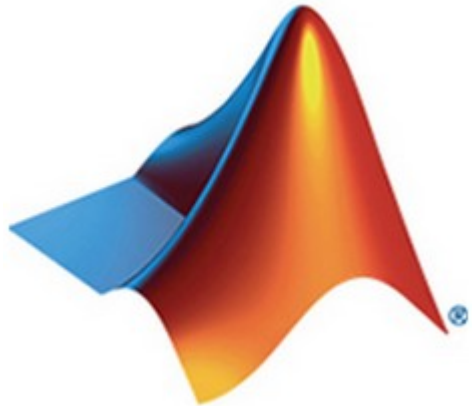
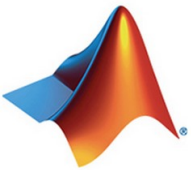


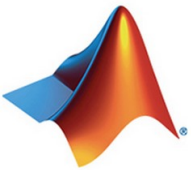
Introduction to Programming in MATLAB

By:
Himanshu Mittal



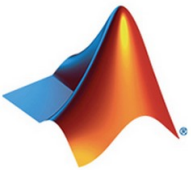
Outline

- MATLAB Basics
- Plotting
- GUI
- Database Handling
- Audio Processing
- Image Processing
- Video Processing
- Data Mining



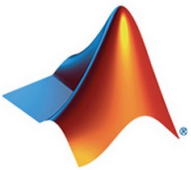
Introduction to MATLAB

- High-performance language for numerical computing
- Stands for **MAT**rix **LAB**oratory
- Developed by MathWorks, 1983
- Easy-to-use environment for:
  - ✓Computation
  - ✓Visualization
  - ✓Programming
- Typical uses include:
  - ➢Mathematical computation
  - ➢Algorithm development
  - ➢Modeling, simulation, and prototyping
  - ➢Data analysis, exploration and visualization



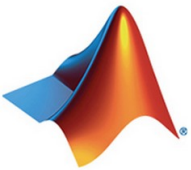
Why MATLAB

- Easy to formulate solutions for computing problems, especially involving matrix representation
- Excellent display capabilities
- Family of application-specific toolbox
- Widely used for research in industry and universities



Why not MATLAB

- Not free
- Great for prototyping but not for developing complete
- Memory inefficient comparatively for small values



MATLAB

- **Some facts regarding MATLAB:**

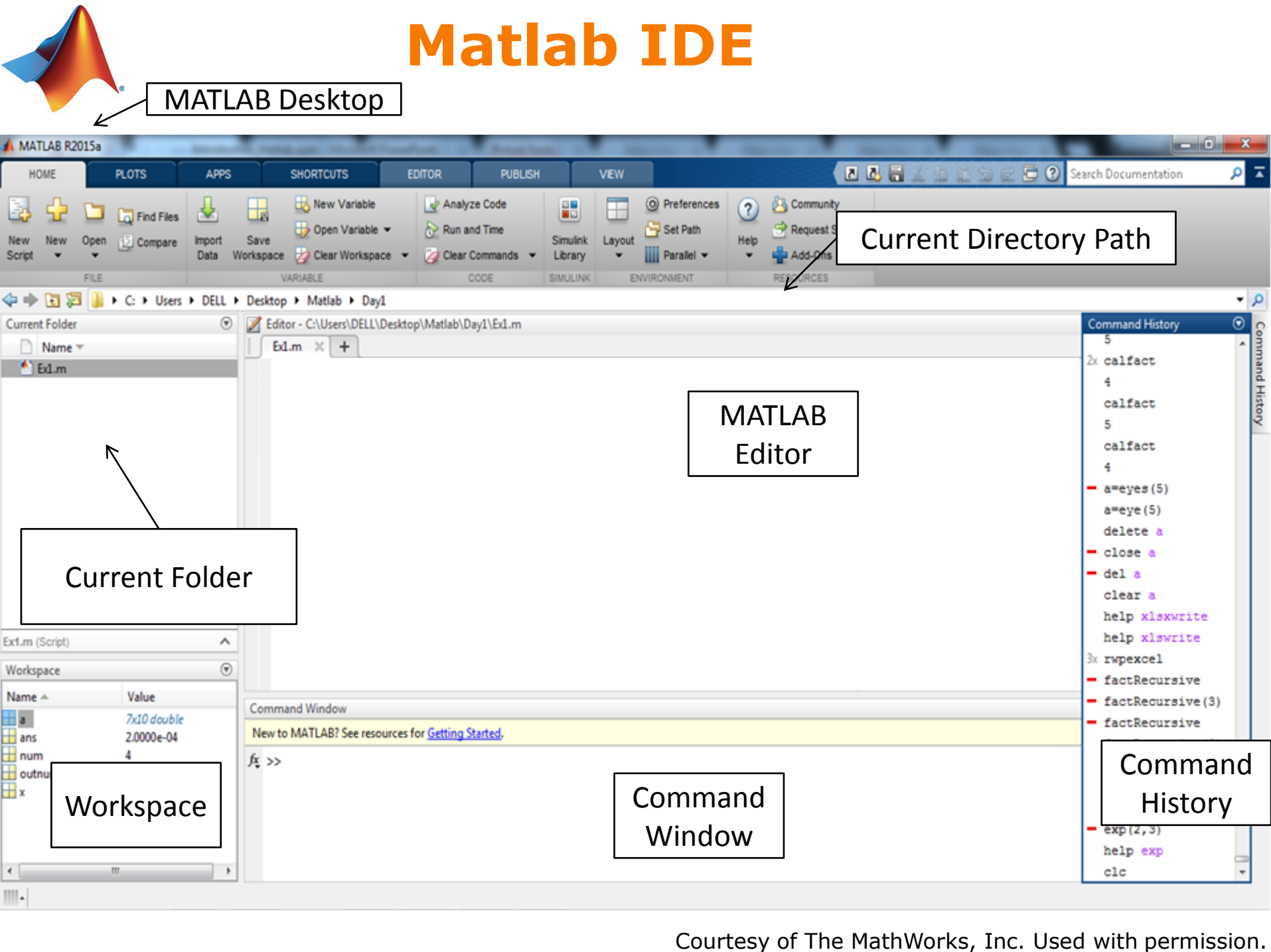
  ✓Everything in MATLAB is matrix

  ✓MATLAB is an interpreted language, executes command line by line

  ✓MATLAB does not need any variable declarations, no storage allocation, no packaging, no pointers

  ✓Indexing in MATLAB starts with 1 instead of 0.

Matlab IDE

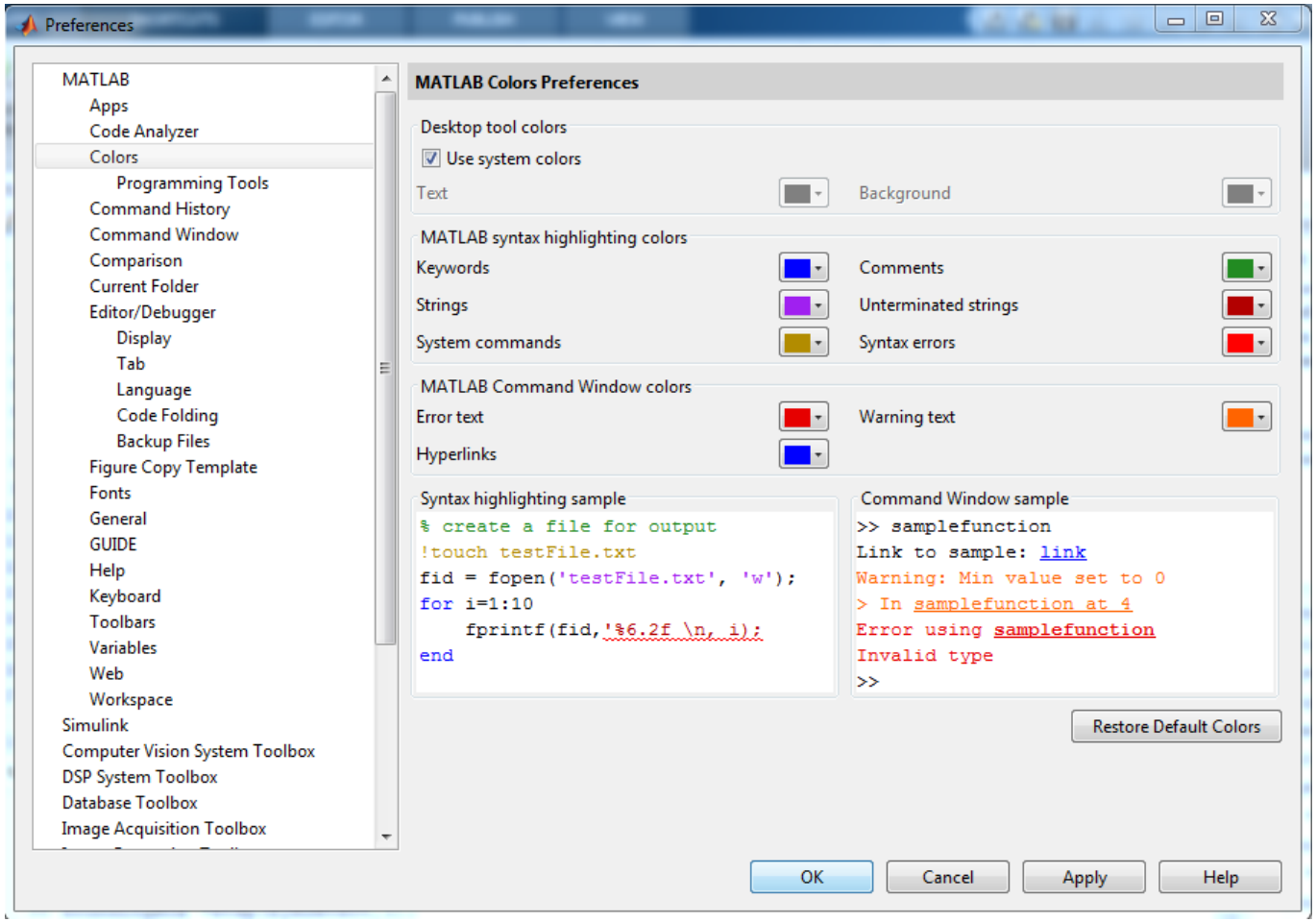


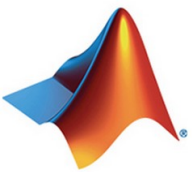
Making Folders Using IDE

- Use folders to keep your programs organized
- To make a new folder, click the ‘Dropdown Arrow’ button next to ‘Current Folder’
- Click the ‘New Folder’, and change the name of the folder. **Do NOT use spaces** in folder names.
- Double-click the folder you just made
- The current folder is now the folder you just created

Customization

- Under *Home* tab, select *Preferences*
  - ➢ Allows you personalize your MATLAB experience





Help/Docs

- **`help`**
  - **The most** important function for learning MATLAB on your own
- To get info on how to use a function:

  `» help sin`
  - Help lists related functions at the bottom and links to the doc
- To get a nicer version of help with examples and easy-to-read descriptions:

  `» doc sin`
- To search for a function by specifying keywords, use search tab of doc:

  `» doc`
- **TRY:**

  » doc Sample Data Sets

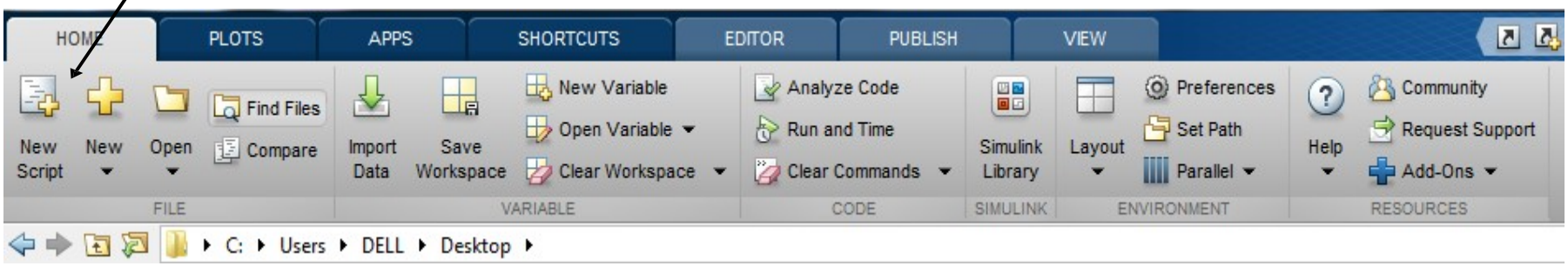
Scripts: Overview

- Scripts are
  - collection of commands executed in sequence
  - written in the MATLAB editor
  - saved as MATLAB files (.m extension)

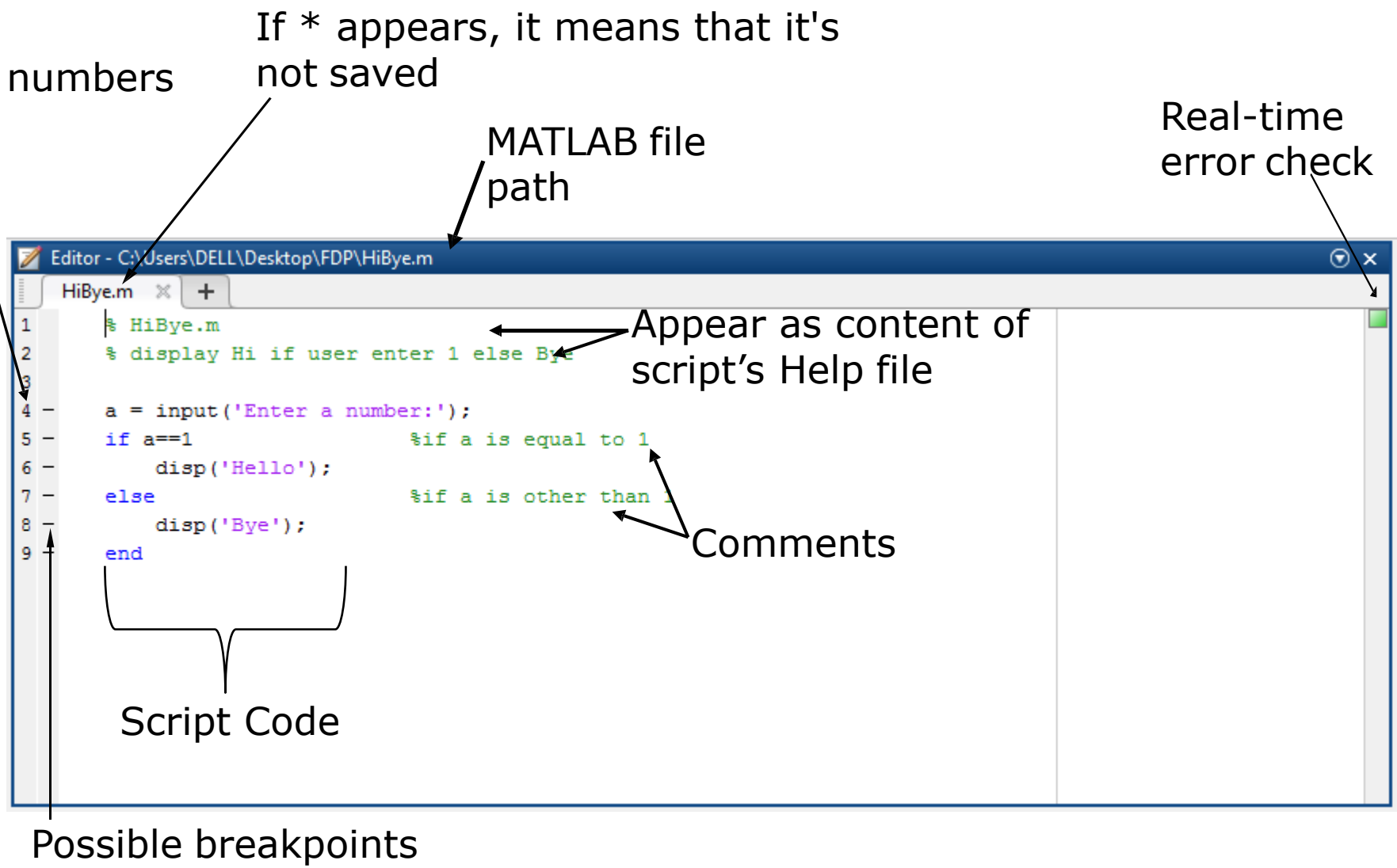
- To create a MATLAB file from command-line

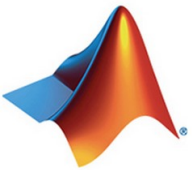
```
» edit HiBye.m
```

- or click



Scripts: the Editor

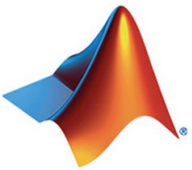




Scripts: Some Notes

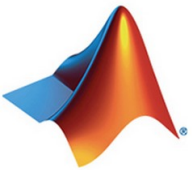
- **COMMENT!**
  - Anything following a **%** is seen as a comment
  - When **% %** appears, the code below it appear as a section
  - Comment thoroughly to avoid wasting time later

- All variables created and modified in a script exist in the workspace even after it has stopped running



display

- Use **disp** to print messages
  - » `disp('starting loop')`
  - » `disp(['loop is over' num2str(9)])`
    - ➢ disp prints the given string to the command window (Strings are written between single quotes, like `'This is a string'`.)



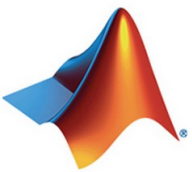
Exercise: Scripts

Ex1: Creating a script file

- Make a `helloWorld.m` script
- When run, the script should display the following text:

Hello World!
I am going to learn MATLAB!

- Hint: use `disp` to display strings.



Exercise: Scripts

Ex1: Creating a script file

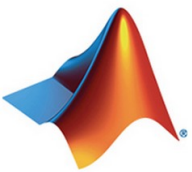
- Make a **`helloWorld.m`** script
- When run, the script should display the following text:

Hello World!
I am going to learn MATLAB!

- Hint: use **`disp`** to display strings.

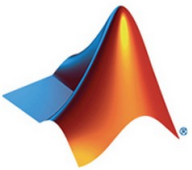
Soln: Open the editor and save a script as **`helloWorld.m`**.
This is an easy script, containing two lines of code:

```
% helloWorld.m
% my first hello world program in MATLAB

disp('Hello World!');
disp('I am going to learn MATLAB!');
```



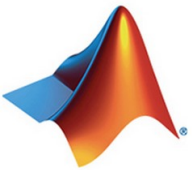
Defining variables

- To create a variable, simply assign a value to a name:
  - » `var1=3.14`
  - » `myString='hello world'`
- Variable names
  - first character must be a LETTER
  - after that, any combination of letters, numbers and _
  - CASE SENSITIVE! (`var1` is different from `Var1`)
- Built-in variables. Don't use these names!
  - `i` and `j` can be used to indicate complex numbers
  - `pi` has the value 3.1415926…
  - `ans` stores the last unassigned value (like on a calculator)
  - `Inf` and `-Inf` are positive and negative infinity
  - `NaN` represents 'Not a Number'



Variable Types

- MATLAB is a weakly typed language
  - No need to declare variables!
- MATLAB supports various types, the most often used are
  - `double`
    - 64-bits (default)
  - `char`
    - 16-bits
- Other types are also supported: complex, symbolic, 16-bit and 8 bit integers, etc.



Scalar Variables

- A variable can be initialized as a scalar, vector or matrix
- A scalar variable can be given a value explicitly

```
» a = 10
```

  - shows up in workspace!
  - To suppress the o/p, use `;` at the end of the statement.

```
» a = 10;
```

- Or as a function of explicit values and existing variables

```
» c = 1.3*45-2*a
```

Arrays

- Like other programming languages, arrays are an important part of MATLAB
- Two types of arrays

(1) matrix of numbers (either double or complex)

(2) cell array of objects (more advanced data structure)

MATLAB makes vectors easy!
That's its power!

Row Vectors

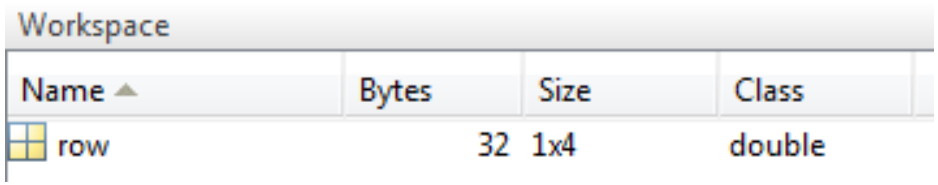
- Row vector: comma or space separated values between square brackets

```
» row = [1 2 5.4 -6.6]
» row = [1, 2, 5.4, -6.6]
   !!!TRY!!!
```

- Command window:

```
>> row=[1 2 5.4 -6.6]

row =

    1.0000    2.0000    5.4000   -6.6000
```

- Workspace:



Column Vectors

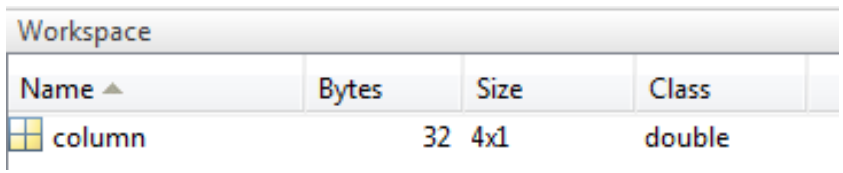
- Column vector: semicolon separated values between brackets

```
» column = [4;2;7;4]
    !!!TRY!!!
```

- Command window:

```
>> column=[4;2;7;4]

column =

     4
     2
     7
     4
```

- Workspace:



size & length

- You can tell the difference between a row and a column vector by:
  - Looking in the workspace
  - Displaying the variable in the command window
  - Using the size function

```
>> size(row)

ans =

     1     4
```

```
>> size(column)

ans =

     4     1
```

- To get a vector's length, use the length function

```
>> length(row)

ans =

     4
```

```
>> length(column)

ans =

     4
```

Matrices

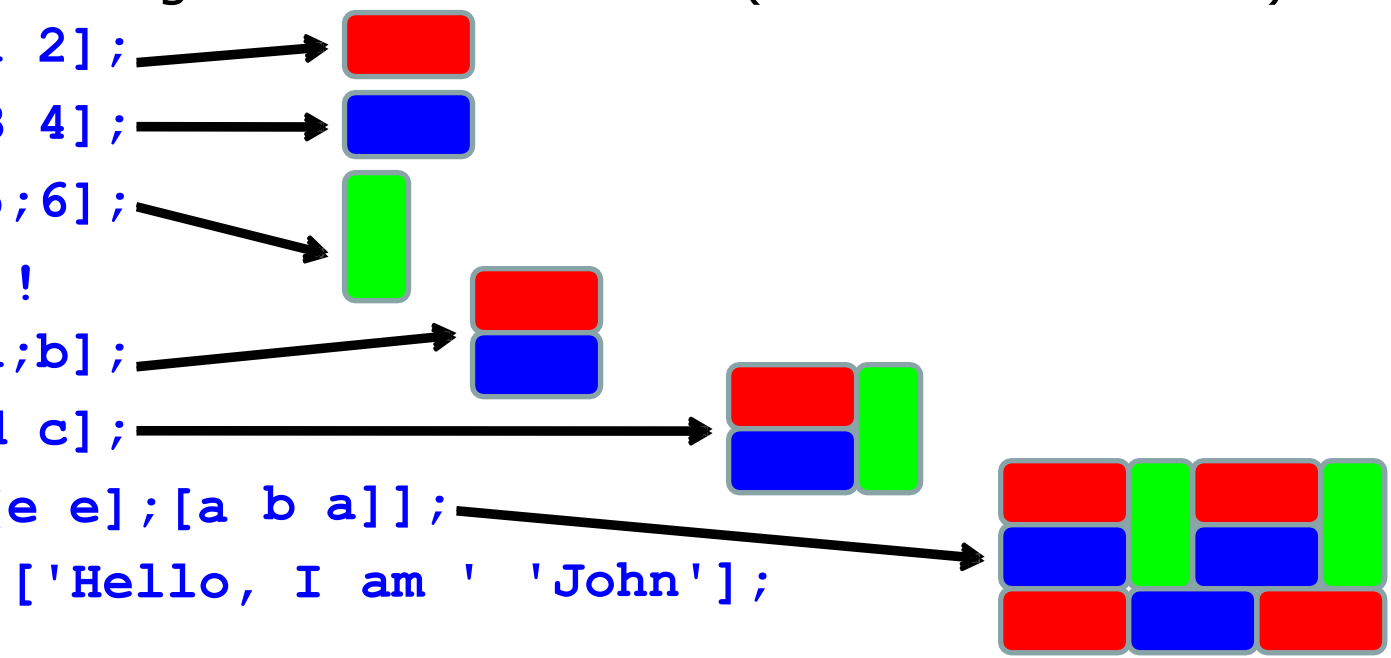
- Make matrices like vectors
- Element by element

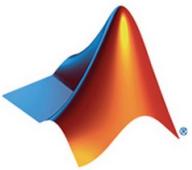
```
» a= [1 2;3 4];
```

$$a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

  - Strings are character vectors
- By concatenating vectors or matrices (dimension matters)

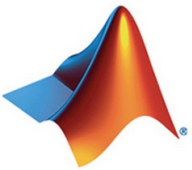
```
» a = [1 2];
» b = [3 4];
» c = [5;6];
!!!TRY!!!
» d = [a;b];
» e = [d c];
» f = [[e e];[a b a]];
» str = ['Hello, I am ' 'John'];
```





save/clear/clc/load

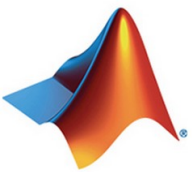
- Use **save** to save variables to a file
  - `» save myFile a b`
    - ➢ saves variables a and b to the file myfile.mat
    - ➢ myfile.mat file is saved in the current directory
    - ➢ Default working directory is
- Use **clear** to remove variables from environment
  - `» clear a b`
    - ➢ look at workspace, the variables a and b are gone
- Use **clc** to clear the command window
  - `» clc`
- Use **load** to load variable bindings into the environment
  - `» load myFile`
    - ➢ look at workspace, the variables a and b are back
- Can do the same for entire environment
  - `» save myenv; clear all; load myenv;`



Exercise: Variables

Ex2: Get and save the current date and time

- Create a variable **`start`** that saves the current date and time. Use the built-in function **`clock`**
- What is the size of **`start`**? Is it a row or column?
- What values in **`start`** signify? See **`help clock`**
- Convert the vector **`start`** to a string. Use the function **`datestr`** and name the new variable **`startString`**
- Save **`start`** and **`startString`** into a mat file named **`startTime`**



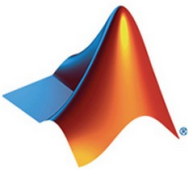
Exercise: Variables

Ex2: Get and save the current date and time

- Create a variable **`start`** that saves the current date and time. Use the built-in function **`clock`**
- What is the size of **`start`**? Is it a row or column?
- What values in **`start`** signify? See **`help clock`**
- Convert the vector **`start`** to a string. Use the function **`datestr`** and name the new variable **`startString`**
- Save **`start`** and **`startString`** into a mat file named **`startTime`**

Soln:

```
» help clock
» start=clock;
» size(start)
» help datestr
» startString=datestr(start);
» save startTime start startString
```

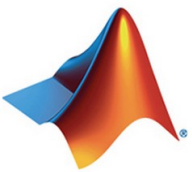


Exercise: Variables

Ex3: Read and display the current date and time in **helloWorld** script file

- In `helloWorld.m`, read in the variables you just saved using `load`
- Display the following text:

  I started learning MATLAB on *start date and time*
- Hint: use the `disp` command. Remember that strings are just vectors of characters so you can join two strings by making a row vector with the two strings as sub-vectors.



Exercise: Variables

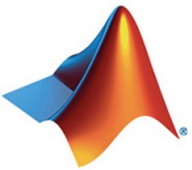
Ex3: Read and display the current date and time in **helloWorld** script file

- In `helloWorld.m`, read in the variables you just saved using `load`
- Display the following text:

  I started learning MATLAB on *start date and time*
- Hint: use the `disp` command. Remember that strings are just vectors of characters so you can join two strings by making a row vector with the two strings as sub-vectors.

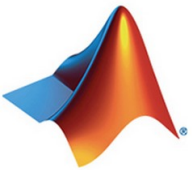
Soln:

```
load startTime
disp(['I started learning MATLAB on ' ...
  startString]);
```



Basic Scalar Operations

- Arithmetic operations (+,-,*,/)
  - » `7/45`
  - » `(1+i)*(2+i)`
  - » `1 / 0`
  - » `0 / 0`
- Exponentiation (^)
  - » `4^2`
  - » `(3+4*j)^2`
- Complicated expressions, use parentheses
  - » `((2+3)*3)^0.1`
- Multiplication is NOT implicit given parentheses
  - » `3(1+0.7) gives an error`

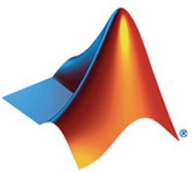


Built-in Functions

- MATLAB has an **enormous** library of built-in functions

- Call using parentheses – passing parameter to function

```
» sqrt(2)
» log(2), log10(0.23)
» cos(1.2), atan(-.8)
» exp(2+4*i)
» round(1.4), floor(3.3), ceil(4.23)
» abs(1+i);
```



Exercise: Scalars

Ex4: Assuming that the learning rate for MATLAB is exponential. Consider the following conditions to code your **helloWorld** script:

- Your learning time constant is 1.5 days. Calculate the number of seconds in 1.5 days and name this variable `tau`
- Assuming that this class lasts for 5 days. Calculate the number of seconds in 5 days and name this variable `endOfClass`

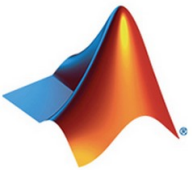
Suppose following equation describes your knowledge as a function of time t:

$$k = 1 - e^{-t/\tau}$$

- How well will you know MATLAB at `endOfClass`? Name this variable `knowledgeAtEnd`. (use `exp`)
- Using the value of `knowledgeAtEnd`, display the phrase:

At the end of 6.094, I will know X% of MATLAB

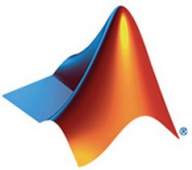
- Hint: to convert a number to a string, use `num2str`



Exercise: Scalars

Soln:

```
secPerDay=60*60*24;
tau=1.5*secPerDay;
endOfClass=5*secPerDay
knowledgeAtEnd=1-exp(-endOfClass/tau);
disp(['At the end of 6.094, I will know' ...
  num2str(knowledgeAtEnd*100) '% of MATLAB'])
```

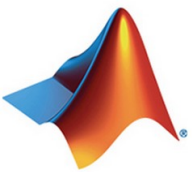


Transpose

- The transpose operators turns a column vector into a row vector and vice versa

```
» a = [1 2 3 4+i]
» transpose(a)
» a.'
» a'
```

- The o/p of `transpose()` function and `.'` is same.
- The `'` gives the Hermitian-transpose, i.e. transposes and conjugates all complex numbers
- For vectors of real numbers, `.'` and `'` give same result



Addition and Subtraction

- Addition and subtraction are element-wise; sizes must match (unless one is a scalar):

$$\begin{array}{r} \begin{bmatrix} 12 & 3 & 32 & -11 \end{bmatrix} \\ +\begin{bmatrix} 2 & 11 & -30 & 32 \end{bmatrix} \\ \hline =\begin{bmatrix} 14 & 14 & 2 & 21 \end{bmatrix} \end{array} \qquad \begin{bmatrix} 12 \\ 1 \\ -10 \\ 0 \end{bmatrix} - \begin{bmatrix} 3 \\ -1 \\ 13 \\ 33 \end{bmatrix} = \begin{bmatrix} 9 \\ 2 \\ -23 \\ -33 \end{bmatrix}$$

- Taking following vectors:

```
» row = [1, 2, 5.4, -6.6];
» column = [4;2;7;4]
```

- **Try** following statement:

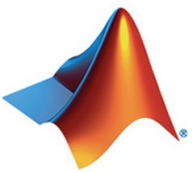
```
» c = row + column   → error
```

- Use the transpose to make sizes compatible

```
» c = row' + column
» c = row + column'
```

- Can sum up or multiply elements of vector

```
» s=sum(row);
» p=prod(row);
```



Element-Wise Functions

- All the functions that work on scalars also work on vectors
  - `» t = [1 2 3];`
  - `» f = exp(t);`
    - is the same as
  - `» f = [exp(1) exp(2) exp(3)];`

- If in doubt, check a function's help file to see if it handles vectors elementwise

- Operators (`* / ^`) have two modes of operation
  - element-wise
  - standard

Operators: element-wise

- To do element-wise operations, use the dot: `.` (`.*`, `./`, `.^`). BOTH dimensions must match (unless one is scalar)!
  - » `a=[1 2 3];b=[4;2;1];`
  - » `a.*b, a./b, a.^b` → **all errors**
  - » `a.*b', a./b', a.^(b')` → **all valid**

$$\begin{bmatrix}1 & 2 & 3\end{bmatrix}.*\begin{bmatrix}4\\2\\1\end{bmatrix} = ERROR$$

$$\begin{bmatrix}1\\2\\3\end{bmatrix}.*\begin{bmatrix}4\\2\\1\end{bmatrix}=\begin{bmatrix}4\\4\\3\end{bmatrix}$$

$$3\times 1.*3\times 1 = 3\times 1$$

$$\begin{bmatrix}1 & 1 & 1\\2 & 2 & 2\\3 & 3 & 3\end{bmatrix}.*\begin{bmatrix}1 & 2 & 3\\1 & 2 & 3\\1 & 2 & 3\end{bmatrix}=\begin{bmatrix}1 & 2 & 3\\2 & 4 & 6\\3 & 6 & 9\end{bmatrix}$$

$$3\times 3.*3\times 3 = 3\times 3$$

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}.^2=\begin{bmatrix}1^2 & 2^2\\3^2 & 4^2\end{bmatrix}$$

Can be any dimension

Operators: standard

- Multiplication can be done in standard way using (*) without dot.
- Standard exponentiation (^) can only be done on square matrices and scalars
- Left and right division (/ \) is same as multiplying by inverse
  - Right (/) division operator: A/B (equivalent to A*inv(B))
  - Left (\) division operator: A\B (equivalent to inv(A)*B)

$$\begin{bmatrix}1 & 2 & 3\end{bmatrix} * \begin{bmatrix}4\\2\\1\end{bmatrix} = 11$$

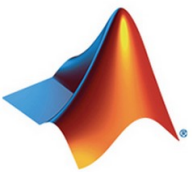
$$1\times 3 * 3\times 1 = 1\times 1$$

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \wedge 2 = \begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} * \begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}$$

Must be square to do powers

$$\begin{bmatrix}1 & 1 & 1\\2 & 2 & 2\\3 & 3 & 3\end{bmatrix} * \begin{bmatrix}1 & 2 & 3\\1 & 2 & 3\\1 & 2 & 3\end{bmatrix} = \begin{bmatrix}3 & 6 & 9\\6 & 12 & 18\\9 & 18 & 27\end{bmatrix}$$

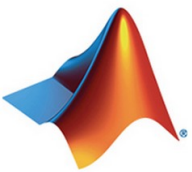
$$3\times 3 * 3\times 3 = 3\times 3$$



Exercise: Vector Operations

Ex5: In **helloWorld** script, calculate how many seconds elapsed since the start of this class?

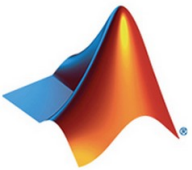
- In helloWorld.m, make variables called **`secPerMin`**, **`secPerHour`**, **`secPerDay`**, **`secPerMonth`** (assume 30.5 days per month), and **`secPerYear`** (12 months in year), which have the number of seconds in each time period.
- Assemble a row vector called **`secondConversion`** that has elements in this order: **`secPerYear`**, **`secPerMonth`**, **`secPerDay`**, **`secPerHour`**, **`secPerMinute`**, **`1`**.
- Make a **`currentTime`** vector by using **`clock`**
- Compute **`elapsedTime`** by subtracting **`currentTime`** from **`start`**
- Compute **`t`** (the elapsed time in seconds) by taking the dot product of **`secondConversion`** and **`elapsedTime`** (transpose one of them to get the dimensions right)



Exercise: Vector Operations

Soln:

```
secPerMin=60;
secPerHour=60*secPerMin;
secPerDay=24*secPerHour;
secPerMonth=30.5*secPerDay;
secPerYear=12*secPerMonth;
secondConversion=[secPerYear secPerMonth ...
  secPerDay secPerHour secPerMin 1];
currentTime=clock;
elapsedTime=currentTime-start;
t=secondConversion*elapsedTime';
```



Exercise: Vector Operations

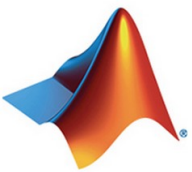
Ex6: In **helloWorld.m** script, also display the current state of your knowledge?

- In helloWorld.m, calculate `currentKnowledge` using the same relationship as before , and the `t` we just calculated:

$$k = 1 - e^{-t/\tau}$$

- Display the following text:

At this time, I know X% of MATLAB



Exercise: Vector Operations

Ex6: In **helloWorld.m** script, also display the current state of your knowledge?

- In helloWorld.m, calculate `currentKnowledge` using the same relationship as before , and the `t` we just calculated:

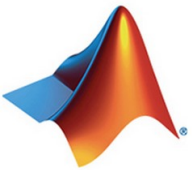
$$k = 1 - e^{-t/\tau}$$

- Display the following text:

At this time, I know X% of MATLAB

Soln:

```
currentKnowledge=1-exp(-t/tau);
disp(['At this time, I know ' ...
  num2str(currentKnowledge*100) '% of MATLAB']);
```



Automatic Initialization

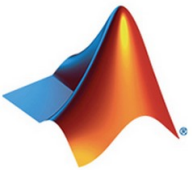
- Initialize a vector of **ones**, **zeros**, or **rand**om numbers
  - » `o=ones(1,10)`
    - ➢ row vector with 10 elements, all 1
  - » `z=zeros(23,1)`
    - ➢ column vector with 23 elements, all 0
  - » `r=rand(1,45)`
    - ➢ row vector with 45 elements (uniform [0,1])
  - » `n=nan(1,69)`
    - ➢ row vector of NaNs (useful for representing uninitialized variables)

The general function call is:

```
var=zeros(M,N);
```

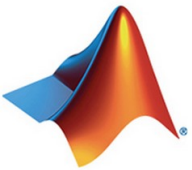
Number of rows

Number of columns



Automatic Initialization

- To initialize a linear vector of values use **linspace**
  - » `a=linspace(0,10,5)`
    - starts at 0, ends at 10 (inclusive), 5 values
- Can also use colon operator (**:**)
  - » `b=0:2:10`
    - starts at 0, increments by 2, and ends at or before 10
    - increment can be decimal or negative
  - » `c=1:5`
    - if increment isn't specified, default is 1
- To initialize logarithmically spaced values use **logspace**
    - similar to **linspace**, but see **help**

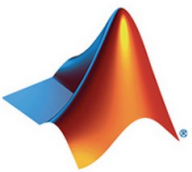


Exercise: Vector Functions

Ex7: In **helloWorld** script, calculate the learning trajectory.

- In helloWorld.m, make a linear time vector **`tVec`** that has 10,000 samples between 0 and **`endOfClass`**
- Calculate the value of your knowledge (call it **`knowledgeVec`**) at each of these time points using the same equation as before:

$$k = 1 - e^{-t/\tau}$$



Exercise: Vector Functions

Ex7: In **helloWorld** script, calculate the learning trajectory.

- In helloWorld.m, make a linear time vector **`tVec`** that has 10,000 samples between 0 and **`endOfClass`**
- Calculate the value of your knowledge (call it **`knowledgeVec`**) at each of these time points using the same equation as before:

$$k = 1 - e^{-t/\tau}$$

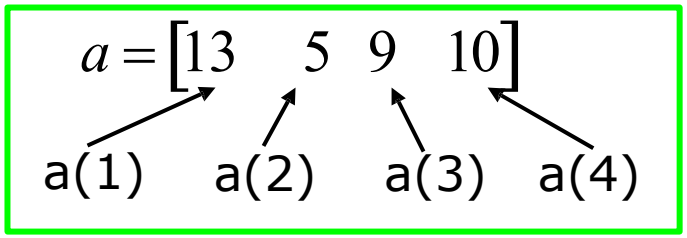
Soln:

```
tVec = linspace(0,endOfClass,10000);
knowledgeVec=1-exp(-tVec/tau);
```

Vector Indexing

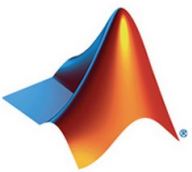
- MATLAB indexing starts with **1**, not **0**

- a(n) returns the n^{th} element



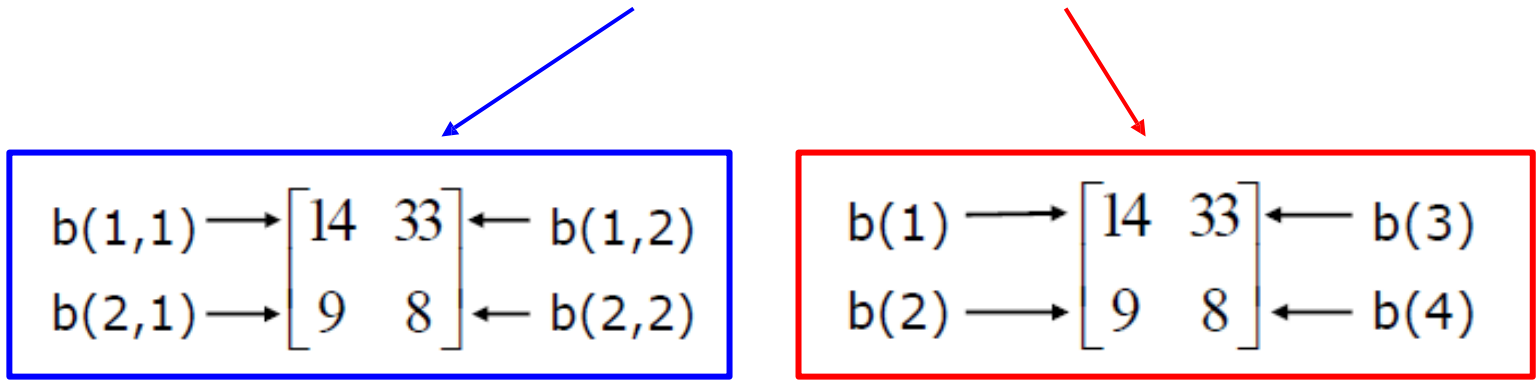
- The index argument can be a vector. In this case, each element is looked up individually, and returned as a vector of the same size as the index vector.

```
» x=[12 13 5 8];
» a=x(2:3);      ------------------------>  a=[13 5];
» b=x(1:end-1);  ------------------------>  b=[12 13 5];
```



Matrix Indexing

- Matrices can be indexed in two ways
  - using **subscripts** (row and column)
  - using linear **indices** (as if matrix is a vector)
- Matrix indexing: subscripts or linear indices



- Picking submatrices

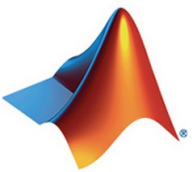
```
» A = rand(5) % shorthand for 5x5 matrix
» A(1:3,1:2) % specify contiguous submatrix
» A([1 5 3], [1 4]) % specify rows and columns
```

Advanced Indexing 1

- To select all the rows (or columns) of a specify column (or row) of a matrix, then use **:**

$$c = \begin{bmatrix} 12 & 5 \\ -2 & 13 \end{bmatrix}$$

```
» d=c(1,:);  ⟶  d=[12 5];
» e=c(:,2);  ⟶  e=[5;13];
» c(2,:)=[3 6];  %replaces second row of c
```



Advanced Indexing 2

- MATLAB contains functions to help you find desired values within a vector or matrix

```
» vec = [5 3 1 9 7]
```

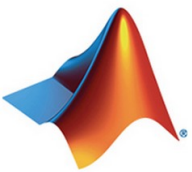
- To get the minimum value and its index:

```
» [minVal,minInd] = min(vec);
```

  - **max** works the same way

- To find any the indices of specific values or ranges

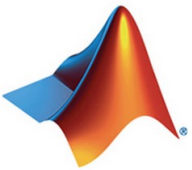
```
» ind = find(vec == 9);
» ind = find(vec > 2 & vec < 6);
```



Exercise: Indexing

Ex8: Code in **helloWorld** script to calculate the time to know 50% of MATLAB?

- In helloWorld.m, find the index where **`knowledgeVec`** is closest to 0.5. Mathematically, what you want is the index where the value of $|knowledgeVec - 0.5|$ is at a minimum (use **`abs`** and **`min`**).
- Next, use that index to look up the corresponding time in **`tVec`** and name this time **`halfTime`**.
- Finally, display the string: I will know half of MATLAB after X days
  Remember to convert **`halfTime`** to days by using **`secPerDay`**



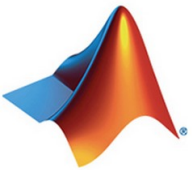
Exercise: Indexing

Ex8: Code in **helloWorld** script to calculate the time to know 50% of MATLAB?

- In helloWorld.m, find the index where **knowledgeVec** is closest to 0.5. Mathematically, what you want is the index where the value of $|knowledgeVec - 0.5|$ is at a minimum (use **abs** and **min**).
- Next, use that index to look up the corresponding time in **tVec** and name this time **halfTime**.
- Finally, display the string: I will know half of MATLAB after X days
  Remember to convert **halfTime** to days by using **secPerDay**

Soln:

```
[val,ind]=min(abs(knowledgeVec-0.5));
halfTime=tVec(ind);
disp(['I will know half of MATLAB after ' ...
    num2str(halfTime/secPerDay) ' days']);
```



Plotting

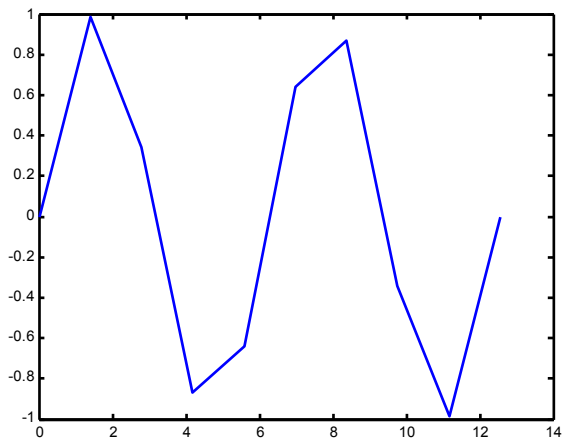
- Usually we want to plot y versus x
  ```
  » plot(x,y);
  ```
- Plot values against their index
  ```
  » plot(y);
  ```
- Example
  ```
  » x=linspace(0,4*pi,10);
  » y=sin(x);
  ```

MATLAB makes visualizing data fun and easy!

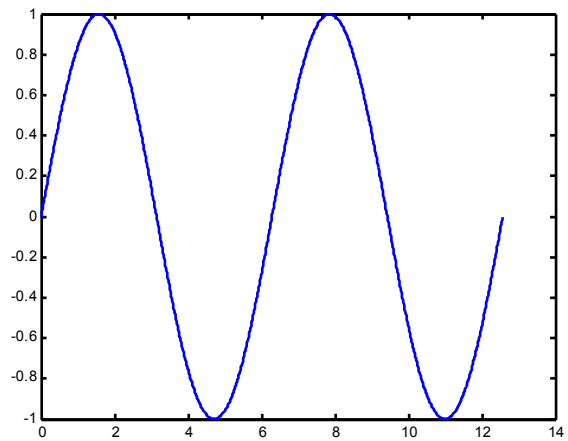
What does plot do?

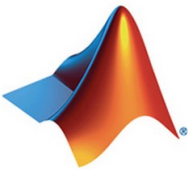
- To make plot of a function look smoother, evaluate at more points
  - `» x=linspace(0,4*pi,1000);`
  - `» plot(x,sin(x));`
- x and y vectors must be same size or else you'll get an error
  - `» plot([1 2], [1 2 3])`
    - ➢ error!!

10 x values:



1000 x values:

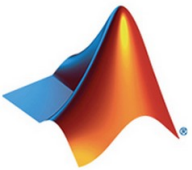




Exercise: Plotting

Ex9: Code to show the learning trajectory in **helloWorld** script!

- In **helloWorld.m**, open a new figure (use `figure`)
- Plot the knowledge trajectory using `tVec` and `knowledgeVec`. When plotting, convert `tVec` to days by using `secPerDay`



Exercise: Plotting

Ex9: Code to show the learning trajectory in **helloWorld** script!

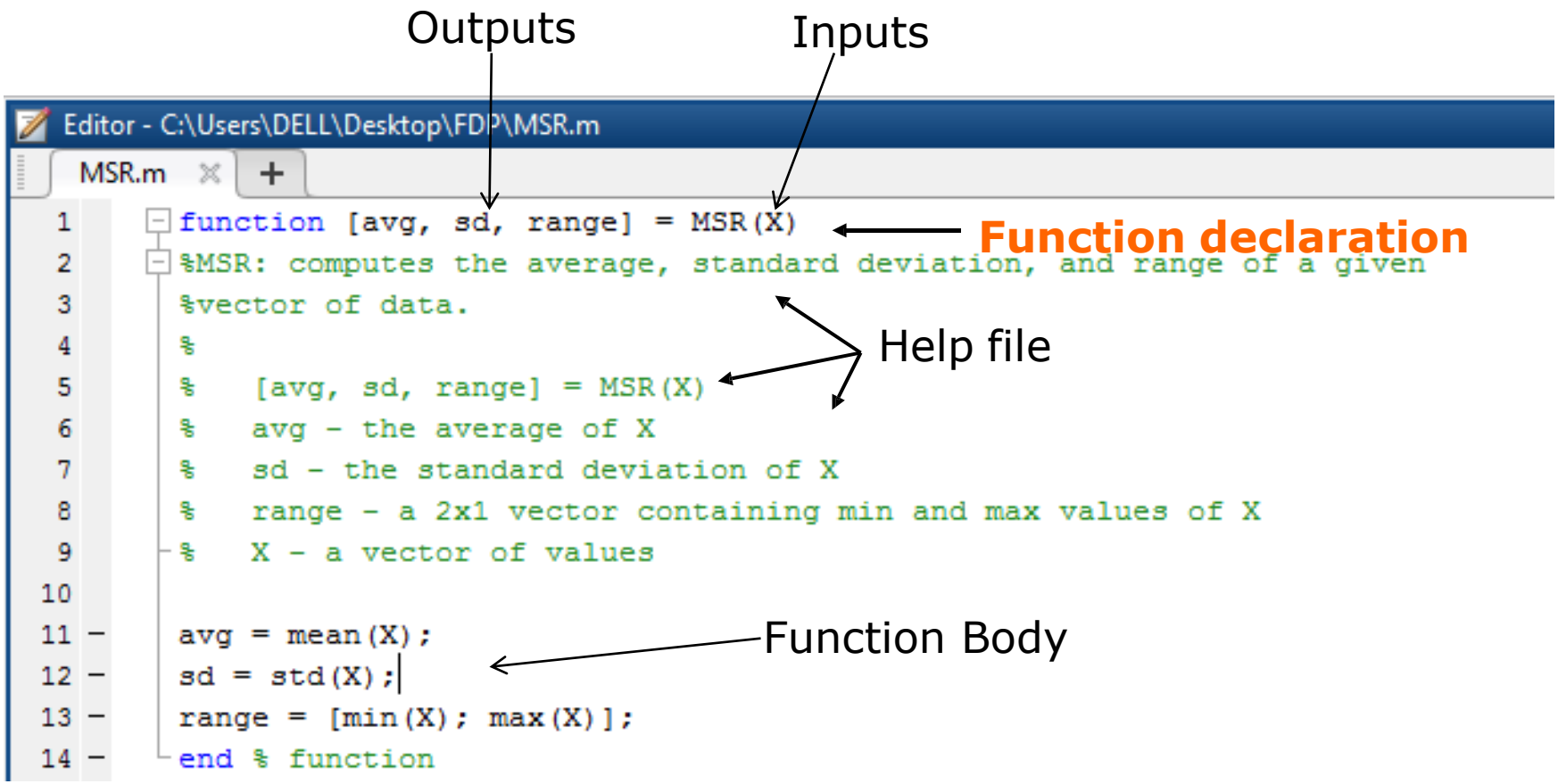
- In **helloWorld.m**, open a new figure (use `figure`)
- Plot the knowledge trajectory using `tVec` and `knowledgeVec`. When plotting, convert `tVec` to days by using `secPerDay`

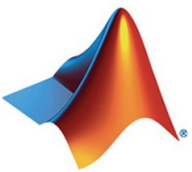
Soln:

```
figure
plot(tVec/secPerDay, knowledgeVec);
```

User-defined Functions

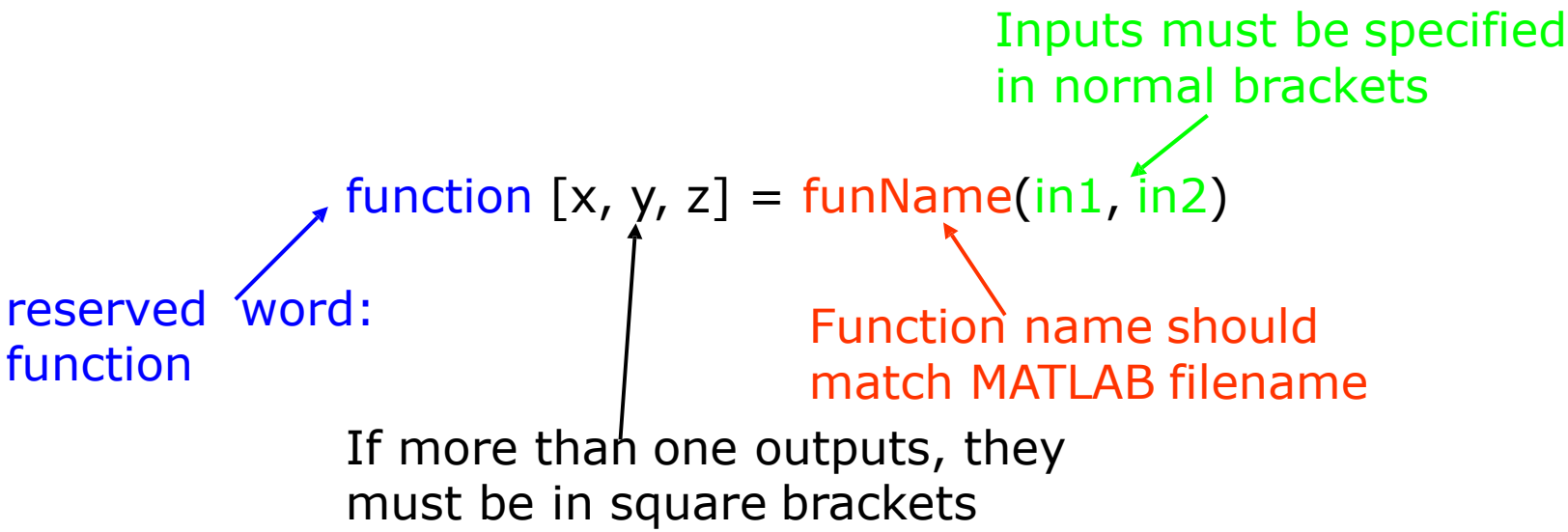
- Functions look exactly like scripts, but for **ONE** difference
  - Functions must have a function declaration





User-defined Functions

- Some comments about the function declaration

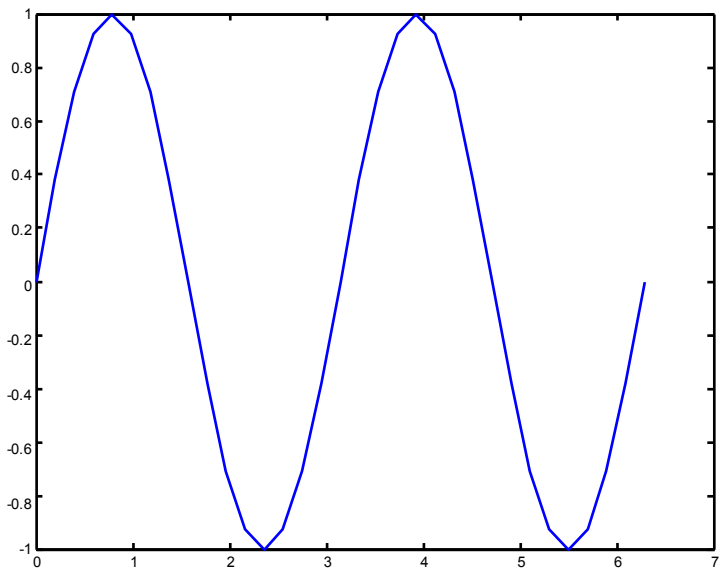


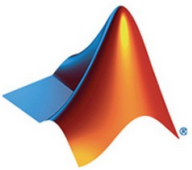
- No need for return: MATLAB 'returns' the variables whose names match those in the function declaration
- Variable scope: Any variables created within the function but not returned disappear after the function stops running

Functions: Excercise

Ex10:

- Write a function with the following declaration :
  `function plotSin(f1)`
- In the function, plot a sin wave with frequency f1, on the range [0,2π]: $\sin(f_1 x)$
- To get good sampling, use 16 points per period.





Functions: Excercise

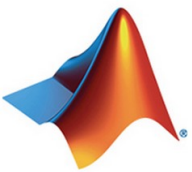
Ex10:

- Write a function with the following declaration :
  `function plotSin(f1)`
- In the function, plot a sin wave with frequency f1, on the range [0,2π]: $\sin(f_1 x)$
- To get good sampling, use 16 points per period.

Soln:

- In an MATLAB file saved as `plotSin.m`, write the following:

```
function plotSin(f1)

x=linspace(0,2*pi,f1*16+1);
figure
plot(x,sin(f1*x))
```

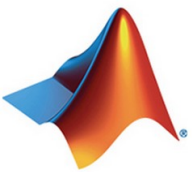


Relational Operators

- MATLAB uses *mostly* standard relational operators
  - equal ==
  - **not** equal ~=
  - greater than >
  - less than <
  - greater or equal >=
  - less or equal <=
- Logical operators

	elementwise	short-circuit (scalars)
And	&	&&
Or	|	||
Not	~	
Xor	xor	
All true	all	
Any true	any	

- Boolean values: zero is false, nonzero is true
- See **help .** for a detailed list of operators



if/else/elseif

- Basic flow-control, common to all languages
- MATLAB syntax is somewhat unique

IF

```
if cond
    commands
end
```

Conditional statement: evaluates to true or false

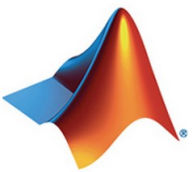
ELSE

```
if cond
    commands1
else
    commands2
end
```

ELSEIF

```
if cond1
    commands1
elseif cond2
    commands2
else
    commands3
end
```

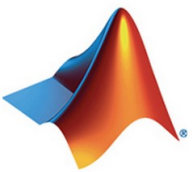
- No need for parentheses: command blocks are between reserved words



Exercise: Conditionals

Ex11:

- Write a function `plotSin2(f1,f2)` function that take two inputs
- If the number of input arguments is 1, execute the plot command you wrote before. Otherwise, display the line `'Two inputs were given'`
- Hint: the number of input arguments are in the built-in variable `nargin`



Exercise: Conditionals

Ex11:

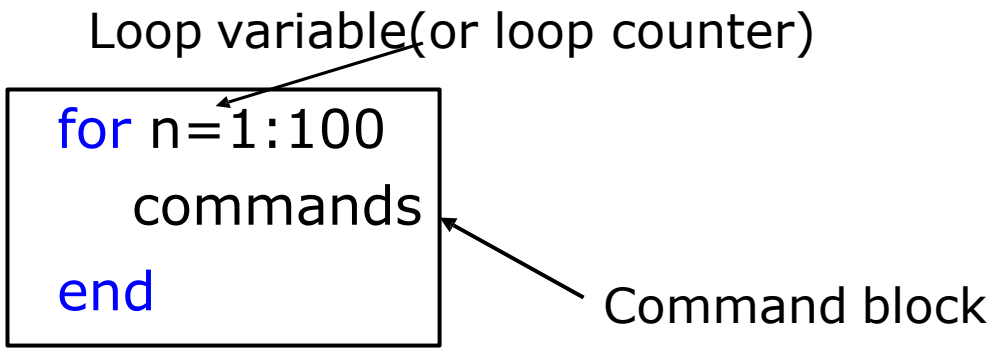
- Write a function `plotSin2(f1,f2)` function that take two inputs
- If the number of input arguments is 1, execute the plot command you wrote before. Otherwise, display the line `'Two inputs were given'`
- Hint: the number of input arguments are in the built-in variable `nargin`

Soln:

```
function plotSin2(f1,f2)
  x=linspace(0,2*pi,f1*16+1);
  figure
  if nargin == 1
       plot(x,sin(f1*x));
  elseif nargin == 2
       disp('Two inputs were given');
end
```

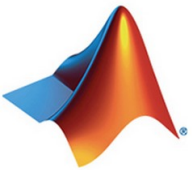
for

- **for** loops: use for a known number of iterations
- MATLAB syntax:



- The loop variable
  - Is defined as a vector
  - Is a scalar within the command block

- The command block
  - Anything between the **for** line and the **end**



while

- The while is like a more general for loop:
  - Don't need to know number of iterations

```
WHILE

while cond
    commands
end
```

- The command block will execute while the conditional expression is true
- Beware of infinite loops!

Exercise: Loops

Ex12: Write a script that will rotate a matrix by 90 degree

- Create a script **`Rotate90.m`** that defines a matrix *A* and stores in matrix *B* the 90 degree rotated A.

$$A = \begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix} \qquad B = \begin{bmatrix} 3 & 1 \\ 2 & 5 \end{bmatrix}$$

B is A, rotated by 90 degrees

- The **`Rotate90.m`** should calls a function **`rotate(a)`** and returns output **`b`** as output.

- `Hint:` Use **`jth`** index of A as **`ith`** index of B and define **`jth`** index of be as **`N-i+1`** where **`i`** corresponds to **`ith`** index of A

Exercise: Loops

Ex12: Write a script that will rotate a matrix by 90 degree

- Create a script **`Rotate90.m`** that defines a matrix *A* and stores in matrix *B* the 90 degree rotated A.

$$A = \begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix} \qquad B = \begin{bmatrix} 3 & 1 \\ 2 & 5 \end{bmatrix}$$

B is A, rotated by 90 degrees

- The **`Rotate90.m`** should calls a function **`rotate(a)`** and returns output **`b`** as output.

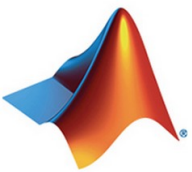
- Hint: Use **`jth`** index of A as **`ith`** index of B and define **`jth`** index of be as **`N-i+1`** where **`i`** corresponds to **`ith`** index of A

Soln:

Rotate90.m

```
A=[1 5; 3 2];
B=rotate(A);
B
```

```
function b= rotate(a)
    N=length(a);
    for i=1:N
     for j=1:N
      b(j,N-i+1)=a(i,j);
     end
    end
```



Revisiting find

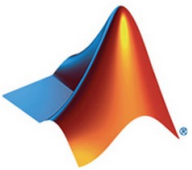
- **find** is a very important function
  - Returns indices of nonzero values
  - Can simplify code and help avoid loops

- Basic syntax:

  index=find(cond)

```
» x=rand(1,100);
» inds = find(x>0.4 & x<0.6);
```

- **inds** will contain the indices at which x has values between 0.4 and 0.6. This is what happens:
  - x>0.4 returns a vector with 1 where true and 0 where false
  - x<0.6 returns a similar vector
  - The & combines the two vectors using an **and**
  - The find returns the indices of the 1's



Example: Avoiding Loops

- Given x= sin(linspace(0,10*pi,100)), how many of the entries are positive?

Using a loop and if/else

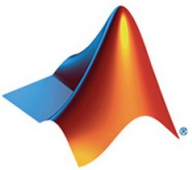
```
count=0;
for n=1:length(x)
    if x(n)>0
        count=count+1;
    end
end
```

Being more clever

```
count=length(find(x>0))
```

length(x)	Loop time	Find time
100	0.01	0
10,000	0.1	0
100,000	0.22	0
1,000,000	1.5	0.04

- Avoid loops!
- Built-in functions will make it faster to write and execute



Efficient Code

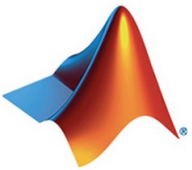
- Avoid loops
  - This is referred to as vectorization
- Vectorized code is more efficient for MATLAB
- Use indexing and matrix operations to avoid loops
- For example, to sum up every two consecutive terms:

```
» a=rand(1,100);
» b=zeros(1,100);
» for n=1:100
»     if n==1
»         b(n)=a(n);
»     else
»         b(n)=a(n-1)+a(n);
»     end
» end
```

  - Slow and complicated

```
» a=rand(1,100);
» b=[0 a(1:end-1)]+a;
```

  - Efficient and clean.



Advanced Data Structures

- We have used 2D matrices
  - Can have n-dimensions
  - Every element must be the same type (ex. integers, doubles, characters…)
  - Matrices are space-efficient and convenient for calculation
  - Large matrices with many zeros can be made sparse:

```
» a=zeros(100); a(1,3)=10;a(21,5)=pi; b=sparse(a);
```

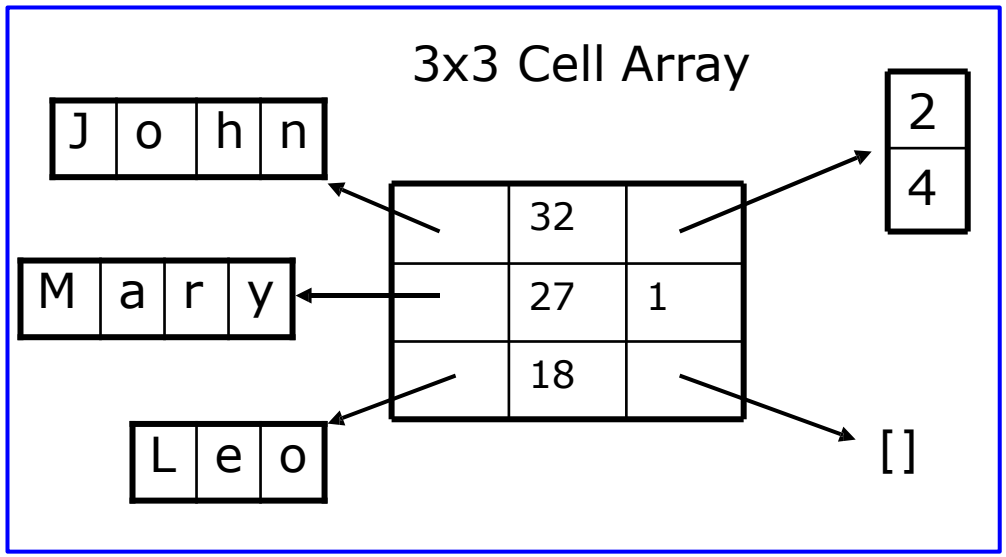
- Sometimes, more complex data structures are more appropriate
  - **Cell array**: it's like an array, but elements don't have to be the same type
  - **Structs**: can bundle variable names and values into one structure
    - Like object oriented programming in MATLAB

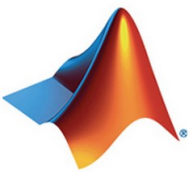
Cells: organization

- A cell is just like a matrix, but each field (or cell) can contain anything (even other matrices):

3x3 Matrix

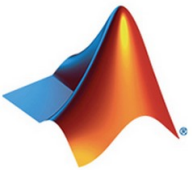
1.2	-3	5.5
-2.4	15	-10
7.8	-1.1	4





Cells: initialization

- To initialize a cell, specify the size
  - » `a=cell(3,10);`
    - ➢ a will be a cell with 3 rows and 10 columns
- or do it manually, with curly braces {}
  - » `c={'hello world',[1 5 6 2],rand(3,2)};`
    - ➢ c is a cell with 1 row and 3 columns
- Each element of a cell can be anything
- To access a cell element, use curly braces {}
  - » `a{1,1}=[1 3 4 -10];`
  - » `a{2,1}='hello world 2';`
  - » `a{1,2}=c{3};`
- Widely used to store *string.*



Structs

- Structs allow you to name and bundle relevant variables
  - Like C-structs, which are objects with fields
- To initialize an empty struct:

```
» s=struct([]);
```

  - size(s) will be 1x1
  - initialization is optional but is recommended when using large structs
- To add fields

```
» s.name = 'Jack Bauer';
» s.scores = [95 98 67];
» s.year = 'G3';
```

  - Fields can be anything: matrix, cell, even struct
  - Widely Used for keeping variables together
- To access struct fields, give name of the field as:

```
» score=s.year;
```

!!! TRY !!!

- For more information, see **doc struct**

Struct Arrays

- To initialize a struct array, give field, values pairs

```
» people=struct('name',{'John','Mary','Leo'},...
  'age',{32,27,18},'childAge',{[2;4],1,[]});
```

  - size(people)=1x3
  - every struct must have the same size

```
» person=people(2);
```

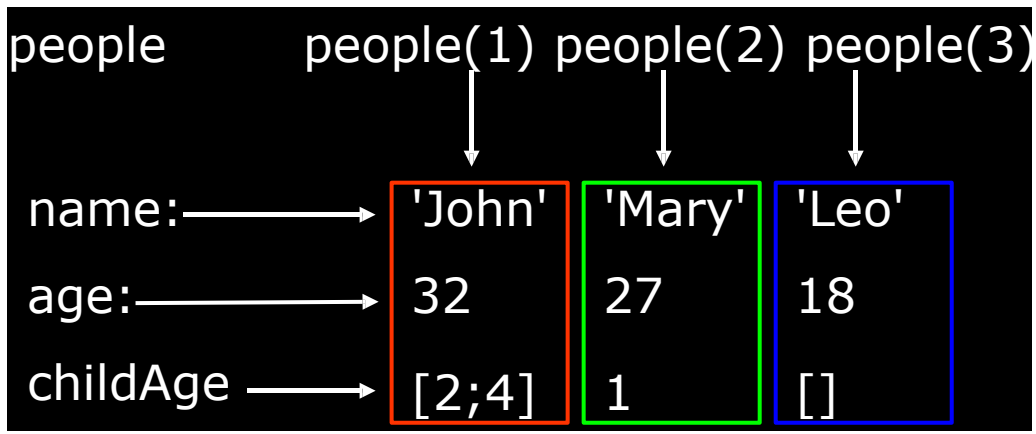
  - person is now a struct with fields name, age, children
  - the values of the fields are the values of the second index

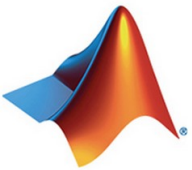
```
» person.name
```

  - returns 'Mary'

```
» people(1).age
```

  - returns 32





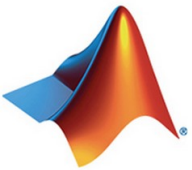
Exercise: Struct Arrays

Ex13:

Initializes a structure array ‘s’ with four fields (s1, s2, s3, s4) and values as following:

`s1` -> matrix of 2x3 `random values`
`s2` -> cell array of two string values `'a'` and `'b'`
`s3` -> cell array of two values `12` and `14.2`
`s4` -> a string `'first'`

- Display the `size` of s.
- Display the `values` of each field of each struct array.



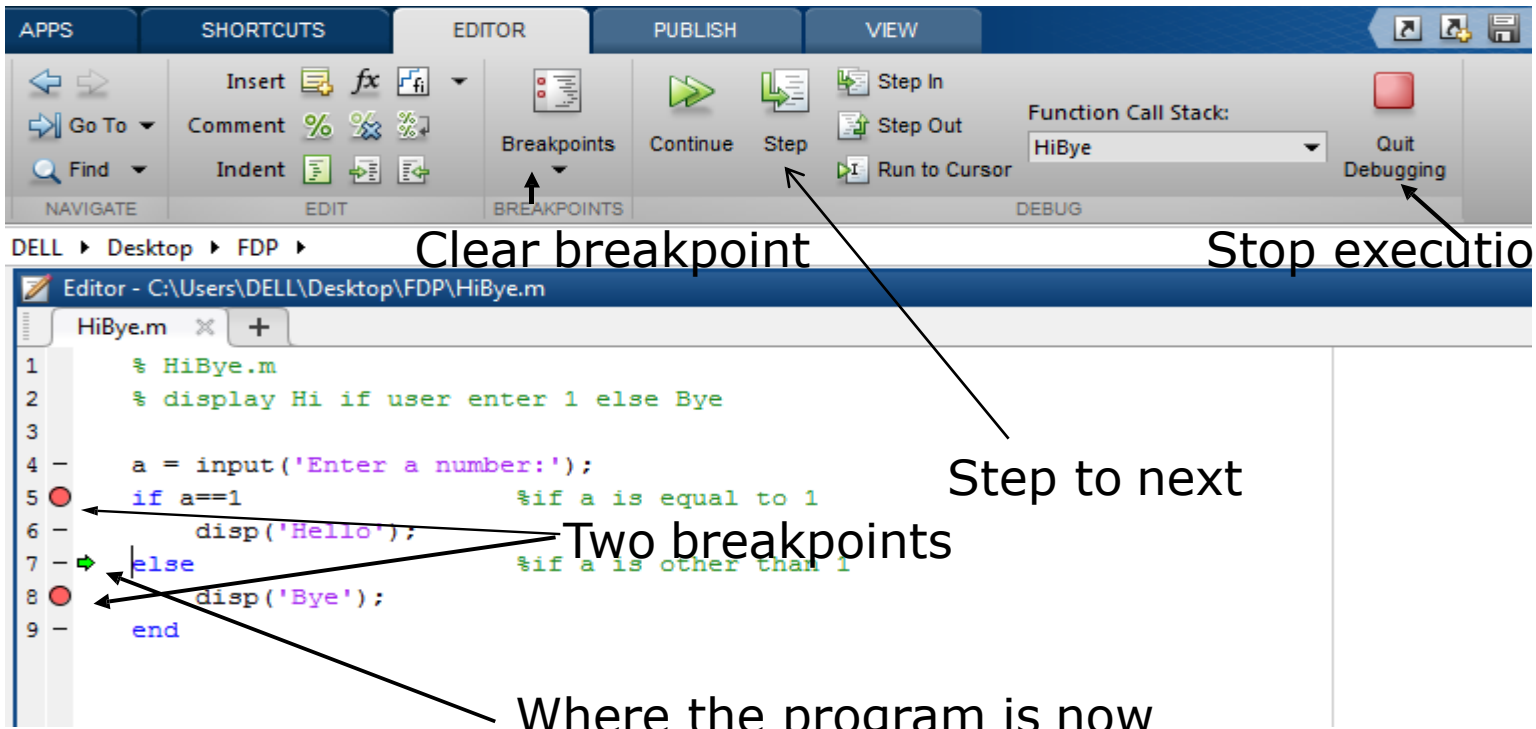
Exercise: Struct Arrays

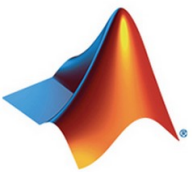
Soln:

```
» field1 = 's1';  value1 = rand(2,3);
» field2 = 's2';  value2 = {'a', 'b'};
» field3 = 's3';  value3 = {12, 14.2};
» field4 = 's4';  value4 = {'first'};
» s = struct(field1,value1,field2,value2,field3,value3,field4,value4);

» size(s)
» s(1).s1
» s(1).s2
» s(1).s3
» s(1).s4

» s(2).s1
» s(2).s2
» s(2).s3
» s(2).s4
```

Debugging

- To use the debugger, set breakpoints
  - Click on (–) next to line numbers in MATLAB files
  - Each red dot that appears is a breakpoint
  - Run the program
  - The program pauses when it reaches a breakpoint
  - Use the command window to print variables
  - Use the debugging buttons to control debugger





Exercise: Debugging

Ex14: Write a **calfact** script that calls a function **factorial** which takes a number as argument and outputs the factorial of that number.

- Also analysis the values of variables when running the **calfact** script and **factorial** function in the workspace.

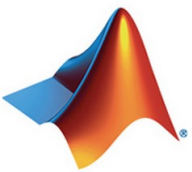
Soln:

calfact Script:

```
clear;
clc;
disp('Welcome to the program
of finding factorial');
num=str2num(input('Enter a
number:','s'));
outnum=factorial(num);
disp('The factorial is:');
outnum
```

factorial Function:

```
b=factorial(a)
    b=1;
while (a>0)
    b=b*a;
    a=a-1;
end
end
```



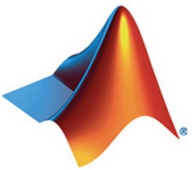
Performance Measures

- It can be useful to know how long your code takes to run
  - To predict how long a loop will take
  - To pinpoint inefficient code

- You can time operations using **tic**/**toc**:

```
» tic
» CommandBlock1
» a=toc;
» CommandBlock2
» b=toc;
```

  - tic resets the timer
  - Each toc returns the current value in seconds
  - Can have multiple tocs per tic



Importing Data

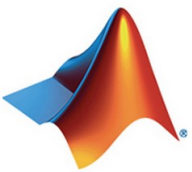
- MATLAB is a great environment for processing data. If you have a text file with some data:

```
jane joe jimmy
10 11 12
5 4 2
5 6 4
```

- To import data from files on your hard drive, use importdata
  - `» a=importdata('textFile.txt');`
    - a is a struct with `data`, `textdata`, and `colheaders` fields

```
a =
            data: [3x3 double]
        textdata: {'jane'  'joe'  'jimmy'}
      colheaders: {'jane'  'joe'  'jimmy'}
```

  - `» x=a.data;`
  - `» names=a.colheaders;`



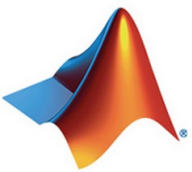
Importing Data

- With **importdata**, you can also specify delimiters. For example, for comma separated values, use:

```
» a=importdata('filename.txt',', ');
```

  - The second argument tells matlab that the tokens of interest are separated by commas or spaces

- **importdata** is very robust, but sometimes it can have trouble. To read files with more control, use **fscanf** (similar to C/Java) . See **help** or **doc** for information on how to use these functions

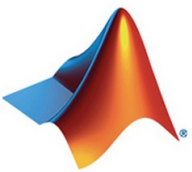


Writing Excel Files

- MATLAB contains specific functions for reading and writing Microsoft Excel files
- To write a matrix to an Excel file, use **xlswrite**

```
» xlswrite('randomNumbers',rand(10,4)); % we
  can also specify the sheet
```

- See **doc xlswrite** for more usage options



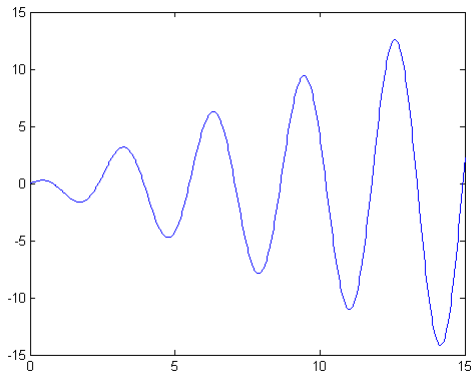
Reading Excel Files

- Reading excel files is equally easy
- To read from an Excel file, use **`xlsread`**
  - » **`X=xlsread('randomNumbers.xls');`**
    - ➢ Reads the first sheet
    - ➢ X contains the values
  - » **`[num,txt,raw]=xlsread('randomNumbers.xls',...`**
    `'mixedData');`
    - ➢ Reads the **`mixedData`** sheet
    - ➢ **`num`** contains numbers, **`txt`** contains strings, **`raw`** is the entire cell array containing everything

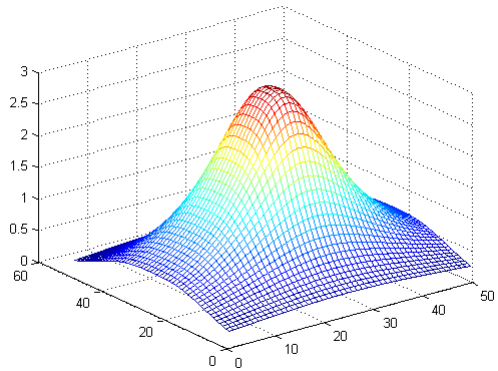
- See **`doc xlsread`** for even more fancy options

Plot

- MATLAB provides functions to create various types of plots.



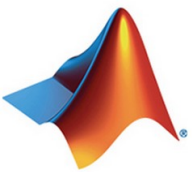
2D plot



3D plot

- To draw 2D-plot, use plot() function
- *plot()* generates dots at each (x,y) pair and then connects the dots with a line

```
» x=linspace(0,4*pi,10);
» y=sin(x);
» plot(x,y);
```



Plot

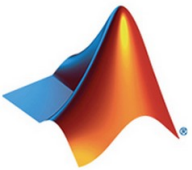
- Can change the line color, marker style, and line style by adding a string argument

plot(x,y,'line specifier')

```
» plot(x,y,'k.-');  !!!TRY!!!
```

color marker line-style

Line Specifier Style		Line Specifier Color		Marker Specifier Type	
Solid	-	red	r	plus sign	+
dotted	:	green	g	circle	o
dashed	--	blue	b	asterisk	*
dash-dot	-.	Cyan	c	point	.
		magenta	m	square	s
		yellow	y	diamond	d
		black	k		

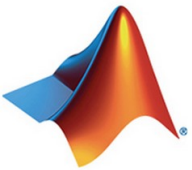


Exercise: Plot

Ex15:

- Draw the same 2D plot using *plot()* without connecting the dots.

- Hint: omit the line style argument



Exercise: Plot

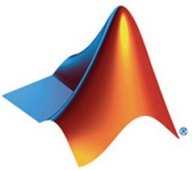
Ex15:

- Draw the same 2D plot using *plot()* without connecting the dots

- Hint: omit the line style argument

Soln:

```
» plot(x,y,'.')
```



Exercise: Plot

Ex16:

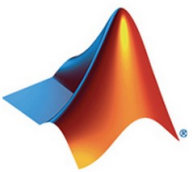
- Draw a 2D plot using *plot()* that shows two lines in the same plot
- Take following values for plotting:

  x = [1:10]

  y= 10*rand(1,10)

  z= 100*rand(1,10)

- Hint: Use **`hold on`** (holds the current plot) and **`hold on`** (returns to default mode) command
- Can we do it with single command!!!!!



Exercise: Plot

Ex16:

- Draw a 2D plot using *plot()* that shows two lines in the same plot
- Take following values for plotting:

  x = [1:10]

  y= 10*rand(1,10)

  z= 100*rand(1,10)
- Hint: Use **`hold on`** (holds the current plot) and **`hold on`** (returns to default mode) command

Soln:

```
» figure
» plot(x,y,'r-s');
» hold on
» plot(x,z,'b-.*');
» hold off
```

Yes, We can do with single command

```
» plot(x,y,'r-s',x,z,'b-.*')
```

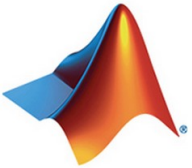


Figure Formatting

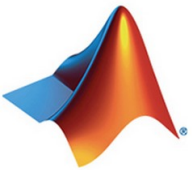
- **title(‘string’):**
  Adds the string as a title at the top of the plot.

- **xlabel(‘string’):**
  Adds the string as a label to the x-axis.

- **ylabel(‘string’):**
  Adds the string as a label to the y-axis.

- **legend(‘string1’, ‘string2’, ‘string3)**
  Creates a legend using the strings to label various curves (when several curves are in one plot).

- **axis([xmin xmax ymin ymax])**
  Sets the minimum and maximum limits of the x- and y-axes.

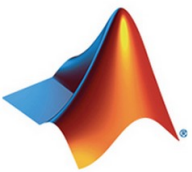


Exercise: Plot

Ex17:

- Format the previous 2D plot as follow.

  - Label the x-axis and y-axis as ‘x variable’ and ‘random variable’ respestively
  - Title the plot as ‘random v/s x’
  - Define legends as ‘y-function’ and ‘z-function’
  - Set the minimum and maximum limits of x- and y-axis as (0,10) and (0,100).



Exercise: Plot

Ex17:

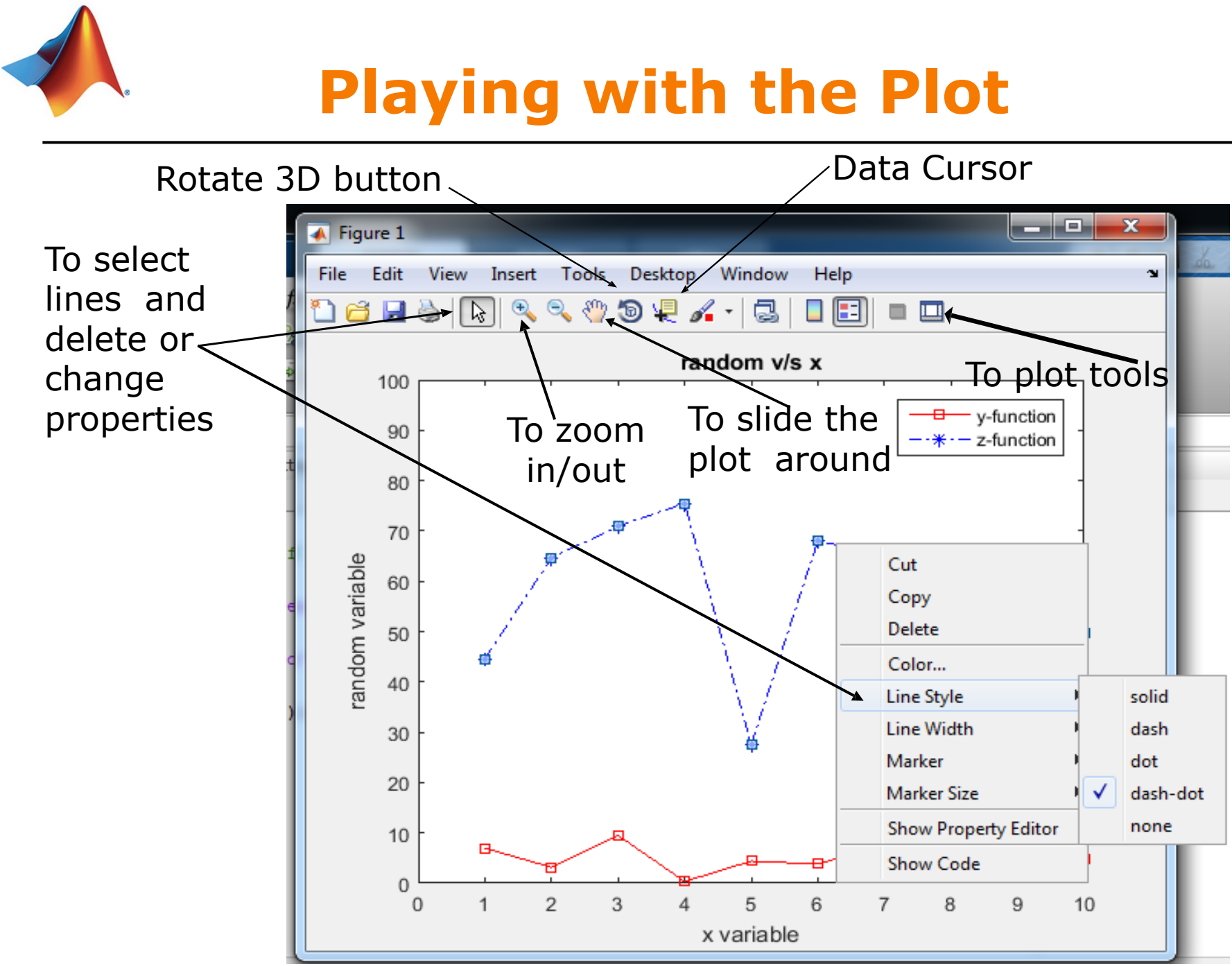
- Format the previous 2D plot as follow.

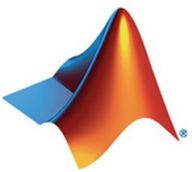
  - Label the x-axis and y-axis as ‘x variable’ and ‘random variable’ respestively
  - Title the plot as ‘random v/s x’
  - Define legends as ‘y-function’ and ‘z-function’
  - Set the minimum and maximum limits of x- and y-axis as (0,10) and (0,100).

Soln:

```
» plot(x,y,'r-s',x,z,'b-.*')
» xlabel('x values')
» ylabel('random variable')
» xlabel('x variable')
» ylabel('random variable')
» title('random v/s x')
» legend('y-function','z-function')
» axis([0 10 0 100])
```

Playing with the Plot

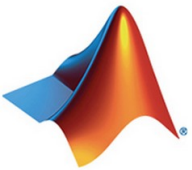




Exercise: Read, Write & Plot

Ex18:

- Write a **RWPExcel.m** script that creates a matrix **`randNum`** of 7x10 containing random values.
- Store these values in excel sheet **`randNumbers`**.
- Delete the **`randNum`** from workspace.
- Read the **`randNumbers`** excel and plots a graph having lines for corresponding row read from the excel.
- Use the figure GUI to:
  - Label the x-axis as **`numbers`** and y-axis as **`Random Numbers`**, title the graph as **`10_Random Numbers`**,
  - Show the legend for each color and change name of each color in legend,
  - Make line style as dash and show square markers on the line corresponds to data of row2,
  - Show the co-odinates of peak of line corresponding to 4th row,
  - Rotate to see the 3D view,
  - Save the file as randfigure.fig and randfigure.bmp,
  - Change the axis scale of y-axis to log.



Exercise: Read, Write & Plot

Soln:

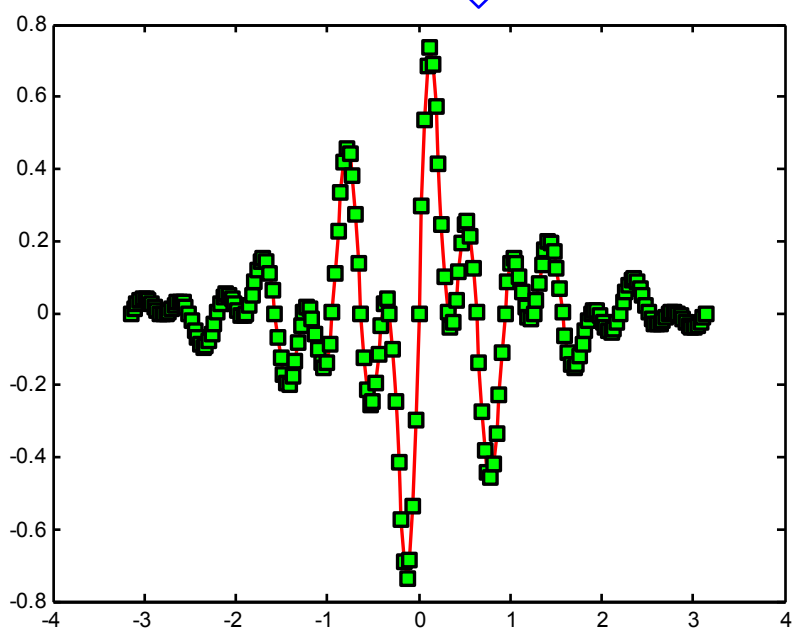
```
randNum=rand(7,10);
xlswrite('randNumbers',randNum);
clear randNum;
a=xlsread('randNumbers');
x=[1:10];
plot(x,a);
```

Line and Marker Options

- Everything on a line can be customized

```
» plot(x,y,'--s','LineWidth',2,...
          'Color', [1 0 0], ...
          'MarkerEdgeColor','k',...
          'MarkerFaceColor','g',...
          'MarkerSize',10)
```

You can set colors by using a vector of [R G B] values or a predefined color character like 'g', 'k', etc.



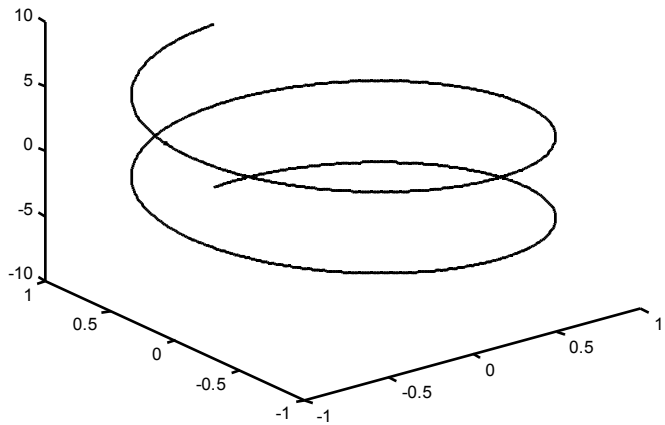
- See **doc** for a full list of properties that can be specified

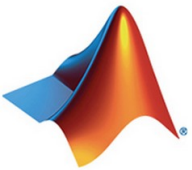
3D Line Plots

- We can plot in 3 dimensions just as easily as in 2

```
» time=0:0.001:4*pi;
» x=sin(time);
» y=cos(time);
» z=time;
» plot3(x,y,z,'k','LineWidth',2);
» zlabel('Time');
```

- Use tools on figure to rotate it





Multiple Plots in one Figure

- To have multiple axes in one figure
  - » `subplot(2,3,1)`
    - ➢ makes a figure with 2 rows and three columns of axes, and activates the first axis for plotting
    - ➢ each axis can have labels, a legend, and a title
  - » `subplot(2,3,4:6)`
    - ➢ activating a range of axes fuses them into one

- To close existing figures
  - » `close([1 3])`
    - ➢ closes figures 1 and 3
  - » `close all`
    - ➢ closes all figures (useful in scripts/functions)

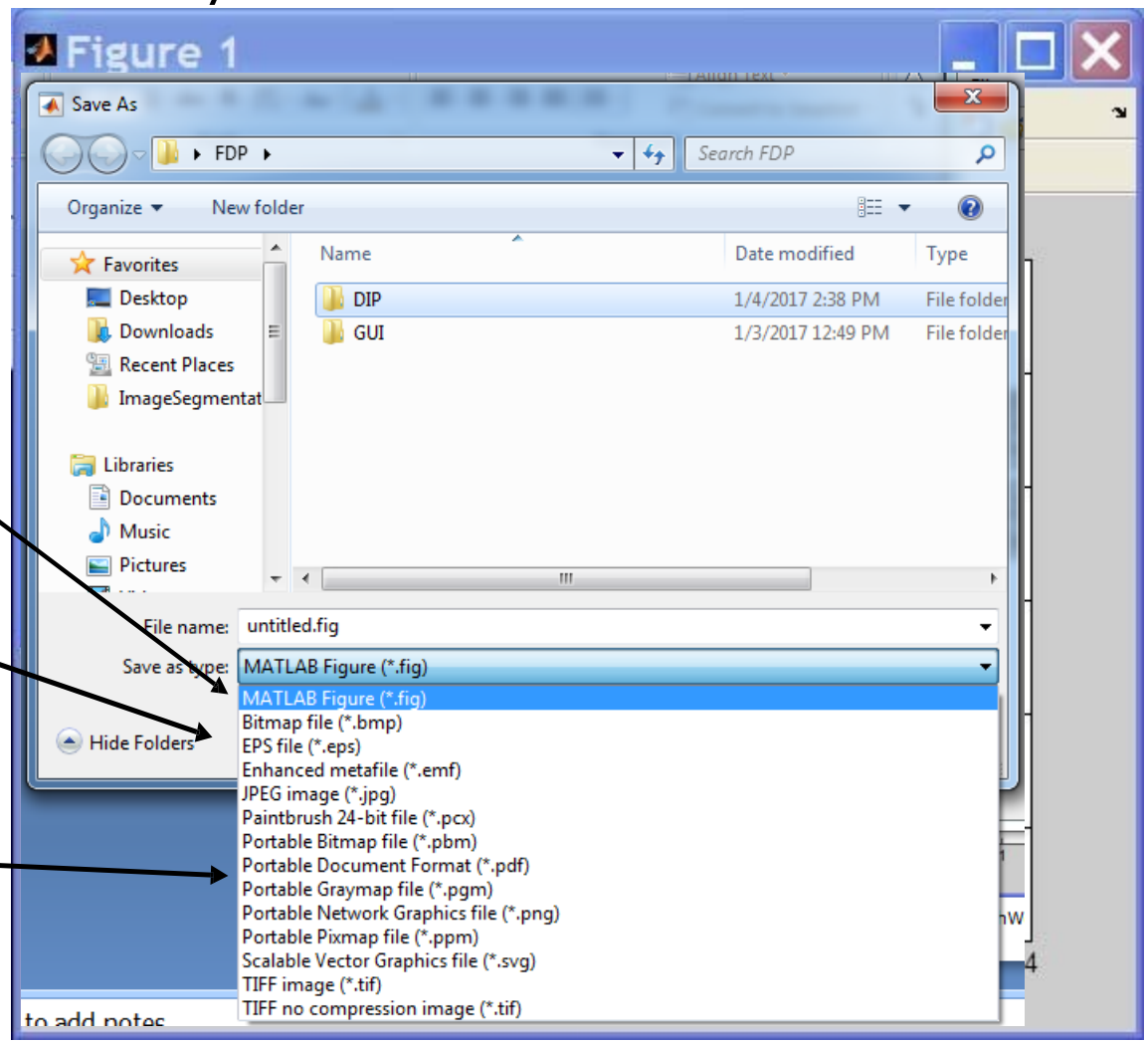
Saving Figures

- Figures can be saved in many formats. The common ones are:

.fig preserves all information

.bmp uncompressed image

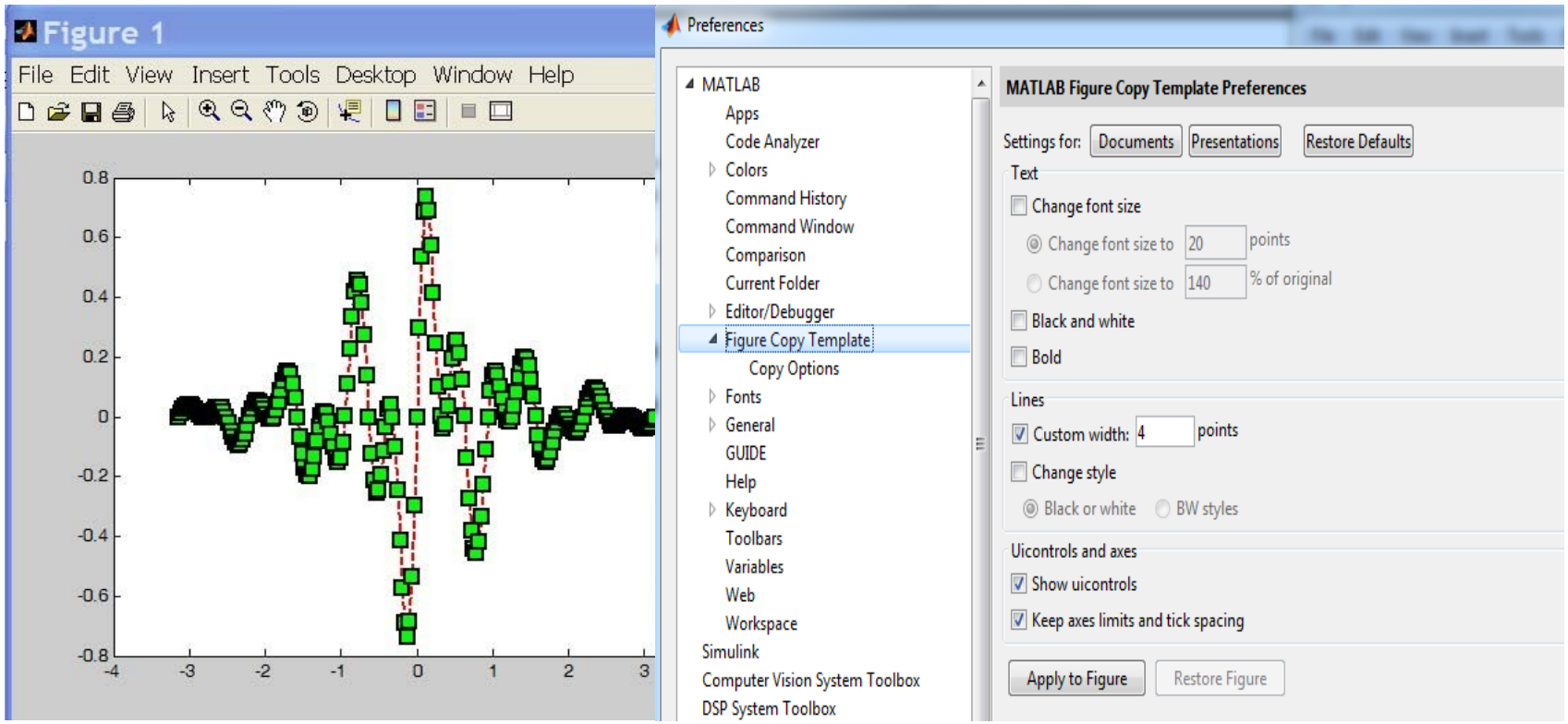
.pdf compressed image



Copy/Paste Figures

- Figure can be copied to clipboard as:

  Edit → copy figure to copy figure
- Paste into any document of interest (word, ppt, etc)

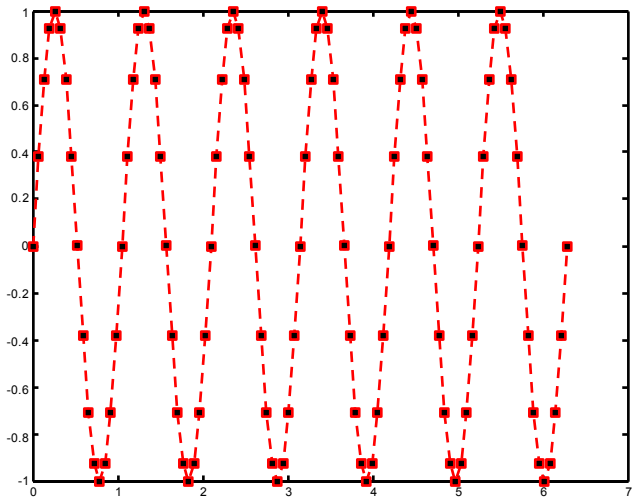


Exercise: Advanced Plotting

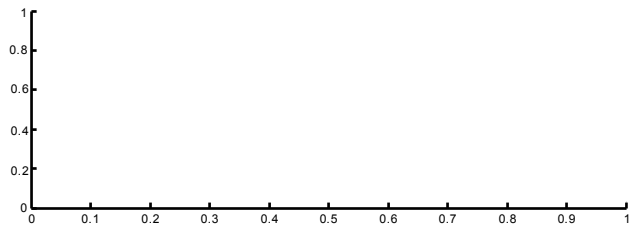
Ex19:

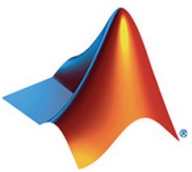
- Modify the plot command in your `plotSin2` function to use **squares** as markers and a **dashed red** line of **thickness** 2 as the line. Set the marker face color to be **black** (properties are `LineWidth`, `MarkerFaceColor`)
- If there are 2 inputs, open a new figure with 2 axes, one on top of the other (not side by side), and activate the top one (`subplot`)
- The output on `plotSin2(6)` and `plotSin2(6,2)` as:

`plotSin2(6)`



`plotSin2(6,2)`





Exercise: Advanced Plotting

Ex19:

- Modify the plot command in your `plotSin2` function to use **squares** as markers and a **dashed red** line of **thickness** 2 as the line. Set the marker face color to be **black** (properties are `LineWidth`, `MarkerFaceColor`)
- If there are 2 inputs, open a new figure with 2 axes, one on top of the other (not side by side), and activate the top one (`subplot`)

Soln:

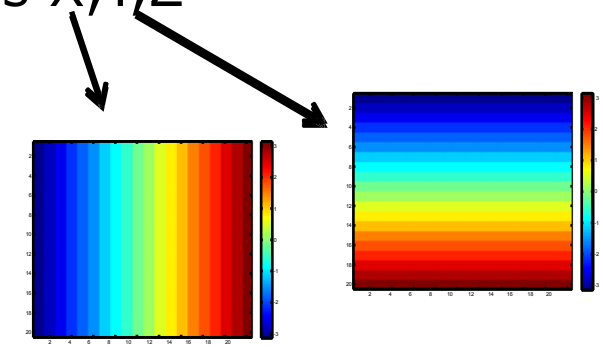
```
if nargin == 1
    plot(x,sin(f1*x),'rs--',...
     'LineWidth',2,'MarkerFaceColor','k');
elseif nargin == 2
    subplot(2,1,1);
end
```

Surface Plots

- It is more common to visualize *surfaces* in 3D
- Example:

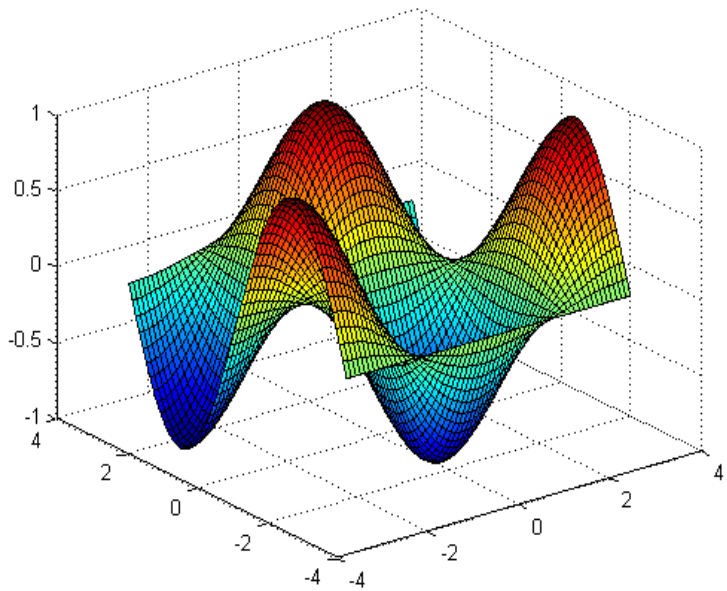
$$f(x,y) = sin(x)cos(y)$$
$$x \in [-\pi, \pi]; y \in [-\pi, \pi]$$

- **surf** puts vertices at specified points in space x,y,z, and connects all the vertices to make a surface
- The vertices can be denoted by matrices X,Y,Z
- How can we make these matrices
  - loop (DUMB)
  - built-in function: **meshgrid**



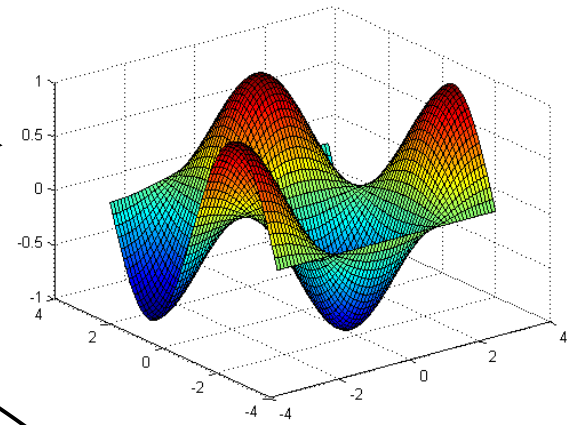
Exercise: surf

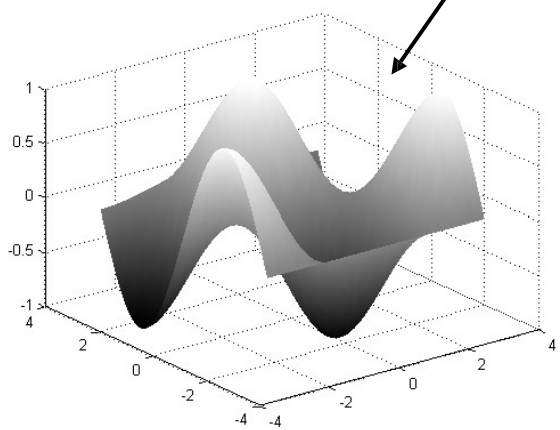
- Make the x and y vectors
  - `» x=-pi:0.1:pi;`
  - `» y=-pi:0.1:pi;`
- Use meshgrid to make matrices (this is the same as loop)
  - `» [X,Y]=meshgrid(x,y);`
- To get function values, evaluate the matrices
  - `» Z =sin(X).*cos(Y);`
- **Try** to plot the surface for X, Y, Z
  - `» surf(X,Y,Z)`
  - `» surf(x,y,Z);`

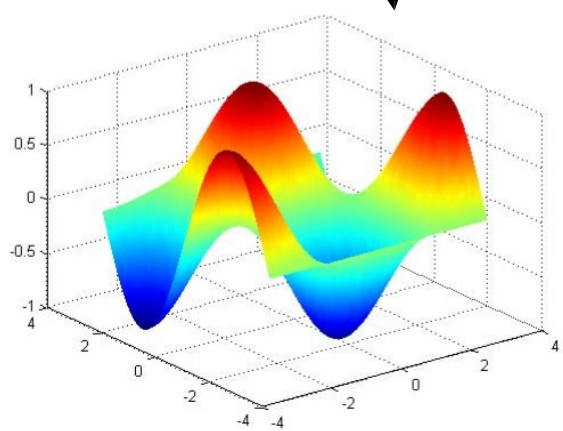


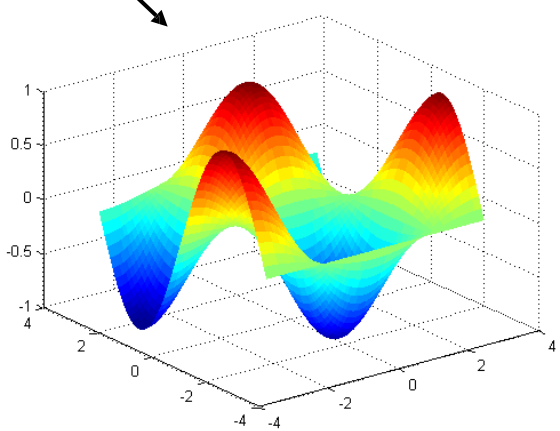
surf Options

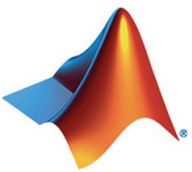
- See **help surf** for more options
- There are three types of surface shading
  - » `shading faceted`
  - » `shading flat`
  - » `shading interp`
- You can change colormaps
  - » `colormap(gray)`











Specialized Plotting Functions

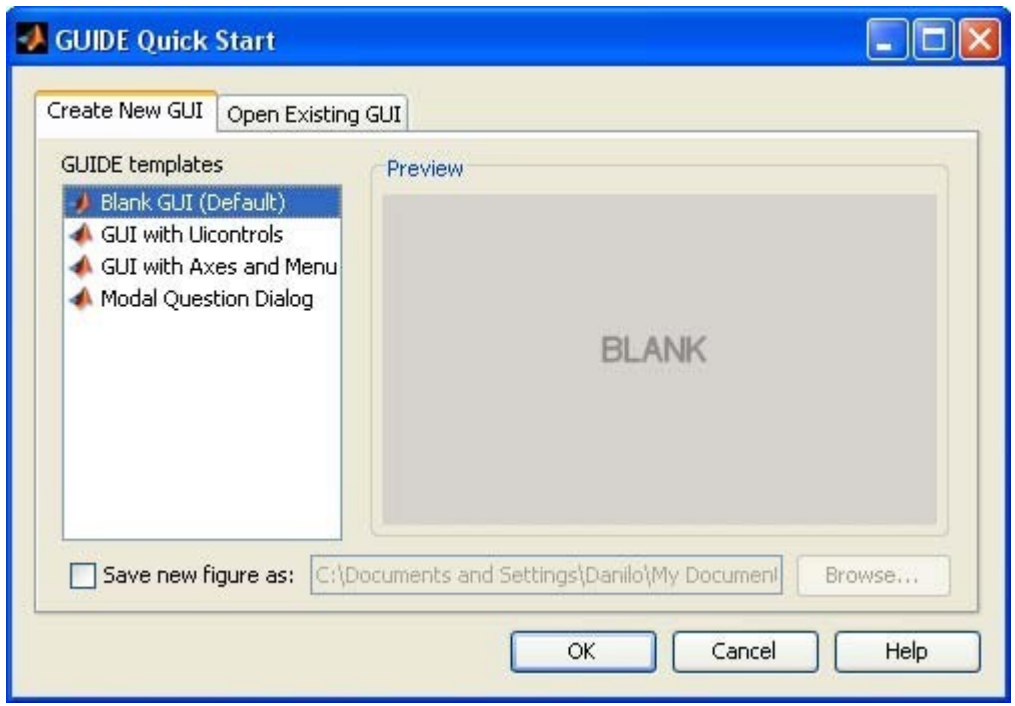
- MATLAB has a lot of specialized plotting functions
- **contour**- make surfaces two-dimensional
  ```
  » contour(X,Y,Z,'LineWidth',2)
  ```
- **polar**-to make polar plots
  ```
  » polar(0:0.01:2*pi,cos((0:0.01:2*pi)*2))
  ```
- **bar**-to make bar graphs
  ```
  » bar(1:10,rand(1,10));
  ```
- **stairs**-plot piecewise constant functions
  ```
  » stairs(1:10,rand(1,10));
  ```
- see help on these functions for syntax
- **doc specgraph** – for a complete list

Making GUIs

- It's really easy to make a graphical user interface in MATLAB
- To open the graphical user interface development environment, type **`guide`**

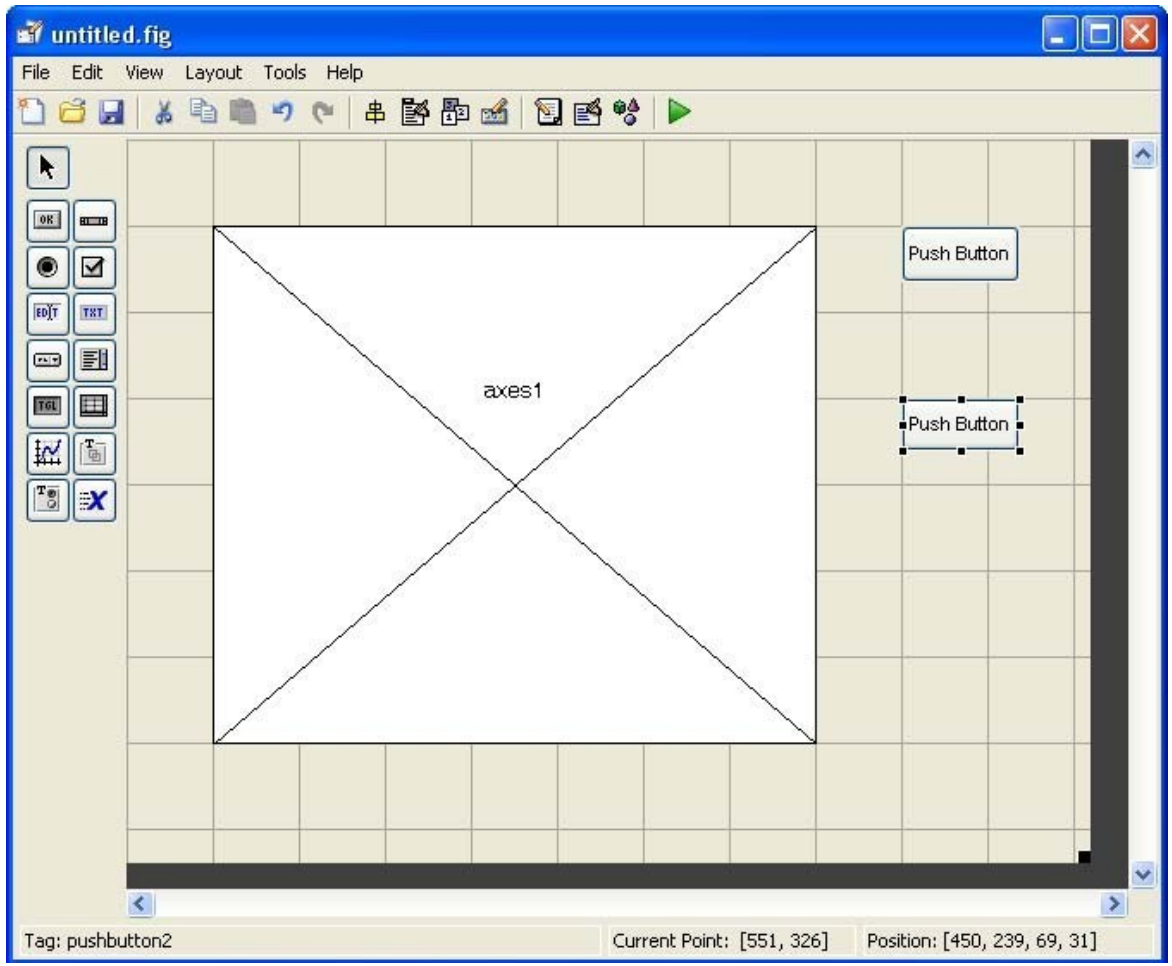
```
» guide
```

  - Select Blank GUI



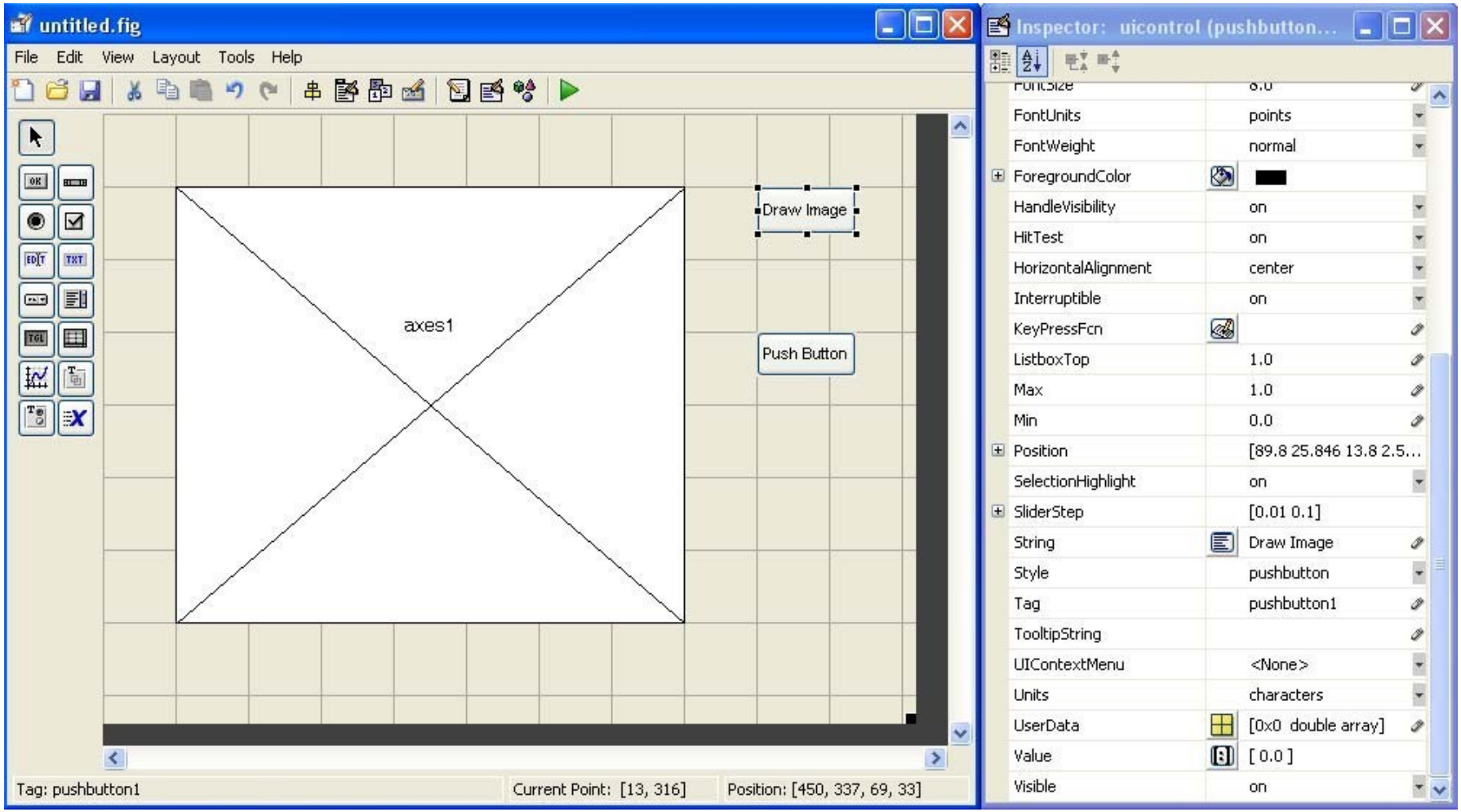
Draw the GUI

- Select objects from the left, and draw them where you want them



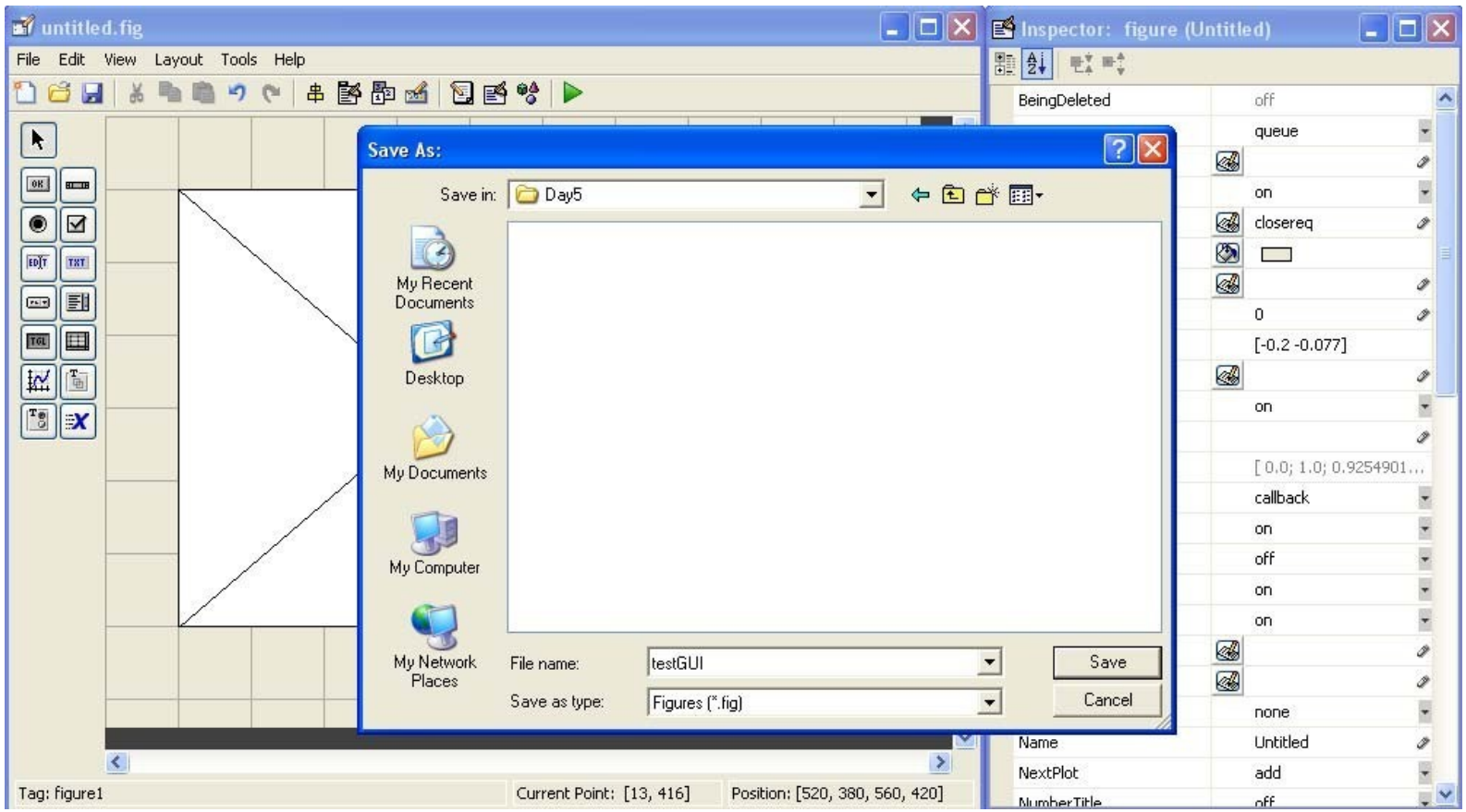
Change Object Settings

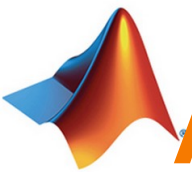
- Double-click on objects to open the Inspector. Here you can change all the object's properties.



Save the GUI

- When you have modified all the properties, you can save the GUI
- MATLAB saves the GUI as a .fig file, and generates an MATLAB file!





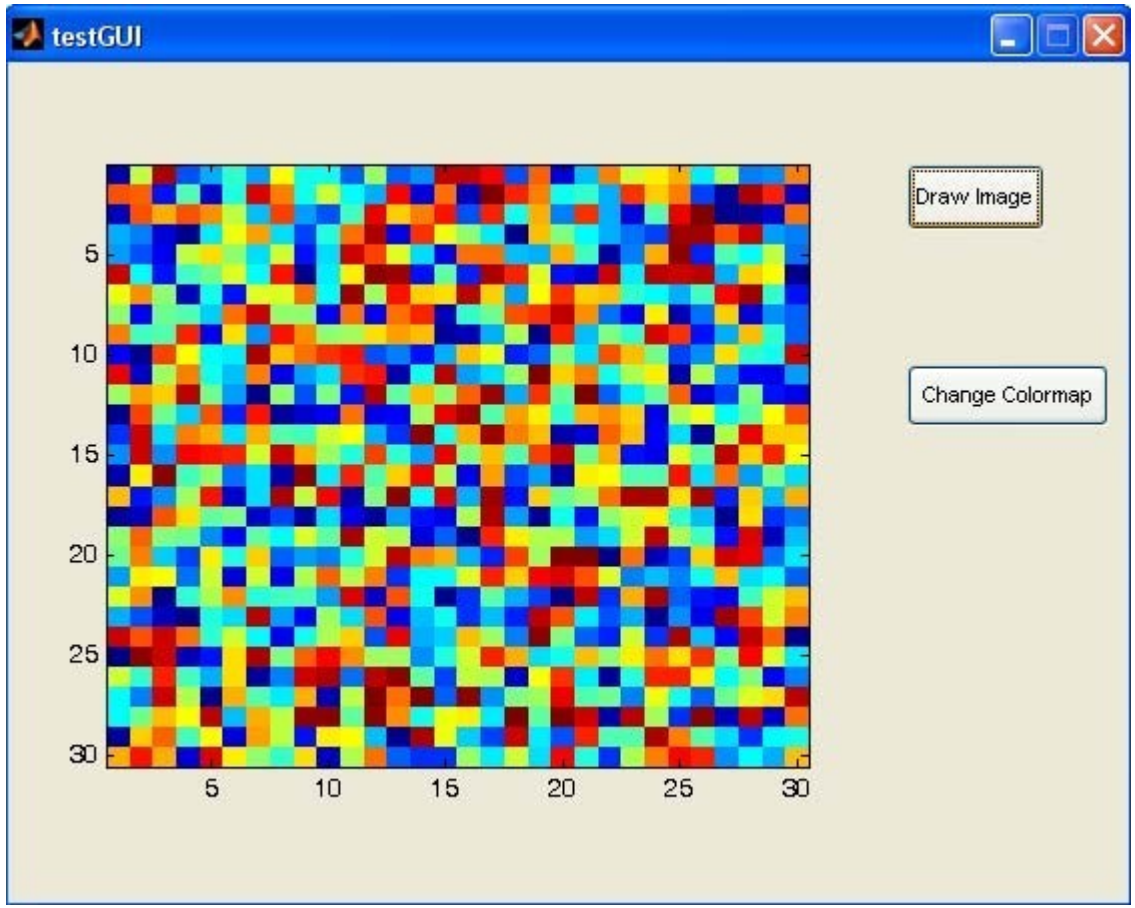
Add Functionality to MATLAB file

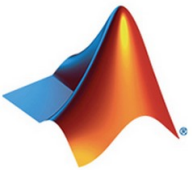
- To add functionality to your buttons, add commands to the 'Callback' functions in the MATLAB file. For example, when the user clicks the Draw Image button, the **`drawimage_Callback`** function will be called and executed

```
75
76      % --- Executes on button press in drawimage.
77      function drawimage_Callback(hObject, eventdata, handles)
78      % hObject    handle to drawimage (see GCBO)
79      % eventdata  reserved - to be defined in a future version of MATLAB
80      % handles    structure with handles and user data (see GUIDATA)
81
82
83      % --- Executes on button press in changeColormap.
84      function changeColormap_Callback(hObject, eventdata, handles)
85      % hObject    handle to changeColormap (see GCBO)
86      % eventdata  reserved - to be defined in a future version of MATLAB
87      % handles    structure with handles and user data (see GUIDATA)
88
```

Running the GUI

- To run the GUI, just type its name in the command window and the GUI will pop up. The debugger is really helpful for writing GUIs because it lets you see inside the GUI





Exercise: Login GUI

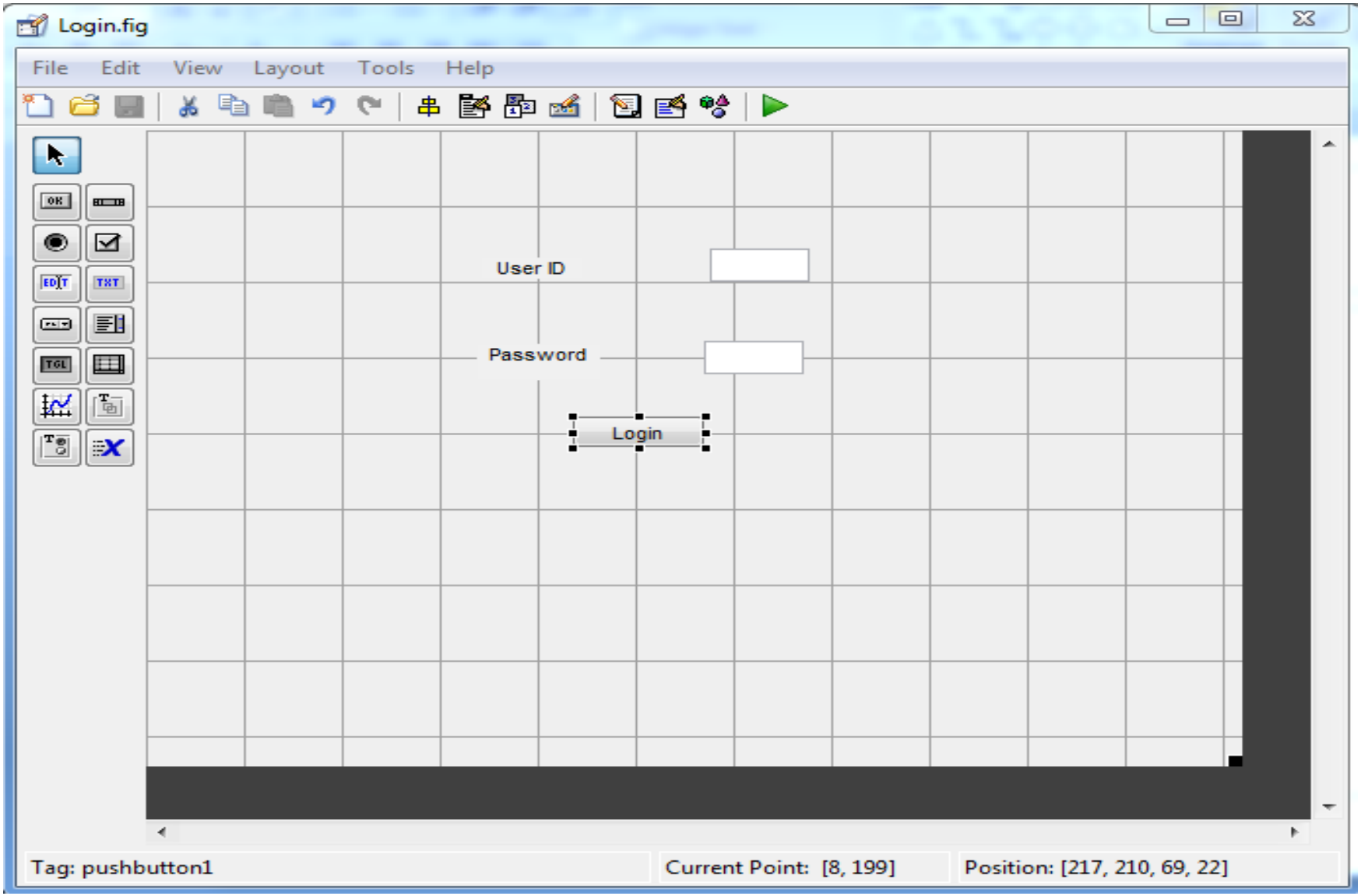
Try:

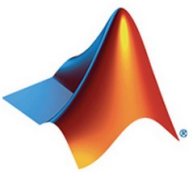
- Type *guide* at the command prompt

  ```
  » guide
  ```

- Select the **`Blank GUI`** from the pop-up window
- Select and edit the following components form the **`Toolbar`**
  - 2 Static Text
  - 2 Edit Text
  - 1 pushbutton
- Save the GUI as **`login.fig`**

Exercise: Login GUI





Exercise: Login GUI

Try:

- Add the following code in the function `pushbutton1_Callback:`

  - Get the data entry in password field

```
a=str2num(get(handles.edit2,'string'));
```

  - Check if entered password is correct

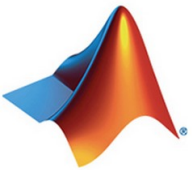
```
if a == 1234
```

  - Popup the respective message

```
msgbox('Login Successful');
```

  - If entered password is incorrect, display the message

```
else
    msgbox('Wrong Password');
end
```

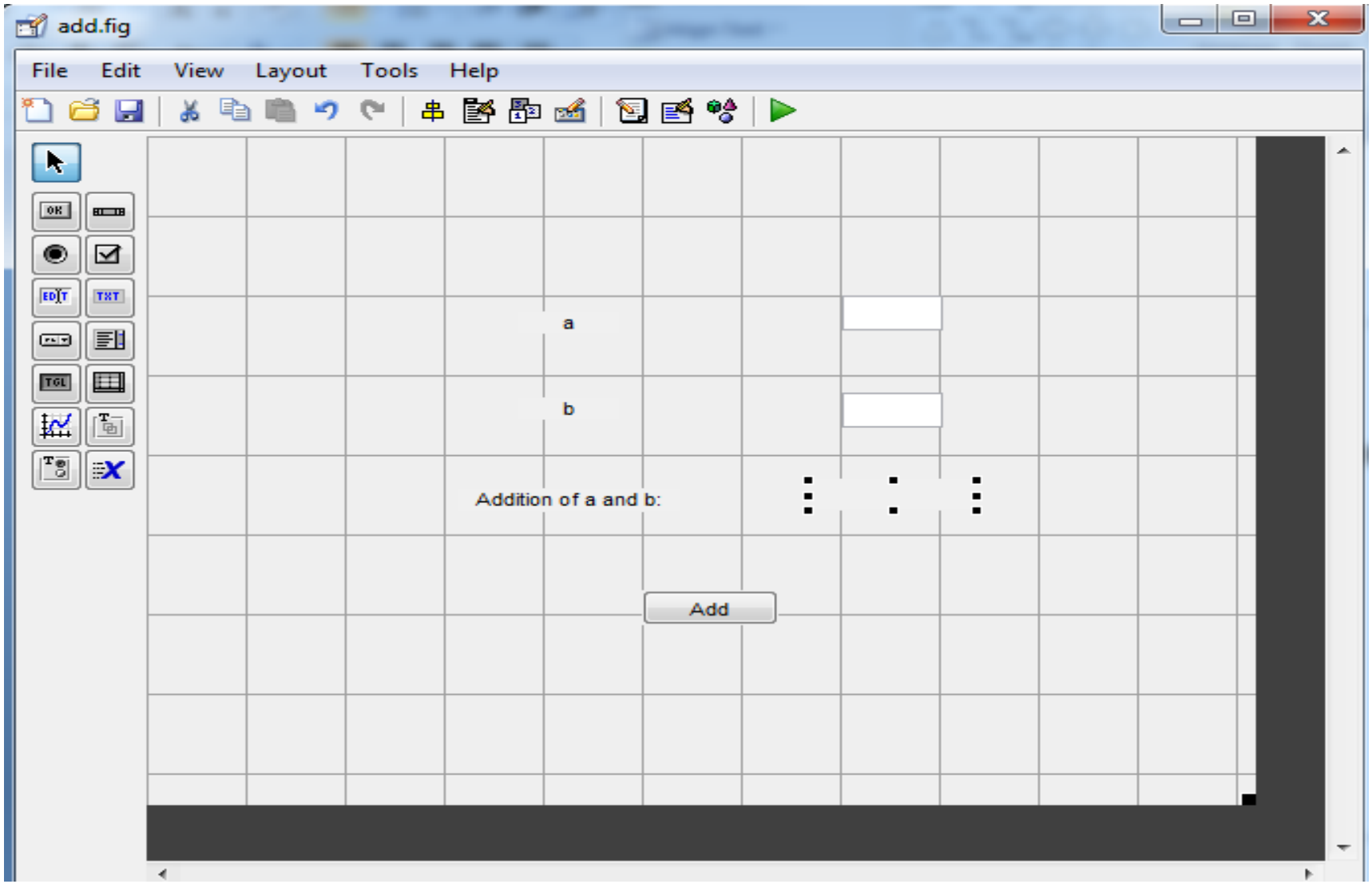


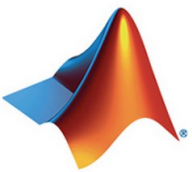
Exercise: GUI for Addition

Try:

- Open another GUI as:
  `File > New`
- Select the **`Blank GUI`** from the pop-up window
- Select and edit the following components form the **`Toolbar`**
  3 Static Text
  2 Edit Text
  1 pushbutton
- Save the GUI as **`add.fig`**

Exercise: GUI for Addition





Exercise: GUI for Addition

Try:

Add the following code in the function `pushbutton1_Callback:`

- Get the data entry in field1 and field2

```
var1=str2num(get(handles.edit1,'string'));
var2=str2num(get(handles.edit2,'string'));
```

- Add the two values

```
sum = var1+var2;
```

- Set the result of addition

```
set(handles.text5,'string',num2str(sum));
```

Exercise: GUI form Another GUI

Try:

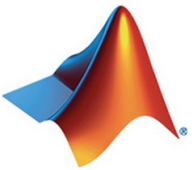
- Call *add* GUI inside *login* GUI

  - In the **login.m**, add following code in **`pushbutton1_Callback`** under the **`if`** condition:

    `add()`

Login.m | add.m

```
74
75
76      % --- Executes on button press in pushbutton1.
77      function pushbutton1_Callback(hObject, eventdata, handles)
78      % hObject    handle to pushbutton1 (see GCBO)
79      % eventdata  reserved - to be defined in a future version of MATLAB
80      % handles    structure with handles and user data (see GUIDATA)
81 -    a=str2num(get(handles.edit2,'string'));
82 -    if a == 1234
83 -        msgbox('Login Successful');
84 -        add();
85 -    else
86 -        msgbox('Wrong Password');
87 -    end
88
```



Database Tool

- Database Toolbox provides functions for exchanging data between relational databases and MATLAB.
- Two ways to interact with a database:
  - Use of SQL commands to read and write data, or
  - Use of Database Explorer interactively.
- Advantages include:

  - The toolbox supports both ODBC-compliant and JDBC-compliant databases, like Oracle, MySQL, Microsoft SQL and many others.

  - Multiple databases can be accessed simultaneously within a single MATLAB session and enables segmented import of large data sets.

Working with Database

- **Steps working with Database:**

  - Install the database and corresponding ODBC or JDBC driver. (List of databases and supported drivers seen at *doc database -> Database Connection -> Configure Environment.)*

  - Define either data source for ODBC-compliant drivers or add full path of driver to the static Java class path for JDBC-compliant drivers.

  - Test the connection to your database using Database Explorer or the command line.

  - Connect to the database.

  - Import the data from database into a MATLAB variable.

  - Insert data into database.

  - Close the connection

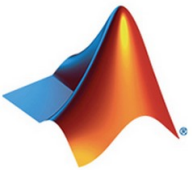
Defining Data Source in Windows

- Defining Data Source Name for database using MS Access and ODBC driver is as follow:

  Assuming MATLAB 32-bits with MS Access and ODBC driver of 32-bits.

  - Go to following path:
    `C:\Windows\SysWOW64\`
  - Search and click the **`odbcad32.exe`**
  - In the pop-up window, click the **`Add`** under User DSN tab.
  - On Create New Data Source pop-up window, select **`Microsoft Access Driver(*.mdb, *.accdb)`** and click **`Finish`**.
  - In the ODBC Microsoft Access Setup dialog box, enter **`dbtoolboxdemo`** as data source name and **`tutorial database`** as the description.
  - Click **`Select`** to open the Select Database dialog box.
  - Select the **`tutorial.mdb`** database in C:\Program Files\MATLAB\.....\toolbox\database\dbdemos
  - Click **`ok`** on all pop-up windows.(Verify that UserDSN tab displays the added dbtoolboxdemo)

- Can also create DSN through DataExplore.
  - Step at *doc database -> Database Connection -> Configure Environment -> Microsoft Access ODBC for Windows*



Connect to Database

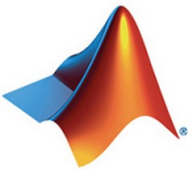
- Connect to Microsoft Access using Database Explorer.
  - After defining DSN, click **Connect** in the Database Explorer tab.
  - In the *Connect to a Data Source* dialog box, connect to your database by selecting the data source name **dbtoolboxdemo** from the Data Sources list.
  - Enter a user name and password and click **Connect**.

- Connect to Microsoft Access using command line
  - Connect to the database with the ODBC data source name

```
conn = database('dbtoolboxdemo','','');
```

  - Close the database connection conn.

```
close(conn)
```



Database Explorer

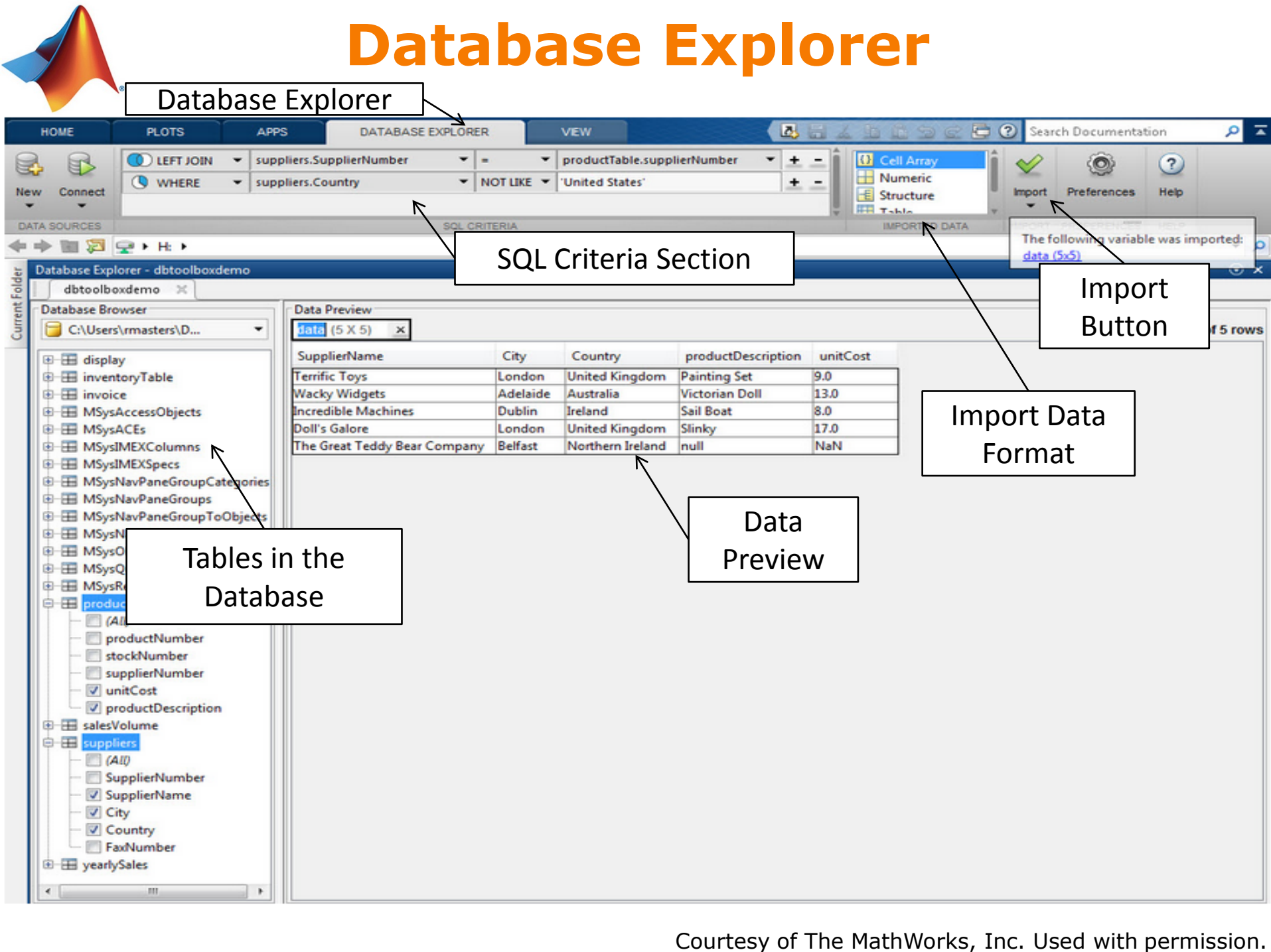
- Enables to quickly connect to a database, explore the database data, and import data from the database to the MATLAB.

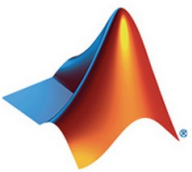
- To open the Database Explorer:
  - Enter `dexplore` at command prompt
  - Under the **Apps** tab, find its *icon* in Database Connectivity and Reporting section.

- Further details can be found at:

```
>> doc dexplore
```

Database Explorer

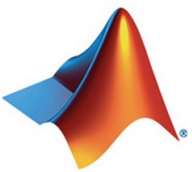




Exercise: Database Explorer

Try:

- Use the database explore to display the all the columns of the `productTable` of `tutorial.mdb` database.
- Refine the results by displaying the entries corresponding to `productNumber=8`.
- Generate the SQL code for above step.
- Save the results in a MATLAB variable `pdata` as string.
- Generate the MATLAB script `prodata.m` for the above steps.



Database Functions

- **database:** To connect to database

```
conn = database(instance,username,password)
```

- **import data:** Use *exec* function (to execute the SQL statement) and *fetch* function (to retrieve the data)

```
curs = exec(conn, SQLquery);
curs = fetch(curs);
```

- **export data:** To insert data into database, use *datainsert*, *fastinsert*, and *insert* functions.

```
fastinsert(conn,tablename,colnames,data)
```

- **update:** To replace data in database table.

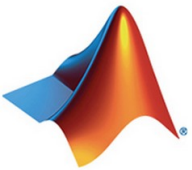
```
update(conn,tablename,colnames,data,whereclause)
```

- **rollback:** To undo the changes made in database.

```
rollback(conn)
```

- **close:** To close the database connection

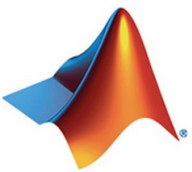
```
close(conn)
```



Exercise: Database Functions

TRY:

- Write a script `dbscript.m` that retrieves sales data from a `salesVolume` table of the **dbtoolboxdemo** database.
- Calculates the sum of sales for the month of March.
- Stores this pair of data in a cell array.
- Exports this data to a table `yearlySales`.



Exercise: Database Functions

- Connect to the database, *dbtoolboxdemo*

```
conn = database('dbtoolboxdemo','','');
```

- Use *setdbprefs* to set the format for retrieved data to *numeric*

```
setdbprefs('DataReturnFormat','numeric');
```

- Import rows of the March column from the *salesVolume* table

```
curs = exec(conn,'select March from salesVolume');
curs = fetch(curs);
```

- Assign the data to variable *AA*

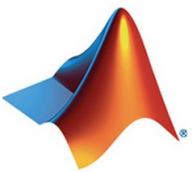
```
AA = curs.Data;
```

- Calculate the sum of *March sales* and assign result to variable *sumA*

```
sumA = sum(AA(:))
```

- Assign the month and sum of sales to a cell array *exdata*

```
exdata(1,1) = {'March'};
exdata(1,2) = {sumA};
```



Exercise: Database Functions

- Assign the cell array *colnames* containing the column names

```
colnames = {'Month','salesTotal'};
```

- Use the *fastinsert* function to export the data into the yearlySales table.

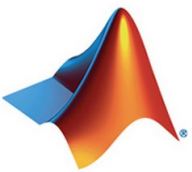
```
fastinsert(conn,'yearlySales',colnames,exdata);
```

- *Close* the cursor.

```
close(curs);
```

- *Close* the connection.

```
close(con);
```



Audio Processing

Specify functions to process audio files.

- To loading sound files:

```
>> [road,fs]=wavread('road.wav')        %.wav file format%
>> [lunch,fs2]=auread('lunch.au');      %.au file format%
```

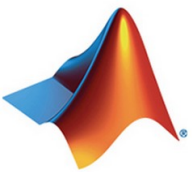
`road` and `lunch` represent stereo sound data;
`fs` is the sampling frequency

- To extract left and right channel road array:

```
>> left=road(:,1);
>> right=road(:,2);
```

- To listen the data:

```
>> soundsc(left,fs)        % plays left channel as mono%
>> soundsc(right,fs)       % plays right channel mono%
>> soundsc(road,fs)        % plays road as stereo%
```



Exercise: Audio Processing

Try:

- To change speed:

  As, **`fs`** used to estimate time between each sample (T=1/fs)

```
>> soundsc(road,fs/1.5)      % slows the speed%
>> soundsc(road,fs*1.5)      % fasts the speed%
```

Try:

- To reverse the track:

  - flip the array upside-down

```
>> left2=flipud(left);
```

  - play the sound

```
>>soundsc(left2,fs)
```

Try:

- To remove voice:

  - vocal track is similar on left and right track:

```
>> soundsc(left-right,fs);      %virtually no vocal%
```