Introduction to MATLAB

Ahmad Asadi

Amirkabir University of Technology
Department of Computer Engineering and Information Technology
ahmad.asadi@aut.ac.ir

Spring 2016

Outlines

Quick Look

What we will see

- What is MATLAB?
- Look around MATLAB
- Applications
- How to work with MATLAB
- Graphical User Interface of MATLAB

What is MATLAB?

- A high level programming language being used for technical sophisticated computations
- Everything is matrix
- Stands for: **MAT**rix **LAB**oratory
- Can be assumed as a powerful super calculator
- Matrix based structure $\rightarrow$ awesome to do linear algebra

Note

Matlab is extremely broader than what we will cover in this course. We just want to understand its basics.

Look around MATLAB

Pros

- Fast and easy prototyping
- A wide variety of provided libraries including wide diversity of applications
- Great easy graphical display facilities
- Providing facilities to quickly make a little tiny application
- Quick to learn & efficient to use

Cons

- It seems slow for some sort of programs (we will see them later)
- A program that is just for personal usages (not available on web, not designed for large scale applications, not designed in a multi-user fashion, etc.)

Applications

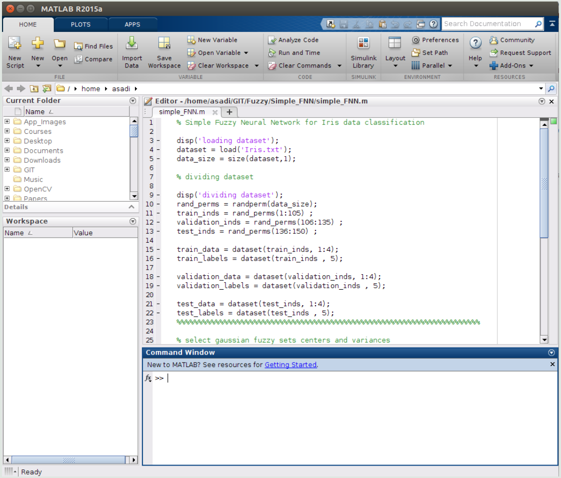
- Math and Computations
- Algorithm Development
- Modeling, Simulation and Prototyping
- Data Analysis, Exploration and Visualization
- Scientific and Engineering Graphics
- Optimized mining operations through modeling and simulation
- Automated data analysis, processing and reporting
- Forecast economical risk and profitability using financial predictive modeling
- Almost, one of the most useful handy applications for engineers and also scientists

How to work with MATLAB?

Big Picture

- Learn Rules (Syntax)
- Decompose interesting problem into simple steps
- Express each step according to MATLAB syntax
- Let MATLAB To do it!

Graphical User Interface (GUI)



Basics

What we will see

- Getting Started!
  - Hello World Again!
  - Simple Calculations
  - Hands on Variables
- Primitive Data Structures
  - Matrices and Vectors
  - Creating Special Vectors
  - Functions to create Matrices
- Operations
  - Matrix Operations
  - Array Operations
- Reading Values of Cells in Matrices and Vectors

Getting Started

Hello World Again

- Using command window
- There is commands for input/output that we will drill into later
- A handy one is
  - disp('this is a message') $\rightarrow$ prints the message string in command window
- The traditional first example!

```
>> disp('Hello World!')
Hello World!
```

Getting Started

Simple Calculations

- You can write any desired expression to be calculated and get its results simply in command window.

```
>> 2 + 3
ans =
5
>> sqrt((pi * 12)^2 / 3 - 57 * cos(pi/3))
ans =
21.007
```

- There exists a complete list of provided functions like *cos*() and *sqrt*() available here in MATLAB documentation: http://mathworks.com/help/matlab/functionlist.html

Getting Started

Hands on Variables

- Variables are named places on memory being used in order to keep a value
- Each variable has a specific data type
- There exists a list of defined data types in MATLAB documents

```
>> a = 2
a =
2
>> b = 3
b =
3
>> c = a + b
c =
5
>> (c * a * b)^2
ans =
900
```

Primitive data structures

Matrices & Vectors

- Almost the most primitive data structures in MATLAB $\rightarrow$ matrices
- Defined as bellow:

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

- Separate rows by ';' and cols by ',' or ' '
- Vectors are special cases of matrices
  - **Row Vector** is an $N * 1$ matrix
  - **Column Vector** is a $1 * M$ matrix
- $size(A)$ returns dimensions of matrix A

Facilities in Creating Vectors

- Creating a vector with equally spaced intervals

```
>> A = 1:0.5:pi
A = 1.0000 1.5000 2.0000 2.5000 3.0000
```

- Creating a vector with *n* equally spaced intervals

```
>> A = linspace(0, pi, 7)
A = 0 0.5236 1.0472 1.5708 2.0944 2.6180 3.1416
```

Note

- MATLAB uses *pi* to represent π and *i* or *j* to represent imaginary unit

Matrices

There is still another useful slide!

There exist a list of useful functions being used to create matrices

- $zeros(m, n)$ creates an $m * n$ matrix of all zeros
- $ones(m, n)$ creates an $m * n$ matrix of all ones
- $eye(m, n)$ creates an $m * n$ identity matrix
- $rand(m, n)$ creates an $m * n$ uniformly distributed randoms
- $randn(m, n)$ creates an $m * n$ normally distributed randoms
- $magic(m)$ creates a square matrix with equal summation of rows, columns and diagonal
- $pascal(m)$ creates a square pascal matrix

Operations

Operations on vectors and matrices are divided into two groups

1. Matrix Operations
   Operands of these kind of operations are matrices as whole.
2. Array Operations
   Operands of these kind of operations are elements of matrices. These kind of operations are being applied to matrices, element by element.

Operations

Matrix Operations

- $+ \rightarrow$ summation
- $- \rightarrow$ subtraction
- $* \rightarrow$ multiplication
- $/ \rightarrow$ division
- $\setminus \rightarrow$ left division$(A \setminus B = INV(A) * B)$
- $\hat{} \rightarrow$ exponentiation

Array Operations

- $.' \rightarrow$ array transpose
- $.\hat{} \rightarrow$ array power
- $.* \rightarrow$ array multiplication
- $./ \rightarrow$ array division

Reading values of a particle matrix

- get value of cell on row 1, col 3 of matrix A

```
>> A(1,2)
```

- get value of cells on row 2, from col 2 to col 5 of matrix A

```
>> A(3,2:5)
```

- get value of cells from row 3 to row 6 on col 3 of matrix A

```
>> A(3:6,3)
```

- get value of cells from row 1 to row 3, from col 2 to row 4 of matrix A

```
>> A(1:3,2:4)
```

Reading values of a particle matrix

- get value of all cells on row 3 of matrix A

```
>> A(3,:)
```

- get value of all cells on col 2 of matrix A

```
>> A(:,2)
```

- get value of all cells of matrix A

```
>> A(:,:)
```

Exercises

You are what you practice more *RichardCarlson*

1. Compute:
   $4[2, 32, 42, 55, 2]^T + (-2)[1, 3, 5, 2, -6]^T + [5, -10, 3, 5, 32]^T$
2. Compute determinant of matrix A without using any function:

$$A = \begin{bmatrix} 12 & 3323 & 411 \\ 30 & -331 & 345 \\ -12.323 & 34.653 & -34 \end{bmatrix}$$

3. Compute determinant of matrix A using $det()$ function for inner $2 * 2$ matrices
4. Do Gauss-Jordan to solve following equations:

$$\begin{cases} x + y + z = 5 \\ 2x + 3y + 5z = 8 \\ 4x + 5z = 2 \end{cases}$$

Solving Equations

What we will see

- How to solve a simple equation
- How to get more than a response from a function
- How to solve a set of equations

Solve a Simple Equation

- Define equations
- Define unknowns with *syms*
- Use the $'=='$ to specify an equation
- Use *solve* method to solve the equations

```
>> syms x
>> eqn = sin(x) == 1;
>> solx = solve(eqn,x)
solx =
pi/2
```

Solve a Simple Equation

- There exists some options for *solve* method
- A useful one $\rightarrow$ *ReturnConditions*
  Returns all solutions under specified conditions

```
>> [solx, params, conds] = solve(eqn, x, 'ReturnConditions', true)
solx =
pi/2 + 2*pi*k

params =
k

conds =
in(k, 'integer')
```

Note

- It is possible to get more than a result from a called function

Solve a Set of Equations

- Use a vector of equations
- Use a vector of unknowns
- Use *solve()* function to solve all of them

```
>> eqn1 = x + y + z == 6;
>> eqn2 = x - 2*y + z == 0;
>> eqn3 = x - z == -2;
>> [solx, soly, solz]=solve([eqn1, eqn2,eqn3],[x,y,z])
solx =
1
soly =
2
solz =
3
```

Structs

- Structs are being used in order to keep a set of different variables packed
- Makes working with a set of different variables more convenient
- Each struct has a unique name (as same as other variables)
- Each variable inside a struct has a unique name
- Value of each inner variables in a struct can be read using '.'

In the example on next page...

- *S* is the name of a struct holding all parameters returned by *solve* function
- The name of inner variables in *S* is the same as their name when returned by *solve* function
- The value of each inner variable is accessible using '.'

Structs

```
>> S = solve([eqn1, eqn2],[x,y,z],'ReturnConditions' , true
    )
S =
x: [1x1 sym]
y: [1x1 sym]
z: [1x1 sym]
parameters: [1x1 sym]
conditions: [1x1 sym]
>> S.x
ans =
4 - z1
>> S.z
ans =
z1
>> S.parameters
ans =
z1
```

Figures and Plots

What we will see

- How to create a new figure
- How to use plot for drawing 2D curves
- How to use scatter for drawing 2D points

Figures

- The instruction *'figure'*, creates an empty window
- The created window will be used by plots
- Each single call on *'figure'* will generate a new window
- It does not take any parameter

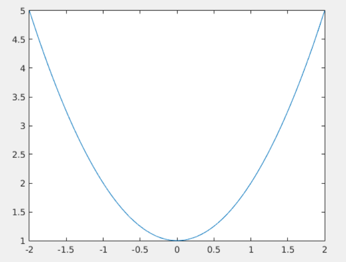
Note that...

- The first figure will be created automatically and does not need to any figure call, but the others should have a figure call in essential cases.

Plot Function

- Plots a line over input points
- For each point the specific (x, y) pair is required
- A separated vector for all x_s and y_s is required

```
>> X = -2:0.0001:2;
>> Y = X.^2 + 1;
>> plot(X,Y)
```

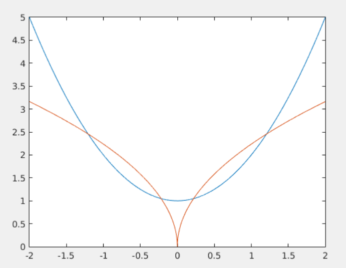


Plot Function

Some options...

- use *'hold on'* to plot more than one curve

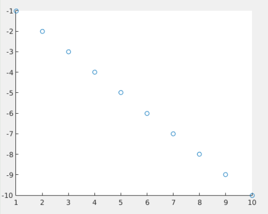
```
>> X = -2:0.0001:2;
>> Y = X.^2 + 1;
>> Z = sqrt(abs(5*X));
>> plot(X,Y)
>> hold on;
>> plot(X,Z)
```



Scatter Function

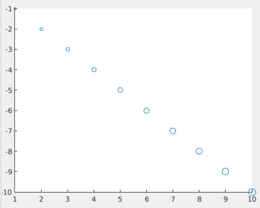
- *Scatter*, draws points given their coordination

```
>> X = 1:10;
>> Y = -X;
>> scatter(X,Y)
```



- The area of each point can be specified by third parameter

```
>> X = 1:10;
>> Y = -X;
>> A = 10*X;
>> scatter(X,Y,A)
```



Exercises

An ounce of practice is generally worth more than a ton of theory $_{Ernst F. Schumacher}$

1. Plot each of following curves in a 3d space (use *plot3* and *scatter3*)
2. Solve the set of equations two by two and find solution points
3. Specify solution points on the plot

$$\begin{cases} sin(x) + cos(y) + z = 3 \\ 2x + 3sin(y) + 5z = 2\pi + 5 \\ 4x + 5z = 4\pi + 5 \end{cases}$$

Programming in MATLAB

What we will see

- Conditional statements
- For Loops
- While Loops

Conditional Statements

- Control the conditions under which some operations needed to be accomplished
- There exist several shapes of statements which are referred to as *'IF' Statements*
- Use *(if-end)* form for statements that are required to be accomplished just in true cases of a certain statement
- Use *(if-else-end)* form for deciding on executing one of two specified statements according to value of the one existing condition
- Use *(if-elseif-end)* form for deciding on executing one of more than two specified statements according to value of the condition corresponding to each statement
- The examples on next pages will through light on the fact.

IF-END

```
>> x = 3;
if(x == 2)
disp('statement 1');
end
>> x = 2;
if(x == 2)
disp('statement 2');
end
statement 2
```

Notes on the example...

- In the above example, the statement inside the first *if* clause won't be executed because there $x \neq 2$.
- On the other hand, in the second *if* clause, as $x == 2$ is true, the inside statement will be executed and *'statement 2'* will be printed on console.

IF-ELSE-END

```
>> x = 3;
if(x == 2)
disp('The X is equal to 2');
else
disp('NOP! X is not equal to 2');
end

NOP! X is not equal to 2

>> x = 2;
if(x == 2)
disp('The X is equal to 2');
else
disp('NOP! X is not equal to 2');
end

The X is equal to 2
```

IF-ELSE-END

Notes on the example...

- In the above example at first attempt, the statement inside the *if* clause won't be executed because $x \neq 2$, therefore, the inner statement of *else* clause will be executed.
- At the second attempt, $x == 2$ so the inner statement of *if* clause will be executed and the left won't be launched.

IF-ELSEIF-END

```
>> x = 3;
if(x == 1)
disp('The X is equal to 1');
elseif (x == 2)
disp('The X is equal to 2');
elseif (x == 3)
disp('The X is equal to 3');
elseif (x == 4)
disp('The X is equal to 4');
else
disp('NOP! X is not equal to 1, 2, 3 or 4');
end

The X is equal to 3
```

IF-ELSEIF-END

Notes on the example...

- In the above example at first attempt, the statement inside the *if* clause won't be executed because $x \neq 1$, therefore, the condition on first *elseif* clause will be checked. As $x \neq 2$, the inner statement of first *elseif* clause would be ignored and the condition of the next *elseif* clause would be checked.
- If no conditions satisfies, the inner statement of *else* clause would be executed.
- The number of *elseif* clauses is unlimited.
- The existence of *else* clause is optional, but at most one *else* clause is allowd.
- At the second attempt, $x == 2$ so the inner statement of *if* clause will be executed and the left won't be launched.

FOR Loops

- Repeat a set of statements a specific number of times
- The general syntax

```
>>
for <counter variable> = <range of values>
<the set of statements need to repeat>
end
```

- *counter variable* proceeds in *range of values* along with repeating the *specified statements*.

FOR Loops

```
>>
for x = 1:5
disp('this is the value of x>>');
disp(x);
end

this is the value of x>>
     1
this is the value of x>>
     2
this is the value of x>>
     3
this is the value of x>>
     4
this is the value of x>>
     5
```

WHILE Loops

- Repeat a set of statements while a specific condition is true
- The general syntax

```
>>
while (<condition>)
<the set of statements need to repeat>
end
```

- The specified condition will be checked, if is true, then all of specified statements inside the *while* will be executed. After each round of execution, the condition will be checked again and the statements will be executed if condition is true. As the condition turns to false, the execution will be stopped and control flow will chase the statements after while.
- The example on next page is an equivalent of the previous example!

WHILE Loops

```
>>
 x = 0;
 while(x < 5)
 x = x + 1;
 disp('This is the value of x>>');
 disp(x);
 end

This is the value of x>>
     1
This is the value of x>>
     2
This is the value of x>>
     3
This is the value of x>>
     4
This is the value of x>>
     5
```

Exercises

Practice puts brain in your muscles. *SamSnead*

1. Given the vector x = [1 8 3 9 0 1], create a short set of commands that will
   a. Add up the values of the elements (Check with sum.)
   b. Computes the running sum (for element j, the running sum is the sum of the elements from 1 to j, inclusive. Check with cumsum.)
   c. computes the sine of the given x-values (should be a vector)
2. Compute the value of pi using the following series

$$\frac{\pi^2-8}{16} = \sum_{n=1}^{\infty} \frac{1}{(2n-1)^2(2n+1)^2}$$

   How many terms are needed to obtain an accuracy of 1e-12? How accurate is the sum of 100 terms of this series?