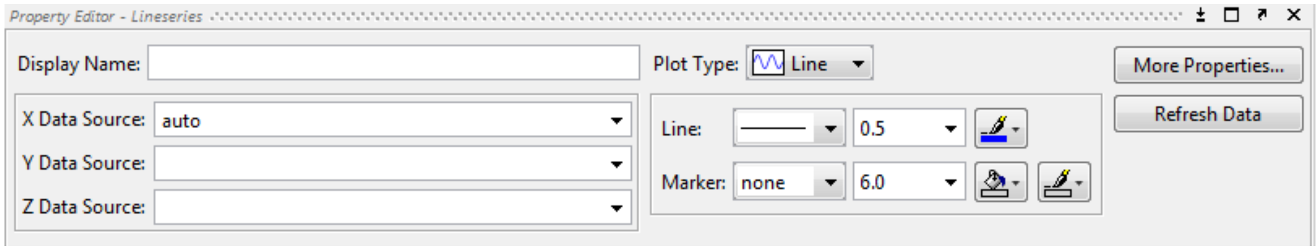




- It is also very useful to plot the lines in different colors and place markers on the lines.

- The graphical approach offers a simple way to do that
  - Click on the arrow icon in the figure you have plotted
  - Double-click on the line you have plotted
  - Use the property-editor at the bottom of your plot to change/insert information.





- "Holding" allows researchers to plot multiple equations on the same graph.

- You may use the following command to "hold" a figure

```
>> hold on;
```

- Example: Plot the following equations for $x = 1 \text{ to } 40$

$$\begin{aligned} y_1 &= 3x - 10 \\ y_2 &= x + 10 \end{aligned}$$

  - Generate an array of x values
  - Generate the output vectors for each equation
  - Plot the figures

$$
\begin{aligned}
y_1 &= 3x - 10 \\
y_2 &= x + 10
\end{aligned}
$$

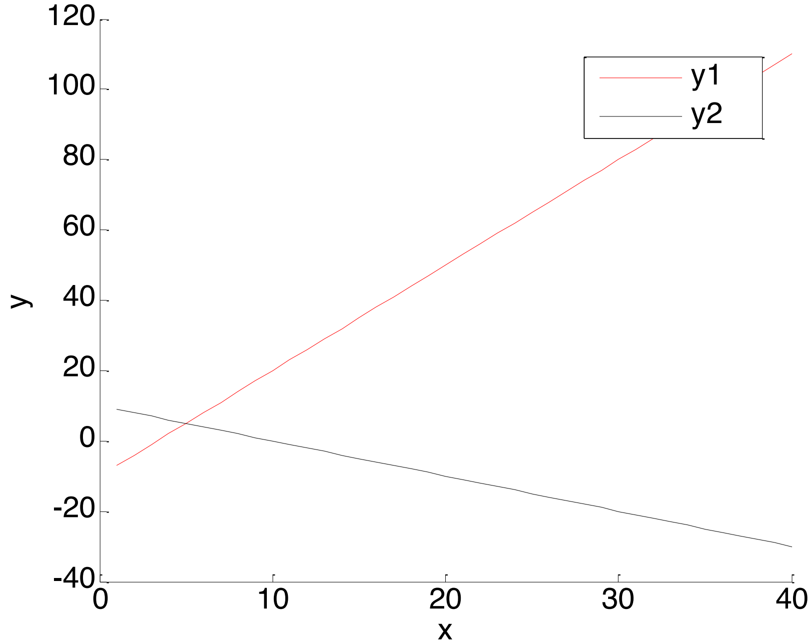
- Generate an array of x values

```
>> x=[1:40];
```

- Generate the output vectors for each equation

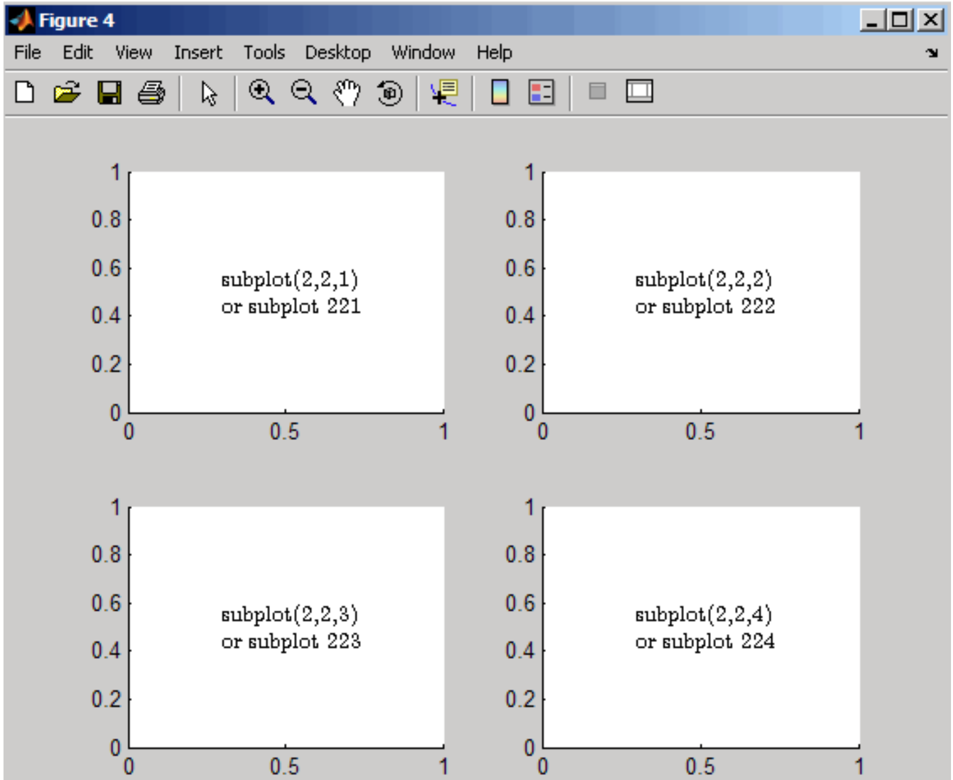
```
>> y1 = 3*x-10;
>> y2 = -x+10;
```

- Plot the figures



- Subplot allows one to plot two or more functions on separate axes.

- The function “subplot(m,n,p)”
  - breaks the figure into m-by-n matrix
  - the integer “p” defines the subplot index



- Consider the case where you may want to plot the following separately.

$$\begin{aligned} y_1 &= 3x - 10 \\ y_2 &= x + 10 \end{aligned}$$

  - Generate a new figure

    >> figure(39);

  - Define the 1st subplot position and plot 1st graph

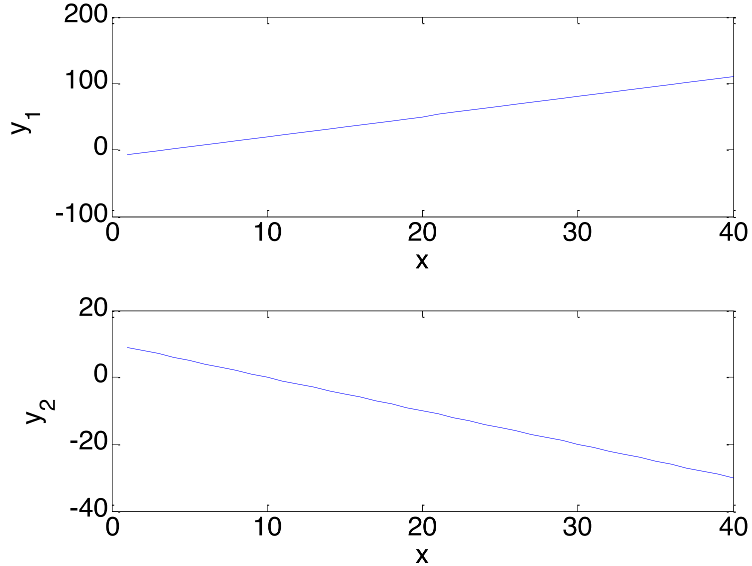
    >> subplot(2,1,1);

    >> plot(x,y1);

  - Define the 2nd subplot position and plot 2nd graph

    >> subplot(2,1,2);

    >> plot(x,y2);



CHAPTER 5

Scripts and Functions

Saving your scripts

- Thus far, all commands have been entered into the “Command window”

- Advantages of using this command window include
  - simple interface for users to type in commands
  - providing a quick way to validate computations

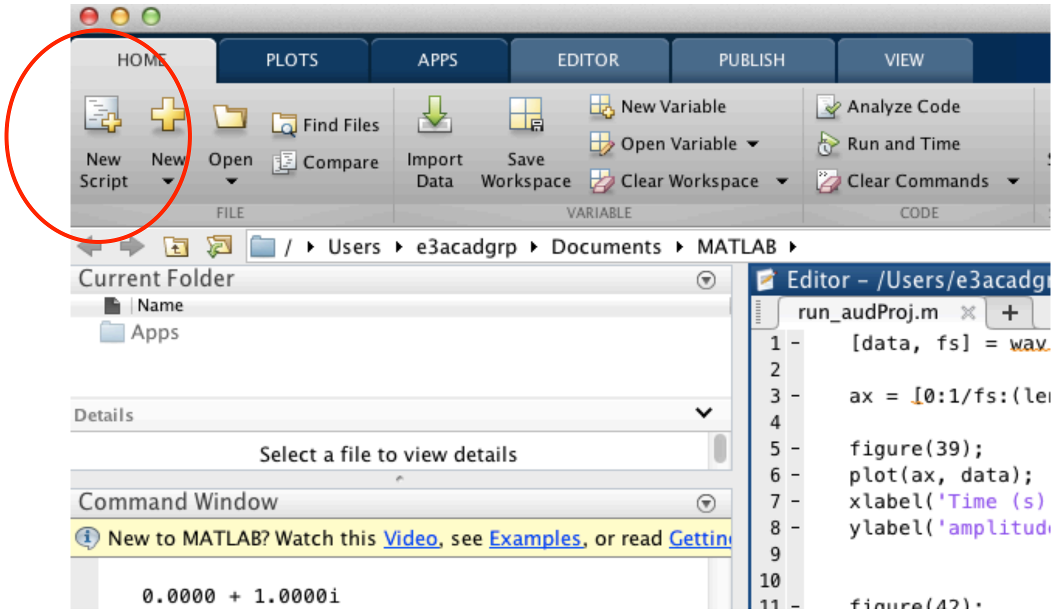
- However, disadvantages of using the command window include
  - not being able to save the commands for future reference
  - the need to re-key commands all over again
    - if there is a typo error
    - on a separate occasion (after you close MatLab)
  - not being able to execute multiple commands in a single run (commands are currently executed after every line)

- The use of scripts allows one to save the commands in a file, and run multiple commands

- Here, we will create a script which solves the following equations and verifies the answer graphically.

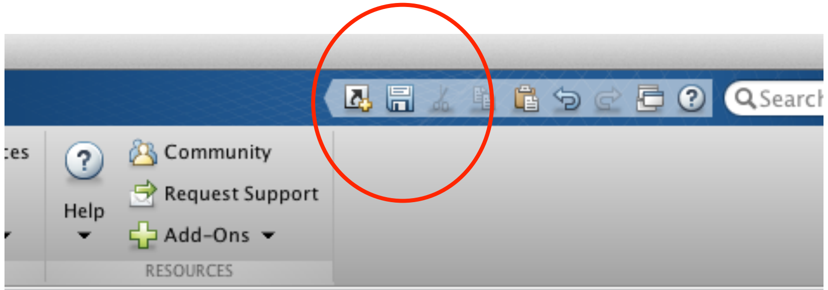
$$
\begin{aligned}
2x + 10y &= 54 \\
3x - 5y &= -19
\end{aligned}
$$

- First create a folder under desktop and name it "projects"
- Create a script by clicking the "New Script" button at the "Home" tab



- You will be brought to the “Editor” page with a filename “Untitled”

- Save this file as “run_solveSimEqns” in the desktop folder you created



- Formulate the problem in Matrix notation according to Section 2.5

$$\begin{aligned} 2x + 10y &= 54 \\ 3x - 5y &= -19 \end{aligned}$$

$$\begin{bmatrix} 2 & 10 \\ 3 & -5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 54 \\ -19 \end{bmatrix}$$

$$\mathbf{A}\mathbf{q} = \mathbf{p}$$

$$\mathbf{q} = \mathbf{A}^{-1}\mathbf{p}$$

- Key in the following commands

```
>> A = [2 10; 3 -5];
>> p = [54; -19];
>> q = inv(A)*p;
```

$$\begin{aligned} 2x + 10y &= 54 \\ 3x - 5y &= -19 \end{aligned}$$

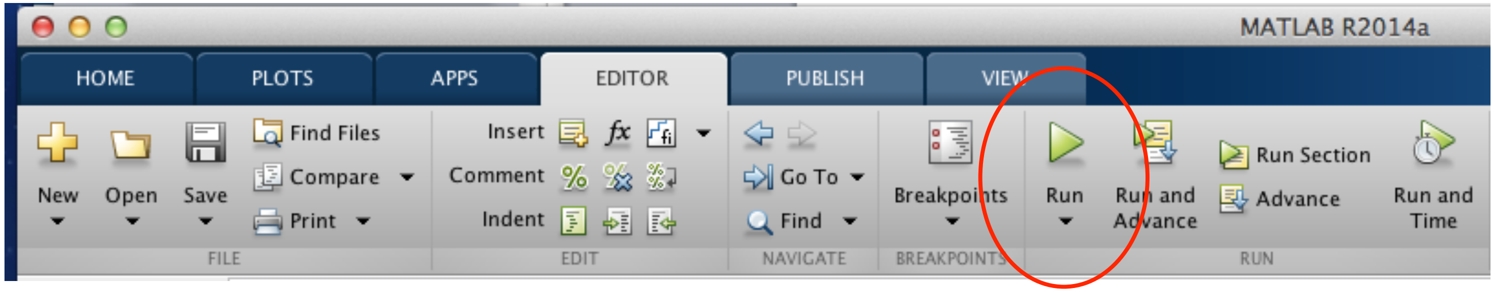
$$\begin{bmatrix} 2 & 10 \\ 3 & -5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 54 \\ -19 \end{bmatrix}$$

$$\mathbf{A}\mathbf{q} = \mathbf{p}$$

$$\mathbf{q} = \mathbf{A}^{-1}\mathbf{p}$$

- To verify the result graphically, we use

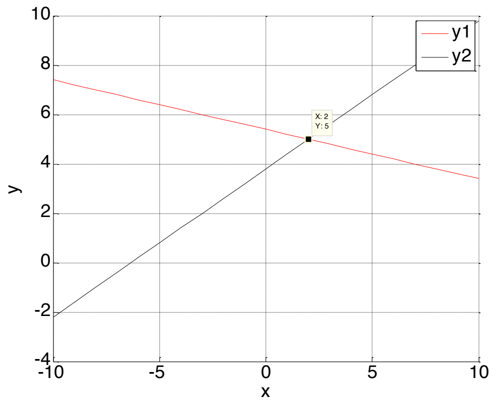
- Click on the “run” icon after all commands have been entered



- Error messages, if any, will appear in the command window.

- The unknowns can be found by double clicking the $\mathbf{q}$ variable in the Workspace.

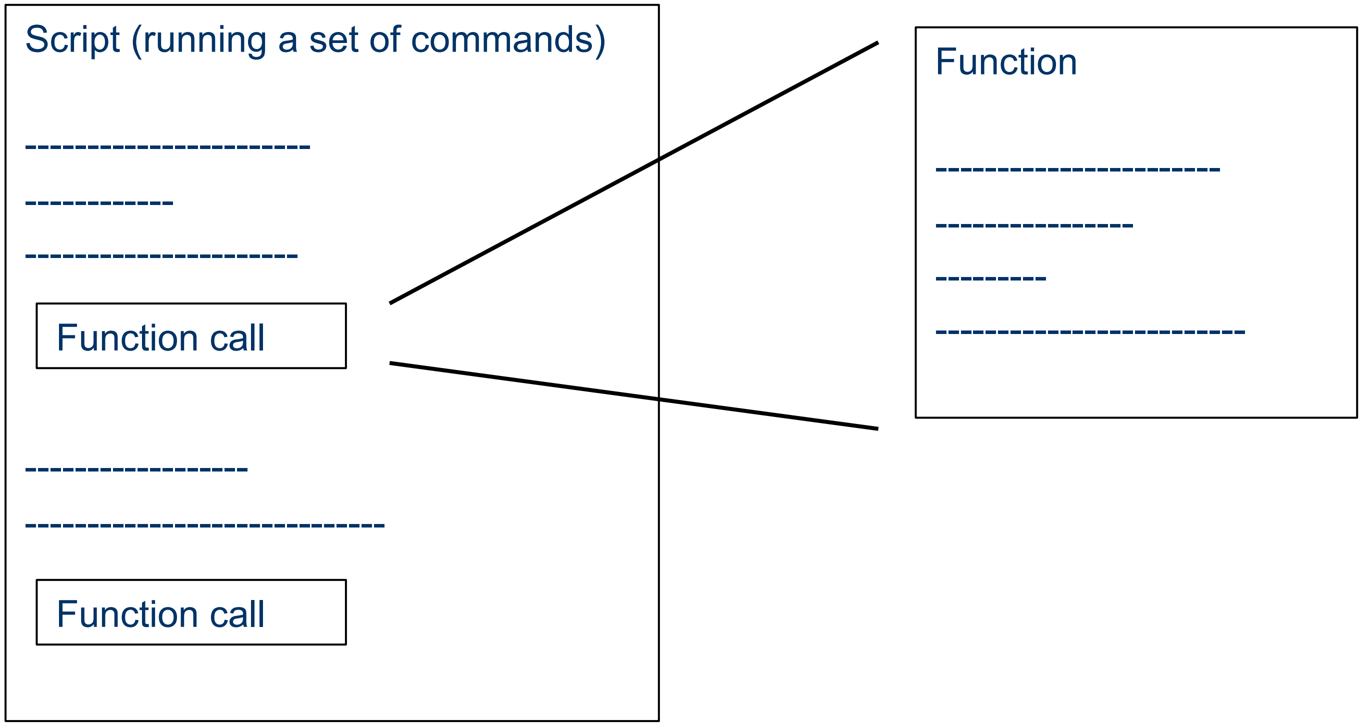
$$\mathbf{q} = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \end{bmatrix}$$



- You may now solve any simultaneous equations by changing the values of variables $\mathbf{A}$ and $\mathbf{p}$ in the script without having to key in all the commands.

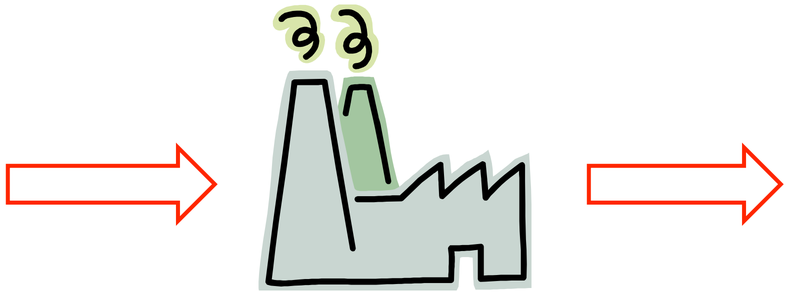
- Functions are a neat way to organize your codes, particularly if you are working on large projects.

- The relationship between “functions” and “scripts” can best be described using the following figure.



- Functions are analogous to an automation factory; it takes in raw materials (input argument(s)) and generates products (output argument(s)).

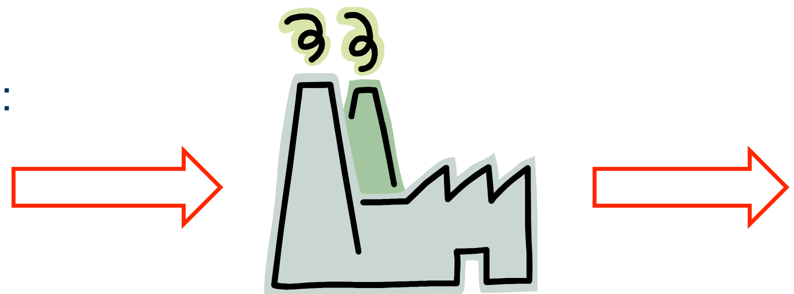
Input argument(s)



output argument(s)

- For example, we can have a function which computes the volume and the surface area of a cuboid.

Input argument(s):
- Length
- Width
- Height



output argument(s):
- Surface area
- Volume

- Example: Write a script and a function to compute the surface area and volume of a cuboid.
  - We first create the script by clicking on “New Script” under the “Home” tab
  - Save this script as “*run_solveCuboid.m*” in the desktop folder you created. Use “Save” under the “Editor” tab.
  - Key in the following commands

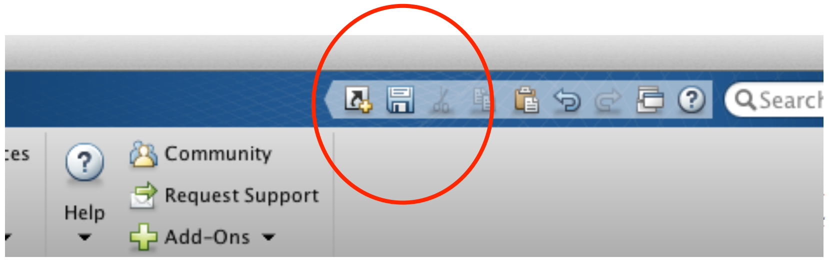
>> length = 4;

>> width = 3;

>> height = 2.5;

$$\text{>> } \underbrace{\text{[volCubd, surfAreaCubd]}}_{\text{Output arguments}} \text{ = } \underbrace{\text{compCubd}}_{\text{Function name}} \text{ } \underbrace{\text{(length, width, height)}}_{\text{Input arguments}}\text{;}$$

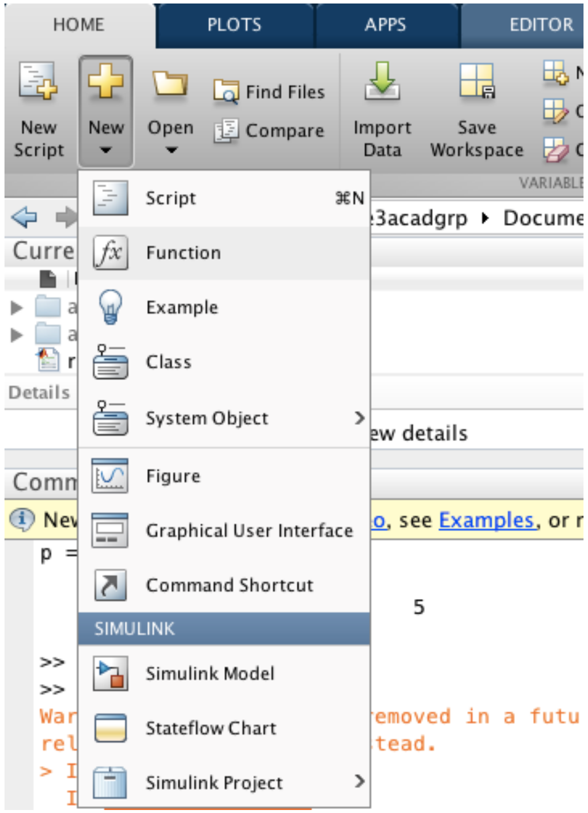
  - Save the file by clicking the save icon

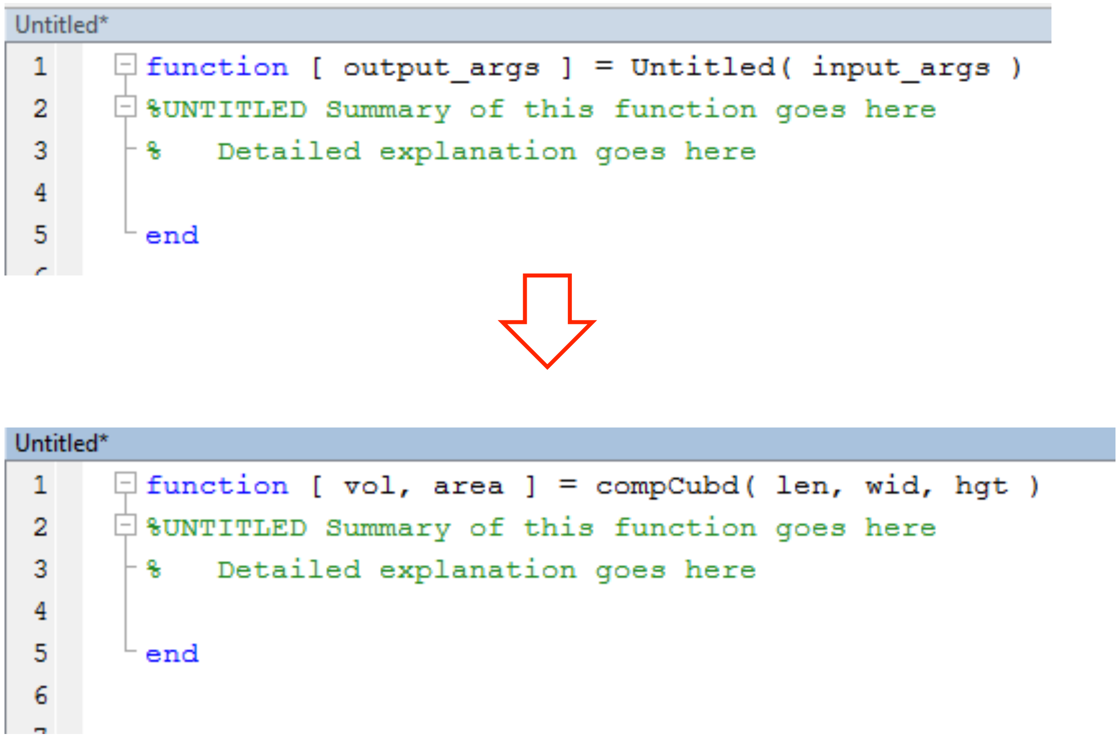


- Create the function using

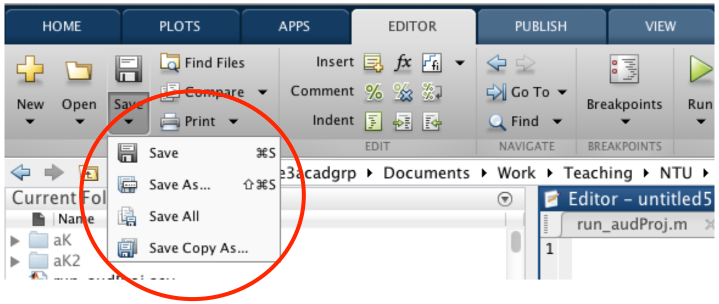
- Change the first line to

  function [vol, area] = compCubd (len, wid, hgt);





- Click the save icon, save the file under the default file name "*compCubd.m*" in the same folder as the script.

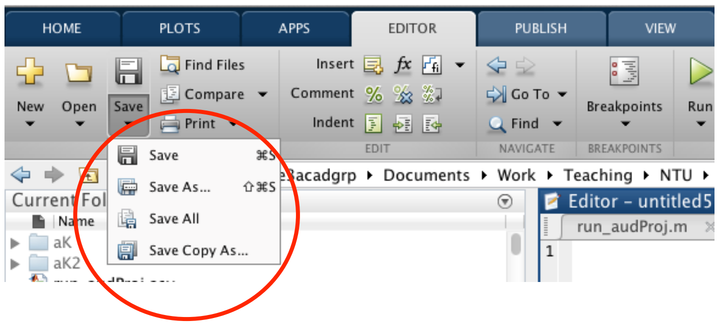


- Key in the following commands

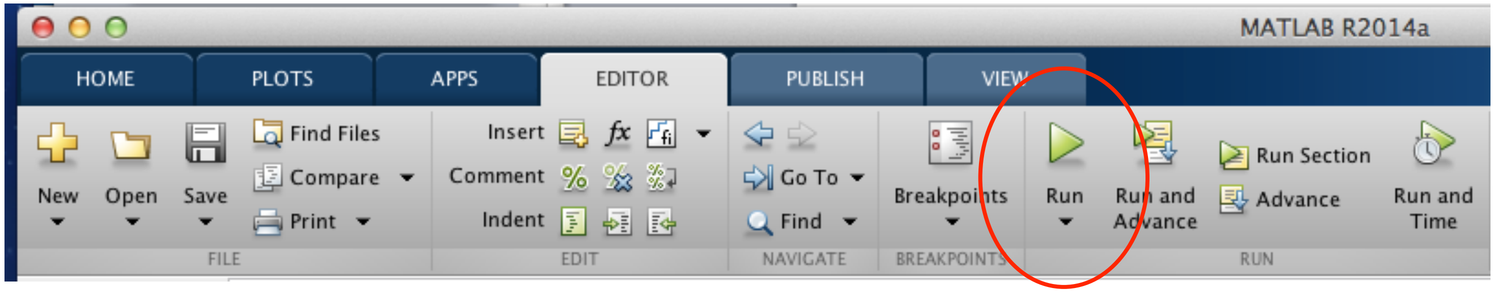
  >> vol = len*wid*hgt;

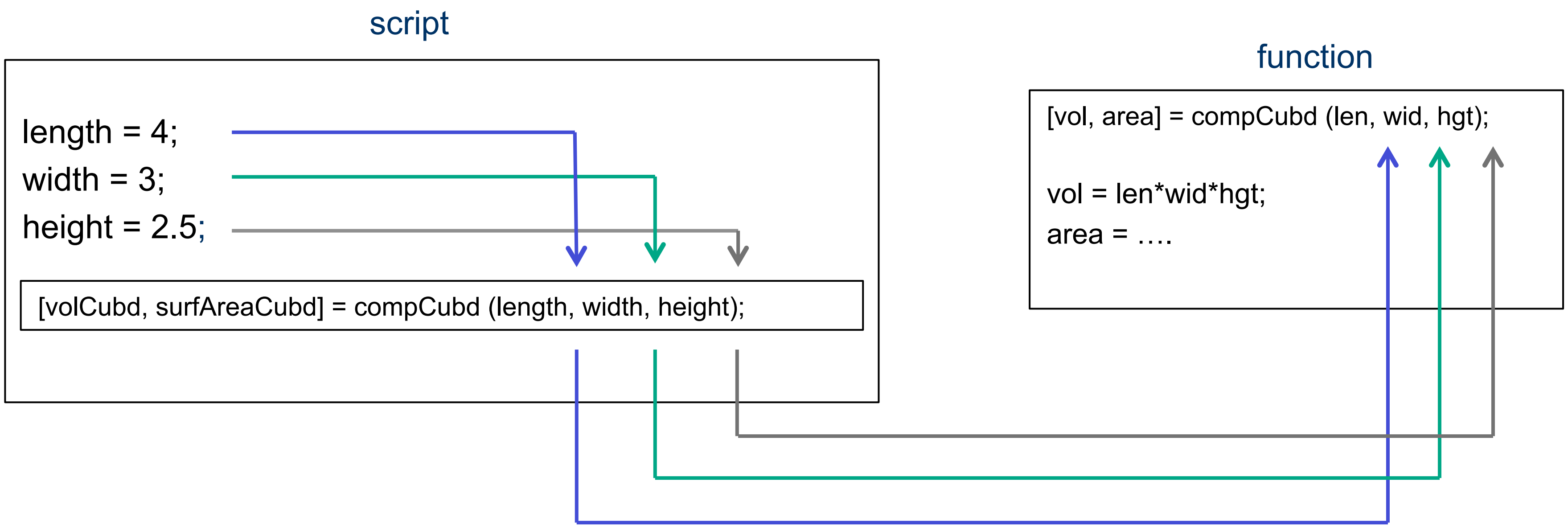
  >> area = 2*(len*wid)+2*(len*hgt)+2*(wid*hgt);

- Click the save icon



- Open the “*run_solveCuboid.m*” script and run it





CHAPTER 6

Sine/Cosine Plots

- An analog sinusoid is in the form of

$$\begin{aligned} x(t) &= A\cos(2\pi f t) \\ &= A\cos(\omega t) \end{aligned}$$

Therefore, by definition,

$$\begin{aligned} f &: \text{analog frequency in cycles/sec (Hz)} \\ \omega &: \text{angular (analog) frequency in rad/sec} \end{aligned}$$

- A digital sinusoid is in the form of

$$\begin{aligned} x[n] &= A\cos(2\pi f_0 n) \\ &= A\cos(\omega_0 n) \end{aligned}$$

Therefore, by definition,

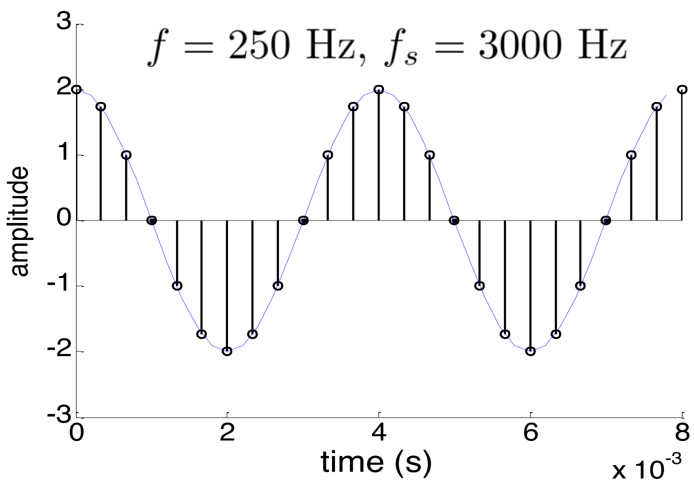
$$\begin{aligned} f_0 &: \text{digital frequency in cycles/sample} \\ \omega_0 &: \text{angular (digital) frequency in rad/sample} \end{aligned}$$

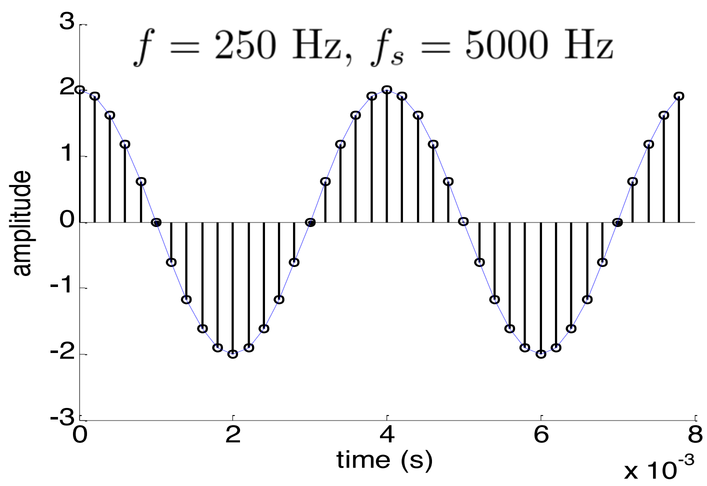
 Discrete-time signal from continuous-time signal

- Sampling a continuous-time signal results in a discrete-time signal.

- We often write
$$\begin{aligned} x[n] &= x_{\text{continuous}}(nT) \\ T &= 1/f_s \text{ is the sampling period in sec} \\ f_s &: \text{ sampling frequency in Hz} \end{aligned}$$

- A higher f_s implies that the analog signal is sampled more frequently.





Discrete-time signal from continuous-time signal

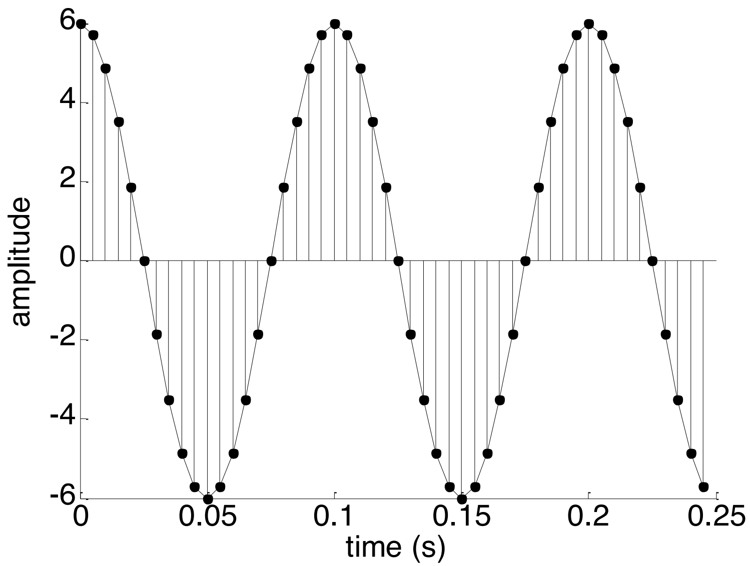
Consider an analog signal $x(t) = 6\cos(20\pi t)$. Given a sampling rate of $f_s = 200\ \text{Hz}$, find the discrete representation of the signal.

A sampling rate of $f_s = 200\ \text{Hz}$ corresponds to a sampling period of $T = 1/200 = 0.005\ \text{sec.}$

This implies that we have a digital signal at sample index n every 0.005 s.

Sampling $x(t)$ at this period will result in

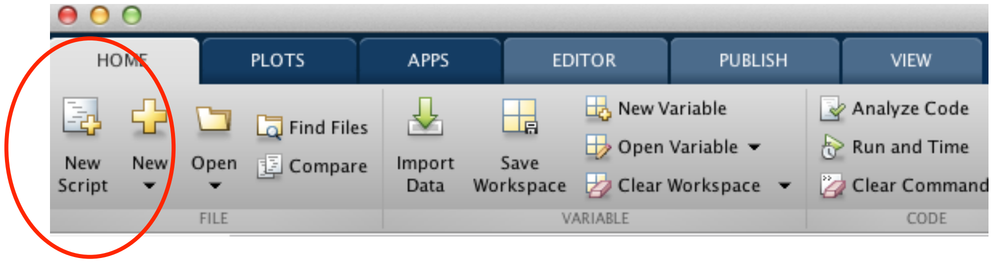
$$
\begin{aligned}
x[n] &= 6\cos(20\pi \times nT) \\
&= 6\cos(0.1\pi n)
\end{aligned}
$$



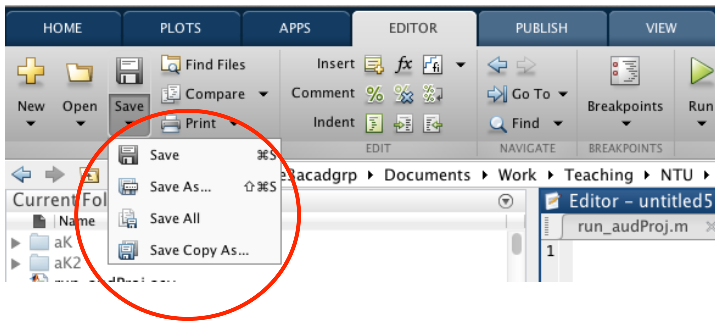
6.3 Plotting a discrete cosine wave

- Example: To plot the first 200 samples of the signal $x(t) = \cos(200\pi t)$ at a sampling rate of $f_s = 200 \text{ Hz}$

  - First create the script using



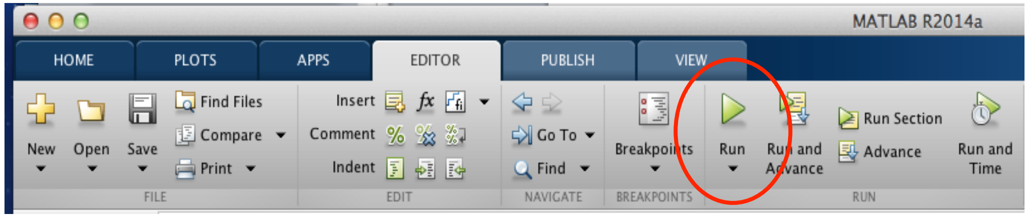
  - Save this script as "*run_sinegen.m*" in the desktop folder you created



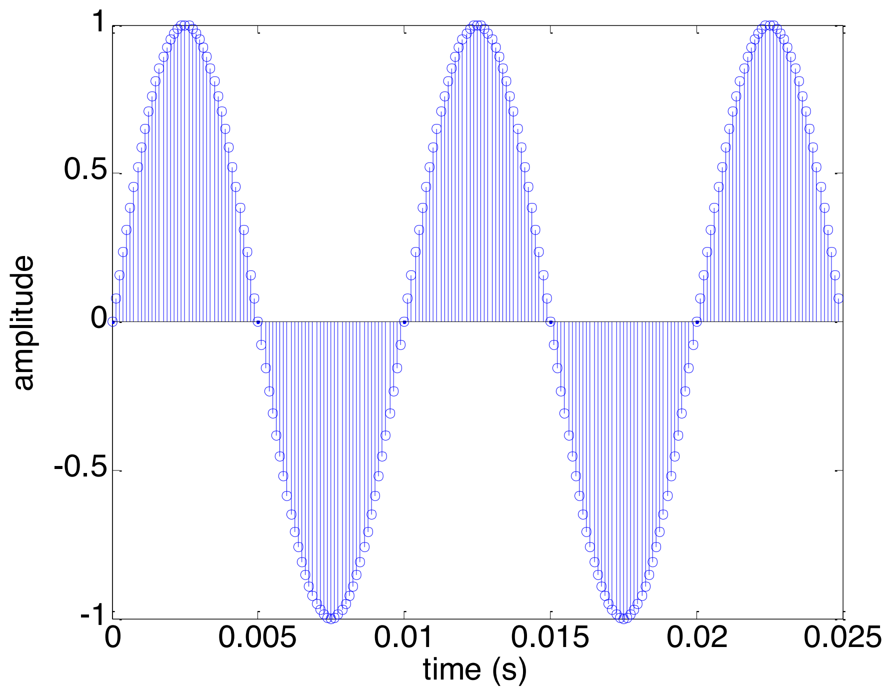
  - Key in the following commands

```
>> fs = 8000;
>> fsig = 100;
>> nsamp = 200;
>> t = [0:1/fs:(nsamp-1)/fs];
>> sig = sin(2*pi*fsig*t);
```

```
>> figure(25);
>> stem(t,sig);
>> xlabel('time (s)');
>> ylabel('amplitude');
```



- Save and run this script



CHAPTER 7

Applications

- It is very easy to read audio files into MatLab and process them using signal processing technique(s).

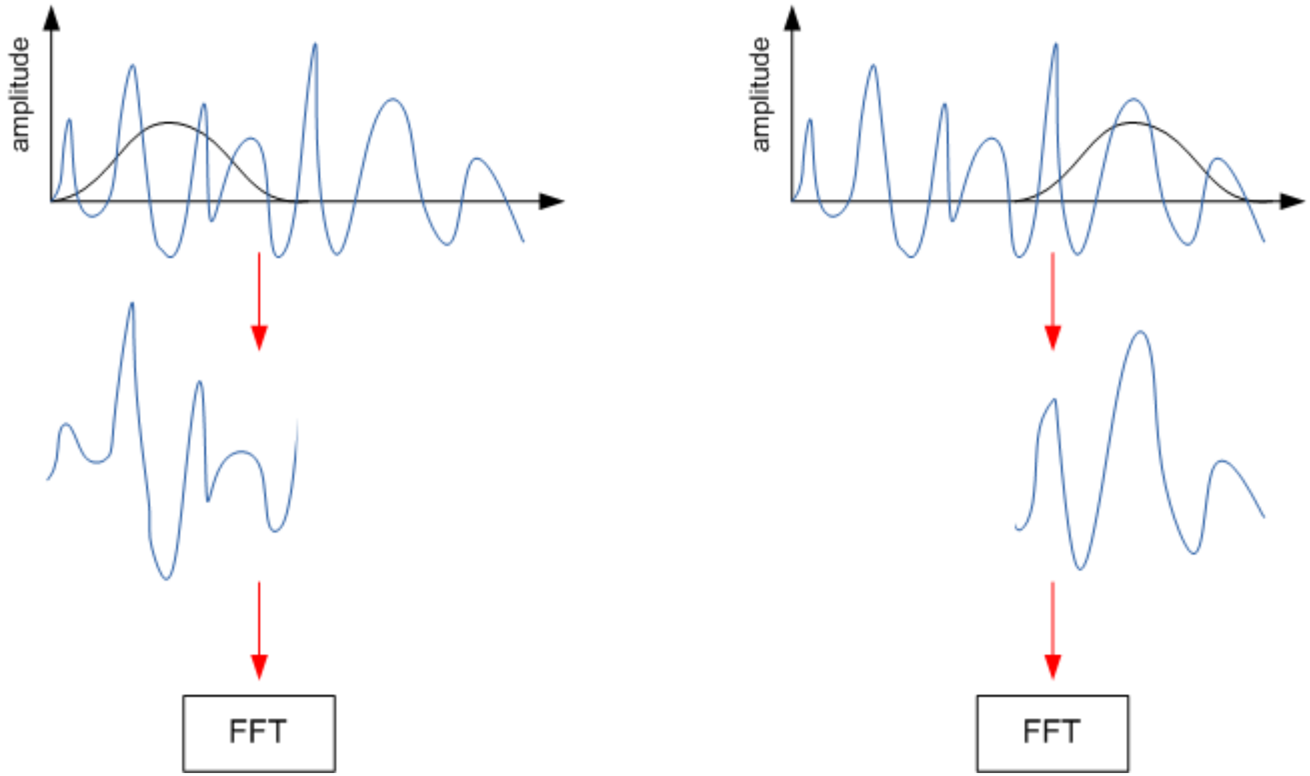
- For versions before MatLab 2012, you may read .wav files using the function “wavread( )”

- For versions MatLab 2012 or later, you may read audio files using the function “audioread( )”. This supports the following file formats
  - AU, SND
  - FLAC
  - OGG
  - WAV
  - MP4 and any formats supported by Microsoft Media Foundation

- You may also write a processed signal using the following functions “wavwrite( )” or “audiowrite( )”

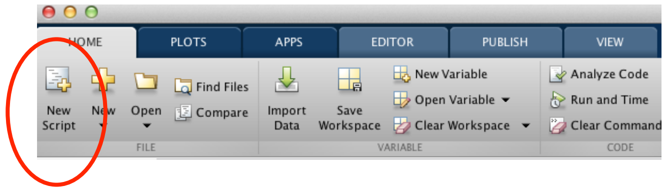
- One important tool when analyzing the audio signal is to determine how the frequency changes with time.

- This is known as the short-time Fourier transform.



- In this example, we will
  - ✓ generate a script,
  - ✓ read in an audio file
  - ✓ plot the signal
  - ✓ view its frequency content of the signal

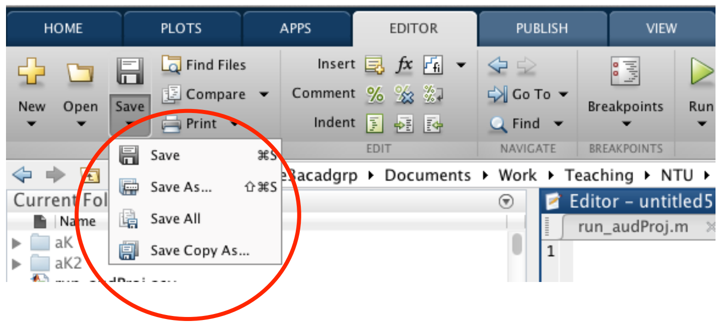
- Copy the wav file “tmtw.wav” into a desktop folder.
- Create a new script



- Save this script as “*run_audProj.m*” in the desktop folder you created

- Read the wav file

  >> [data, fs] = wavread('tmtw');

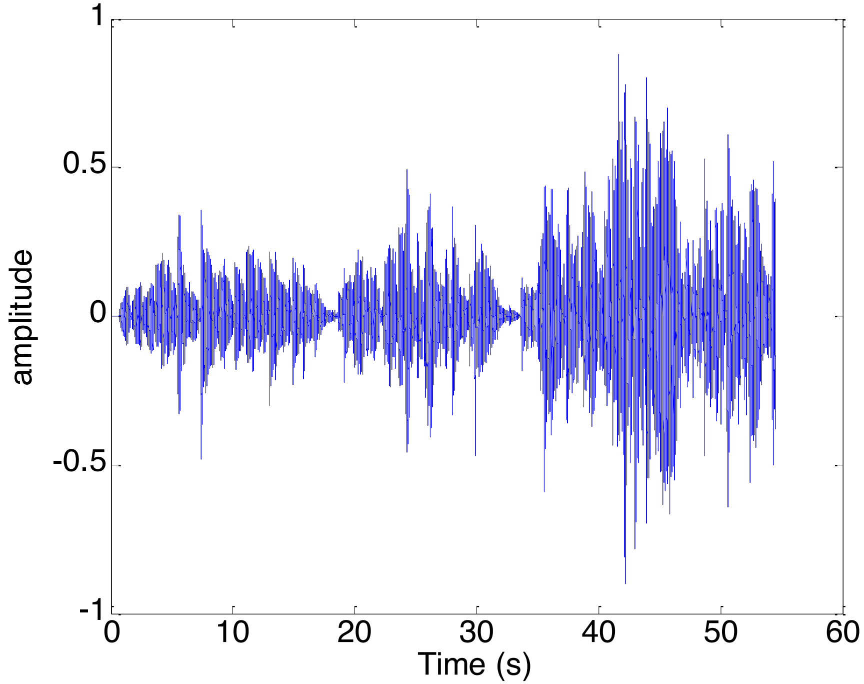


NANYANG TECHNOLOGICAL UNIVERSITY

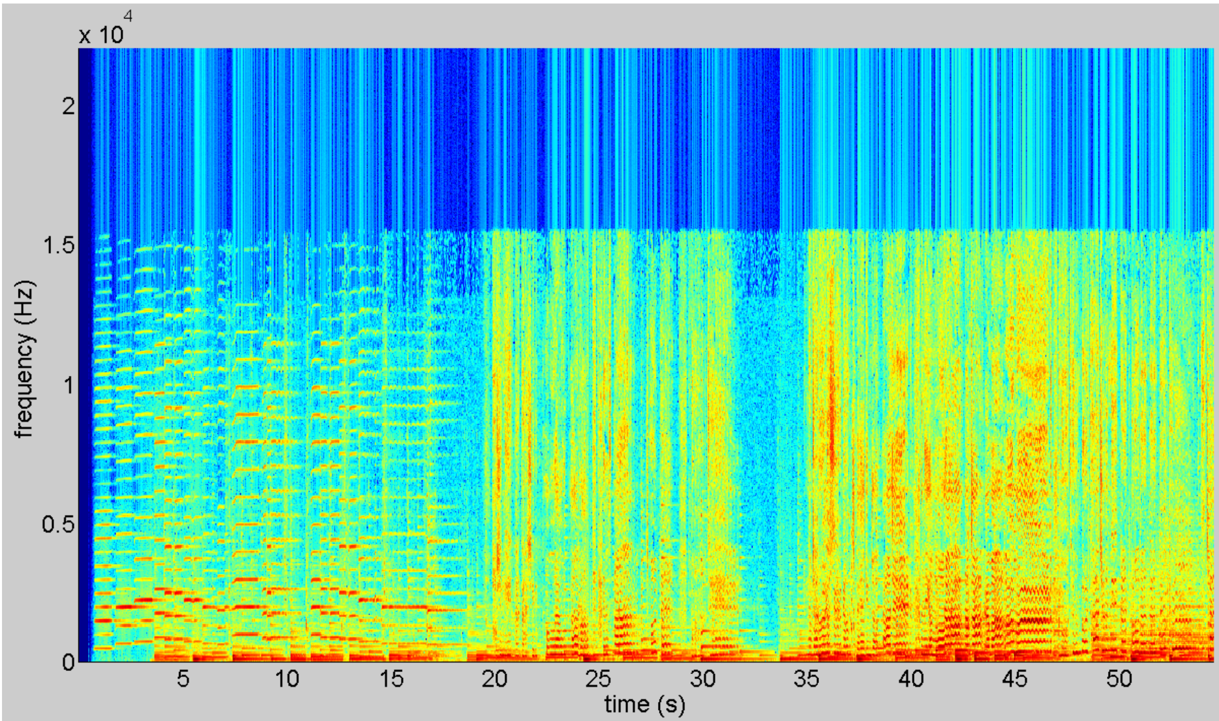
- Generate a vector containing the time sequence starting from 0 to the length of data minus 1. Since the sampling rate is f_s , the time step will be $1/f_s$.

```
>> ax = [0:1/fs:(length(data)-1)/fs];
```

- We can now plot the figure



- Generate the spectrogram using the following



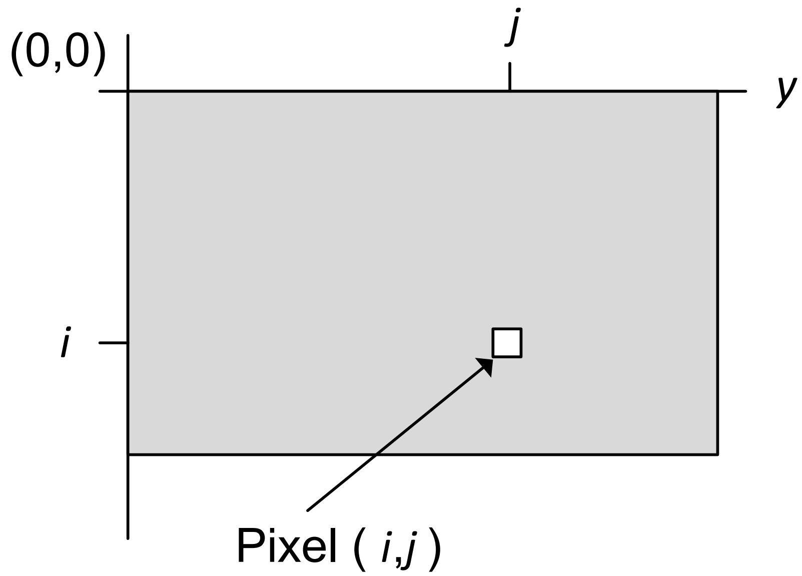
- It is very easy to read image files into MatLab and process them using signal processing technique(s).

- The function “imread( )” supports the following file formats

BMP — Windows Bitmap	JPEG — Joint Photographic Experts Group	PNG — Portable Network Graphics
CUR — Cursor File	JPEG 2000 — Joint Photographic Experts Group 2000	PPM — Portable Pixmap
GIF — Graphics Interchange Format	PBM — Portable Bitmap	RAS — Sun Raster
HDF4 — Hierarchical Data Format	PCX — Windows Paintbrush	TIFF — Tagged Image File Format
ICO — Icon File	PGM — Portable Graymap	XWD — X Window Dump

- The function “imshow( )” displays the image in MatLab.

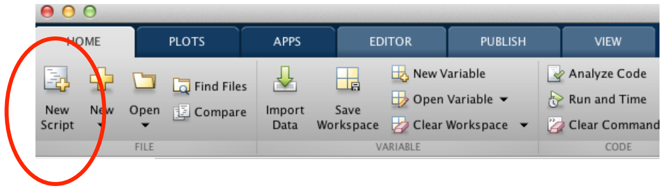
- An image in MatLab is treated as a matrix, i.e., every pixel is a matrix element.



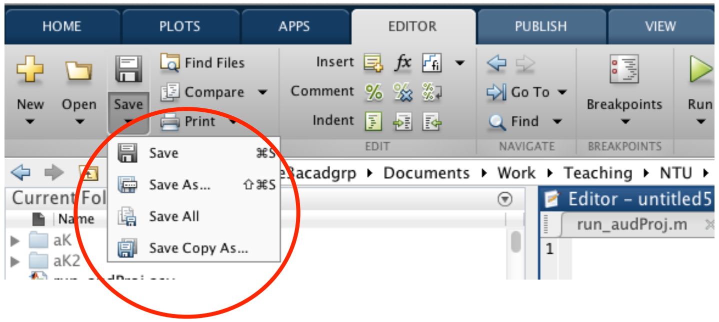
- Note that the (0,0) position is located at the top-left corner of the image.

- Example: We can generate a 500×500 matrix corresponding to greyscale (black-white) values via the following.

  - Create the script



  - Save this script as "*run_genGreyScale.m*" in the desktop folder you created



  - Define the size of the matrix

```
>> size = 500;
```

  - Generate a vector containing integers from 1 to 500^2. The last element will hold the value 500^2 since, for a 500×500 matrix, there will be 500^2 pixels.

```
>> y = [1: size^2];
```

- We next reshape the 1×500^2 vector into a 50×50 matrix
- `>> Y = reshape(y,size,size);`

1	2	…	500^2

1	501	…	249501
2	…	…	249502
…	…	…	…
500	…	…	500^2

- Normalize each value by dividing all elements in the vector by the largest number, i.e., 5002. This is to ensure that the largest value in the matrix is 1.

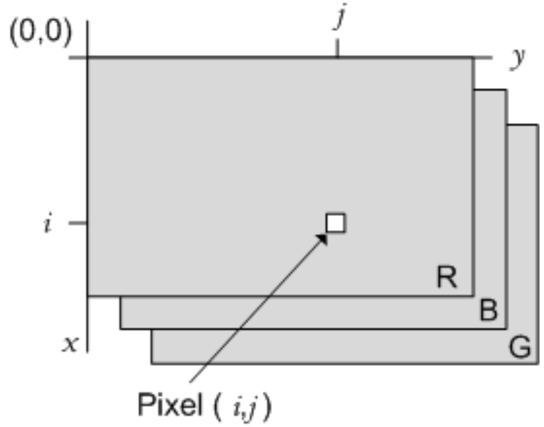
  `>> Y = Y./Y(end);`

- We now show the image using

  `>> imshow(Y);`

- Images, in general, are not greyscale, i.e., its values do not vary between 0 and 1.

- True color images comprise of RGB values.



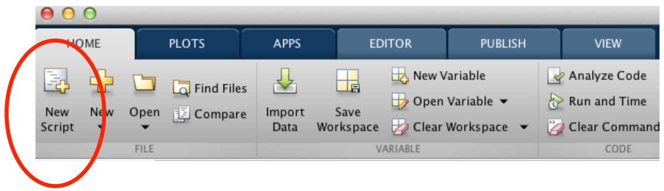
- The function "imread( )" for true color image will therefore result in a 3D matrix with the 3rd dimension corresponding to the RBG values.

- http://web.njit.edu/~kevin/rgb.txt.html

- In this example, we will
  - ✓ generate a script,
  - ✓ read in an image file
  - ✓ show the image on MatLab
  - ✓ adjust its RBG values

- Copy the jpg file "zoo.jpg" into a desktop folder.

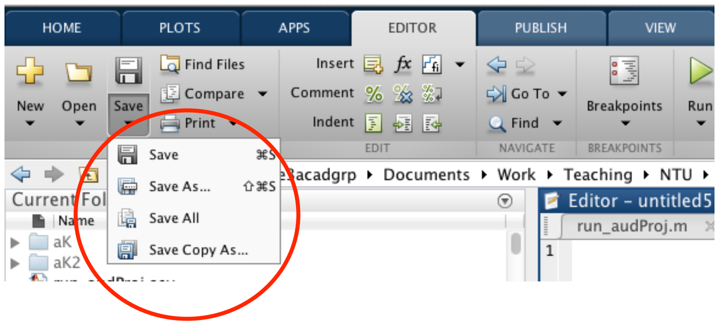
- Create the script using



- Save this script as "*run_imgProj.m*" in the desktop folder you created

- Read the image file

  >> image = imread('zoo.jpg');



- Plot the image file

```
>> figure(32);
>> imshow(image);
>> title('original image');
```

- To reduce the blue component, we first create a dummy variable and equate it to the original image

```
>> imageA = image;
```

- Next we change the blue component by changing the 2nd component of "imageA"

```
>> imageA(:,:,2)= 0.5*imageA(:,:,2);
>> figure(34)
>> imshow(imageA);
>> title('Image A- reduced Blue component');
```

original image

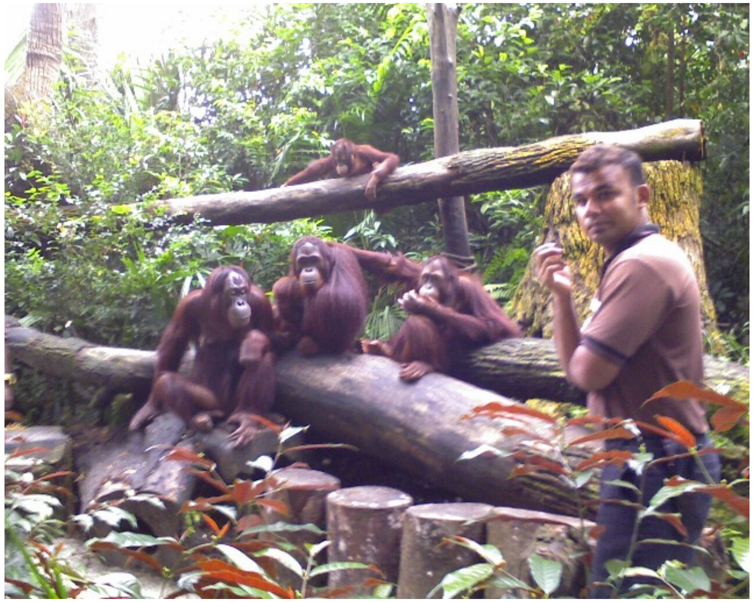


Image A- reduced Blue component

धन्यवाद

Merci

谢谢

Gracias

Obrigado!

Ευχαριστώ

شكراً