INTRODUCTION TO MATLAB

Intensive Computation

2015-2016

Annalisa Massini

Introduction

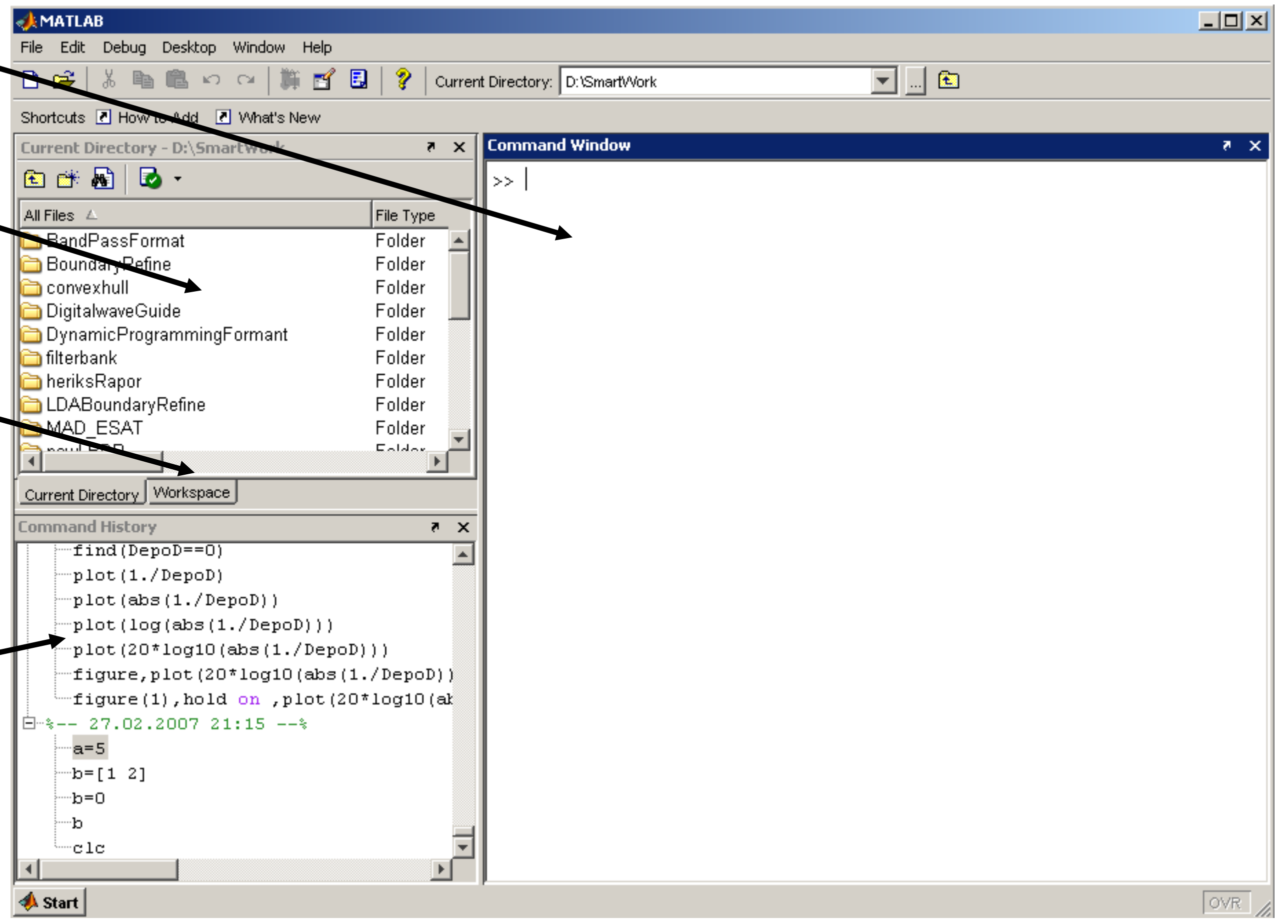
- MATLAB stands for MATrix LABoratory
- MATLAB is a **high-level interpreted language** and **interactive environment** for numerical computation, data analysis, visualisation and algorithm development
- MATLAB enables you to perform computationally intensive tasks faster than with traditional programming languages such as C, C++ and Fortran

Introduction

- MATLAB started its life in the late 1970s as an interactive calculator built on top of LINPACK and EISPACK, which were then **state-of-the-art Fortran subroutine libraries for matrix computation**
- In the **80s Cleve Moler** write the first version of MATLAB to give his students at the University of New Mexico easy access to these libraries without writing Fortran
- Matlab has many specialized **toolboxes**

Matlab Screen

- **Command Window**
  - Type commands
- **Current Directory**
  - View folders and m-files
- **Workspace**
  - View program variables
  - Double click on a variable to see it in the Array Editor
- **Command History**
  - view past commands
  - save a whole session using diary



Helpful commands

- **help** lists all the help topic – *the most **important** function to learn Matlab*
  - help *name* the help text for the functionality specified by *name*, such as a function, method, class, or toolbox
- **who/whos** show the current variables in the workspace
- **dir** list files in the current directory
- **clear all** delete all the variables present in the workspace
  - clear *var1 var2* clear variables *var1* and *var2*
- **lookfor** search for keyword in all help entries
  - lookfor *topic*

Variables and expressions

- In the Command window, the command **prompt** is " **>>** "
- Two types of statement:
  - **evaluation of an expression**

    "**>>** *expression*"
  - **assignment** "**>>** *variable* = *expression*"
- The evaluation of an expression generates a **matrix** assigned to the specified **variable**
- If you do not specify the name of the *variable* associated to the result, the system "***ans***" is used

Examples:

- **>> 8+2**

  ans =

  10

- **>> a = 5*ans**

  a =

  50

- **>> 6.9**

  ans =

  6.9000

Variables and expressions

- If an expression ends with symbol **“;”** its value is not displayed on the screen
- MATLAB names are **case-sensitive**

- **No need to declare variables**
- **No need for types**

Examples:

```
» b = 6+a;

» b
b =
  56
```

- 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’

Variables and expressions

- All variables are created with **double precision**
- The **variables are 1x1 matrices** with double precision
- Double precision values consist of **8 bytes**

- The default display format for variables is 5-digit scaled, fixed-point values
- We can ask for different display formats with command **`format`**
- The **format** function affects only how numbers display in the Command Window, **not how MATLAB computes or saves them**

The command FORMAT

Command **`format`** changes the display format to the specified ***style***

Let us consider x = 4/3

- **`format short`** 1.3333 0.0000 - 5-digit scaled, fixed-point ***default***
- **`format long`** 1.33333333333333 - 15-digit fixed point
- **`format short e`** 1.3333e+000 - 5-digit floating point
- **`format long e`** 1.333333333333333e+000 - 15-digit floating point
- **`format short g`** 1.3333 – best between fixed point and floating point
- **`format long g`** 1.33333333333333 – best between fixed and floating pt
- **`format bank`** 1.33 – currency format (dollar or euro)
- **`format rat`** 4/3 - ratio of small integers
- **`format hex`** 3ff5555555555555 - hexadecimal (double-precision)

Double precision values

- Only a number of double precision values can be represented
- There is always a **small gap** between two consecutive values
- The command `eps` provides the floating-point relative accuracy
- **eps** returns the distance from 1.0 to the next largest double-precision number, that is eps = 2^(-52)
- **eps**(x) is the positive distance from abs(X) to the next larger in magnitude floating point number of the same precision as X
- **realmin** returns the smallest positive normalized floating-point number in IEEE double precision about **2.2251e-308** that is **2^(-1022)**
- **realmax** returns the largest finite floating-point number in IEEE double precision, about **1.7976e+308** that is **2^1023**

Matrices

- The simplest way to create a matrix is to use the matrix constructor operator **[ ]**
- Create a row in the matrix by entering elements within the brackets
- Separate **row elements** with a **comma** or **space**
- For a **new row**, terminate the current row with a **semicolon** or **return**

```
» A = [7  8; 8.9  7; 9  8]

A =
        7.0000    8.0000
        8.9000    7.0000
        9.0000    8.0000
```

```
» B = [1  2  3
4  5  6]

B =
     1     2     3
     4     5     6
```

Matrices

- Examples of **functions** for creating **different kinds of matrices**
  - **zeros(n,m)** matrix *nxm* of all zeros
  - **ones(n,m)** matrix *nxm* of all ones
  - **eye(n,m)** matrix with ones on the diagonal (zeros elsewhere)
  - **rand(n,m)** matrix of uniformly distributed random numbers
  - **diag([a11, a22, a33, ..., aNN])** diagonal matrix
  - **....**

Matrices

- Increase matrices by adding a row or a column having the correct size

- **Column**
  - Given **A = [1 2; 3 4; 5 6];**
  - Add the column of elements 7 8 9

A = [A [7; 8; 9]] oppure **A=[A [7 8 9]']**)

1 2		**1 2 7**
3 4	→	**3 4 8**
5 6		**5 6 9**

Matrices

To access elements of a matrix → matrices' name followed by round brackets containing a reference to the row and column number

- » A = [7 8; 8.9 7; 9 8]

  A =

  7.0000   8.0000
  8.9000   7.0000
  9.0000   8.0000

- **A(n,m)** access **element** (n,m) of matrix A

  » A(1,2)

  ans =

  8

Note that elements of the matrix are displayed as 5-digit values

Matrices

The colon operator

- The colon operator (`first:last`) generates a 1-by-n matrix (or *vector*) of sequential numbers from the first value to the last

- The default sequence is made up of values **incrementing by 1**

  A = 10:15 → A = 10 11 12 13 14 15

- The numeric sequence can include negative and fractional numbers

  A = -2.5:2.5 → A = -2.5000 -1.5000 -0.5000 0.5000 1.5000 2.5000

Matrices

The colon operator

- You can also specify a ***step*** value with the colon operator in between the starting and ending value (`first:step:last`).
- To generate a series of numbers from 10 to 50 incrementing by 5:

  A = 10:5:50 → A = 10 15 20 25 30 35 40 45 50

- You can increment by ***noninteger*** values

  A = 3:0.2:3.8 → A = 3.0000 3.2000 3.4000 3.6000 3.8000

- Yo can ***decrement***, specifying a negative step value:

  A = 9:-1:1 → A = 9 8 7 6 5 4 3 2 1

Matrices

Accessing matrix rows or matrix columns

- **A(n,:)** extracts **row n** of matrix A

» A(2,:)

ans =

8.9000 7.0000

A(:,m) extracts **column m** of matrix A

» A(:,1)

ans =

7.0000

8.9000

9.0000

The colon notation**":"** allows to specify a sequence of values

The whole row (column) is extracted because the interval is not specified

Matrices

diag(A)

- If A is a square matrix, diag(A) returns the main diagonal of A

```
» A=[5 6 ; 7 8]
A =
    5 6
    7 8
```

```
» diag(A)
ans =
    5
    8
```

- If A is a vector with n components, returns an n-by-n diagonal matrix having A as its main diagonal

```
» diag(ans)
ans =
    5 0
    0 8
```

Matrices

- **sum(A)**
- If A is a ***vector***, then sum(A) returns the sum of the elements

```
» sum(A)
ans =
      36
```

- If A is a ***matrix***, then sum(A) treats the columns of A as vectors and returns a row vector whose elements are the sums of each column

```
» A=[0 1 2 ;3 4 5 ;6 7 8 ]
A =
    0 1 2
    3 4 5
    6 7 8
```

```
» B=sum(A)
B =
9 12 15
```

Vectors

- A matrix with only one row or column (that is, a **1-by-n** or **n-by-1** array) is a ***vector***, such as:

C = [1, 2, 3]	row vector
D = [10; 20; 30]	column vector

- An array can be created with the colon operator

x = 1:6 → x = 1 2 3 4 5 6

x = 0.5:0.1:0.7 → x = 0.5000 0.6000 0.7000

Vectors

- A vector can be created by using `linspace(a,b)` or `linspace(a,b,N)` that generates vectors of (N) points linearly spaced between and including a and b

x = linspace(-1,1) → -1 0 1

x = linspace(-1,1,4) → -1.0000 -0.3333 0.3333 1.0000

- The logspace functions – `logspace(a,b)` or `logspace(a,b,N)` –generate *logarithmically spaced vectors*
- The logspace function is useful for creating frequency vectors
- It is a logarithmic equivalent of linspace and the ":" or colon operator

Vector Indexing

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

- 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**

$$\begin{matrix} b(1,1) \rightarrow \\ b(2,1) \rightarrow \end{matrix} \begin{bmatrix} 14 & 32 \\ 11 & 81 \end{bmatrix} \begin{matrix} \leftarrow b(1,2) \\ \leftarrow b(2,2) \end{matrix} \qquad \begin{matrix} b(1) \rightarrow \\ b(2) \rightarrow \end{matrix} \begin{bmatrix} 14 & 32 \\ 11 & 81 \end{bmatrix} \begin{matrix} \leftarrow b(3) \\ \leftarrow b(4) \end{matrix}$$

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 columns143398
```

Matrix Indexing

- 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);
- To convert between subscripts and indices, use **ind2sub** and **sub2ind**

Scalar operators and functions

- Mathematical operators on scalars
  add +, subtract -, divide /, multiply *, power ^

- Trigonometric function
  - sin, cos
  - tan
  - asin, acos
  - atan

The list of ***elementary math functions***

- help **elfun:** trigonometric, esponential, complex, rounding and remainder

The list of ***specialized math functions***

- help **specfun:** specialized, number theoretic, coordinate transforms

Scalar operators and functions

- Some mathematical operators on scalars:
  - **abs** Absolute value and complex magnitude
  - **conj** Complex conjugate
  - **real**, **imag** Real and Imaginary part of complex number
  - **exp** Exponential
  - **log**, **log10** Natural and base 10 logarithm
  - **sqrt** Square root
  - **ceil** Round toward positive infinity
  - **floor** Round toward negative infinity
  - **round** Round to nearest integer

- Variables **i** and **j** are both functions denoting the **imaginary unit** and are the square-root of -1

Matrix operations

Matrix operations:

- + **addition** of vectors or matrices (element-by-element)
- - **subtraction** of vectors or matrices (element-by-element)
- * **multiplication** of vectors or matrices (row-by-column)

Note that:

- **addition / subtraction**: matrices with the same number of rows and columns
- **addition / subtraction** with a **scalar:** the scalar is added/subtracted to each element of the matrix
- **multiplication**: the number of columns in the first matrix must be the same as the number of rows in the second matrix

Matrix operations

Matlab has a set of **dot operators**, a dot and a normal algebraic operator, performing element-wise algebraic operations on a matrix

- .* element-wise product
- ./ element-wise division
- .^ element-wise power

\ and / operators for the **solution of linear systems**:

- x = B/A is the solution of the equation x*A = B
- x = A\B denote the solution to the equation A*x = B

Systems of Linear Equations

- Given a system of linear equations

  x+2y-3z=5

  -3x-y+z=-8

  x-y+z=0

- Construct matrices so the system is described by Ax=b

```
»A=[1 2 -3;-3 -1 1;1 -1 1];
»b=[5;-8;0];
```

- And solve with a single line of code!

```
»x=A\b;
```

  - x is a 3x1 vector containing the values of x, y, and z
- The \ **will work with square or rectangular systems**
  - Gives least squares solution for rectangular systems

Matrix functions

- **Matrix functions**:
  - Transpose matrix **A'**
  - Inverse matrix **inv(A)**
  - Matrix determinant **det(A)**
  - Eigenvalues **eig(A)**
  - Rank of matrix **rank(A)**
  - Dimensions **size(A)**

The list of elementary matrices and matrix manipulation

- help **elmat**: elementary matrices, basic array information, matrix manipulation, special variables e costants, specialized matrices, …

MATLAB Programming

Script and Function

- The simplest type of MATLAB program is called a *script*
- A script is a file that contains multiple sequential lines of MATLAB commands and function calls
- You can run a script by typing its name at the command line
- **Script** and **Function** are **M-files** with a **.m extension**
- **Scripts**
  - have no input or output arguments
  - use workspace data
- **Functions**
  - accept input arguments and produce output
  - have their own workspace, separate from the base workspace
  - function variables are local

MATLAB Programming

You can:

- Add **comments** to code using the percent symbol **%**.
- Create **help text** by inserting **comments at the beginning** of your program.
- Help text appears in the Command Window when you use the help function → **`help`** ***`ProgramName`***

- If your program includes a **function**, position the help text immediately below the function definition line (the line with the *function keyword*)

MATLAB Programming

Function - The definition statement is the first executable line

Each function definition includes:

- `function` keyword (***required***) (lowercase characters)
- Output arguments (***optional***)
  - `function output= myfunction(x)`
  - `function [one,two,three] = myfunction(x)`
  - `function myfun(x)` or `function []=myfunction(x)`
- Function name (***required***)
- Input arguments (***optional***)
  - `function y = myfunction(one,two,three)`

Remark: use the same name for both the file and the function

MATLAB Programming

Example

```
% mean computes the
% mean of a random
% values array and the
% mean among the
% minimum and maximum
v=rand(50,1)
mean=valmean(v)
meanmm=minmax(v)
```

```
function m=valmean(v)
n=length(v)
m=sum(v)/n

function mm=minmax(v)
mini=min(v)
maxi=max(v)
mm=(mini+maxi)/2
```

Relational and logical operators

The **relational operators** are:

- **<, >, <=, >=, ==,** and **~=**

Relational operators perform element-by-element comparisons between two arrays

They return a logical array of the same size, with elements set to:

- logical **1**(true) where the relation is true
- logical **0** (false) where the relation is false

The **logical operators** are:

- **& (and), | (or), ~ (not)**
- **xor (xor), all (all true), any (any true)**

Relational and logical operators

- **Examples**

```
>> a=10; b=3; c=25;

>> a==b
ans=
    0
>> a>b
ans=
    1
>> a+b > c
ans=
    0
```

Programming: loop control

With ***loop control statements***, you can repeatedly execute a block of code

`for` statements loop a specific number of times, and keep track of each iteration with an incrementing index variable

- ```
  for index=starting value:increment:final value
      program statements
  end
  ```

Remark *indent* the loops for readability, especially when they are nested
  ```

Programming: loop control

- **Example**

```
x = ones(1,10);
for n = 2:10
    x(n) = 2 * x(n - 1);
end
```

- **Example**

```
for i=1:m
    for j=1:n
        H(i,j)=1/(i+j-1);
    end
end
```

Programming: loop control

while repeatedly executes one or more program statements in a loop as long as an expression remains true

```
while expression
    statements
end
```

- Expressions can include ***relational operators*** (such as < or ==) and ***logical operators*** (such as &&, ||, or ~)

- To programmatically **exit the loop**, use a **break** statement
- To skip the rest of the instructions in the loop and begin the next iteration, use a **continue** statement

Programming: loop control

Examples

- ```
  x = 3.;
  while x < 25
      x = x + 2
  end
  ```

- Fibonacci
  ```
  a(1)=1; a(2)=1; c=15;
  n=2;
  while a(n) < c
      a(n+1) = a(n) + a(n-1);
      n=n+1;
  end
  ```
  ```

Programming: loop control

- **`if`** ***`expression, statements,`*** **`end`**

  evaluates an expression, and executes the statements when the expression is true
- **`elseif`** and **`else`** are optional, and execute statements only when previous expressions in the if block are false
- An *if block* can include multiple **`elseif`** statements

```
if expression
     statements
elseif expression
    statements
else
    statements
end
```

Programming: loop control

Example

```
if x > 0
   y = sqrt(x);
elseif x == 0
   y = 0;
else
   y = NaN;
  disp('y undefined')
end
```

Programming: loop control

switch case otherwise

Switch among several cases based on expression

```
switch switch_expr
case case_expr
 statements
case {case_expr1,case_expr2,case_expr3,...}
 statements
...
otherwise
 statements
end
```

Programming: loop control

Example

```
name='rose';
switch name
case 'rose'
 disp('the flower is a rose')
case 'tulip'
 disp('the flower is a tulip')
case 'daisy'
 disp('the flower is a daisy')
otherwise
 disp('it's a flower')
end
```

Advanced Data Structures

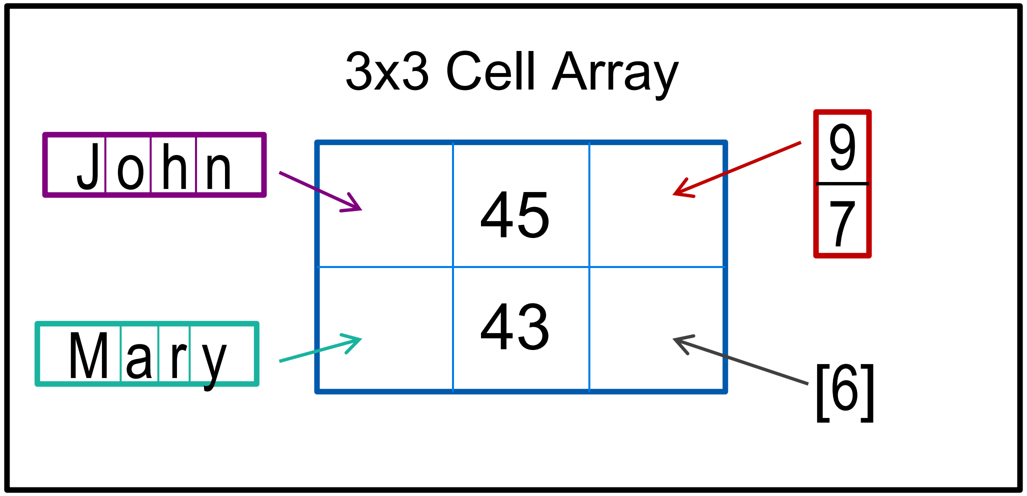
- We have used 2D matrices
  - Can have n-dimensions
  - Every element must be the same type (ex. integers, doubles, characters…)
  - Matrices are space-efficient and convenient for calculation

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**

Cell

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



- One cell can contain people's names, ages, and the ages of their children

Cell

- 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};
```

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***
  - Useful for keeping variables together

Structs

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

»ppl=struct('name',{'John','Mary','Leo'},...

'age',{32,27,18},'childAge',{[2;4],1,[]});

  - size(s2)=1x3
  - every cell must have the same size

»person=ppl(2);

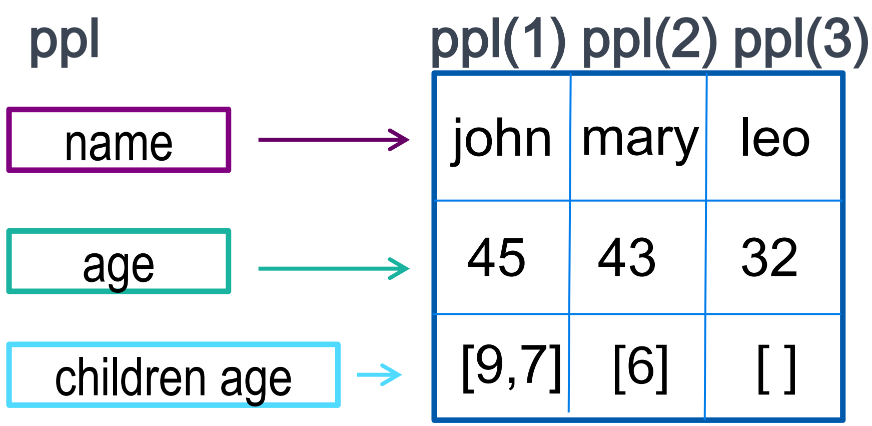
  - person is now a struct with fields name, age, children
  - the values of the fields are the second index into each cell

»person.name

  - returns 'Mary'

»ppl(1).age

  - returns 45



Structs

- To access 1x1 struct fields, give name of the field

  »**stu=s.name;**

  »**scor=s.scores;**

  - 1x1 structs are useful when passing many variables to a function. put them all in a struct, and pass the struct

- To access nx1 struct arrays, use indices

  »**person=ppl(2);**

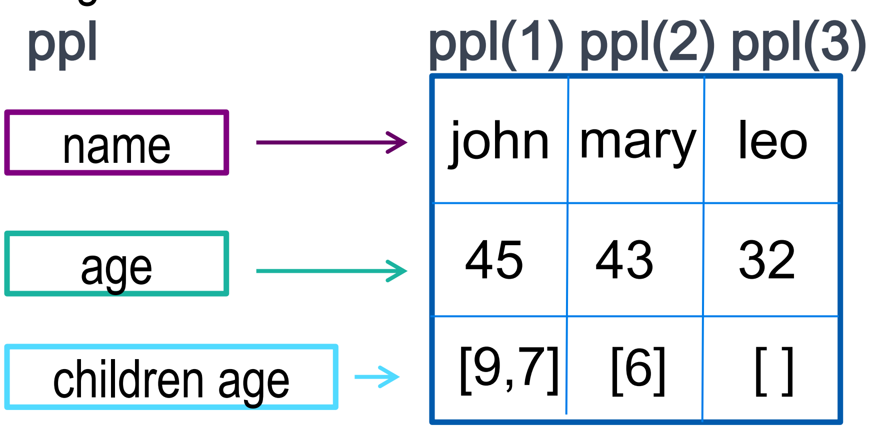
  - person is a struct with name, age, and child age

  »**personName=ppl(2).name;**

  - personName is 'Mary'

  a=[ppl.age];

  - a is a 1x3 vector of the ages



Polynomial

- A polynomial is represented by an array containing the coefficients of the polynom in descending powers of the polynomial decreasing order
- The polynomial $3x^3 + 2x + 8$ can be represented as:

```
» pol= [3 0 2 8]
```

- To evaluate a polynomial in `x`, where x can be a vector, you can use `polyval(p,x)` where `p` is the polynomial

```
» polyval(pol, 1)
   ans =
      13
```

Polynomial

- `roots` computes the roots of the polynomial
- `r=roots(p)` returns a column vector whose elements are the roots of the polynomial `p`
- Row vector `p` contains the coefficients of the polynomial
- Example: the polynomial $x^3 -6 x^2 + 11 x – 6$

```
» p= [1 -6 11 -6]; format long;
» roots(p)
    ans =
     3.00000000000000
     3.00000000000000
     3.00000000000000
```

Polynomial

Remark There are some complications with **multiple roots**

The polynomial $\mathbf{r^3+3r^2+3r+1}$ have just one root **r = -1**, but

```
roots([1 3 3 1])
```

returns three different (though close) values

```
ans =
 -1.00000913968880
 -0.99999543015560 + 0.00000791513186i
 -0.99999543015560 - 0.00000791513186i
```

Even worse for $\mathbf{p(x)=(x+1)^7}$ (coefficients [1 7 21 35 35 21 7 1])

Polynomial

Operations with polynomials

- `p=conv(u,v)` multiplication of the polynomials whose coefficients are the elements of `u` and `v`
- `[q,r]=deconv(u,v)` polynomial division - the quotient is returned in vector `q` and the remainder in vector `r` such that `v = conv(u,q)+r`
- `p=polyfit(x,y,n)` finds the coefficients of a polynomial `p(x)` of degree `n` that fits the data, `p(x(i))` to `y(i)`, in a least squares sense. The result `p` is a row vector of length n+1 containing the polynomial coefficients in descending powers

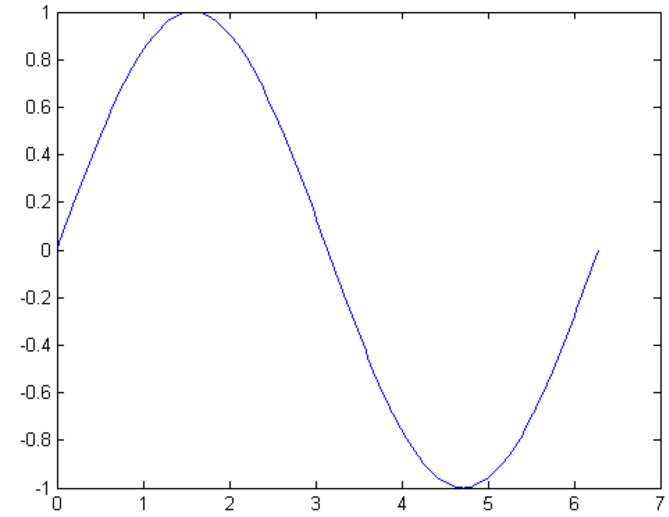
Polynomial

- **`poly`** gives the polynomial with specified roots
- **`p=roots(r)`** where **`r`** is a vector, returns a row vector whose elements are the coefficients of the polynomial whose roots are the elements of **`r`**
- **`p=roots(A)`** where **`A`** is an n-by-n matrix, returns an n+1 element row vector whose elements are the coefficients of the characteristic polynomial, det(λI – A)

Remark **`poly(A)`** generates the characteristic polynomial of **`A`**, and **`roots(poly(A))`** finds the roots of that polynomial, which are the **eigenvalues** of **`A`**

Plotting

The function **`plot`** creates a 2D line plot - it can be used in different ways



- **Example**

```
» n = 31
» x = linspace(0,2*pi,n)
» y = sin(x)
» plot(x,y)
```

x is a vector of linearly spaced values between 0 and 2π

y is the vector of values of sine function evaluated at the values in x

Plotting

- Command **plot** is:
  - **plot(X, Y, options)**

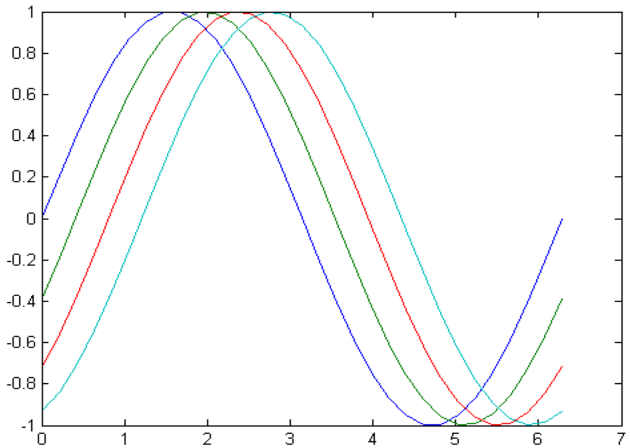
  Where **X** is for abscissas and **Y** is for ordinates

  options sets the *line style*, *marker symbol*, and *color*

- To plot ***multiple lines*** in the same windows, we can use two ways:

```
y2 = sin(x - .4);
y3 = sin(x - .8);
y4 = sin(x - 1.2);
```

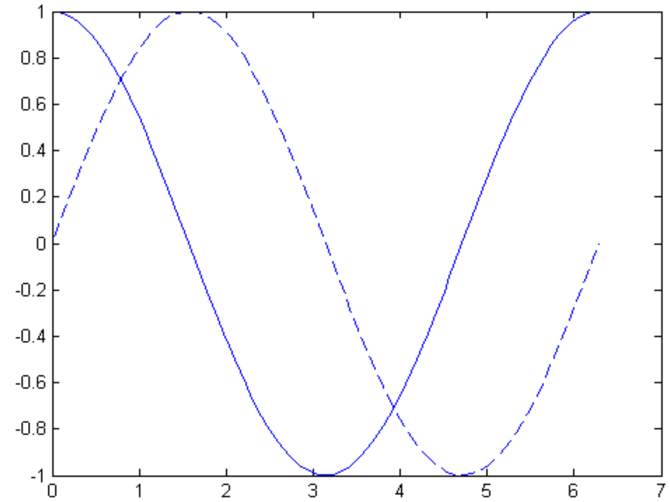
- `plot(x,y,x,y2,x,y3,x,y4)`
- `plot(x,[y;y2;y3;y4])`



Plotting

- Another way to plot **multiple line** in the same window is by using commands **`hold on`** and **`hold off`**:

```
» x = linspace(0,2*pi)
» y1 = cos(x)
» y2 = sin(x)
» plot(x,y1,'-')
» hold on
» plot(x,y2,'--')
» hold off
```



Plotting

- You can add a *title* and and ***axis labels*** to the graph

```
» title('title of the graph')
» xlabel('x axis')
» ylabel('y axis')
```

- **axis** - axis scaling and appearance
- **legend** - graph legend
- **text** - create text object in current axes

```
» text(x(70)+0.5,r(70),'r = -2x')
```

- **grid on** add grid lines for 2D and 3D plots

Plotting

Other functions for graphs are:

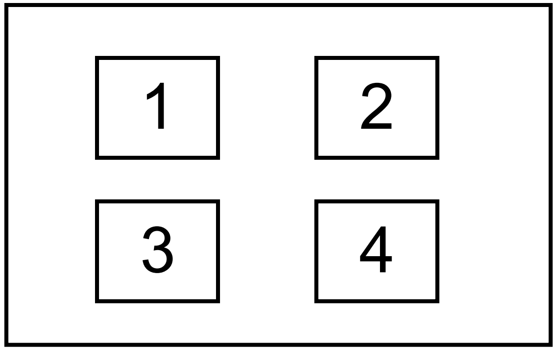
- `loglog` Log-log scale plot
- `semilogx` Semilogarithmic plot (x logarithmic, y linear)
- `semilogy` Semilogarithmic plot (x linear, y logarithmic)
- `errorbar` Plot error bars along curve
- `bar` Bar graph
- `stairs` Stairstep graph
- `scatter` Scatter plot

Plotting

`subplot` divides the current figure into ***grid***, it numbers the cells by rows

» **`subplot(m,n,p)`**

divides the current figure into an **m-by-n** grid and plots in the **grid position** specified by p



Plotting

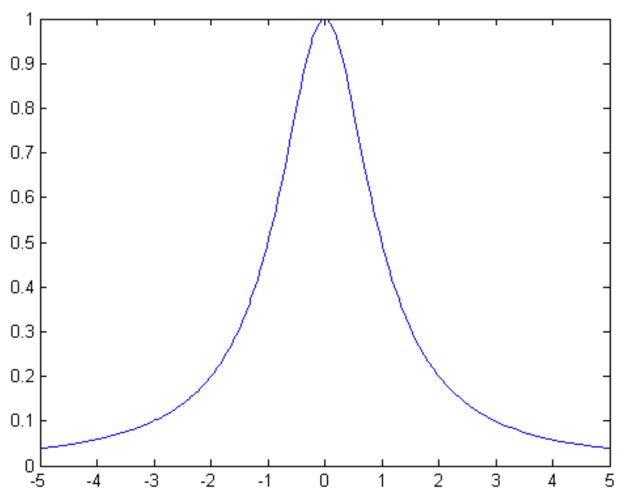
fplot(fun, lims) plots a function

- **fun**, that must be ***a string***
- between the limits specified by **lims**, specifying the ***x-axis limits*** ([xmin xmax]), or the ***x- and y-axes limits***, ([xmin xmax ymin ymax])

```
» fun='1/(1+x^2)';
» lims=[-5,5];
» fplot(fun,lims);
```

or the equivalent

```
» fplot('1/(1+x^2)', [-5,5]);
```



Plotting

- **`fplot(fun,limits,`*`LineSpec`*`)`** plots fun using the line specification ***`LineSpec`***

  `fplot(fun, lims, '- -')`

  `fplot(fun, lims, 'r -')`

- **`fplot`** can plot a vector of functions

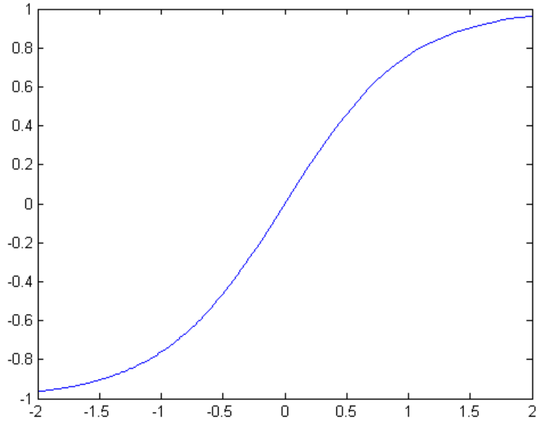
  `fplot('[sin(t), sin(t-.25), sin(t-.5)]',[0,2*pi])`

Plotting

- **ezplot** plots the expression fun(x) over the default domain -2π < *x* < 2π, where fun(x) is an explicit function of only *x*
- **ezplot(fun,[xmin,xmax])** plots fun(x) over the domain: xmin < *x* < xmax

- Both for **fplot** and **ezplot** **fun** can be a **function handle**

```
fh = @tanh;
fplot(fh,[-2,2])
```

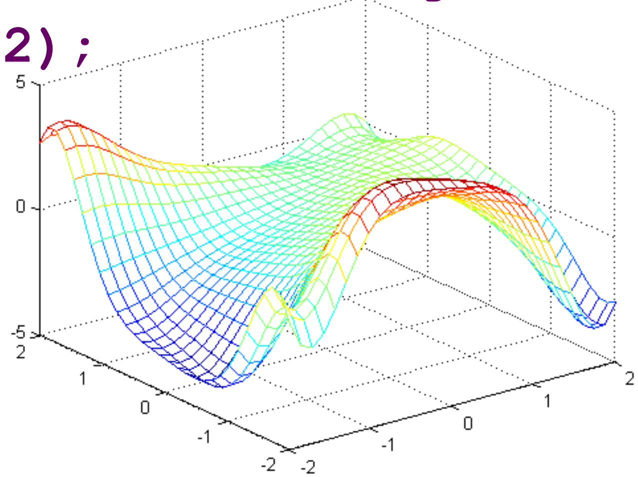


Plotting

3D plot with **mesh** and **surf**

- **mesh** and **surf** plot a surface
- **mesh** and **surf** create 3D surface plots of matrix data generated by the command **meshgrid**

```
» n=30; m=n;
» x=linspace(-2,2,n);
» y=linspace(-2,2,n);
» [X,Y]=meshgrid(x,y); % matrices X e Y for the grid
» Z=(1-Y).*cos(X.^2)+(X-1).*cos(Y.^2);
» mesh(X,Y,Z);
```



Data and file management

You can load variables from file into workspace with **`load`**

For example if you want analyze data coming from a program, like the following, that are in the file data.dat

```
1       0.2000          -5
2       0.2500          -9
3       0.0740         -23
4       0.0310         -53
5       0.0160        -105
6       0.0090        -185
7       0.0050        -299
8       0.0030        -453
9       0.0020        -653
10      0.0020        -905
```

Data and file management

If you load these data with the function **load**, a matrix is created of size 10x3

```
>> load data.dat
>> whos
Name Size Bytes Class
data 10x3 240 double array
Grand total is 30 elements using 240 bytes
```

load filename is the command form

load 'filename' is the function form

Data and file management

```
>> M = load('data.dat')
M =
1.0000  2.0000   -5.0000
2.0000  0.2500   -9.0000
3.0000  0.0740  -23.0000
4.0000  0.0310  -53.0000
5.0000  0.0160 -105.0000
6.0000  0.0090 -185.0000
7.0000  0.0050 -299.0000
8.0000  0.0030 -453.0000
9.0000  0.0020 -653.0000
10.0000 0.0020 -905.0000
```

Data and file management

`save` save workspace variables to file

- **`save (`*`filename`*`)`**
  saves all variables from the current workspace in a formatted binary file (MAT-file) called ***`filename`***
  if ***`filename`*** is not specified the file **`Matlab.mat`** is created

- **`save(filename,variables)`**
  saves only the variables or fields of a structure array specified by variables

- **`save(`*`filename`*`,`*`variables`*`,`*`fmt`*`)`**
  saves in the file format specified by ***`fmt`*** - ***`variables`*** is optional

Data and file management

Example

```
% mytable.m
n=input('Insert the number of values n:');
x=linspace(0,pi,n);
s=sin(x);
c=cos(x);
v=(1:n);
save mytable.dat v x s c -ascii
```

Data and file management

Example

To visualize the table saved in the previous example with save we can load the file and display the table

```
% viewtable.m
load mytable.dat
A=mytable;
disp('--------------------------------------');
fprintf('k\t x(k)\t sin(x(k))\t cos(x(k))\n');
disp('--------------------------------------');
fprintf('%d\t %3.2f\t %8.5f\t %8.5f\n',A);
```

Improving performance

Techniques for Improving Performance

- **Preallocating Arrays**
  - `for` and `while` loops that incrementally increase the size of a data structure each time through the loop can adversely affect performance and memory use
  - resizing arrays often requires MATLAB to spend extra time looking for larger contiguous blocks of memory, and then moving the array into those blocks
  - you can improve code execution time by **preallocating** the maximum amount of space required for the array

Improving performance

Techniques for Improving Performance

- **Preallocating a Nondouble Matrix**
  - When you preallocate a block of memory to hold a matrix of some type other than double, avoid using the method

```
A = int8(zeros(100))
```

  - This statement preallocates a 100-by-100 matrix of `int8`, first by creating a full matrix of double values, and then by converts each element to `int8`

  - Creating the array as int8 values saves time and memory

```
A = zeros(100, 'int8')
```

Improving performance

Techniques for Improving Performance

- **Vectorization**
  - MATLAB is optimized for operations involving matrices and vectors
  - The process of revising loop-based, scalar-oriented code to use MATLAB matrix and vector operations is called ***vectorization***
- Vectorizing your code is worthwhile for several reasons:
  - *Appearance*: Vectorized mathematical code appears more like the mathematical expressions, making the code easier to understand
  - *Less Error Prone*: Without loops, vectorized code is often shorter, and fewer lines of code mean fewer programming errors
  - *Performance*: Vectorized code often runs much faster

Improving performance

- **Vectorizing Code for General Computing**

  - This code computes the sine of 1,001 values ranging from 0 to 10:

```
i = 0;
for t = 0:.01:10
i = i + 1;
  y(i) = sin(t);
End
```

  - This is a vectorized version of the same code:

```
t = 0:.01:10;
y = sin(t);
```

Improving performance

- **Vectorizing Code for Specific Tasks**

  - This code computes the cumulative sum of a vector at every fifth element:

```
x = 1:10000;
ylength = (length(x) - mod(length(x),5))/5;
y(1:ylength) = 0;
for n= 5:5:length(x)
  y(n/5) = sum(x(1:n));
end
```

  - This code shows one way to accomplish the task:

```
x = 1:10000;
xsums = cumsum(x);
y = xsums(5:5:length(x));
```

Improving performance

- **Array Operations**
  - Array operators perform the same operation for all elements in the data set
  - **Example**
    - collect the volume (V) of various cones by recording their diameter (D) and height (H)
    - The volume for that single cone: `V = 1/12*pi*(D^2)*H`
    - Consider 10,000 cones
    - The vectors D and H each contain 10,000 elements

```
for n = 1:10000
    V(n) = 1/12*pi*(D(n)^2)*H(n));
end
```

  - Vectorized Calculation

```
V = 1/12*pi*(D.^2).*H;
```

More examples

Use built-in Matlab functions

- **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>0.4returns a vector with 1 where true and 0 where false
    - x<0.6returns a similar vector
    - The & combines the two vectors using an **and**
    - The find returns the indices of the 1's

More examples

- 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));
```

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