AN INTRODUCTION TO MATLAB

FRANCISCO DE LA HOZ AND FERNANDO VADILLO

Abstract. In these notes, we give a basic introduction to Matlab, by means of a tutorial and different short programs, paying special attention to construction of and operations with matrices, graphical representation of functions and numerical methods.

Contents

1. Introduction

Matlab **(MATrix LABoratory)** is a language for scientific computation whose fundamental data are vectors and matrices. Unlike other programs like Fortran and C, it works on a very high level, including hundreds of operations within a single command.

From the first versions by Cleve Moler at the late seventies, MATLAB has become a powerful tool for conducting research and solving practical problems. In `http://www.mathworks.com/`, it is possible to see the enormous amount of information available on MATLAB: published books, programs, links, news groups, scientific meetings, etc.

MATLAB is an interactive system. When the computer starts, the screen shows Figure 1 (the actual interface can change a bit, depending on the version), at whose right-hand side the command window appears. In order for the system to execute an instruction, we type it after the prompt `>>`, and then press `enter`; moreover, if a certain instruction was not given a name, it is stored in a variable with the name `ans`. Variables are kept in the `workspace`, which is visible at the left-hand side of Figure 1.

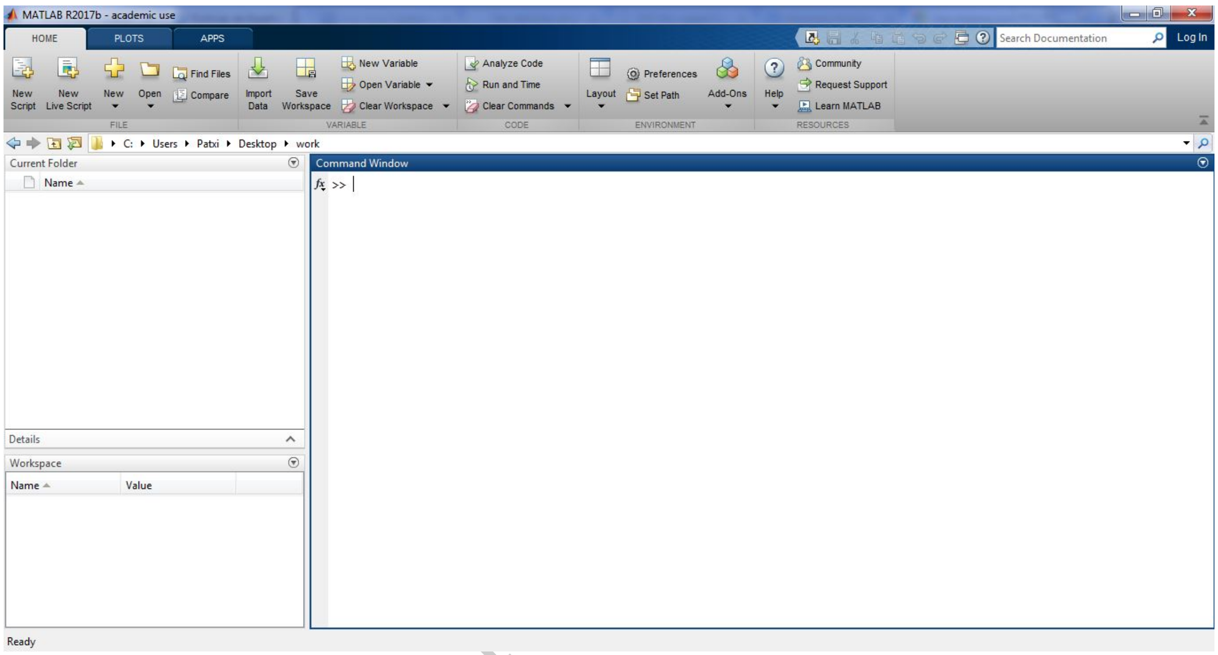

FIGURE 1. MATLAB screen

MATLAB has three very important characteristics, which differenciate it from other programming languages:

(1) The variables need not be previously declared.
(2) It contains a large collection of mathematical functions, with a number of arguments not necessarily the same.
(3) The basic data type are vectors and matrices of complex numbers, stored in double precision floating-point format. Additionally, `NaN` denotes a ”not-a-number” type, and `inf` denotes a positive infinity.

Some observations that should be always borne in mind:

(1) MATLAB is case sensitive.
(2) A semicolon at the end of a command hides the screen output.
(3) () and [] are not exchangeable. Each one has its own uses.
(4) The upward arrow recovers previously typed commands.
(5) Command `edit` opens a text editor.
(6) `^C` kills the current command execution.
(7) Commands `exit` or `quit` quit MATLAB.

The following commands are very useful.

Help commands:

demo: runs the demo program.

doc: opens the help browser.
help: lists topics on which help is available.
help `topic`: shows information on a topic.
lookfor `string`: lists help topics containing a string.
why: provides a funny answer to almost anything.

File system commands:

pwd: shows the current working directory.
cd: changes the current working directory.
dir, ls: list files within the current directory.
path: gets or sets Matlab search path.
editpath: edits Matlab search path.
copyfile: copies a file.
movefile: moves a file.
mkdir: creates a directory.
rmdir: removes a directory.
rm: removes a file.

Workspace commands:

who: lists variables currently in the workspace.
whos: lists variables in the workspace, together with their size.
clear: clears all variables.
clear `variable`: clears only the specified variable.
clear all: clears all variables and functions from workspace.
mlock `function`: locks a function, so that `clear` cannot remove it.
munlock `function`: unlocks a function, so that `clear` can remove it.
clc: clears the command window.
close all: closes all figure windows.

General information:

bench: tests the speed of the computer.
clock: returns the time and date as a vector.
computer: indicates the computer where Matlab runs.
ver: displays the current Matlab, Simulink and toolbox version information.

Additionally, thanks to the **Symbolic Math Toolboxes**, we have access to hundreds of Maple commands that supplement Matlab numeric and graphical facilities with other types of mathematical computation, in areas such as calculus, linear algebra, simplification of algebraic expressions, solutions of equations, special mathematical functions, variable-precision arithmetics, transforms, etc.

Finally, let us also mention the **Chebfun project** [20] by Trefethen and collaborators, which is a collection of algorithms and an open-source software system in object-oriented MATLAB, which extends familiar powerful methods of numerical computation involving numbers to continuous or piecewise-continuous functions.

2. A short Matlab tutorial

The use of a tutorial is a frequent option when using a scientific programming language for the first time. Its philosophy is that the student can acquire a general idea of the syntax of a language by typing directly commands and observing the results. The tutorial that follows is similar to that in [7]; other interesting references are [3, 10, 11, 13, 22].

As said above, when Matlab is started on a computer, Figure 1 appears on the screen. The commands that follow have to be typed at the command window, which is located at the right-hand side.

```
% tutorial.m
% A FIRST TUTORIAL       % "%" indicates a comment line
clear                    % clear variables and functions from memory
close all                % close all the open figure windows
a=[1 2 3]                % or, equivalently, a=[1, 2, 3]
```

```
c=[4+1i;5-2i;6]          % type 2i or 2j instead of 2*i or 2*j
c'
c.'
a*c
dot(a,c)
A=c*a
size(A)
openvar('A')
b=a.^2
exp(a)
sqrt(a)
format long
sqrt(a)
format
sum(a), sum(c)
pi
B=[-3 0 1; 2 5 -7; -1 4 8]
size(B)
%                        solve a linear system
x=B\c                    % equivalently x=c'/B';x=x';
norm(B*x-c)
%                        compute the eigenvalues and eigenvectors of a matrix
e=eig(B)
[V,D]=eig(B)
%                        other ways of creating vectors and matrices
v=1:6
w=2:3:10, y=1:-.25:0
C=[A,[8;9;10]]
D=[B;a]
%                        access the elements of a matrix
C(2,3)
C(2:3,1:2)
%                        other types of matrices
I3=eye(3), Y=zeros(3,5), Z=ones(2)
%                        random
rng('default')           % restart the random number generator (recommended)
rand('state',20)         % obsolescent
randn('state',15)        % obsolescent
F=rand(3), G=randn(1,5)
%                        access the workspace
who
whos
%                        sum of continuous fractions
g=2;
for k=1:10, g=1+1/g; end
g
%                        plot a curve and define a handle to the plot
t=0:.005:1;
z=exp(10*t.*(t-1)).*sin(12*pi*t);
h=plot(t,z)
get(h)                   % show the properties of the previous plot
%                        definition of anonymous functions (recommended)
f=@(x,y)x^2+y^2          % f is a function handle
```

```
f(5,7)
%                        definition of inline functions (obsolescent)
g=inline('x^2+y^2')
g(5,7)
%                        definition of cell arrays
E = {A, f, 'this is a sentence'}
H = cell(3, 1);
H{2} = E{3};
%                        numerical integration of y' = y, y(0) = 1
[x,y]=ode45(@(x,y)y,[0,1],1);
norm(y - exp(x), inf)
```

In the next subsections, we will show several brief programs in MATLAB. We encourage the student to copy them with the MATLAB editor (which is opened by typing `edit`) and to run them on a computer.

2.1. **The Fibonnaci Numbers.** In this first example, we build the sequence of Fibonacci numbers $\{x_n\}$ generated by the iteration

$$x_{n+1} = x_n \pm x_{n-1}, \qquad n \geq 2, \tag{2.1}$$

where $x_1 = 1, x_2 = 2$, and **we have introduced an element of randomness**, because the positive or negative sign is chosen with identical probability. Viswanath showed in 1999 that, for n large enough, $| x_n | \sim c^n$, where $c = 1.1319882487943\ldots$ is the so-called Viswanath constant.

```
% randomfibonacci.m      Fibonacci sequence with randomness
rng('default')
m = 5000;                % Number of iterations
x = [1 2];               % Initial data
for n = 2:m-1
     x(n+1) = x(n) + sign(rand-0.5)*x(n-1);
end
semilogy(1:m,abs(x))
c = 1.1319882487943;     % Viswanath's constant (1999)
hold on
semilogy(1:m,c.^(1:m))
hold off
```

Figure 2 shows the results obtained after executing the program.

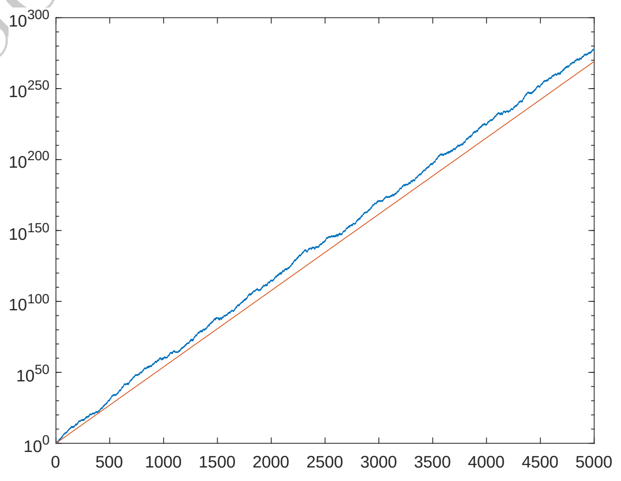

FIGURE 2. Results from `randomfibonacci.m`

2.2. **The Collatz iteration.** The following example analyzes the Collatz iteration: given a positive integer x_1, we build the sequence $x_{n+1} = f(x_n)$, where

$$f(x) = \begin{cases} 3x+1, & \text{if } x \text{ is odd}, \\ x/2, & \text{if } x \text{ is even}. \end{cases} \tag{2.2}$$

This process has been called **Half Or Triple Plus One** or **HOTPO**. The so-called Collatz conjecture is that no matter what number we start with, we will always eventually reach 1.

```
% collatz.m              Collatz iteration
n = input('Enter an integer larger than 2: ');
narray = n;
count = 1;
while n ~= 1
    if rem(n,2) == 1     % remainder modulo 2
        n = 3*n+1;
    else
        n = n/2;
    end
    count = count + 1;
    narray(count) = n;   % store the iteration
end
plot(narray,'*-')        % draw stars and join them with a continuous line
title(['Collatz iteration starting at ' int2str(narray(1))],'FontSize',14)
```

Figure 3 shows the results obtained after executing the program, starting with 2000.

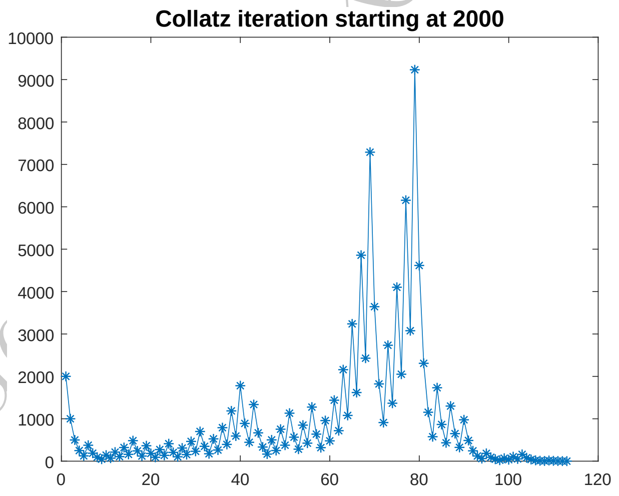

FIGURE 3. Results from the Collatz iteration, with initial number 2000.

2.3. **The Mandelbrot set.** Figure 4 shows the Mandelbrot set, which can be plotted in a few lines of code with MATLAB. Remember that it is formed by the values c in the complex plane for which $z_{n+1} = z_n^2 + c$ is bounded when $z_1 = c$. Try more functions to get other spectacular fractal designs.

```
% mandelbrot.m
% the Mandelbrot set
h = waitbar(0,'Computing...');
x = linspace(-2.1,0.6,301);
y = linspace(-1.1,1.1,301);
[X,Y] = meshgrid(x,y);
C = complex(X,Y);
```

```
Z_max = 1e6; it_max = 50;
Z = C;
for k = 1:it_max
    Z = Z.^2 + C;
    waitbar(k/it_max)
end
close(h)
contourf(x,y,abs(Z)<Z_max,1)
title('Mandelbrot Set','FontSize',14)
```

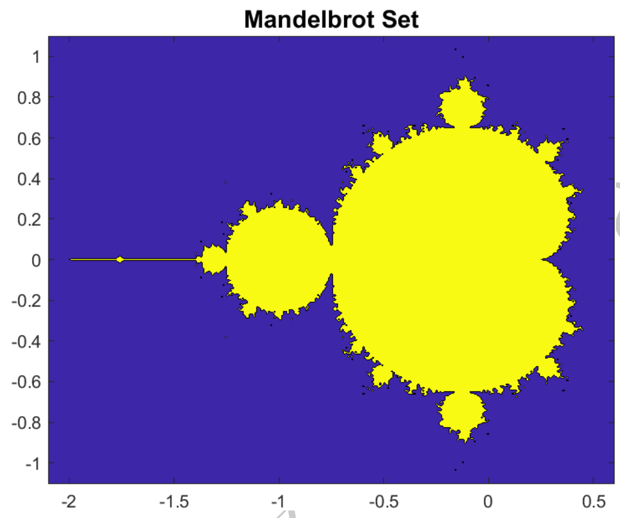

Figure 4. The Mandelbrot set

2.4. **Lorenz equations.** In the following example, we solve a system of differential equations

$$\frac{dy_1(t)}{dt} = 10(y_2(t) - y_1(t)),$$
$$\frac{dy_2(t)}{dt} = 28y_1(t) - y_2(t) - y_1(t)y_3(t),$$
$$\frac{dy_3(t)}{dt} = y_1(t)y_2(t) - 8y_3(t)/3,$$

which is an example of the family of Lorenz equations which build his famous attractor. Taking $(0, 1, 0)$ as initial data, the following MATLAB program constructs the attractor in just a few seconds:

```
% mainlorenz.m                         Lorenz system
tspan = [0 50];                        % solve the Lorenz equations for t\in[0,50]
yzero = [0;1;0];                       % initial conditions
[t,y] = ode45(@lorenzequations,tspan,yzero);
plot(y(:,1),y(:,3))                    % plot the result
xlabel('y_1','FontSize',14)
ylabel('y_3 ','FontSize',14,'Rotation',0,'HorizontalAlignment','right')
title('Lorenz equations','FontSize',14)
```

where the Lorenz equations have been defined in

```
% lorenzequations.m
% Y' = lorenzequations(T,Y)
function yprime = lorenzequations(~,y)
yprime = [10*(y(2)-y(1))
          28*y(1)-y(2)-y(1)*y(3)
```

```
             y(1)*y(2)-8*y(3)/3];
```

Figure 5 shows the Lorenz attractor.

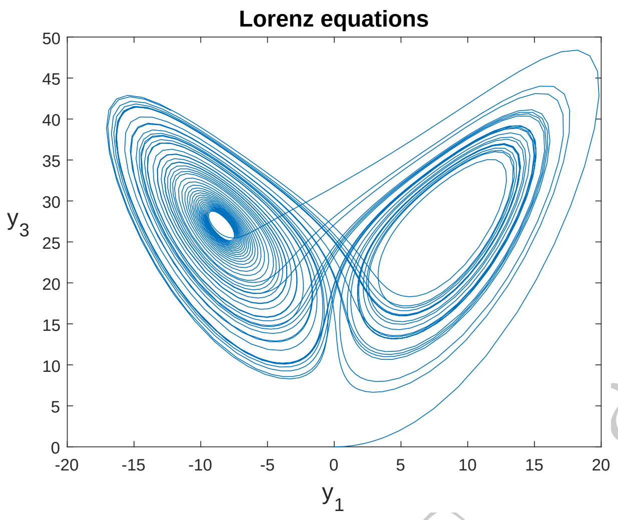

FIGURE 5. The Lorenz attractor

Creating an animation of the evolution is also straightforward:

```
% lorenzanimation.m
% creates a 3D animation:
close all
for m = 2:length(y)
    plot3(y(1:m,1),y(1:m,2),y(1:m,3))
    axis([-20, 20, -40, 40, 0, 50])
    drawnow
end
```

2.5. **A hyperbolic paraboloid.** In this last example, we plot the hyperbolic paraboloid defined by the equation $x^2 + y^2 = 1 + z^2$, and plotted in Figure 6.

```
% hyperbolicparaboloid.m      draw a hyperbolic paraboloid
N = 10;                       % number of sides
z  = linspace(-5,5,N)';
radius = sqrt(1+z.^2);
theta = 2*pi*linspace(0,1,N);
X = radius*cos(theta);
Y = radius*sin(theta);
Z = z(:,ones(1,N));
surf(X,Y,Z)
axis equal
```

3. LOGICAL RELATIONS AND OPERATIONS

In MATLAB, there is also a logical type of data. The easiest way to construct it is by applying the function `logical`.

The comparison between two scalars produces a result of logical type equal to 1 if it is true, and to 0 if it is false. The relation operations in MATLAB are the following ones:

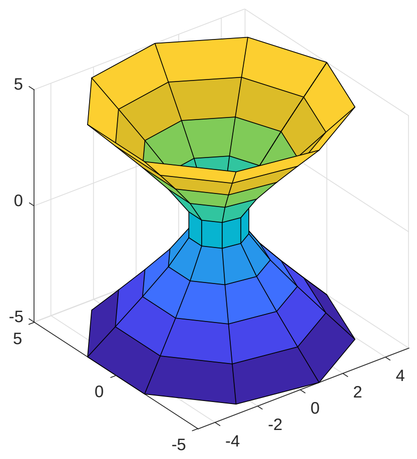

FIGURE 6. A hyperbolic paraboloid

$==$	equality
$\sim=$	inequality
$<$	less than
$>$	greater that
$<=$	less than or equal to
$>=$	greater than or equal to

Remember that the single equality $=$ assigns values to variables. The comparison between matrices obtains a matrix which is also logical. In order to obtain a list containing the logical functions, type `lookfor 'True for'`.

The most important operators involving logical variables are:

&	and
|	or
$\sim$	negation
`xor(x, y)`	logical symmetric difference of elements `x` and `y`
`all(x)`	true if all elements of a vector are nonzero
`any(x)`	true if any element of a vector is a nonzero number or a logical TRUE (1)

For example, `all(all(A==B))` is a way of testing whether $\mathbf{A} = \mathbf{B}$; and `any(any(A==B))` allows knowing whether **A** and **B** have any element in common.

4. Flow control and Matlab m-files

Matlab has four flow controls, whose exact formats can be consulted by typing `help`:

```
if
for
while
switch
```

The m-files or script files are the equivalents of the programs, functions, subroutines and procedures of other languages. In order to write m-files, Matlab has its own editor, which appears after typing `edit`. All the files bear the extension `.m`, and there are two types of m-files:

***scripts*:** these are sets of commands that work on the variables of the workspace.

***functions*:** these contain the command `function`, with input variables and output variables, although all the output variables are not required. The variables are local, unless they are declared as global, and the name of an m-file has to coincide with the name of its corresponding function.

The following short example describes a function that calculates the derivative of the logistic function $f(x) = x(1 - ax)$ at a point:

```
% flogist.m
% compute the logistic function and its derivative
function [f,fprime]=flogist(x,a)
f=x.*(1-a*x);
fprime=1-2*a*x;
```

In the next example, the m-file contains an implementation of the Newton method to calculate the square root of a number $a \geq 0$.

```
% sqrtn.m
% calculate the square root of a number using Newton's method
% a >= 0 is assumed
% TOL is the tolerance for convergence (eps by default)
% iter is the number of iterations needed for convergence
function [x,iter] = sqrtn(a,tol)
if nargin < 2, tol = eps; end
x = a;
iter = 0;
xdiff = inf;
fprintf(' k             x_k           relative change\n')
while xdiff > tol
    iter = iter + 1;
    xold = x;
    x = (x + a/x)/2;
    xdiff = abs(x-xold)/abs(x);
    fprintf('%2.0f:  %20.16e  %9.2e\n', iter, x, xdiff)
    if iter > 50
       error('No convergence achieved after 50 iterations!')
    end
end
```

If we want to calculate the square root of 10, we invoke it by typing `sqrtn(10)`, and the following table is generated:

```
 k             x_k           relative change
 1:  5.5000000000000000e+00   8.18e-01
 2:  3.6590909090909092e+00   5.03e-01
 3:  3.1960050818746470e+00   1.45e-01
 4:  3.1624556228038898e+00   1.06e-02
 5:  3.1622776651756750e+00   5.63e-05
 6:  3.1622776601683791e+00   1.58e-09
 7:  3.1622776601683791e+00   0.00e+00

ans =

    3.1623
```

5. Graphics

Matlab has powerful and versatile tools to represent graphically different types of data. In this section, we will briefly summarize the most common ones.

5.1. **Two-dimensional graphics.**
The most simple graphic in two dimensions joins the points with coordinates the vectors `x, y`, by means of the command `plot(x,y)`. When we run that command, Matlab opens a graphical window. The command `plot` has several options that can be consulted under `help`. Its general format is `plot(x,y,'chain')`, where the chain has three elements that indicate the color, the mark at the points, and the type of line. If `X`, `Y` are matrices $m \times n$, then `plot(X,Y)` plots the

graphics columnwise. For a vector `y` of complex numbers, `plot(y)` is equivalent to `plot(Re(y),Im(y))`; and if `y` is real, `plot(y)` plots `y` against its indices.

Other commands frequently used are:

`loglog`: is similar to `plot`, but in logarithmic scale.
`semilogx, semilogy`: plots in the corresponding semi-logarithmic scale.
`axis`: controls the axes.
`grid on`: shows a net.
`xlabel, ylabel`: label the axes.
`title`: puts a title over the graphic.
`legend`: creates a legend that identifies the graphics.
`text, gtext`: allow to write a text at any position in the graphic.

In the following example, we plot the first four Legendre polynomials, with a legend that identifies each of them. The result can be seen in Figure 7.

```
% legendre4.m
x=-1:.01:1;
p1=x;
p2=(3/2)*x.^2-1/2;
p3=(5/2)*x.^3-(3/2)*x;
p4=(35/8)*x.^4-(15/4)*x.^2+3/8;
plot(x,p1,'r:',x,p2,'g--',x,p3,'b-',x,p4,'m-')
box on
h=legend('\itn=1','n=2','n=3','n=4');
set(h,'location','southeast')
xlabel('x','Fontsize',12,'FontAngle','italic')
ylabel('P_n','FontSize',12,'FontAngle','italic')
title('Legendre polynomials','Fontsize',14)
text(-.6,.7 ,'(n+1)P_{n+1}(x)=(2n+1)xP_n(x)-n P_{n-1}(x)',...
         'Fontsize',12,'FontAngle','italic')
```

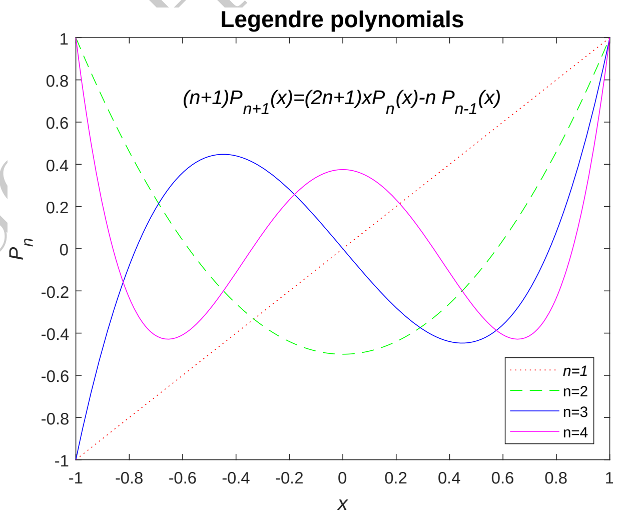

FIGURE 7. First four Legendre polynomials

The command `subplot` allows to plot different graphics in the same window. On the other hand, `fplot` plots mathematical functions. The following example generates Figure 8, and shows how both instructions work.

```
% plotmult.m
% example of multiple plots
```

```
subplot(221), fplot(@(x)exp(sqrt(x).*sin(12*x)),[0 2*pi])
subplot(222), fplot(@(x)sin(round(x)),[0 10],'--')
subplot(223), fplot(@(x)cos(30*x)./x,[0.01 1],'-'), ylim([-15 20])
subplot(224), fplot(@(x)[sin(x),cos(2*x),1./(1+x)],[0 5*pi]), ylim([-1.5 1.5])
```

Observe that it is also possible to type `fplot('[sin(x),cos(2*x),1/(1+x)]',[0 5*pi])`, etc. However, this syntax, even if slightly simpler, is labeled as obsolescent by the last MATLAB versions.

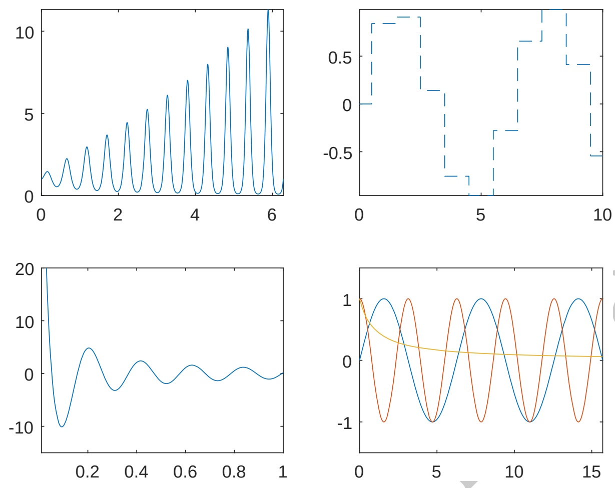

FIGURE 8. An example illustrating the usage of `subplot` and `fplot`

5.2. **Three-dimensional graphics.** The command `plot3` is the three-dimensional equivalent of `plot`. The following simple piece of code uses this instruction and draws the curve in Figure 9:

```
% draw3d.m
% example using the command plot3
t=-5:.005:5;
x=(1+t.^2).*sin(20*t);
y=(1+t.^2).*cos(20*t);
z=t;
plot3(x,y,z)
grid on
xlabel('x(t)'), ylabel('y(t)'), zlabel('z(t)')
title('\it{Example using plot3}','FontSize',14)
```

In order to draw surfaces, there are different options; although the most usual ones are those represented in Figures 10 and 11, which correspond respectively to the programs `surface1.m` and `surface2.m`:

- `mesh, meshgrid`: draw meshed surfaces.
- `surf, surfc`: draw surfaces.
- `waterfall`: draws traveling waves.

```
% surface1.m
% example of surface plotting
x=1:.1:pi; y=x;
[X,Y]=meshgrid(x,y);
Z=sin(Y.^2+X)-cos(Y-X.^2);
subplot(221), mesh(Z)
subplot(222), meshc(Z)
subplot(223), mesh(x,y,Z), axis([0 pi 0 pi -5 5])
subplot(224), mesh(Z), hidden off
```

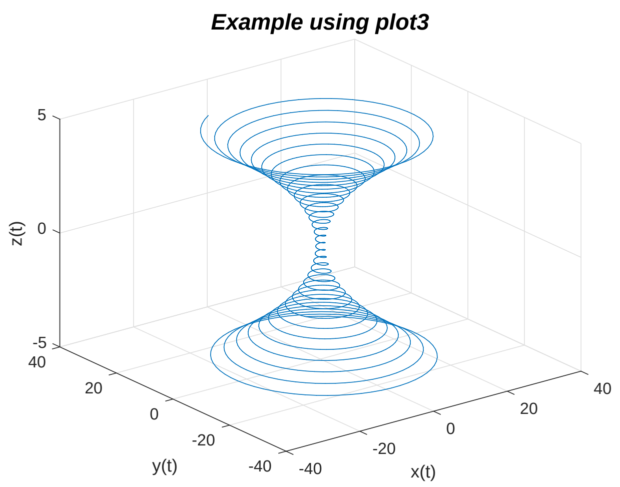

FIGURE 9. An example illustrating the usage of `plot3`

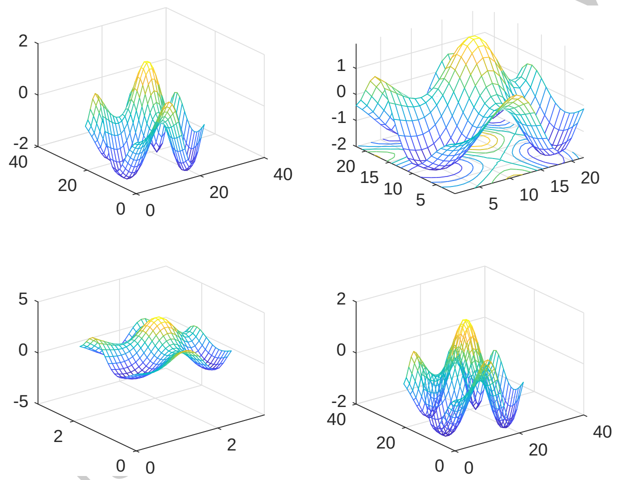

FIGURE 10. Results of Program surface1.m

```
% surface2.m
% another example with surfaces
Z=membrane;
subplot(221), surf(Z), title('surf','FontSize',14)
subplot(222), surfc(Z), title('surfc','FontSize',14),colorbar
subplot(223), surf(Z), shading flat, title('shading flat','FontSize',14)
subplot(224), waterfall(Z), title('waterfall','FontSize',14)
```

6. Linear algebra

The fundamental data type in MATLAB are vectors and matrices. Therefore, it is fundamental to learn how to generate and manipulate those types of data efficiently.

Besides constructing matrices by enumerating each of their elements, as we saw in the tutorial, there are functions that create specific matrices. Let us enumerate some of them:

- `zeros(m,n)`: creates a matrix of zeros.
- `ones(m,n)`: creates a matrices of ones.
- `eye(m,n)`: creates an identity matrix.

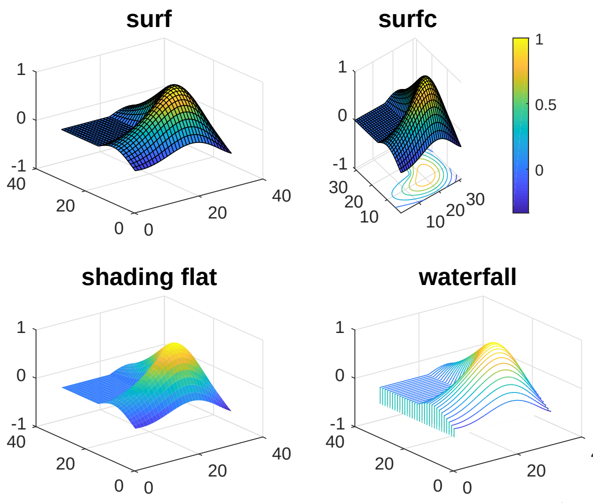

FIGURE 11. Results of Program surface2.m

- `rand(m,n)`: creates a matrix of random uniformly distributed numbers.
- `randn(n,m)`: creates a matrix of random normally distributed numbers.
- `linspace(m:n:p)`: creates a vector of equally spaced elements.
- `lgspace(m:n:p)`: creates a vector of logarithmically spaced elements.

In general, parameters m and n indicate the dimensions of the created matrices.

The determinant of a matrix $\mathbf{A}$ is calculated by means of the function `det(A)`, and, if $\mathbf{A}$ is invertible, its inverse is given by `inv(A)`, although in scientific computing the explicit calculation of inverses is very uncommon, so the test of the determinant is not very advisable.

Matrices can be also manipulated blockwise: `blkdiag` creates a diagonal matrix blockwise, while `repmat(A,m,n)` creates a matrix $m \times n$, where $\mathbf{A}$ is repeated in each block.

The Kronecker product of two matrices $\mathbf{A}$ and $\mathbf{B}$ is `kron(A,B)`.

Matrices can be reshaped. `A(:)` transforms the matrix $\mathbf{A}$ into a long column vector of dimension $1 \times (m\,n)$. `reshape(A,m,n)` orders the elements of the matrix $\mathbf{A}$ into m rows and n columns. The diagonal elements can be obtained with the function `diag(A)`; the upper triangular or lower triangular matrices, with the functions `tril(A) and triu(A)`, respectively.

Finally, let us mention that the command `gallery` gives access to a large collection of test matrices created by N. J. Higham, which can be consulted in [8].

6.1. **Systems of linear equations.** The norm of a matrix measures its size. The command `norm(A,p)` calculates the p-norm, where $p = 1, 2, \inf$.

The condition number of a matrix $\mathbf{A}$, $\kappa(\mathbf{A}) = \|\mathbf{A}\| \cdot \|\mathbf{A}^{-1}\|$, measures the sensitivity of the solution of a system $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ with respect to perturbations of $\mathbf{A}$ and $\mathbf{b}$. The command `cond(A,p)` returns an estimate of that number.

The fundamental tool to solve a system of linear equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ is the operator backslash \, which also solves the more general system $\mathbf{A} \cdot \mathbf{X} = \mathbf{B}$, where $\mathbf{B}$ is a matrix with m columns; in that case, the m systems are solved for each column of $\mathbf{B}$. Equivalently, the slash / solves systems of the form $\mathbf{X} \cdot \mathbf{A} = \mathbf{B}$.

Three types of systems of linear equations have to be considered: square systems (with the same number of equations and unknowns), overdetermined systems (with more equations than unknowns), and systems with more unknowns than equations. The numerical methods involved can be consulted in [1, 2, 4, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 10, 21].

6.2. **Square systems.** If $\mathbf{A}$ is a non-singular matrix $n \times n$, then $\mathbf{A}\backslash\mathbf{b}$ gives the solution of the system $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. MATLAB recognizes two special types of matrices:

- Triangular matrices or permutations of triangular matrices
- Positive definite Matrices

6.3. **Overdetermined systems.** If the matrix of the system is $m \times n$, with $m > n$, the system may have or may not have a solution. When the rank of the matrix $\mathbf{A}$ is n, $\mathbf{A}\backslash\mathbf{b}$ calculates a solution in the least square sense and, if the rank of $\mathbf{A}$ is less than n, it calculates a basic solution.

Another way of obtaining a least square solution is by calculating the pseudoinverse of the matrix $\mathbf{A}$ with the command `pinv(A)`, the solution being now `x=pinv(A)*b` (see for instance [10, 13, 17, 19, 21, 22]).

6.4. **Underdetermined systems.** When there are more unknowns than equations ($m < n$), the system may have or may not have a solution, or there may be infinitely many solutions. In this case, $\mathbf{A}\backslash\mathbf{b}$ gives a solution with at most k nonzero elements, where k is the rank of $\mathbf{A}$.

6.5. **Matrix factorization.** The functions to calculate the classical factorizations are the following ones:

- `[L,U,P]=lu(A)`: calculates the factorization LU, such that $\mathbf{P} \cdot \mathbf{A} = \mathbf{L} \cdot \mathbf{U}$.
- `R=chol(A)`: obtains the factorization of Cholesky of a positive definite matrix.
- `[Q,R,P]=qr(A)`: calculates the factorization of QR, such that $\mathbf{A} \cdot \mathbf{P} = \mathbf{Q} \cdot \mathbf{R}$, where $\mathbf{P}$ is a permutation matrix chosen in such a way that the diagonal elements of $\mathbf{R}$ are non-increasing in modulus.
- `[U,S,V]=svd(A)`: computes the singular value decomposition of the matrix $\mathbf{A}$, i.e., constructs unitary matrices $\mathbf{U}$ and $\mathbf{V}$, and a diagonal matrix $\mathbf{S}$, such that $\mathbf{A} = \mathbf{U} \cdot \mathbf{S} \cdot \mathbf{V}^*$.

6.6. **Sparse matrices.** A sparse matrix is a matrix with a large percentage of zeros. These are very frequent matrices in lineal systems that appear when discretizing differential equations by using finite differences, finite elements or spectral methods. Since the dimension of the system can be very large, it is unavoidable to optimize the use of memory by reducing the name of operations.

Matlab has a sparse data type, which only stores the nonzero elements of a matrix, together with their row and column indices. The algorithms for sparse matrices are different to those used with full matrices, although they receive sometimes the same name.

A habitual way of constructing sparse matrices is by means of the function `A=sparse(i,j,x)`, where `x` is a vector of values located according to the vectors of indices `i` and `j`. In order to turn a sparse matrix `A` into another full one `B`, it is enough to type `B=full(A)`, and, conversely, `A=sparse(B)`. The number of nonzero elements of a matrix is given by the function `nnz`.

The sparse identity matrix is `speye(n)` or`speye(m,n)`, while `spones(A)` places ones where `A` is not zero. The functions `spdiag`, `sprand` and `sprandn` are the sparse equivalents of those studied above.

6.7. **Sparse matrix linear algebra.** Matlab has functions that allow solving systems, calculating eigenvalues, and singular values of sparse matrices. Moreover, although the name coincides sometimes with the name used with full matrices, the implemented algorithm is not the same.

The backslash operator $\mathbf{x} = \mathbf{A}\backslash\mathbf{b}$ solves the linear system $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$; and the function `condest` calculates the condition number of a matrix.

The functions `lu`, `chol`, `eig`, etc., act as in the case of full matrices, and the function `spy(A)` draws a scheme indicating where the nonzero elements of a matrix are located.

When working with sparse matrices, a very important question is their possible rearrangement, in order that the algorithms are more efficient. Matlab includes functions that implement usual algorithms as Cuthill-McKee's, called `symrcm`, which is used to reduce the bandwidth of sparse symmetric matrices

7. Numerical methods

In this section, we describe the related Matlab functions for manipulating polynomials, solving nonlinear equations, calculating numerical quadrature and solving differential equations. The numerical analysis books of general type that deal with the implemented methods are [1, 2, 9, 10, 11, 13, 14, 15, 17, 22].

7.1. **Polynomials.** Matlab represents the polynomial

$$p(x) = p_1 x^n + p_2 x^{n-1} + \ldots + p_n x + p_{n+1}$$

by means of a row vector $p = [p(1), \ldots, p(n+1)]$, formed by its coefficients. The polynomials can be multiplied and divided through `conv` and `deconv`, respectively.

7.2. **Polynomial evaluation.** The function `polyval(p,x)` calculates the value of the polynomial with coefficients `p` at the point `x`. If `x` were a matrix `X`, then the matrix `Y=polyval(p,X)` would be obtained.

7.3. **Root finding of polynomials.** The roots of the polynomial with coefficients `p` are found with `z=root(p)`, and the function `q=poly(z)` performs the inverse operation, i.e., it constructs a polynomial with coefficients `q` and leading coefficient equal to one, whose roots are precisely `z`. The function `poly` admits a matricial argument `poly(A)`, which is the characteristic polynomial of the matrix `A`.

7.4. **Interpolation.** Given the data set $\{x_i, y_i\}_{i=1}^m$, with different points x_i, `p=polyfit(x,y,n)` constructs the least square polynomial of degree $\leq n$. If $n \geq m$, then `p` is the interpolating polynomial.

The command `yy=spine(x,y,xx)` returns the cubic spline for the data `x`, `y`, evaluated at the points `xx`. The function `interp1` admits several options that can be consulted with the command `help` .

Matlab has also functions to interpolate in two dimensions, `griddata` and `interp2`; and to interpolate in three dimensions, `interp3`.

7.5. **Minimization and root finding for nonlinear equations.** Matlab has routines to calculate the zeros of a function of one variable, `fzero`, and to minimize functions of one variable, `fminbnd`, or of n variables, `fminsearch`.

The command `fzero(f,x0)` obtains a zero of the function `f` near a point `x0`, and, in order to find a zero inside an interval, we type `fzero(f,[x1,x2])`. The parameters for convergence are controlled through `optimset`.

In order to obtain a local minimum in the interval `[x1,x2]`, we type `fminbnd(f,x1,x2)`.

More complicate optimization problems may be studied by means of the package `Optimization Toolbox`, which implements more sophisticated techniques.

7.6. **Numerical quadrature.** The two most important functions to estimate the value of the definite integral $\int_a^b f(x)dx$ are `quad` and `quadl`. Both functions use formulas of adaptive composite quadrature; `quad` is based on Simpson's rule, while `quadl` uses the four-point Gauss-Lobatto formula.

The format is the same for both functions: `q=quad(f,a,b,tol)`, where tol controls the absolute error. For example: `f = @(x)sin(x); quad(f,0,pi,eps)`, or `f = inline(sin(x)); quad(f,0,pi,eps)`, or, simply, `quad(@sin,0,pi,eps)`, where we have to add `@` before `sin`.

Another quadrature function is `trapz`, which applies the composite trapezoidal rule.

Double integrals can also be evaluated through `dblquad`.

7.7. **Differential equations.** In Matlab, there are several options to solve initial value problems:

$$\begin{aligned} \frac{d}{dt}y(t) &= f(t, y(t)), \\ y(t_0) &= y_0, \end{aligned}$$

where t is the time and $y(t)$ is an m-dimensional vector. The easiest way to solve this problem is to write a function in Matlab that defines the differential equation and to call then one of the Matlab solvers.

7.8. **An example with ode45.** The function `ode45` implements the Dormand-Prince method, which is an embedded pair of orders 4 and 5 (see [5]). The technique of embedded pairs is also explained in [17], [14] and [13].

In the following example, we will solve the equation for the simple pendulum:

$$\frac{d^2}{dt^2}\theta(t) + \sin\theta(t) = 0,$$

which, after the change of variable $y_1(t) = \theta(t)$, $y_2(t) = \frac{d}{dt}\theta(t)$, becomes the system

$$\begin{aligned} \frac{d}{dt}y_1(t) &= y_2(t), \\ \frac{d}{dt}y_2(t) &= -\sin y_1(t), \end{aligned}$$

or, written in Matlab,

```
% pendulum.m
% simple pendulum
function yprime=pendulum(~,y)
yprime=[y(2); -sin(y(1))];
```

Figure 12 represents the plane of phases and three trajectories. The command sequence is:

```
% mainpendulum.m
tspan=[0 10];
yazero=[1; 1]; ybzero=[-5; 2]; yczero=[5; -2];
[ta,ya]=ode45(@pend,tspan,yazero);
[tb,yb]=ode45(@pend,tspan,ybzero);
[tc,yc]=ode45(@pend,tspan,yczero);
[y1,y2]=meshgrid(-5:.5:5,-3:.5:3);
Dy1Dt=y2; Dy2Dt=-sin(y1);
quiver(y1,y2,Dy1Dt,Dy2Dt)
hold on
plot(ya(:,1),ya(:,2),yb(:,1),yb(:,2),yc(:,1),yc(:,2))
axis equal, axis([- 5 5 -3 3])
xlabel y_1(t), ylabel y-2(t),
hold off
```

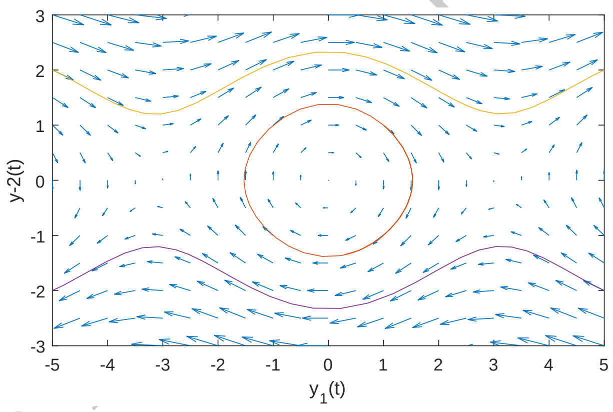

FIGURE 12. Phase plane of the simple pendulum

The general format for `ode45` is `[t,y]=ode45(@f,tspan,yzero,options,p1,p2,..)`, where `p1,p2...` are parameters of `f`; and `options=odeset('AbsTol',valor1,'RelTol',valor2)` controls the tolerance, which means that the error test will be

$$err(i) \leq \max\{\texttt{AbsTol},\ \texttt{RelTol} \times \mid x(i) \mid\}.$$

The values of the options can be known with the function `odeset`. In the following example, we will see another very interesting application of `options`.

7.9. **A predator-prey problem.** In this example, we suppose that the rabbit follows a preestablished path $(r_1(t), r_2(t))$ and that the fox pursues it under two hypotheses:

- At every instant, the tangent of the path of the fox points toward the rabbit.
- The velocity of the fox is k times the rabbit's.

If we suppose that $(y_1(t), y_2(t))$ is the path followed by the fox, according to the previous hypotheses, their components have to be the solution of the following system of differential equations:

$$\frac{d}{dt}y_1(t) = s(t)(r_1(t) - y_1(t)),$$
$$\frac{d}{dt}y_2(t) = s(t)(r_2(t) - y_2(t)),$$

where

$$s(t) = \frac{k\sqrt{(\frac{d}{dt}r_1(t))^2 + (\frac{d}{dt}r_2(t))^2}}{\sqrt{(r_1(t) - y_1(t))^2 + (r_2(t) - y_2(t))^2}}.$$

When the fox is faster than the rabbit ($k > 1$), the fox has to reach its pray in such a way that the denominator of $s(t)$ gets very small, so the numerical integration cannot be completed. For those cases, `ode45` allows introducing an option called `event`, so if during the execution of the program a precise situation arises, the execution stops, giving the results at that time. In our example, the event could be defined by saying that the distance from the fox to the rabbit is smaller than 10^{-4}, which is defined by the following function:

```
% event.m
% Event function for mainfoxrabbit.m
% Detect when the fox is close to the rabbit
function [value,isterminal,direction] = event(t,y)
r = sqrt(1+t)*[cos(t);sin(t)];
value = norm(r-y) - 1e-4;     % The fox is close to the rabbit.
isterminal = 1;               % Stop the integration when
                              % the event function becomes zero.
direction = -1;               % It locates a zero where
                               % the event function is decreasing.
```

The second member of the differential system is

```
% foxrabbit.m
% Fox-rabbit pursuit simulation with relative speed parameter
function yprime = foxrabbit(t,y)
k = 1.1;
r = sqrt(1+t)*[cos(t);sin(t)];
r_p = (0.5/sqrt(1+t))*[cos(t)-2*(1+t)*sin(t);sin(t)+2*(1+t)*cos(t)];
dist = max(norm(r-y),1e-6);
factor = k*norm(r_p)/dist;
yprime = factor*(r-y);
```

The main file `mainfoxrabbit.m` draws Figure 13, which represents the paths of both animals; the hunting happens at time $t = 5.071$, at the point $(0.8646, -2.3073)$.

```
% mainfoxrabbit.m
% fox & rabbit example, with event
k=1.1;
tspan=[0 6];yzero=[3;0];
options=odeset('RelTol',1e-6,'AbsTol',1e-6,'Events',@event);
[tfox,yfox]=ode45(@foxrabbit,tspan,yzero,options);
plot(yfox(:,1),yfox(:,2)),hold on
plot(sqrt(1+tfox).*cos(tfox),sqrt(1+tfox).*sin(tfox),'--')
plot([3 1],[0 0],'o'), plot(yfox(end,1),yfox(end,2),'*')
axis equal, axis([-3.5 3.5 -2.5 3.1])
h=legend('Fox','Rabbit'); set(h,'location','best')
hold off
```

7.10. **Stiff problems.** The function `ode45` does not solve all the differential problems in an efficient way; for instance, it is not well suited for the stiff problems (see for instance [6, 16]). For these types of initial value problems, MATLAB implements other numerical methods that are indicated in the following table:

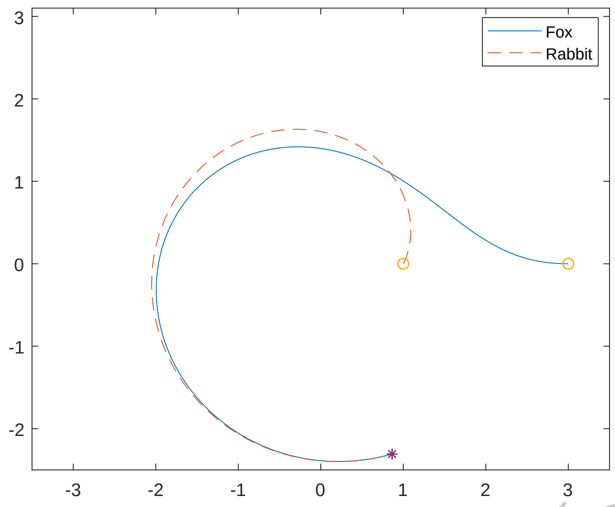

FIGURE 13. Paths of the fox and of the rabbit

Name	Type of problem	Type of algorithm
ode45	nonstiff	Runge-Kutta method of orders 4 and 5
ode23	nonstiff	Runge-Kutta method of orders 2 and 3
ode113	nonstiff	Linear multistep method of orders 1 to 3
ode15s	stiff	BDF method of orders 1 to 5
ode23s	stiff	Modified Rosenbrock method of orders 2 and 3
ode23t	moderately stiff	Trapezoidal rule
ode23tb	stiff	Implicit Runge-Kutta method of orders 2 and 3

In general, these functions are designed to be easily exchanged, i.e., the characteristics of `ode45` can be extended to the others, although it is recommended to try `ode45` first and, if the problem is stiff, to use `ode15s`.

REFERENCES

1. K.E. Atkinson, *An Introduction to Numerical Analysis*, Wiley, 1989.
2. J.D. Faires and R.L. Burden, *Numerical analysis*, International Thomson Publishing, 1991.
3. A. Gilat, *MATLAB: An Introduction with Applications*, third ed., Wiley, 2008.
4. G.H. Golub and C.F. Van Loan, *Matrix Computations*, third ed., The Johns Hopkins University Press, 1996.
5. E. Hairer, S.P. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*, second ed., Springer, 2008.
6. E. Hairer and G. Wanner, *Solving Ordinary Differential Eqautions II: Stiif and D.A. Problems*, Springer, 1991.
7. D.J. Higham and N.J. Higham, *MATLAB Guide*, third ed., SIAM, 2017.
8. N.J. Higham, *Accuracy and Stability of Numerical Algorithms*, second ed., SIAM, 2002.
9. E. Isaacson and H.B. Keller, *Analysis of Numerical Methods*, John Wiley & Sons Inc, 1966.
10. C.F. Van Loan, *Introduction to Scientific Computing: A Matrix Vector Approach Using MATLAB*, Prentice-Hall, 1996.
11. J.H. Mathews and K.D. Fink, *Numerical Methods Using MATLAB*, third ed., Prentice Hall, 1999.
12. C.D. Meyer, *Matrix Analysis and Applied Linear Algebra*, SIAM, 2001.
13. C.B. Moler, *Numerical Computing with MATLAB*, SIAM, 2004.
14. A. Quarteroni, R. Sacco, and F. Saleri, *Numerical Mathematics*, Springer, 2000.
15. J.M. Sanz-Serna, *Diez lecciones de Cálculo Numérico*, Universidad de Valladolid, 1998, in Spanish.
16. L.F. Shampine, I. Gladwell, and S. Thompson, *Solving ODEs with MATLAB*, Cambridge University Press, 2003.
17. J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, third ed., Springer, 2002.
18. G. Strang, *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, 1986.
19. L.N. Trefethen and D. Bau III, *Numerical Linear Algebra*, SIAM, 1997.
20. Nick Trefethen, Zachary Battles, and many others, *chebfun*, `http://www.chebfun.org`.
21. D.S. Watkins, *Fundaments of Matrix Computions*, second ed., Wiley-Interscience, 2002.
22. W.Y. Yang, W. Cao, T.S. Chung, and J. Morris, *Applied Numerical Methods Using MATLAB*, Wiley-Interscience, 2005.

Department of Applied Mathematics and Statistics and Operations Research, University of the Basque Country UPV/EHU
E-mail address: francisco.delahoz@ehu.eus; fernando.vadillo@ehu.eus