Workshop on MATLAB

Raju Pal
Avinash Pandey
Ashish Tripathi

Department of Computer Science
Jaypee Institute of Information Technology, Noida
Weblink: https://sites.google.com/site/rajupalcse/

June 05-09, 2018

Outline

Programming Languages

- First High Level Language : FORTRAN (1954)
  - To make programming easier, especially to solve numerical problems
- Todays languages: 1970s-
  - For commercial,"Shrink-wrapped", large, complex programs: C,C++, JAVA, C#, etc.
  - To make programming easier, especially to solve numerical problems: MATLAB
  - same purpose as FORTRAN but far easier to program

MATLAB History

- Invented by Prof. Cleve Moler to make programming easy for his students
  - Late 1970s
  - University of New Maxico
- The MathWorks, Inc. was formed in 1984
  - By Moler and Jack Little
  - One Product: MATLAB
- Today
  - 100 products
  - Over 1 Million users
  - users come from various backgrounds of engineering, science, and economics.

MATLAB

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

MATLAB

- Operating System: Windows, Linux like ubuntu and MacOS
- Version: 32 bit and 64 bit
- Similar softwares: Octave, Scilab, Mathematica
- MathWorks has been putting out two releases of MATLAB per year for the past few years. The first one early in the year is *a* and the second one later on is *b*.
- They started this versioning scheme in 2006 and still maintain a numerical version number in the background.
  e.g. Recent release of MATLAB is 2017*b* and its verison number is MATLAB 9.3
- Both versions can be installed on the same machine in different folders.

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

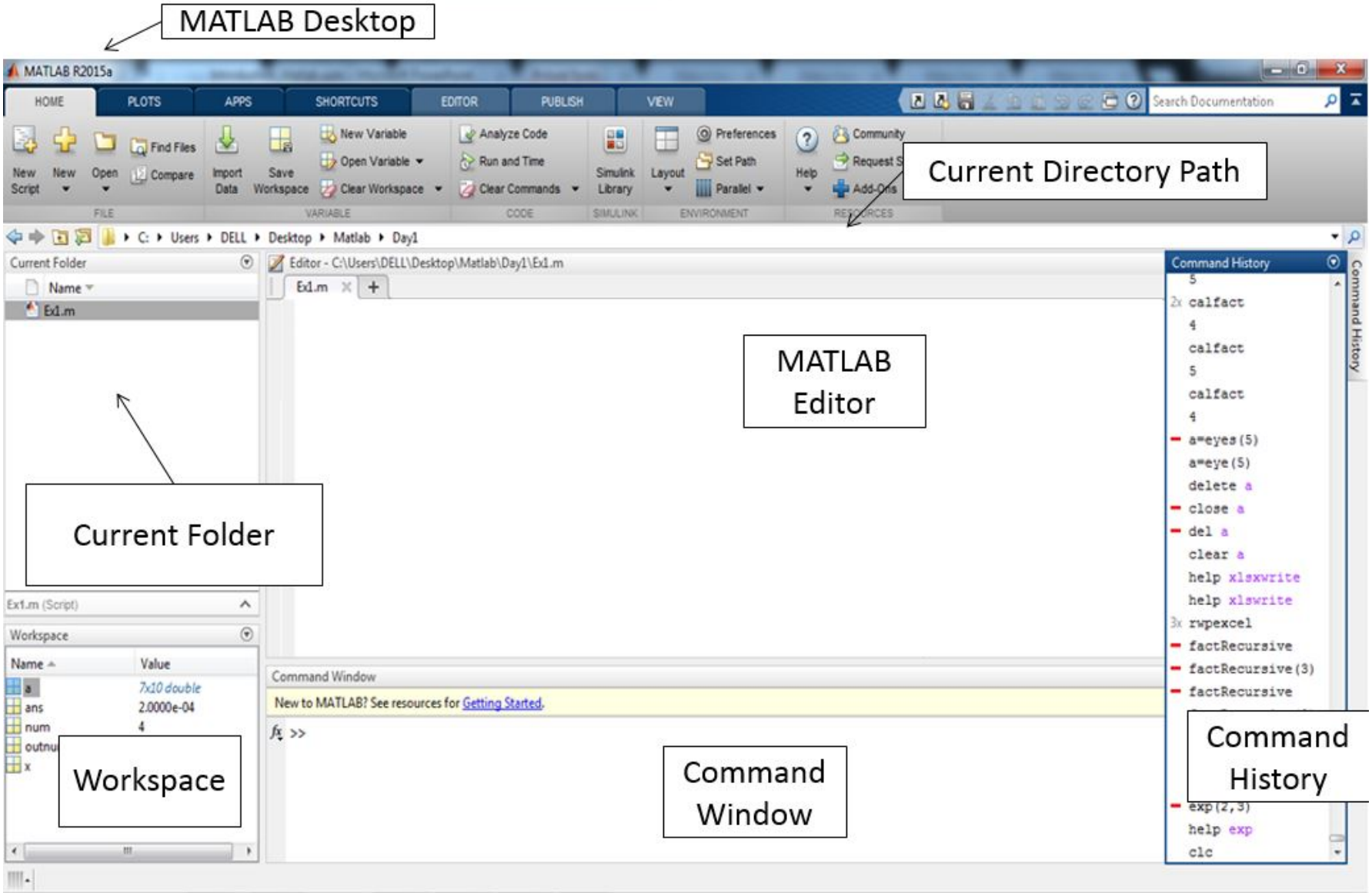
Some facts regarding MATLAB

- MATLAB is an interpreted language, executes commands line by line
- MATLAB is a dynamically typed language
  - Means that you do not have to declare any variables
  - All you need to do is initialize them and they are created
- MATLAB is "4G programming language", that is, a programming language designed with a *specific* purpose; in contrast to 3rd generation programming languages (FORTRAN/C/C++) which are general purpose in nature.
- MATLAB treats all variables as matrices
  - Scalar – 1 x 1 matrix. Vector – 1 x N or N x 1 matrix
  - Why? Makes calculations a lot faster
- Indexing in MATLAB starts with 1 instead of 0.

Toolboxes

- Aerospace Toolbox, Fuzzy Logic Toolbox
- Antenna, Communication systems, DSP, RF Toolbox
- Computer Vision System, Data Acquisition, Image Acquisition Toolbox
- Bioinformatics, Database
- Optimization, Parallel Computing, Real Time workshop and many others.

MATLAB Desktop

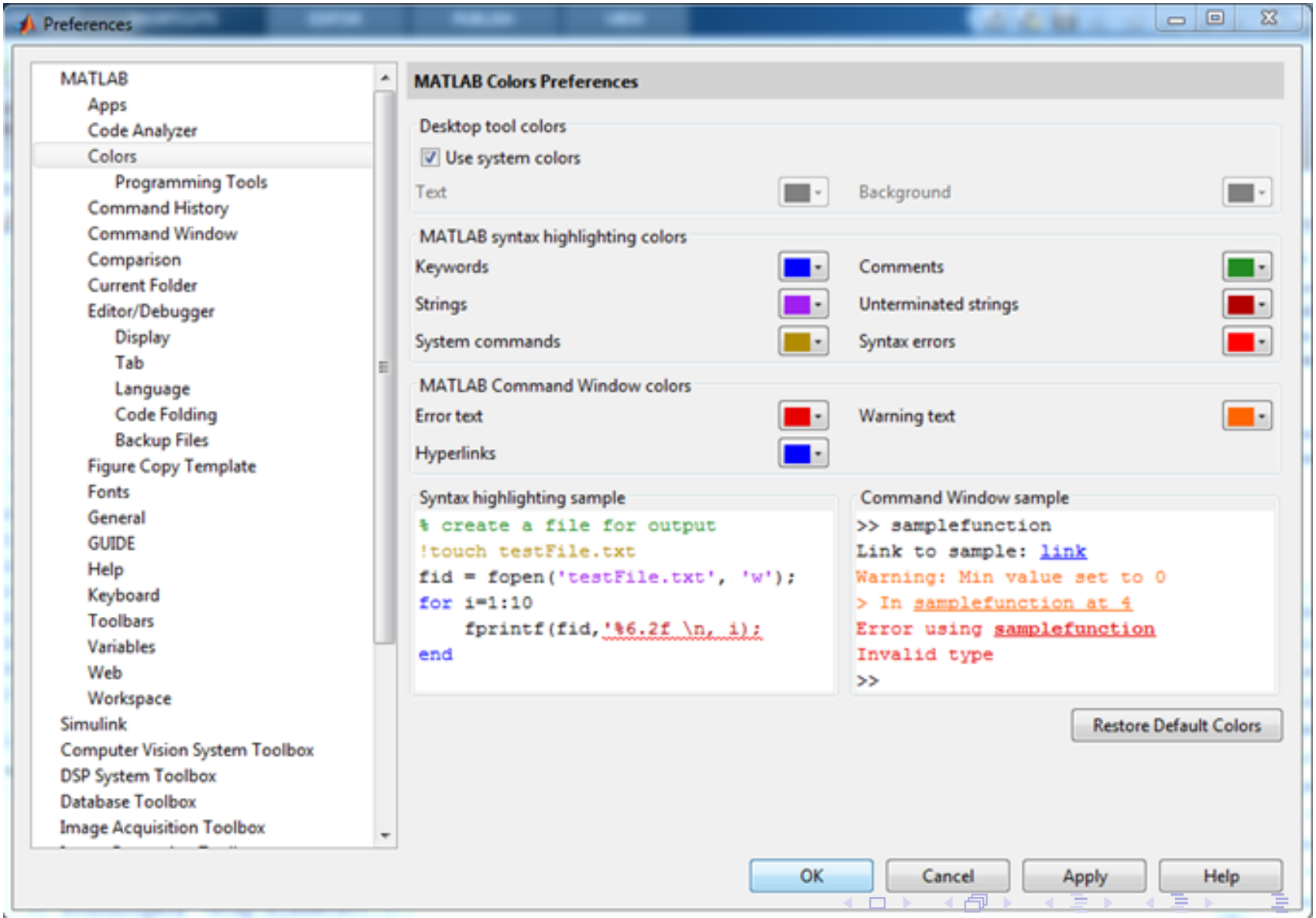


Making Folders

- 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



Let's start some hands on

- You can execute commands by entering them in the command window after the MATLAB prompt (>>) and pressing the Enter key.
  Task 1:Try multiplying the numbers 3 and 5 together with the command 3*5.
- Unless otherwise specified, MATLAB stores calculations in a variable named **ans**.
  $>> 7 + 3$
  $ans = 10$
  Task 2: Try assigning the 3*5 calculation to a variable named m.
  Task 3: Try entering the command m = m + 1 to see what happens (assignment operator)

Let's start some hands on

- The **Workspace** window shows all the variables currently in the workspace.
  Task 4: Create a variable named y that has the value $m/2$.
- Adding a semicolon to the end of a command will suppress the output, but it can be seen in the **workspace**

```
>> x = 5+1
x =
     6
```

```
>> x = 5+1;
```

Task 5: Enter $k = 8 - 2$;

Let's start some hands on

- You can recall previous commands by pressing the Up arrow key on your keyboard. Note that the Command Window must be the active window for this to work.
  Task 6: Press the Up arrow to return to the command m = 3*5 and edit the command to be m = 3*k
- When you enter just a variable name at the command prompt, MATLAB returns the current value of that variable.
- You can name your MATLAB variables anything you'd like as long as they start with a letter and contain only letters, numbers, and underscores (_).
  e.g.

```
>> 3sq = 9
 3sq = 9
  ↑
Error: Unexpected MATLAB expression.
```

Using Built-in Functions and Constants

- MATLAB contains built-in constants, such as pi to represent π
  Task 1: Create a variable called x with a value of $\pi/2$.
- MATLAB contains a wide variety of built-in functions such as abs (absolute value) and eig (calculate eigenvalues)
  Task 2: Try using the *sin* function to calculate the sine of x. Assign the result to a variable named y.
  Task 3: Now try using the *sqrt* function to calculate the square root of -9. Assign the result to a variable named z.
  Task 4: Try following commands:
  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);

Help/Docs

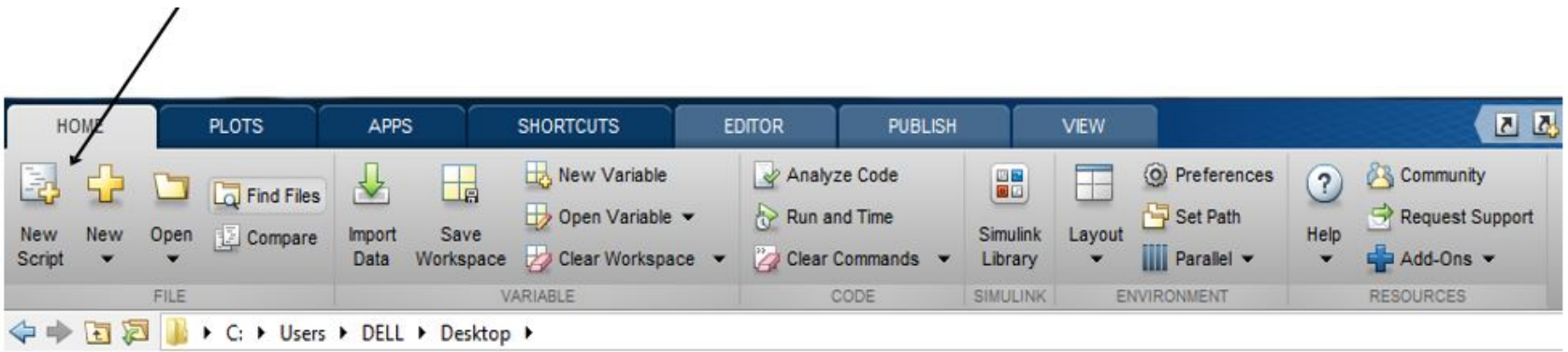
- The most important function for learning MATLAB on your own
  - help
- To get info on how to use a function:
  - help sin
  - Help lists related functions at the bottom and links to the doc
- To get a nicer version of help with examples and easy-to- read descriptions:
  - doc sin
- To search for a function by specifying keywords, use search tab of doc:
  - doc

save/clear/clc/load

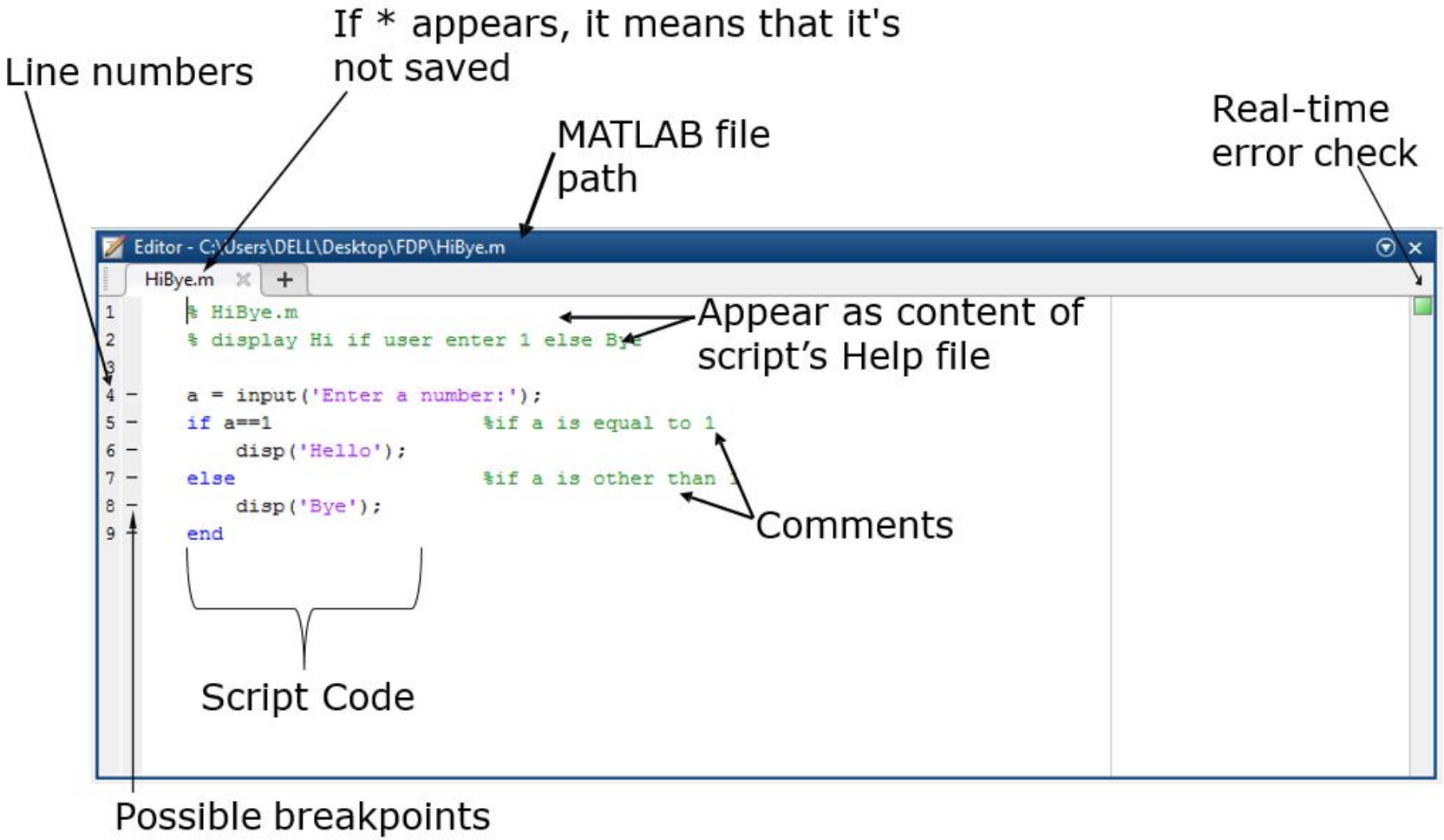
- Use save to save variables to a file
  save myFile a b
  - saves variables a and b to the file myfile.mat
  - myfile.mat file is saved in the current directory
- Use clear to remove variables from environment
  clear a b
  - look at workspace, the variables a and b are gone
- Use clc to clear the command 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;

Introduction

- 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 First.m
  - or click



Scripts: the Editor



Comments

- 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
  Task 1: Type the following command in command window to see the properly commented script file.
  open mean.m

Input and Display

- Use input function to request for user input
  - V1=input('Enter the value');
  - V2=2*V1*V1;
- Use disp to print messages
  - disp('starting loop')
  - disp(['loop is over' num2str(9)])
  - disp(V2);
- disp prints the given string to the command window (Strings are written between single quotes, like 'This is a string'.)

Exercise 1: Scripts

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

  Hello World!

  I am going to learn MATLAB!

Exercise 1: Scripts

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

  Hello World!

  I am going to learn MATLAB!

Solution

```
% helloWorld.m
% my first hello world program in MATLAB
disp('Hello World!');
disp('I am going to learn MATLAB!');
```

Defining variables

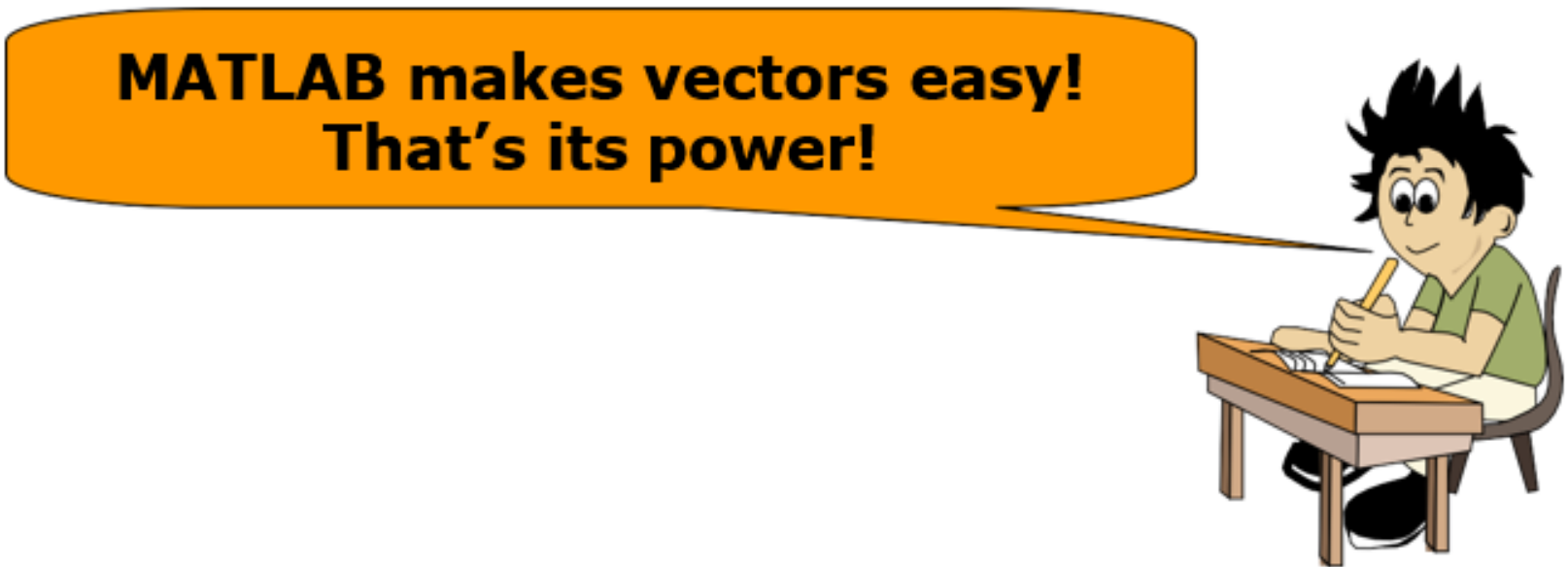
- To create a variable, simply assign a value to a name.
  var1=3.14; var2=-2;
- Default variable type is **double**
- To convert variable type to integer:
  $y1 = int32(var1)$; $y2 = uint8(var2)$;
- Other datatypes supported by MATLAB are as follows:
  - Floating point number (single and double) e.g $A = 1.25125$
  - Integers (int8, uint8, int16, uint16, int32, uint32, int64, uint64) e.g. $A = uint16(256)$
  - Characters and Strings(char) e.g. $myString = 'helloworld'$
  - Boolean (logical) e.g. $A = true$;

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

- All MATLAB variables are arrays, meaning that each variable can contain multiple elements.
- A single number, called a scalar, is actually a 1-by-1 array, meaning it contains 1 row and 1 column.
- Two types of arrays
  - matrix of numbers (either double or complex)
  - cell array of objects (more advanced data structure)



Defining Vectors

- vector: list of values between square brackets
  - row = [1 2 5.4 -6.6]; row = [1, 2, 5.4, -6.6] % row vectors
  - y1 = [1; 2; 5.4; -6.6]; y2 = [1, 2, 5.4, -6.6]'; % column vectors
  - $x_3 = ones(3, 1)$; %Special vectors
  - $x_4 = zeros(1, 4)$; %Special vectors
  - $i = 1 : 1 : 10$ OR $j = -1 : 0.2 : 3$ %index vectors
  - $r = rand(1, 45)$ %random array
- Workspace:

Workspace

Name	Bytes	Size	Class
row	32	1x4	double

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

- Matrices are like vectors
- Initialized with Element by element
- A= [1 2;3 4;0 -1];
- Strings are character vectors
- Diagonal arrays:A1=diag([1 2 -4]);
- Indentity matrix: A2=eye(2); OR A3=eye(5,3);
- In MATLAB, you can perform calculations within the square brackets. x = [abs(-4) 4^2];
- If you know the number of elements you want in a vector,you could use the linspace function:
  linspace(*first*, *last*, *number_of_elements*).

Exercise 2: Variables

- 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 2: Variables

- 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
- Solution
  - help clock
  - start=clock;
  - size(start)
  - help datestr
  - startString=datestr(start);
  - save startTime start startString

Advanced Data Structure

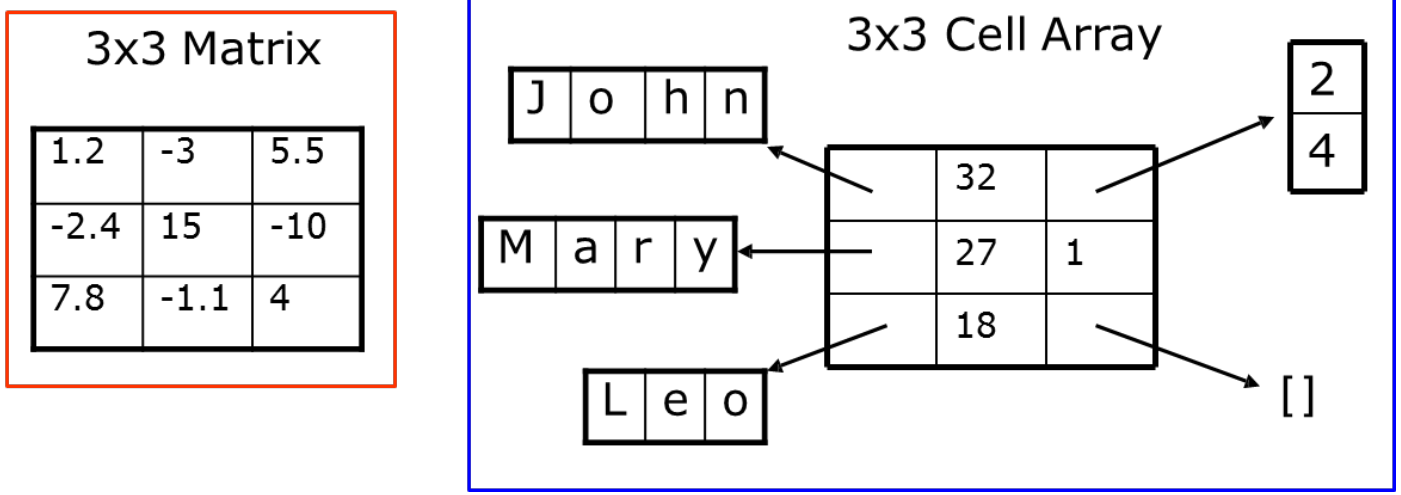
- We have used 2D matrices
  - Can have n-dimensions
  - Every element must be the same type (ex. integers, doubles, characters...)
  - Matrices are space-efficient and convenient for calculation
  - Large matrices with many zeros can be made sparse:

```
» 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):



Cells: organization

- 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 Array

- 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);
```

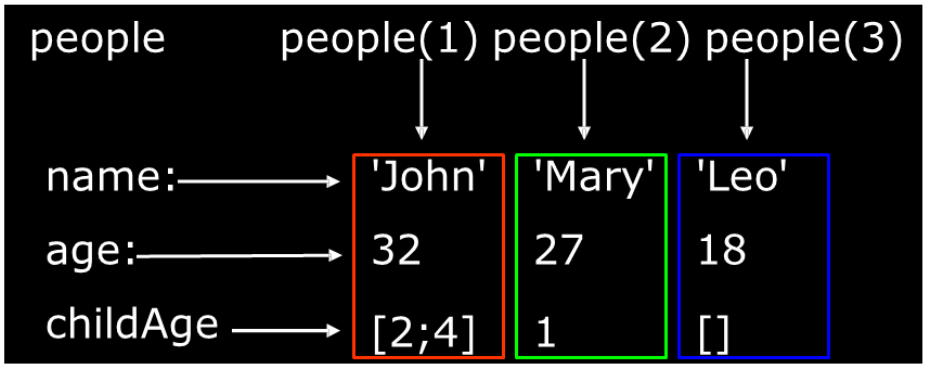
  - person is now a struct with fields name, age, children
  - the values of the fields are the values of the second index

```
» person.name
```

  - returns 'Mary'

```
» people(1).age
```

  - returns 32



Exercise 3: Struct Arrays

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 3: Struct Arrays

```
» 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
```

Basic Scalar Operations

- Arithmetic operations (+,-,*,/)
  7/45
  (i+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

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):

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

- 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

Binary Operators

- 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`

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

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

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

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

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

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

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

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

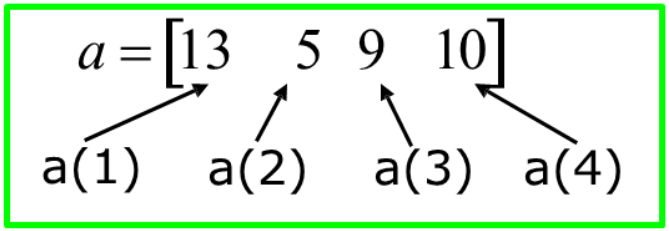
$$3\times 3 * 3\times 3 = 3\times 3$$

Exercise 4: Arrays

- Create a row vector named x that starts at 1, ends at 10, and contains 5 elements.
- Create a variable named x that is a 5-by-5 matrix of random numbers.
- Create a variable having random numbers within a particular range.
- Try Basic math operations
  - x^3
  - 2*y1
  - A*y2
  - A*A or A^2
  - A^(-1)
  - y1.*y2

Vector Indexing

- MATLAB indexing starts with **1**, not **0**
- a(n) returns the n^{th} element

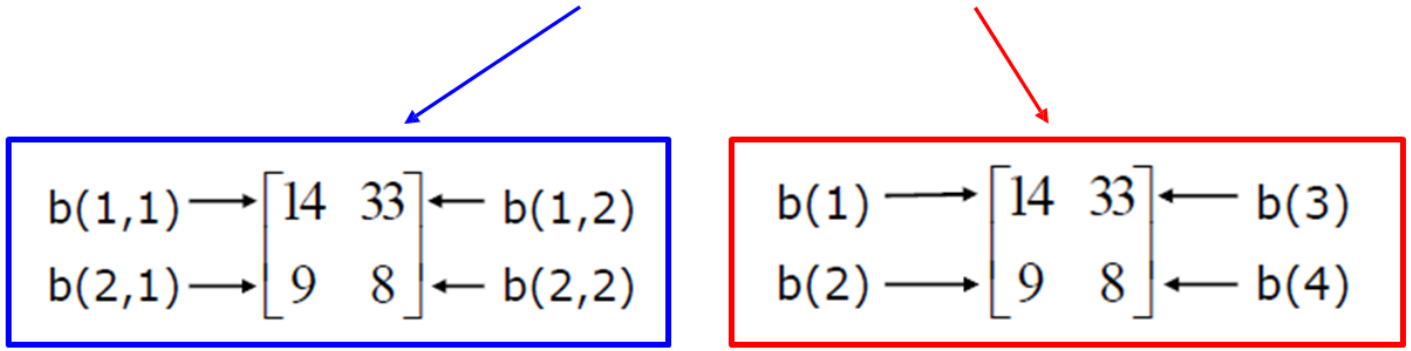


- The index argument can be a vector. In this case, each element is looked up individually, and returned as a vector of the same size as the index vector.

```
» 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 5: Arrays

- Calculate how many seconds elapsed since the start of this class?
  - 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 5: Arrays

- **Solution:**

```
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';
```

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
```

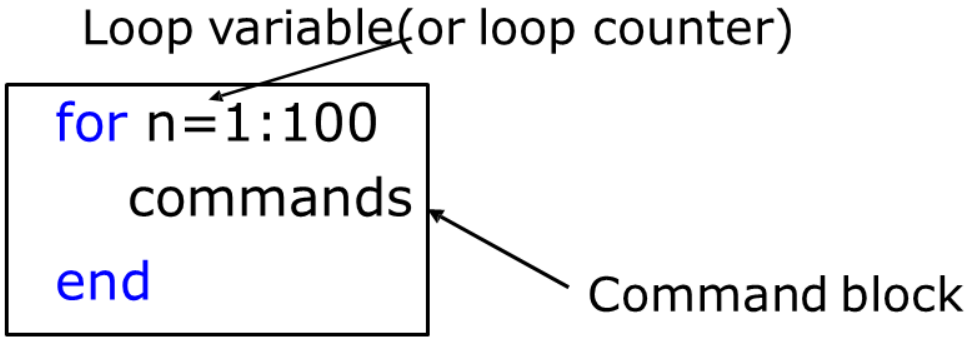
ELSEIF

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

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

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!

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

Advantage of *find* function

- 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

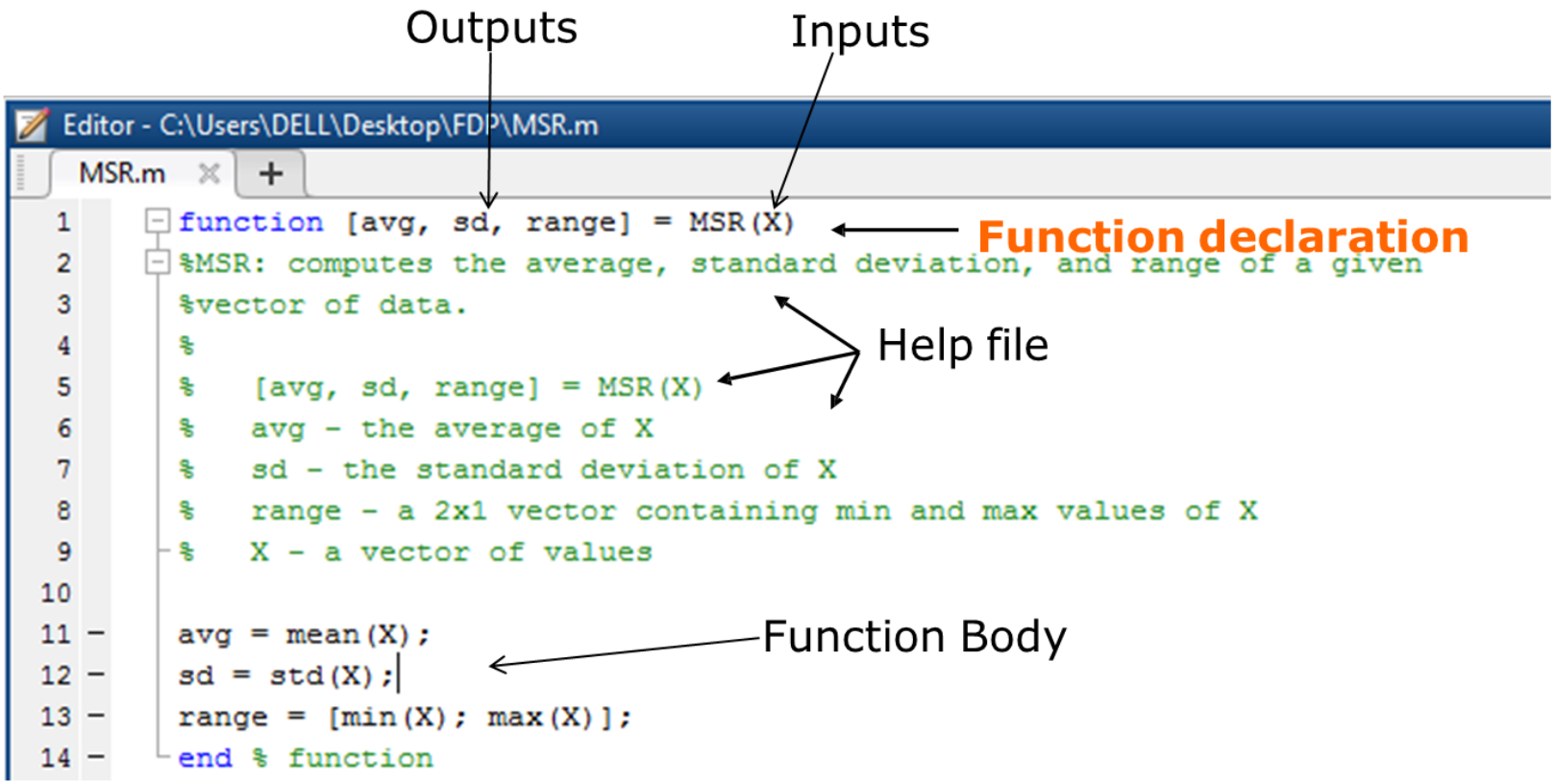
```
count=length(find(x>0))
```

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

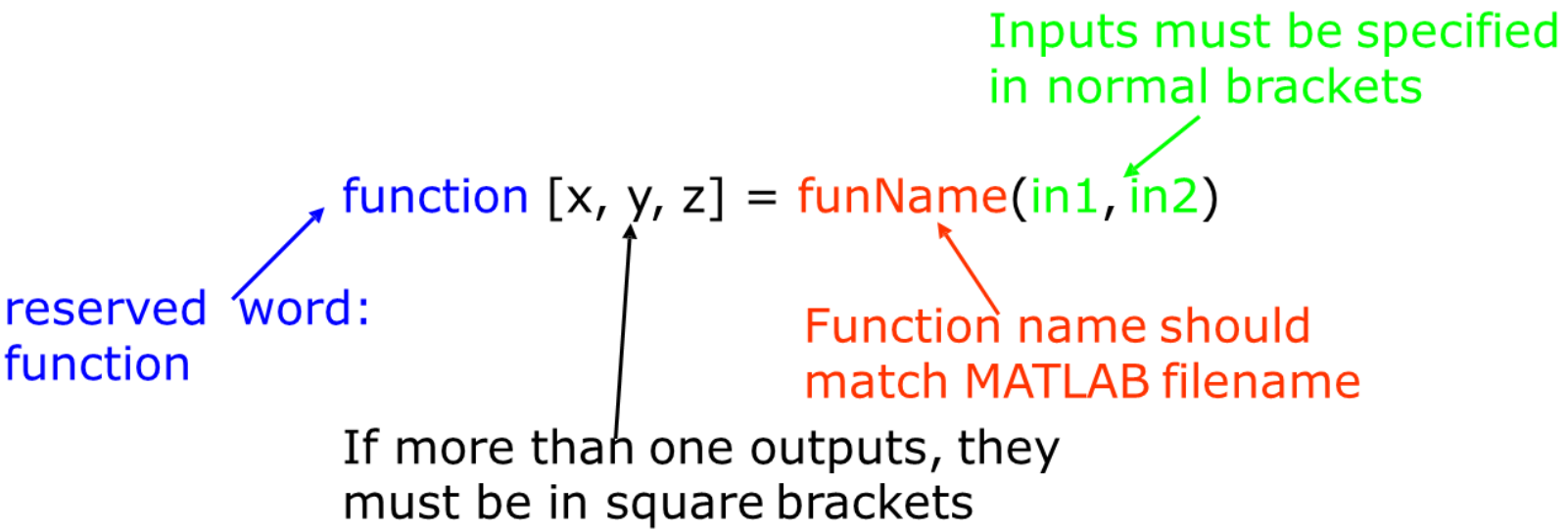
Functions

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



User-defined Functions

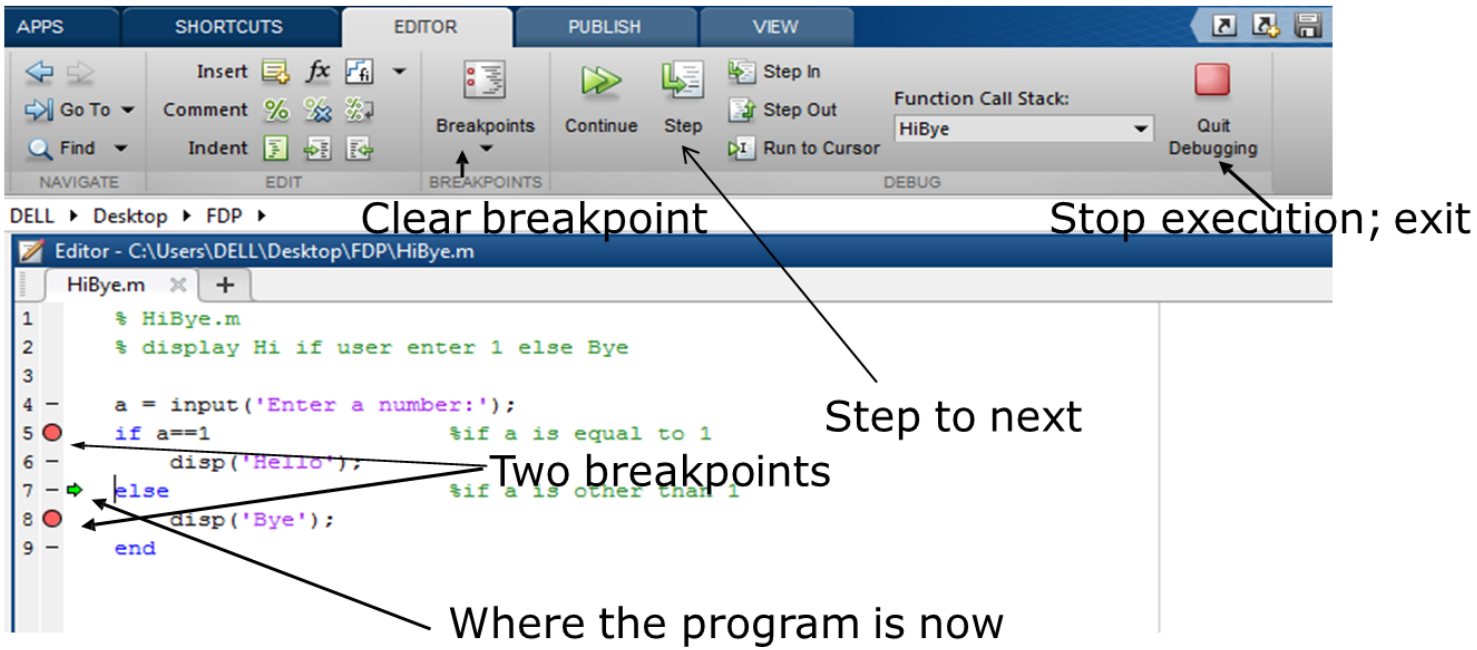
- Some comments about the function declaration



- 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

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 6: Debugging

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;
```

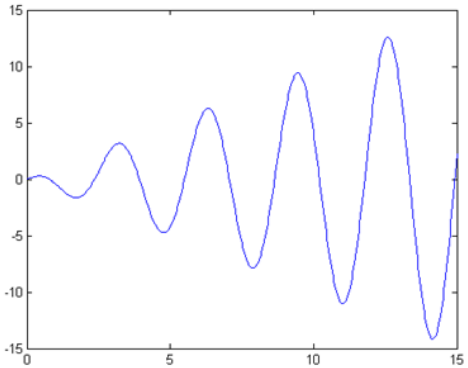
  - tic resets the timer
  - Each toc returns the current value in seconds
  - Can have multiple tocs per tic

Exercise 7: Functions and Loops

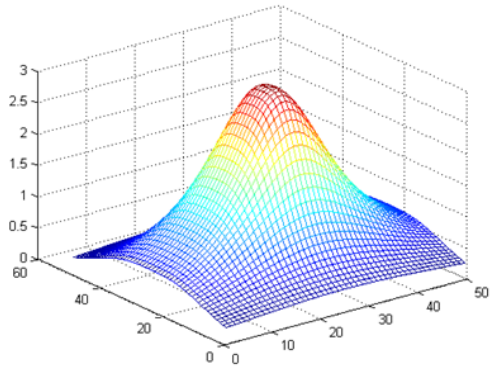
- **Task 1:** Write is a simple function to convert temperatures given in degrees Fahrenheit into degrees Celsius.
- **Task 2:** Create a function that computes the volume of a sphere with a certain radius. Call the function *'volsphere'*. Now compute the volumes of the spheres with radii 0.1, 0.2, 0.3, through to 1.2 using a for-loop. Display the computed volumes. Store the program in a script file called *volumes.m*.
- **Task 3:** Change the function and script file of task 2 so that you also compute and display the surface areas of the spheres.
- **Task 4:** Write a script that will rotate a matrix by 90 degree.

Plot

- MATLAB provides functions to create various types of plots.



2D plot



3D plot

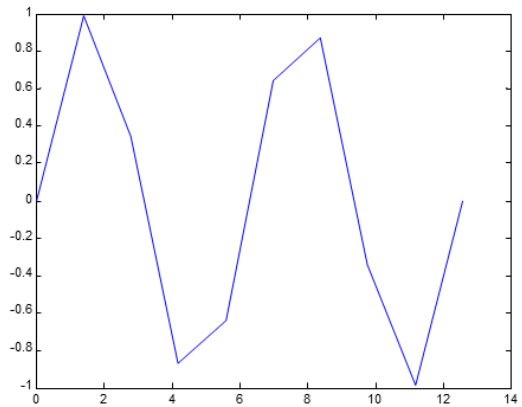
- To draw 2D-plot, use plot() function
- *plot()* generates dots at each (x,y) pair and then connects the dots with a line

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

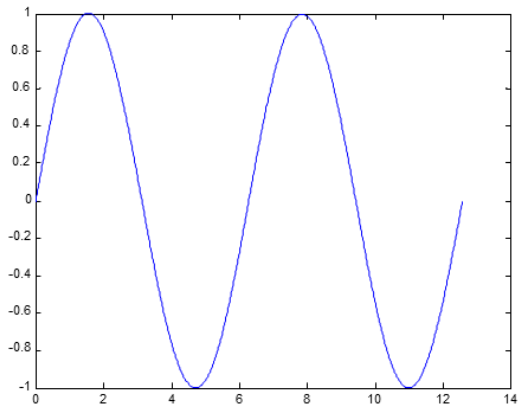
What does plot do?

- 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:



Plot

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

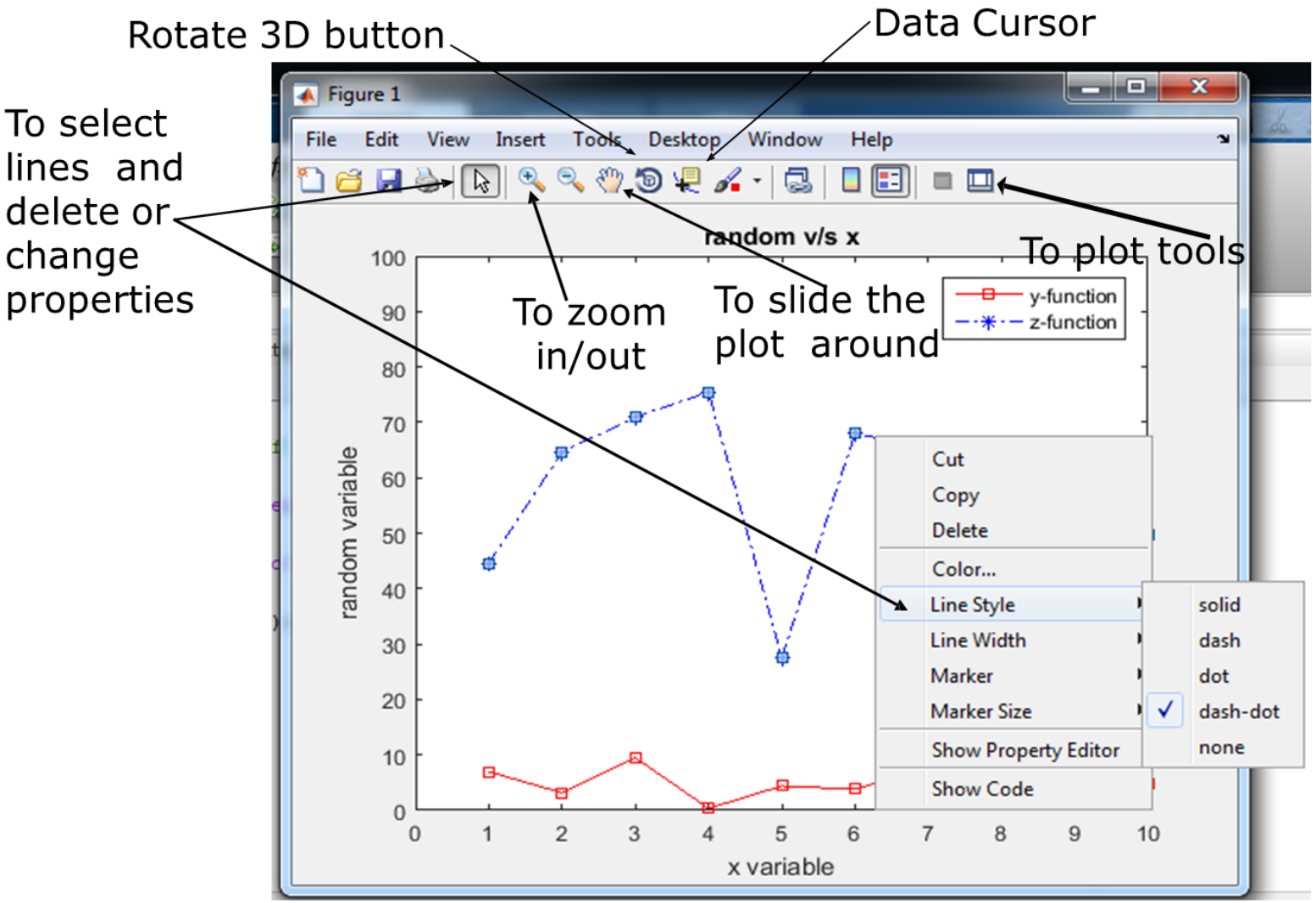
plot(x,y,'line specifier')

```
» 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		

Playing with Plot

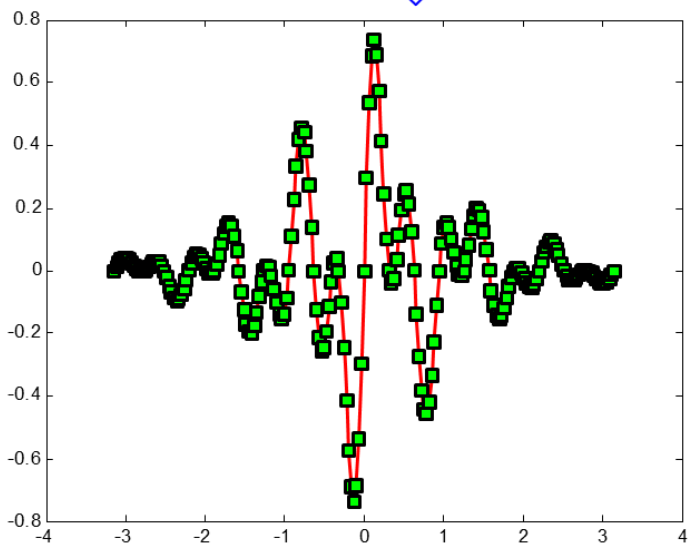


Line and Marker Options

- 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.



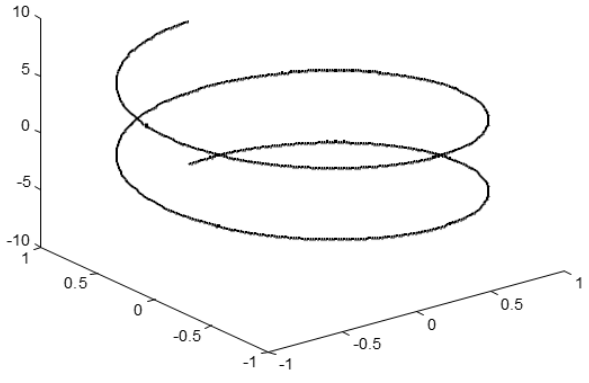
- See **doc** for a full list of properties that can be specified

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 I

Create axes in tiled positions

Syntax

subplot(m,n,p): divides the current figure into an m-by-n grid and creates axes in the position specified by p

Examples:

- Upper and lower subplots

```
subplot(2,1,1);
x = linspace(0,10);
y1 = sin(x);
plot(x,y1)

subplot(2,1,2);
y2 = sin(5*x);
plot(x,y2)
```

Multiple Plots in one Figure II

- Quadrant of Subplots

  Q: Create a figure divided into four subplots. Plot a sine wave in each one and title each subplot.

  A:

```
subplot(2,2,1)
x = linspace(0,10);
y1 = sin(x);
plot(x,y1)
title('Subplot 1: sin(x)')

subplot(2,2,2)
y2 = sin(2*x);
plot(x,y2)
title('Subplot 2: sin(2x)')
```

```
subplot(2,2,3)
y3 = sin(4*x);
plot(x,y3)
title('Subplot 3: sin(4x)')

subplot(2,2,4)
y4 = sin(8*x);
plot(x,y4)
title('Subplot 4: sin(8x)')
```

Multiple Plots in one Figure III

- Subplots with Different Sizes

Create a figure containing with three subplots. Create two subplots across the upper half of the figure and a third subplot that spans the lower half of the figure. Add titles to each subplot.

```
subplot(2,2,1);
x = linspace(-3.8,3.8);
y_cos = cos(x);
plot(x,y_cos);
title('Subplot 1: Cosine')

subplot(2,2,2);
y_poly = 1 - x.^2./2 + x.^4./24;
plot(x,y_poly,'g');
title('Subplot 2: Polynomial')

subplot(2,2,[3,4]);
plot(x,y_cos,'b',x,y_poly,'g');
title('Subplot 3 and 4: Both')
```

Multiple Plots in one Figure IV

- Replace Subplot with Empty Axes

  Create a figure with four stem plots of random data. Then replace the second subplot with empty axes.

```
for k = 1:4
data = rand(1,10);
subplot(2,2,k)
stem(data)
end
```

```
subplot(2,2,2,'replace')
```

Multiple Plots in one Figure V

- Subplots at Custom Positions

  position: [left bottom width height]

  Create a figure with two subplots that are not aligned with grid positions. Specify a custom position for each subplot.

```
pos1 = [0.1 0.3 0.3 0.3];
subplot('Position',pos1)
y = magic(4);
plot(y)
title('First Subplot')

pos2 = [0.5 0.15 0.4 0.7];
subplot('Position',pos2)
bar(y)
title('Second Subplot')
```

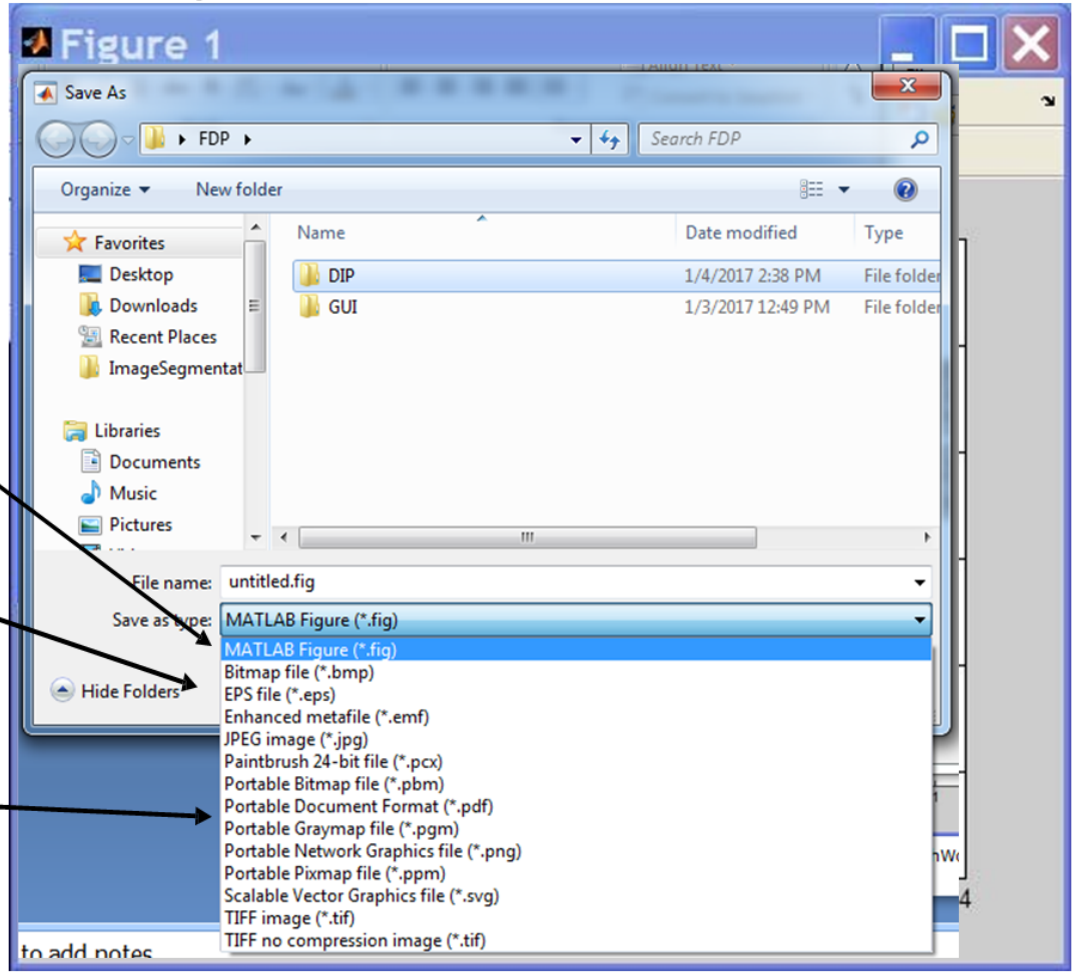
Saving Figures

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

.fig preserves all information

.bmp uncompressed image

.pdf compressed image



Surface Plots

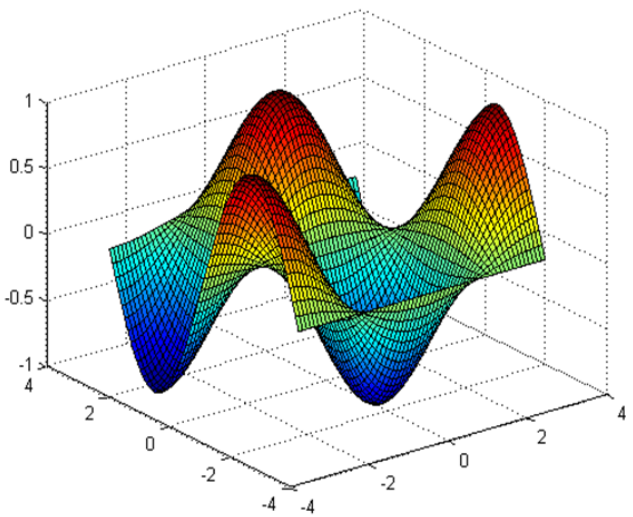
- It is more common to visualize surfaces in 3D
- Command:
  surf(X,Y,Z) creates a three-dimensional surface plot

Example: Create X, Y, and Z as matrices of the same size. Then plot the data as a surface. The surface uses Z for both the height and color data.

```
[X,Y] = meshgrid(1:0.5:10,1:20);
Z = sin(X) + cos(Y);
surf(X,Y,Z)
```

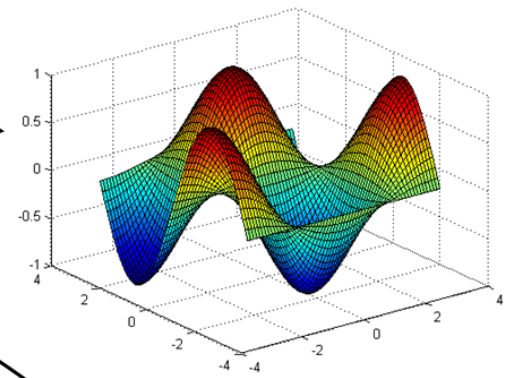
Exercise 8: surf

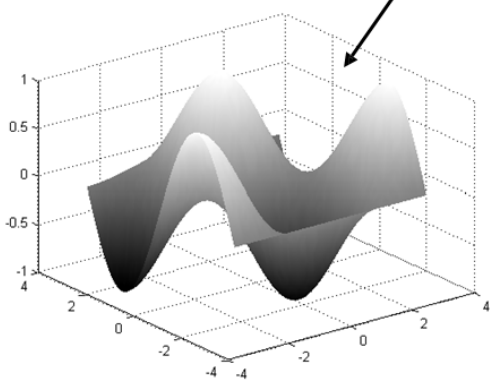
- Make the x and y vectors
  ```
  » x=-pi:0.1:pi;
  » y=-pi:0.1:pi;
  ```
- Use meshgrid to make matrices (this is the same as loop)
  ```
  » [X,Y]=meshgrid(x,y);
  ```
- To get function values, evaluate the matrices
  ```
  » Z =sin(X).*cos(Y);
  ```
- **Try** to plot the surface for X, Y, Z
  ```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

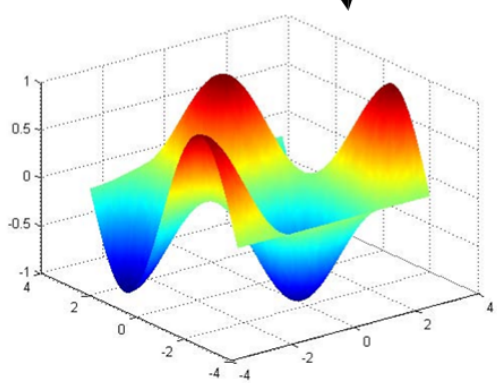


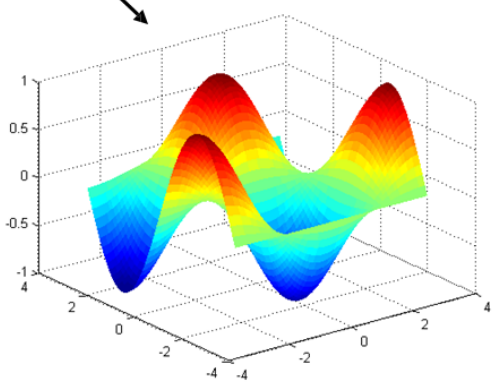
surf Options

- 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

Exercise 9: Plots

- 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 off`** (returns to default mode) command
- Can we do it with single command!!!!!

Exercise 9: Plots

- 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 off`** (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

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 *randomv/sx*
- 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

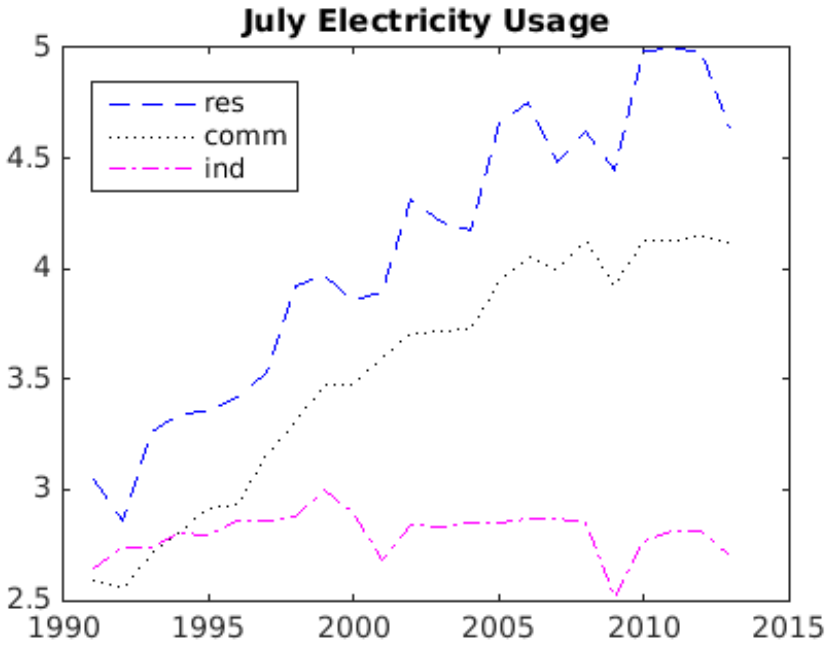
Format the previous 2D plot as follow.

- Label the x-axis and y-axis as 'x variable' and 'random variable' respestively
- Title the plot as *randomv/sx*
- 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)

```
» 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])
```

Project:Electricity Usage

In this project, you will plot electricity usage for various economic sectors - residential, commercial, and industrial. Which economic sector's usage do you think will be the largest?



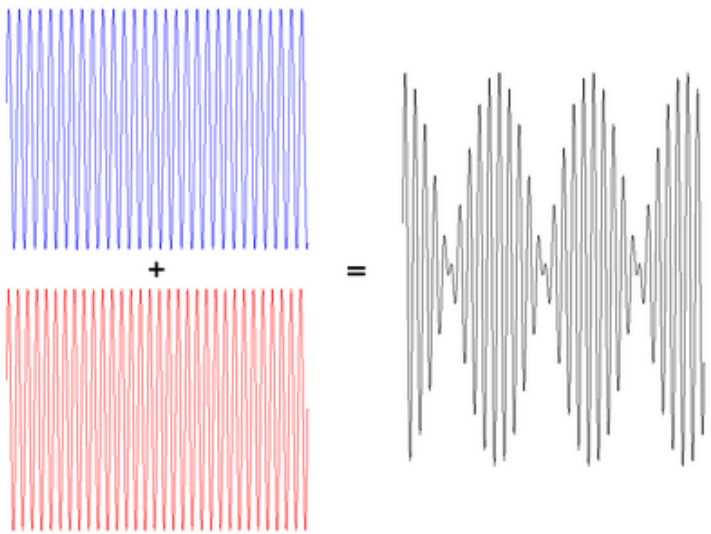
The usage data represents the US electricity consumption for different years in the month of July. The usage data are in 10^9 kWh/day, and the price data is in US cents per kWh.

Task to complete

- load electricity.mat
- One of the elements in the usage variable has a value of NaN. Replace this value with the value 2.74.
- The residential data is stored in the first column. Create a variable *res* that contains the first column of usage.
- The commercial data is stored in the second column. Create a variable *comm* that contains the second column of usage.
- The industrial data is stored in the third column. Create a variable *ind* that contains the third column of usage.
- Create a column vector named yrs that represents the years starting at 1991 and ending with 2013.
- Plot res (y-axis) against yrs (x-axis) with a blue (b) dashed line (–).
- hold on
- Plot comm (y-axis) against yrs (x-axis) with a black dotted line
- Plot ind (y-axis) against yrs (x-axis) with a magenta dash-dot line.
- Add the title 'July Electricity Usage' to the existing plot.
- Add the legend values 'res', 'comm', and 'ind' to the existing plot.

Project:Audio Frequency

Audio signals are usually composed of many different frequencies. For example, in music, the note 'middle C' has a frequency of 261.6 Hz, and most music consists of several notes (or frequencies) being played at the same time.
Typically, the frequencies that make up a signal are different enough that they do not interfere substantially with each other.
However, when a signal contains two frequencies that are close together, they can cause the signal to appear to have a 'beat' - a pulsing pattern in the amplitude.



In this project, you will create a signal that contains this beat phenomenon and then analyze the signal's frequency content.

Tasks to complete

- fs=10;
- Create a vector named t that starts at 0, ends at 20, and whose elements are spaced by $1/fs$
- Create a variable named y that contains the sum of two sine waves: $sin(1.8 * 2\pi t) + sin(2.1 * 2\pi t)$
- Plot the y vector (y-axis) against the t vector (x-axis).
- A Fourier transform returns information about the frequency content of the signal. You can use the fft function to compute the discrete Fourier transform of a vector. e.g. fft(y)
- Create a variable named *yfft* that contains the discrete Fourier transform of y.
- Create a variable named n that contains the number of elements in y.

Tasks to complete

- The **fft** function in MATLAB uses only the sampled data to compute the Fourier transform. The f variable will represent the frequencies that correspond to the values in **yfft**.
- Create a variable named f that contains a vector which starts at 0, ends at fs*(n-1)/n, and whose elements are spaced by fs/n.
- Plot the expression *abs*(*yfft*) (y-axis) against *f* (x-axis).

Importing Data from text files

- 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 from text files

- 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

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

Writing Excel Files

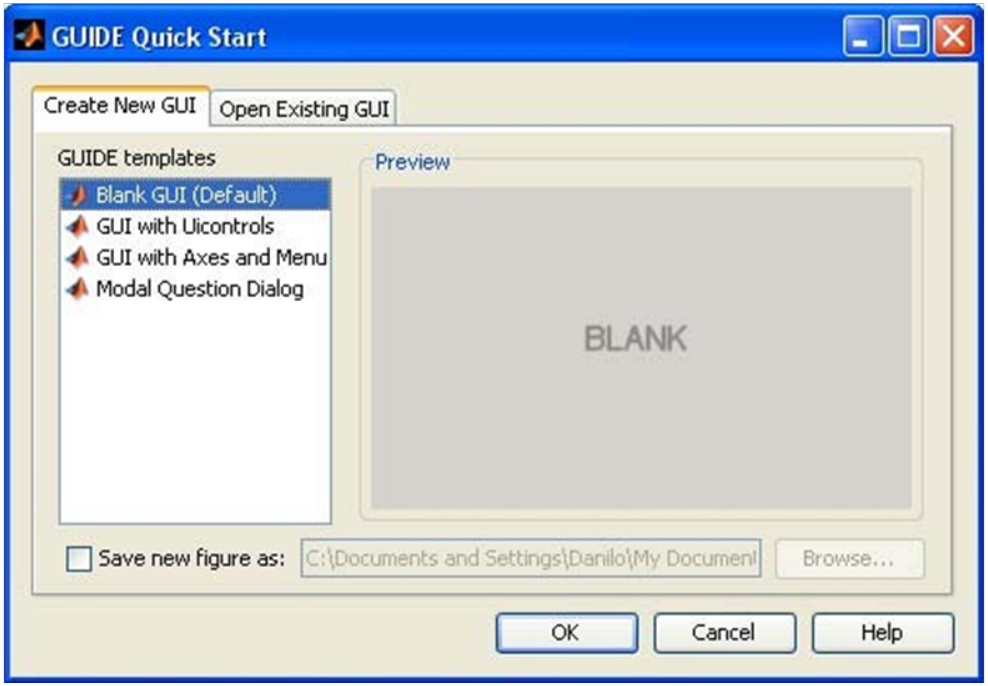
- MATLAB contains specific functions for reading and writing Microsoft Excel files
- To write a matrix to an Excel file, use **`xlswrite`**

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

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

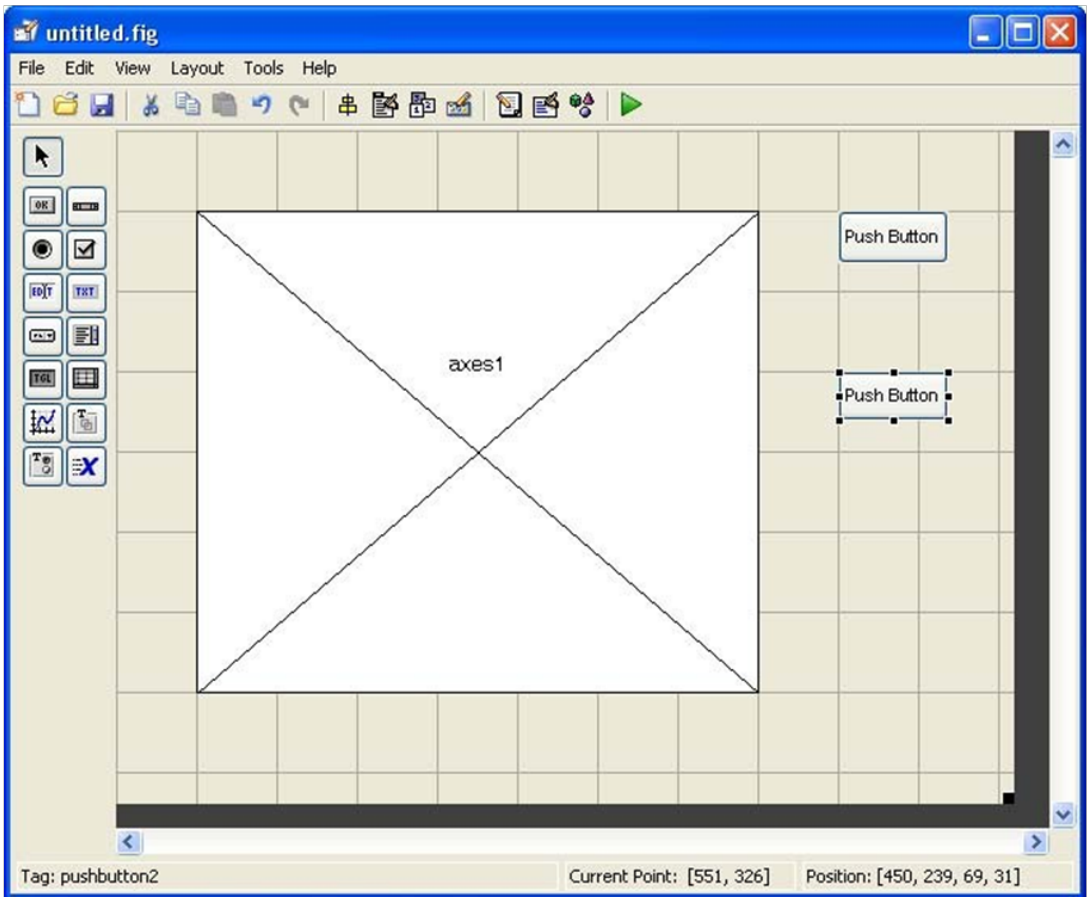
Making GUIs

- 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



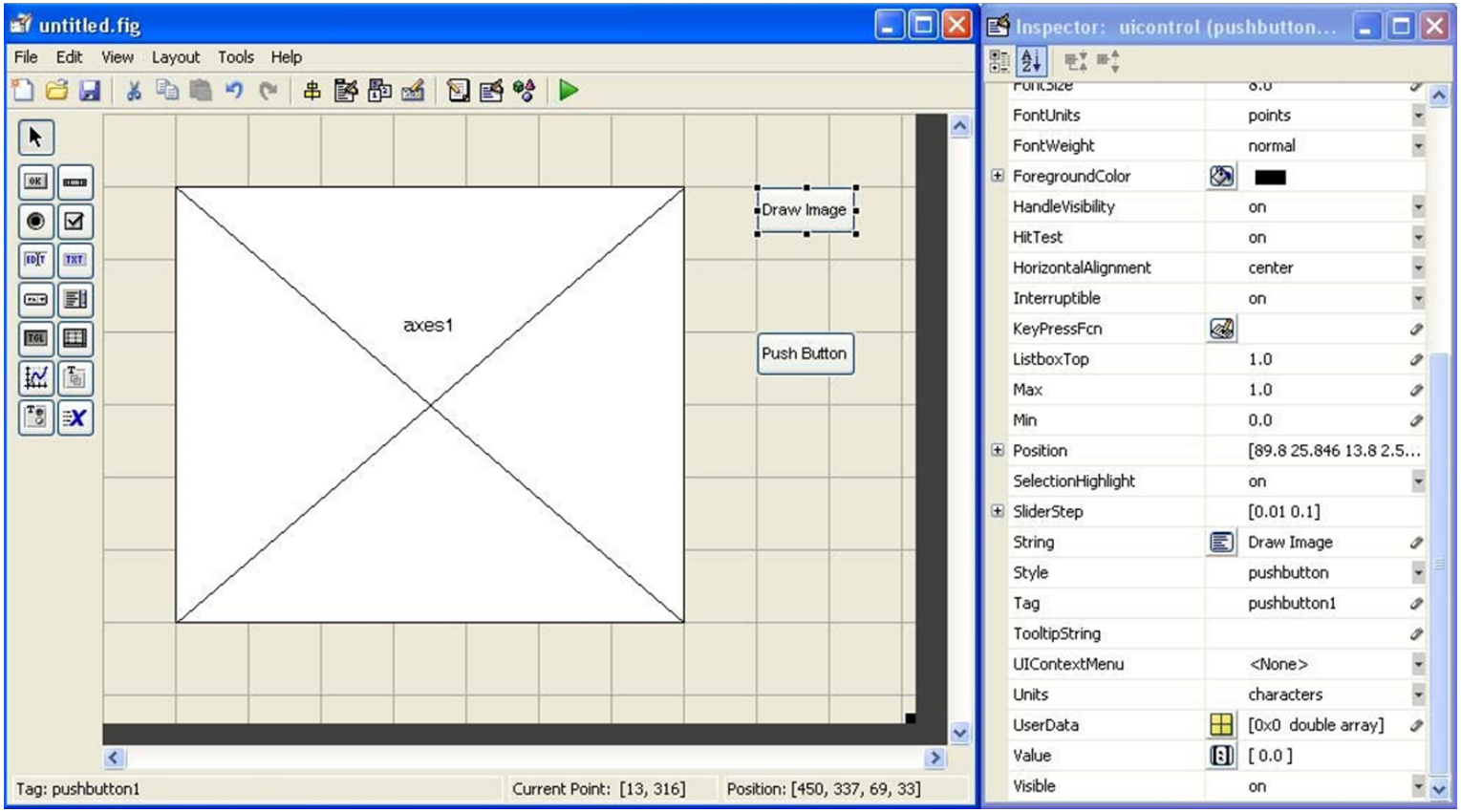
Draw the GUI

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



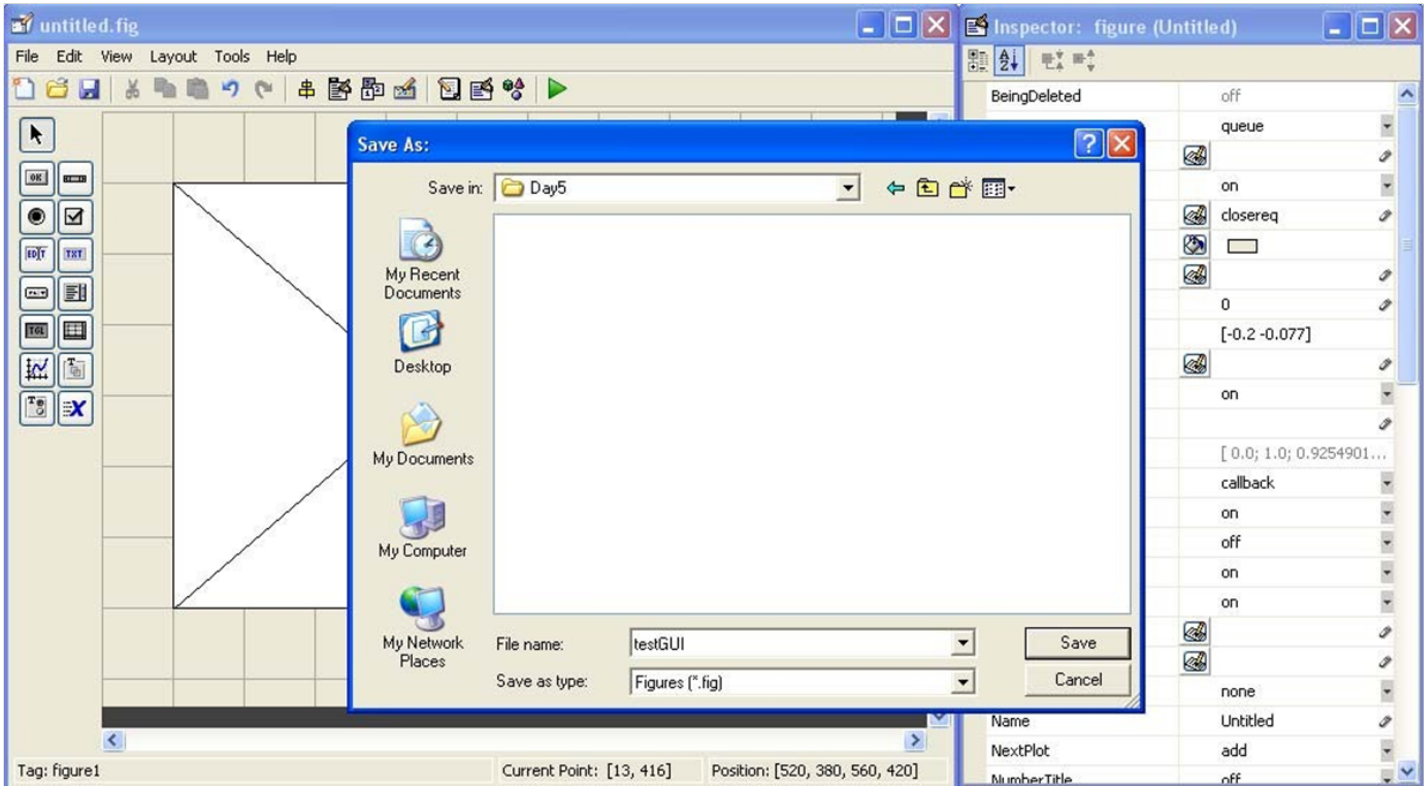
Change Object Settings

- 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!



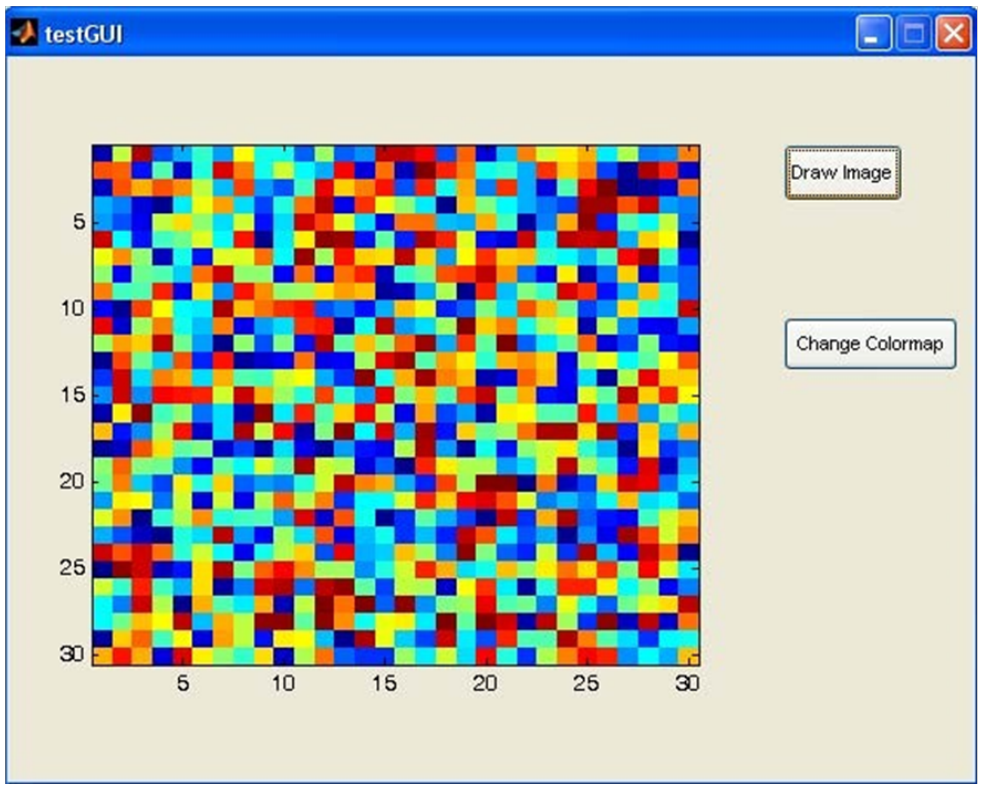
Add Functionality to MATLAB file

- 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 10: Login GUI

Try:

- Type *guide* at the command prompt

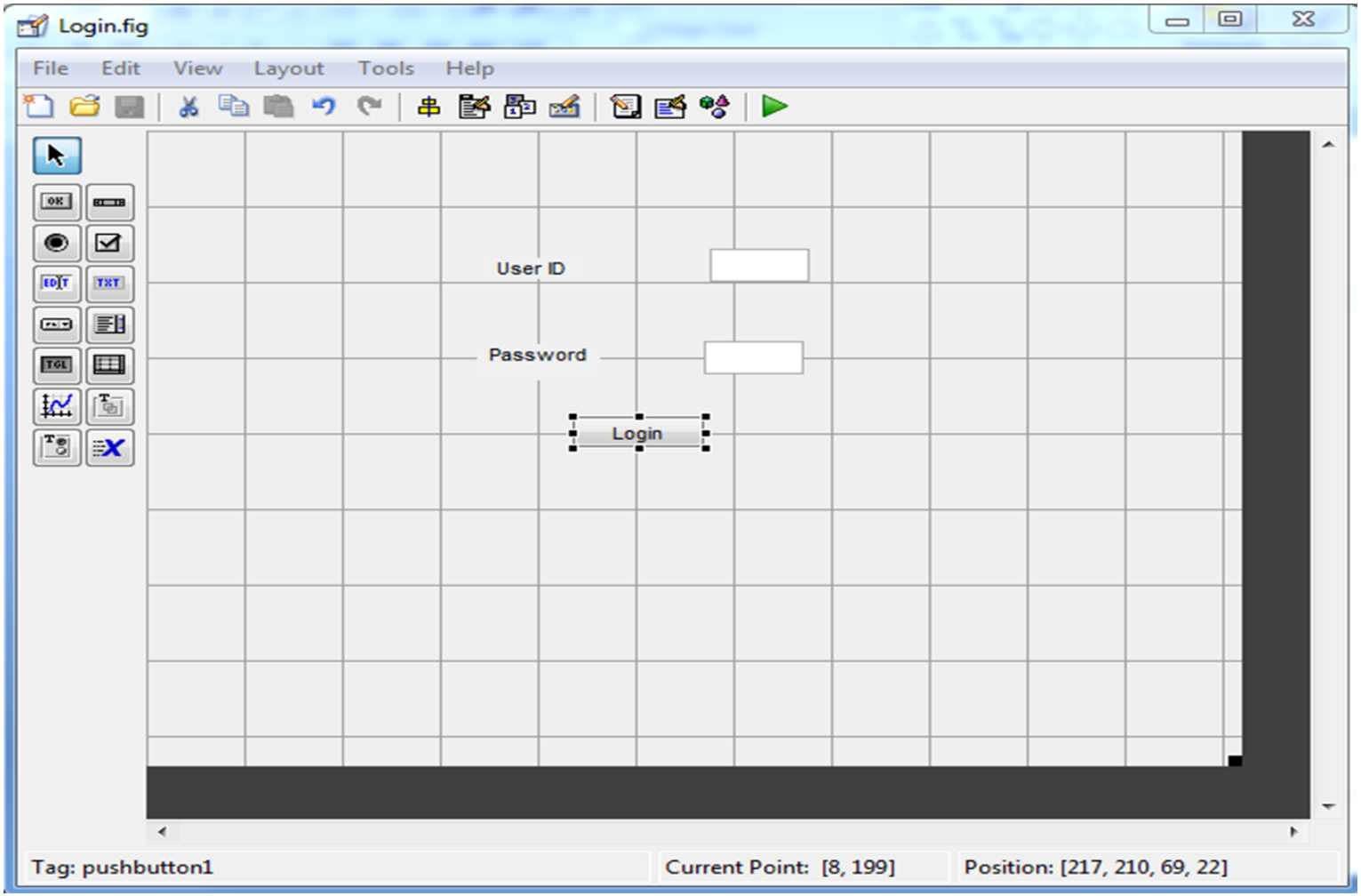
  `» guide`
- Select the **`Blank GUI`** from the pop-up window
- Select and edit the following components form the **`Toolbar`**

  2 Static Text

  2 Edit Text

  1 pushbutton
- Save the GUI as **`login.fig`**

Exercise 10: Login GUI



Exercise 10: 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

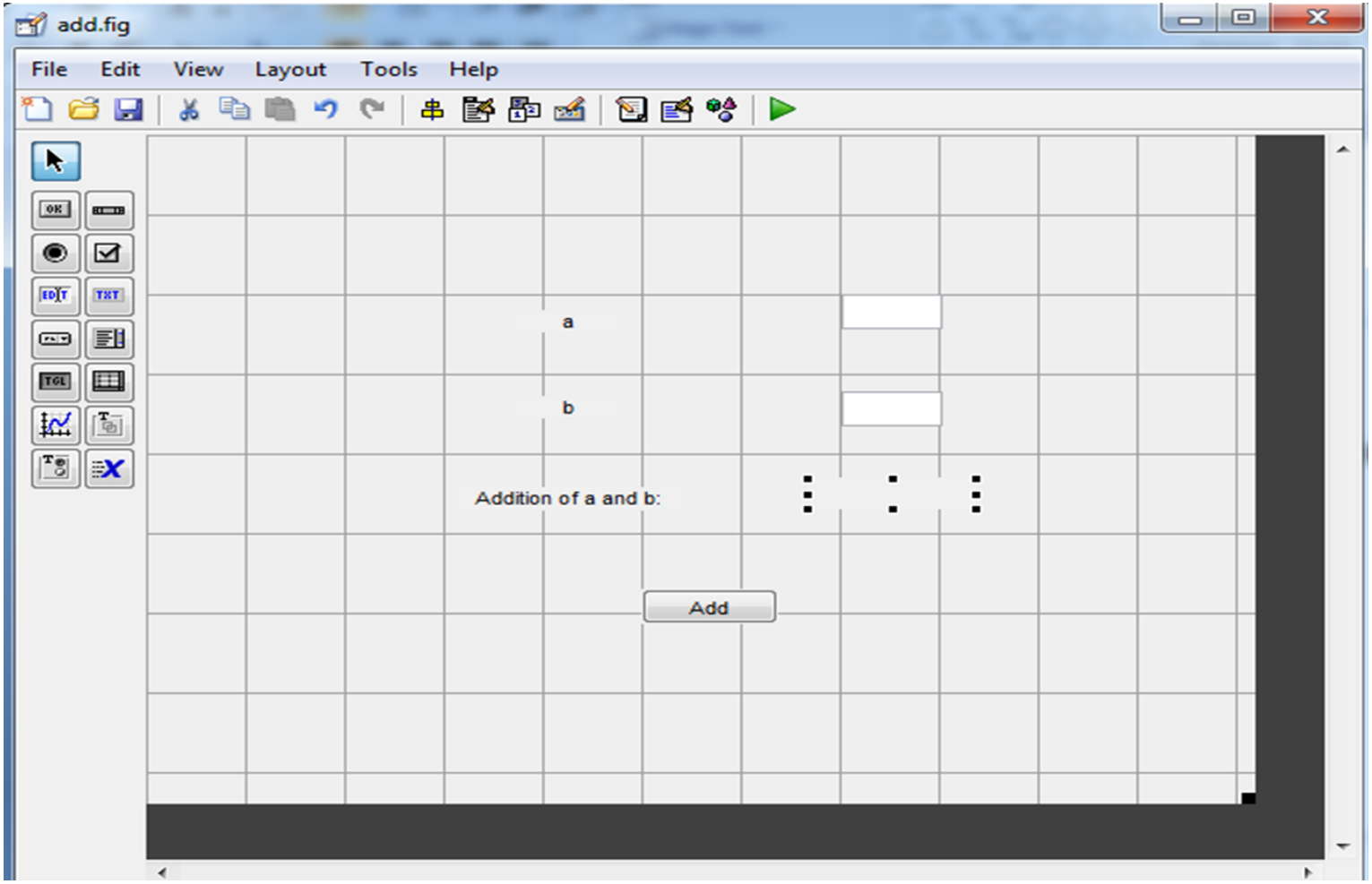
```
else
    msgbox('Wrong Password');
end
```

Exercise 11: 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
  - 3 Edit Text
  - 1 pushbutton
- Save the GUI as `add.fig`

Exercise 11: GUI for Addition



Exercise 11: GUI for Addition

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 12: GUI form Another GUI

- 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
```