INTRODUCTION TO MATLAB

Gerardo Rodriguez
Universidad de Salamanca

MATLAB (**MAT**rix **LAB**oratory) is an interactive software system for numerical computations and graphics. As the name suggests, MATLAB is especially designed for matrix computations: solving systems of linear equations, factoring matrices, and so forth. In addition, it has a variety of graphical capabilities, and can be extended through programs written in its own programming language.

MATLAB is built around the MATLAB language. The simplest way to execute this code is to type it in at the prompt,>>, in the Command Window, one of the elements of the MATLAB Desktop. Sequences of commands can be saved in a text file, typically using the MATLAB Editor, as a script or encapsulated into a function, extending the commands available.

In the following sections, you will have an introduction to some of the most useful features of MATLAB. There are plenty of examples. The best way to learn to use MATLAB is to read this while running MATLAB, trying the examples and experimenting.

Entering vectors and matrices.

The basic data type in MATLAB is an *n*-dimensional array of double precision numbers. The new data types include structures, classes, and "cell arrays", which are arrays of possibly different data types.

The following commands show how to enter numbers, vectors and matrices, and assign them to variables

```
>> a = 2  //scalar
```

If you press enter, you will see:

```
a =
     2
```

```
>> x = [1;2;3] //Vector
```

Press enter,

```
x =
     1
     2
     3
```

```
>> A = [1 2 3;4 5 6;7 8 0] //Matrix
```

Press enter,

```
A =
     1     2     3
     4     5     6
     7     8     0
```

Notice that the rows of a matrix are separated by semicolons, while the entries on a row are separated by spaces (or commas).

A useful command is "whos", which displays the names of all defined variables and their types:

```
>> whos
  Name      Size         Bytes  Class

  A         3x3             72  double array
  a         1x1              8  double array
  x         3x1             24  double array

Grand total is 13 elements using 104 bytes
```

Note that each of these three variables is an array; the "shape" of the array determines its exact type. The scalar 'a' is a 1×1 array, the vector 'x' is a 3×1 array, and the matrix 'A' is a 3×3 array (see the "size" entry for each variable).

One way to enter a *n*-dimensional array (*n*>2) is to concatenate two or more (*n*-1)-dimensional arrays using the `cat` command. For example, the following command concatenates two 3×2 arrays to create a $3\times 2\times 2$ array:

```
>> C = cat(3,[1,2;3,4;5,6],[7,8;9,10;11,12])

C(:,:,1) =
     1     2
     3     4
     5     6

C(:,:,2) =
     7     8
     9    10
    11    12

>> whos

  Name      Size         Bytes  Class

  A         3x3             72  double array
  C         3x2x2           96  double array
  a         1x1              8  double array
  x         3x1             24  double array

Grand total is 25 elements using 200 bytes
```

Note that the argument "3" in the `cat` command indicates that the concatenation is to occur along the third dimension. If *D* and *E* were $k\times m\times n$ arrays, the command

```
>> cat(4,D,E)
```

would create a $k \times m \times n \times 2$ array (you can try it).

MATLAB allows arrays to have complex entries. The complex unit $i = \sqrt{-1}$ is represented by either of the built-in variables `i` or `j`:

```
>> sqrt(-1)

ans =
        0 + 1.0000i
```

This example shows how complex numbers are displayed in MATLAB; it also shows that the square root function is a built-in feature.

The result of the last calculation not assigned to a variable is automatically assigned to the variable `ans`, which can then be used as any other variable in subsequent computations. Here is an example:

```
>> 100^2-4*2*3 //Press enter

ans =
        9976

>> sqrt(ans) //Press enter

ans =
   99.8799

>> (-100+ans)/4 //Press enter

ans =
    -0.0300
```

The arithmetic operators work as expected for scalars. A built-in variable that is often useful is π:

```
>> pi  //Press enter

ans =
    3.1416
```

Some common useful functions, such as sine, cosine, tangent, exponential, and logarithm are pre-defined. For example:

```
>> cos(.5)^2+sin(.5)^2 //Press enter

ans =
     1

>> exp(1)  //Press enter

ans =
    2.7183

>> log(ans)  //Press enter

ans =
     1
```

If you have any doubts about any command or function an extensive online help system can be accessed by commands of the form `help <command-name>`. For example:

```
>> help ans

 ANS    The most recent answer.
        ANS is the variable created automatically when expressions
        are not assigned to anything else. ANSwer.

>> help pi

 PI     3.1415926535897....

 PI = 4*atan(1) = imag(log(-1)) = 3.1415926535897....
```

A good place to start is with the command `help help`, which explains how the help systems works, as well as some related commands. Typing `help` by itself produces a list of topics for which help is available; looking at this list we find the entry "elfun--elementary math functions". Typing `help elfun` produces a list of the math functions available.

Arithmetic operations on matrices.

MATLAB can perform the standard arithmetic operations on matrices, vectors, and scalars: addition, subtraction, and multiplication. In addition, MATLAB defines a notion of matrix division as well as "vectorized" operations. All vectorized operations (these include addition, subtraction, and scalar multiplication, as explained below) can be applied to *n*-dimensional arrays for any value of *n*, but multiplication and division are restricted to matrices and vectors ($n \leq 2$).

Standard operations.

If `A` and `B` are arrays, then MATLAB can compute `A+B` and `A-B` *when these operations are defined*. For example, consider the following commands:

```
>> A = [1 2 3;4 5 6;7 8 9];
>> B = [1 1 1;2 2 2;3 3 3];
>> C = [1 2;3 4;5 6];

>> whos

  Name      Size         Bytes  Class

  A         3x3             72  double array
  B         3x3             72  double array
  C         3x2             48  double array

Grand total is 24 elements using 192 bytes

>> A+B
ans =
     2     3     4
     6     7     8
```

```
    10    11    12
```

But if you type:

```
>> A+C

??? Error using ==> + because Matrix dimensions must agree.
```

Matrix multiplication is also defined:

```
>> A*C

ans =
    22    28
    49    64
    76   100

>> C*A

??? Error using ==> * because Matrix dimensions must agree.
```

If `A` is a square matrix and `m` is a positive integer, then `A^m` is the product of `m` factors of `A`.

However, no notion of multiplication is defined for multi-dimensional arrays with more than 2 dimensions:

```
>> C = cat(3,[1 2;3 4],[5 6;7 8])

C(:,:,1) =
     1     2
     3     4

C(:,:,2) =
     5     6
     7     8

>> D = [1;2]

D =
     1
     2

>> whos
  Name      Size         Bytes  Class

  C         2x2x2           64  double array
  D         2x1             16  double array

Grand total is 10 elements using 80 bytes

>> C*D
??? Error using ==> *

No functional support for matrix inputs.
```

By the same token, the exponentiation operator `^` is only defined for square 2-dimensional arrays (matrices).

Solving matrix equations using “matrix division”.

If A is a square, nonsingular matrix, then the solution of the equation $Ax = b$ is $x = A^{-1}b$. MATLAB implements this operation with the backslash operator:

```
>> A = rand(3,3)

A =
    0.2190    0.6793    0.5194
    0.0470    0.9347    0.8310
    0.6789    0.3835    0.0346

>> b = rand(3,1)

b =
    0.0535
    0.5297
    0.6711

>> x = A\b

x =
 -159.3380
  314.8625
 -344.5078
```

Note: the use of the built-in function `rand`, which creates a matrix with entries from a uniform distribution on the interval (0,1). (See `help rand` for more details.)

This `A\b` is (mathematically) equivalent to multiplying *b* on the left by A^{-1} (however, MATLAB does *not* compute the inverse matrix; instead it solves the linear system directly). When used with a non-square matrix, the backslash operator solves the appropriate system in the least-squares sense; see `help slash` for details.

Of course, as with the other arithmetic operators, the matrices must be compatible in size. The division operator is not defined for *n*-dimensional arrays with *n*>2.

“Vectorized” functions and operators.

MATLAB has many commands to create special matrices; the following command creates a row vector whose components increase arithmetically:

```
>> t = 1:5
t =
     1     2     3     4     5
```

The components can change by non-unit steps:

```
>> x = 0:.1:1

x =
  Columns 1 through 7
         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000
  Columns 8 through 11
```

```
    0.7000    0.8000    0.9000    1.0000
```

A negative step is also allowed.

The command `linspace` has similar results. It creates a vector with linearly spaced entries. Specifically, `linspace(a,b,n)` creates a vector of length *n* with entries $a, a+\frac{b-a}{n-1}, a+\frac{2(b-a)}{n-1}, ..., b$:

```
>> linspace(0,1,11)

ans =
  Columns 1 through 7
         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000
  Columns 8 through 11
    0.7000    0.8000    0.9000    1.0000
```

There is a similar command `logspace` for creating vectors with logarithmically spaced entries:

```
>> logspace(0,1,11)

ans =
  Columns 1 through 7
    1.0000    1.2589    1.5849    1.9953    2.5119    3.1623    3.9811
  Columns 8 through 11
    5.0119    6.3096    7.9433   10.0000
```

See `help logspace` for details.

A vector with linearly spaced entries can be regarded as defining a one-dimensional grid, which is useful for graphing functions. To create a graph of $y = f(x)$ and connect them with line segments, one can create a grid in the vector `x` and then create a vector `y` with the corresponding function values.

It is easy to create the needed vectors to graph a built-in function, since MATLAB functions are *vectorized*. This means that if a built-in function such as sine is applied to a array, the effect is to create a new array of the same size whose entries are the function values of the entries of the original array. For example (see Figure 1):

```
>> x = (0:.1:2*pi);
>> y = sin(x);
>> plot(x,y)
```

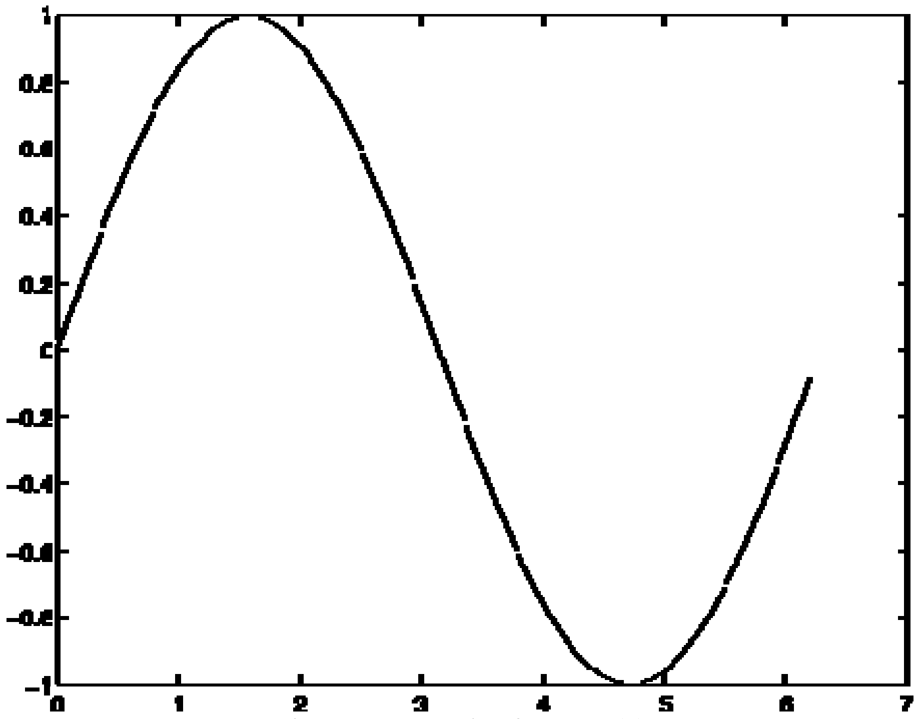


Figure 1: Graph of $y = sin(x)$

MATLAB also provides vectorized arithmetic operators, which are the same as the ordinary operators, preceded by "**.**". For example, to graph $y = \frac{x}{1+x^2}$:

```
>> x = (-5:.1:5);
>> y = x./(1+x.^2);
>> plot(x,y)
```

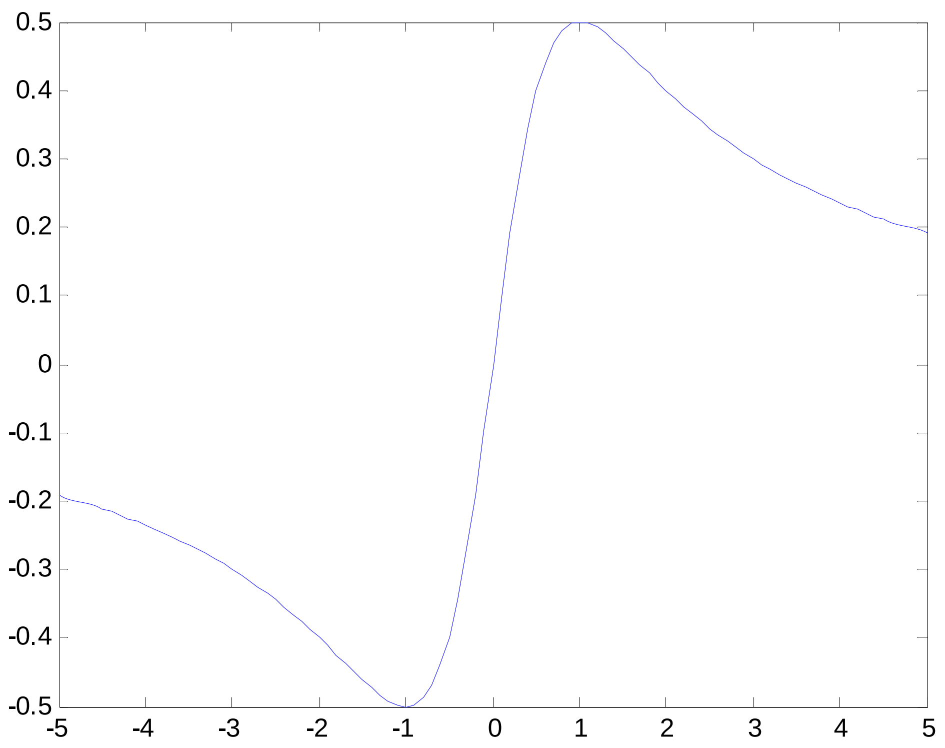


Figure 2: Graph of **y = x / (1 + x^2)**

This `x.^2` squares each component of `x`, and `x./z` divides each component of `x` by the corresponding component of `z`. Addition and subtraction are performed component-wise by definition, so there are not ".+" or ".-"operators.

Note the difference between `A^2` and `A.^2`. The first is only defined if `A` is a square matrix, while the second is defined for any *n*-dimensional array `A`.

Some miscellaneous commands.

An important operator in MATLAB is the single quote " `'` ", which represents the (conjugate) transpose:

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

A =
     1     2
     3     4

>> A'

ans =
     1     3
     2     4

>> B = A + i*.5*A

B =
   1.0000 + 0.5000i   2.0000 + 1.0000i
   3.0000 + 1.5000i   4.0000 + 2.0000i

>> B'
ans =
   1.0000 - 0.5000i   3.0000 - 1.5000i
   2.0000 - 1.0000i   4.0000 - 2.0000i
```

In the rare event that the transpose, rather than the conjugate transpose, is needed, the "**.**'" operator is used:

```
>> B.'

ans =
   1.0000 + 0.5000i   3.0000 + 1.5000i
   2.0000 + 1.0000i   4.0000 + 2.0000i
```

(note that `'` and `.'` are equivalent for matrices with real entries).

The following commands are frequently useful. More information can be obtained from the on-line help system.

Creating matrices.

- `zeros(m,n)` creates an $m \times n$ matrix of zeros;
- `ones(m,n)` creates an $m \times n$ matrix of ones;
- `eye(n)` creates the $n \times n$ identity matrix;
- `diag(v)` (`v` is an *n*-vector) creates an $n \times n$ diagonal matrix with `v` on the diagonal.

For example:

```
>> ones(3,4)

ans =
     1     1     1     1
     1     1     1     1
     1     1     1     1
>> v=[-1 2 3.5]
v =
   -1.0000    2.0000    3.5000

>> diag (v)
ans =
   -1.0000         0         0
         0    2.0000         0
         0         0    3.5000
```

The commands `zeros` and `ones` can be given any number of integer arguments; with *k* arguments, they each create a *k*-dimensional array of the indicated size.

Formatting display and graphics.

The following commands supply different appearances to the distinct outputs.

- `format` : Set output format

```
>> format short, pi
ans =
    3.1416
>> format short e, pi
ans =
      3.1416e+000
>> format long, pi
ans =
      3.14159265358979
>> format long e, pi
ans =
    3.141592653589793e+000
```

`format compact` suppresses extra line feeds (all of the output in this paper is in compact format).

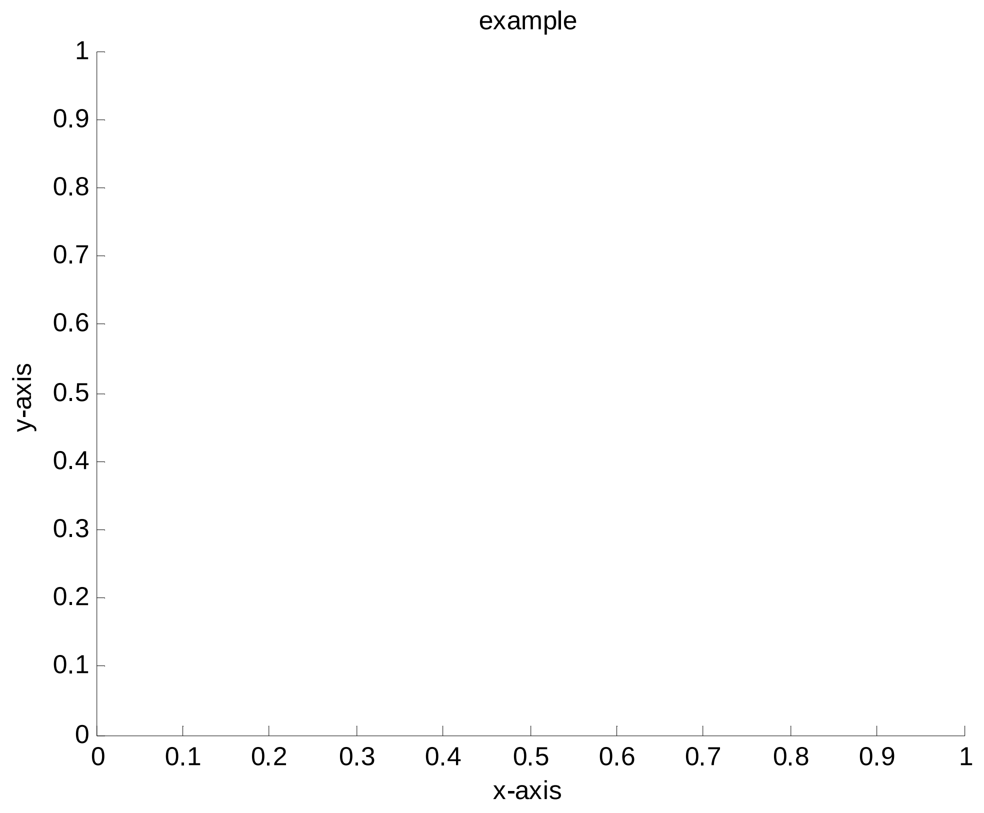
`format loose` puts the extra line-feeds back in.

```
>> format loose, pi



ans =
    3.1416
```

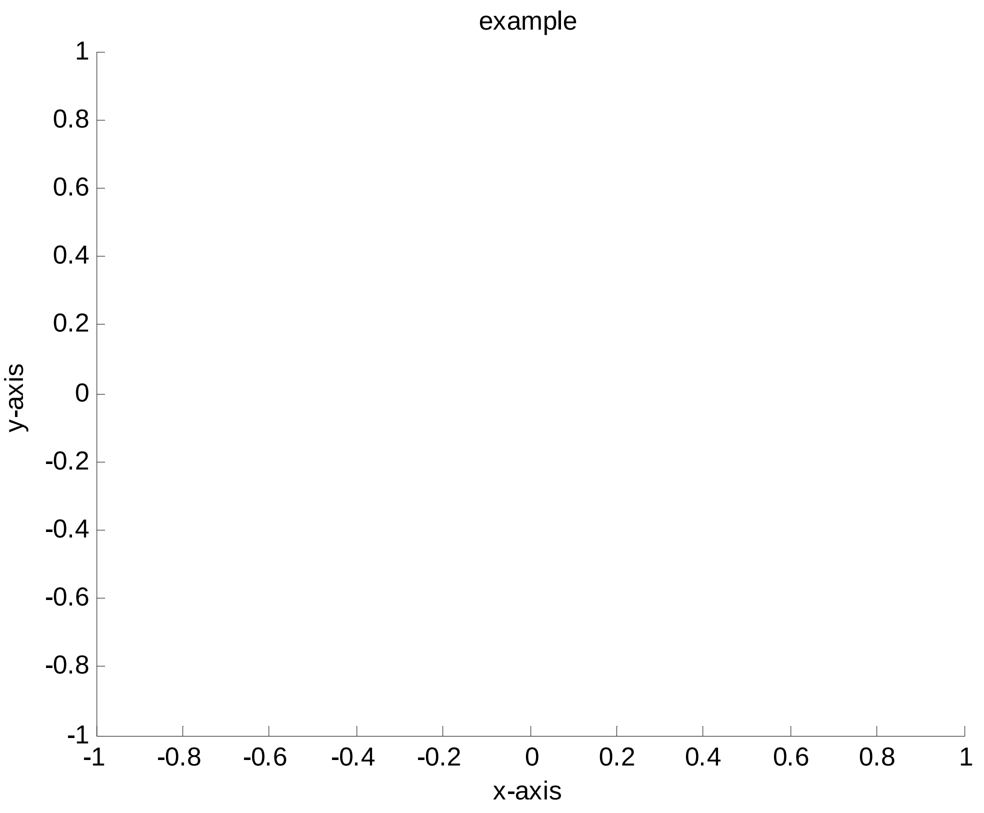
- `xlabel('string'), ylabel('string')` label the horizontal and vertical axes, respectively, in the current plot;
- `title('string')` add a title to the current plot;
- `axis([a b c d])` change the window on the current graph to $a \le x \le b, c \le y \le d$;
- `grid` adds a rectangular grid to the current plot;
- `hold on` freezes the current plot so that subsequent graphs will be displayed with the current;
- `hold off` releases the current plot; the next plot will erase the current before displaying;
- `subplot` puts multiple plots in one graphics window.

For example:

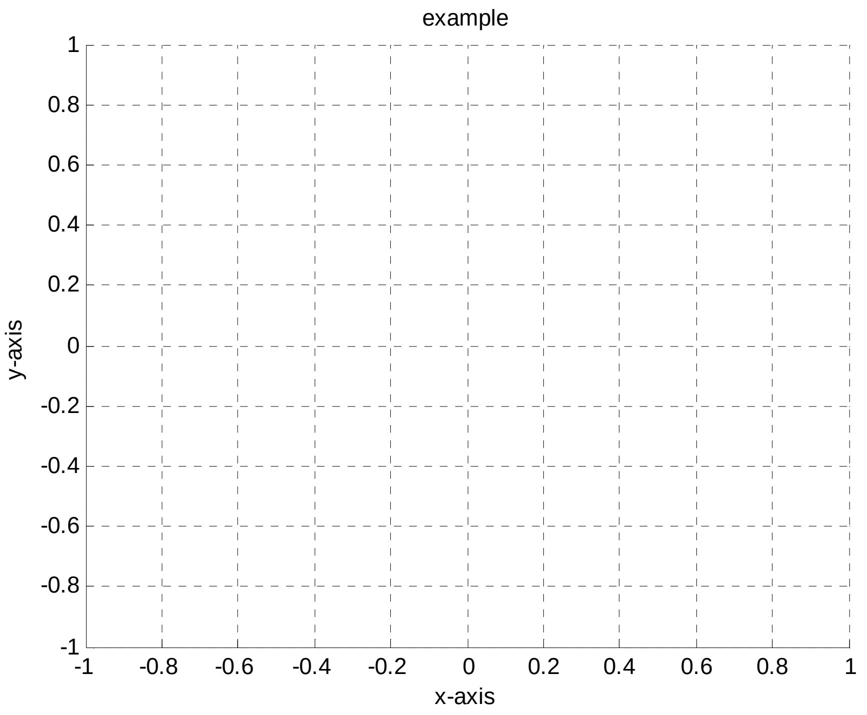
```
>> xlabel('x-axis');
>> ylabel('y-axis');
>> title('example');
```



```
>> axis([-1 1 -1 1]);
```

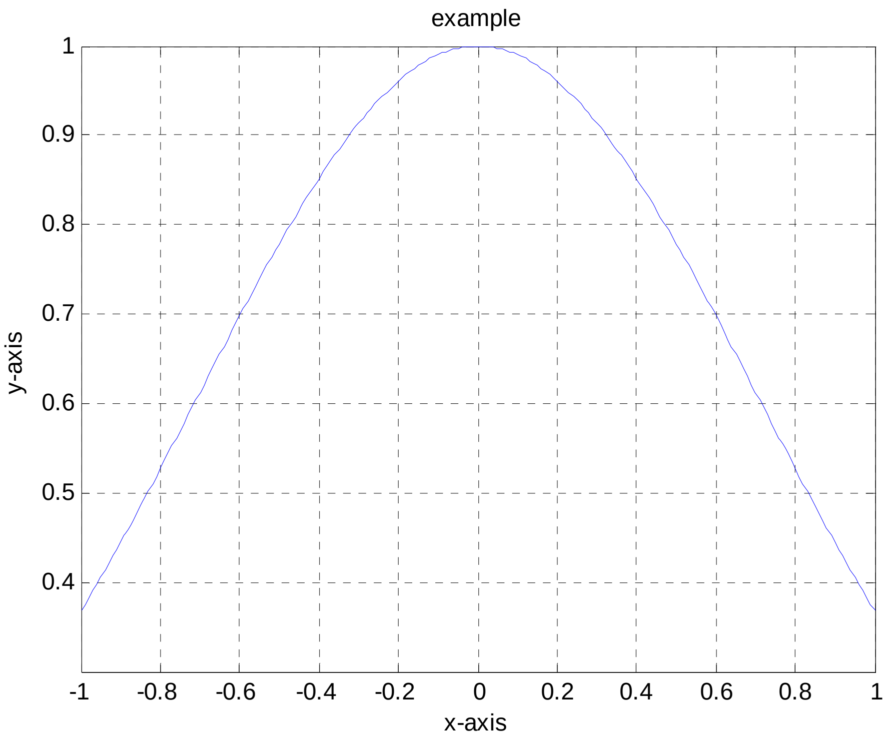


```
>> grid on
```



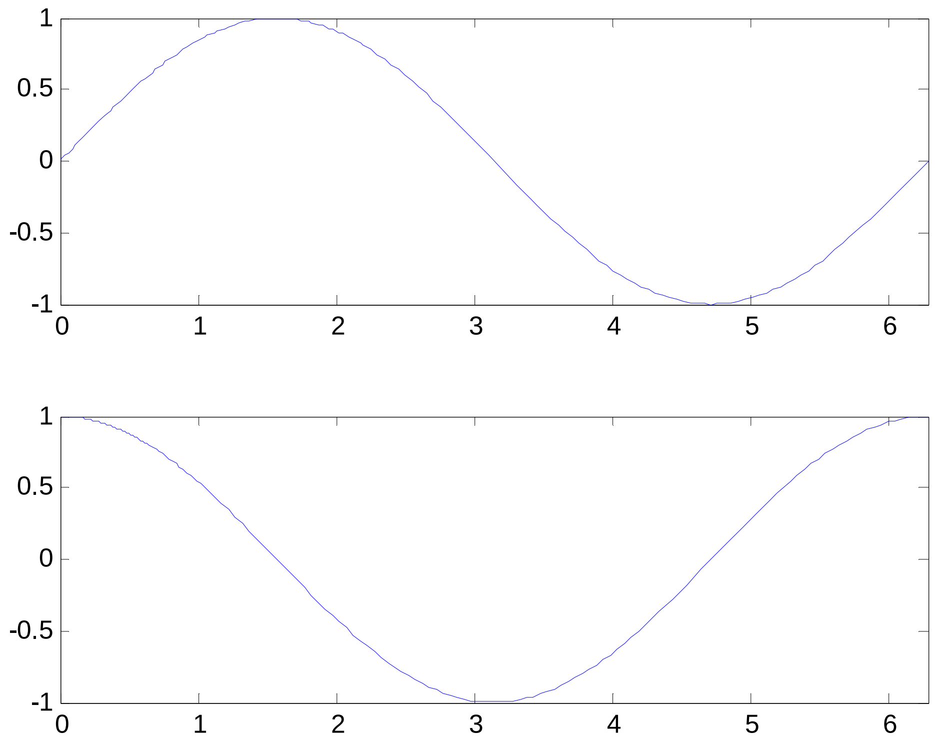
The function y= exp (-x^2) will be drawn, with name at the axis and grid.

```
>> u=x.^2;
>> y= exp(-u);
>> plot (x,y);
```



Now, the `subplot` function will be used to visualize different functions at the same time.

```
>> subplot(2,1,1);
>> fplot('sin(x)',[0 2*pi])
>> subplot(2,1,2);
>> fplot('cos(x)',[0 2*pi])
```



Miscellaneous

- `max(x)` returns the largest entry of x, if x is a vector; see `help max` for the result when x is a *k*-dimensional array;
- `min(x)` analogous to `max`;
- `abs(x)` returns an array of the same size as `x` whose entries are the magnitudes of the entries of `x`;
- `size(A)` returns a $1 \times k$ vector with the number of rows, columns, etc. of the *k*-dimensional array `A`;
- `length(x)` returns the ``length'' of the array, i.e. `max(size(A))`.
- `save fname` saves the current variables to the file named fname.mat;
- `load fname` load the variables from the file named fname.mat;
- `quit` exits MATLAB

For example:

```
>> x=[ 3 -4 60 -71 -13 12];
>> max(x)
ans =
    60
>> min(x)
ans =
   -71
>> abs(x)
ans =
     3     4    60    71    13    12
```

One example: Numerical integration.

Numerical integration is the approximate computation of integral using numerical techniques. There are a wide range of methods available for numerical integration. In this example three of them will be used: the rectangle rule, the trapezoidal rule and Simpson’s rule to approximate the integral:

$$\int_0^1 (1-x)^{1/2}\,dx$$

1.- Rectangle rule.

A first, very simple, scheme is to approximate the function $f(x)$ on each interval by a constant, and considering the area of the rectangle of width $h = x_{i+1} - x_i$ and height $f(x_i)$ for the i^{th} interval. It is then a simple matter to compute the sum of the areas of all the rectangles in the interval.
The interval will be divided into 2, 4, 8, 16, 32, 64 and 128 subintervals.
First an example with 2 subintervals:

Three points in the interval (0,1) are necessary.

```
>> X=linspace(0,1,3)

X =
         0    0.5000    1.0000
```

The values of the function $\sqrt{1-x}$ are saved in the variable *Y*.

```
>> Y=sqrt(1-X)

Y =
    1.0000    0.7071         0
```

The first rectangle has height *f(0),* so the first point of the vector *Y* is saved.

```
>> q=Y(1)

q =
     1
```

The second rectangle has height *f(1),* so the second point of the vector *Y* is saved.

```
>> q=q+Y(2)

q =
    1.7071
```

The width of the rectangles is 0.5, so the sum of the heights is multiplied by this value to calculate the area.

```
>> q=q*1/2

q =
    0.8536
```

In order to extend the number of subintervals, a loop is necessary to manage the vectors. 129 points are necessary to divide de interval in 128 subintervals.

```
>> X=linspace(0,1,129);
```

The values of the function $\sqrt{1-x}$ are saved in the vector *Y* again.

```
>> Y=sqrt(1-X);
>> q=Y(1);
```

In this case, the sum is from *f(0)* to *f(127)*, that is from the first component *Y(1)* to the next to last *Y(128).*

```
>> for j=1:127
q=q+Y(j+1);
end
>> q=q*1/128;
```

Finally, the following program will be used to calculate the integral using 2, 4, 8, 16, 32, 64 and 128 subintervals:

```
function rectangleRule=rectangleRule(Y)
q=zeros(7,1);
N=2; %Number of subintervals
for i=1:7
    h=1/N;
    q(i)=Y(1);
    for j=1:N-1
        q(i)=q(i)+Y(j*(256/N)+1);
    end
    q(i)=q(i)*h;
    N=N*2;
end
disp('Approximations with the rectangle rule:')
      q
```

The input of this function is a vector of 256 components with the function to integrate values in the interval [0,1].

```
>> X=linspace(0,1,257);
>> Y=sqrt(1-X);
>> rectangleRule(Y)
```

```
Approximations with the rectangle rule:

q =
   0.85355339059327
   0.76828304624275
   0.72063022162445
   0.69483119687723
   0.68118393627894
   0.67408331137851
   0.67043190729683
```

2. - Trapezoidal Rule.

In this example, the last program is modified to approximate the integral with the trapezoidal rule.
The following program will be used:

```
function trapezoidalRule=trapezoidalRule(Y)
q=zeros(7,1);
N=2; %Number of subintervals
for i=1:7
   h=1/N;
   q(i)=0.5*Y(1);
   q(i)=q(i)+0.5*Y(257);
   for j=1:N-1
      q(i)=q(i)+Y(j*(256/N)+1);
   end
   q(i)=q(i)*h;
   N=N*2;
end
disp('Approximations with the trapezoidal rule:')
q
```

The input of this function is a vector of 256 components with the function to integrate values in the interval [0,1].

```
>> X=linspace(0,1,257);
>> Y=sqrt(1-X);
>> trapezoidalRule(Y)
Approximations with the rectangle rule:

q =

   0.60355339059327
   0.64328304624275
   0.65813022162445
   0.66358119687723
   0.66555893627894
   0.66627081137851
   0.66652565729683
```

3. – Simpson’s Rule

Finally, Simpson’s Rule will be used. This way, the program is modified as follows:

```
function simpsonRule=simpsonRule(Y)
q=zeros(7,1);
```

```
N=2; %Number of subintervals
for i=1:7
   h=1/N;
   q(i)=Y(1);
   q(i)=q(i)+Y(257);
   k=0;
      for j=1:N-1
         k=k+1;
         q(i)=q(i)+4*Y(k/2*(256/N)+1);
         k=k+1;
         q(i)=q(i)+2*Y(k/2*(256/N)+1);
      end
      k=k+1;
      q(i)=q(i)+4*Y(k/2*(256/N)+1); %Last interval middle point.
      q(i)=q(i)*h/6;
      N=N*2;
end
disp('Approximations with Simpson rule:')
q
```

The input of this function is a vector of 256 components with the function to integrate values in the interval [0,1].

```
>> X=linspace(0,1,257);
>> Y=sqrt(1-X);
>> simpsonRule(Y)
Approximations with Simpson rule:

q =

   0.65652626479257
   0.66307928008502
   0.66539818862815
   0.66621818274618
   0.66650810307836
   0.66661060593627
   0.66664684620310
```

The results of the three approximations are presented in the following table:

Number of subintervals	Rectangle Rule	Trapezoidal Rule	Simpson's Rule
2	0.85355339059327	0.60355339059327	0.65652626479257
4	0.76828304624275	0.64328304624275	0.66307928008502
8	0.72063022162445	0.65813022162445	0.66539818862815
16	0.69483119687723	0.66358119687723	0.66621818274618
32	0.68118393627894	0.66555893627894	0.66650810307836
64	0.67408331137851	0.66627081137851	0.66661060593627
128	0.67043190729683	0.66652565729683	0.66664684620310