Matlab course: hands-on workshop

Jérôme Hoepffner

April 2011

Contents

1 Lecture notes **5**

1.1 Introduction 5
1.2 Basic Matlab 6
1.2.1 Graphs 8
1.2.2 Scripts 8
1.2.3 Current directory 9
1.2.4 Loops and tests 9
1.2.5 Simple functions of Matlab 11
1.2.6 Create your own functions 11
1.2.7 Matlab Help 12
1.2.8 Special characters 14
1.3 Arrays 14
1.3.1 Build arrays by concatenation 15
1.3.2 Access to subarrays 16
1.3.3 Array operations 18
1.3.4 Special array functions 20
1.4 Vectorization 22
1.4.1 More advanced vectorization 25
1.4.2 Even more advanced 27
1.5 Computing faster 30
1.6 Graphs 32
1.6.1 Lines 33
1.6.2 surfaces 36
1.6.3 Isovalues 39
1.6.4 Vector fields 40
1.6.5 The current graph 41
1.6.6 Saving figures 41
1.7 Set and get 43
1.8 Your report 44

Chapter 1

Lecture notes

1.1 Introduction

The computer is a tool at the heart of science: writing articles, analyzing and visualizing data, but also creating data by simulating accurate models of our physical systems. Creating data is typically done by intensive computations, by solving partial differential equations with a large number of degrees of freedom. This is done using compiled languages: the restriction is simulation time, these codes are large, complex, of long development time and hopefully long lifetime. Modern experimental measurement techniques as well produce large fields of data: Particle Image Velocimetry for full velocity fields, the large number of pixels of high speed cameras...

The data obtained this way is heavy and complex, and stored in the memory of computers. It is clear that special programming skills are needed to manipulate this data; on one hand to visualize it (data must be seen), and on the other hand to extract the relevant information: statistics, automatic identification of special events...

This use of the computer goes beyond post-processing: to understand our physical systems we need to compare them to simple models, often ordinary differential equations; light systems of equations which must be themselves solved or simulated.

The goal of this course is to set-up a standard of programing using Matlab to give you the tool you need for all that. These codes are typically simple, of rapid development time and often of short lifetime: the qualities of interactivity.

Organisation: Four consecutive mornings, each one structured as: presentation of the general ideas, examples coded ”live” and discussed, hands-on

where you apply the techniques on various types of physical systems, put your results (graphics) together in a report. We keep the physics in mind at all times.

1) Compiled versus interpreted languages. Basic syntax: loops, tests, arrays, logicals, graphics. Using the documentation. Manipulation of arrays.

2) Graphics: overview of the possibilities. Set/get commands to affect all properties of the graphics through programing. Animations.

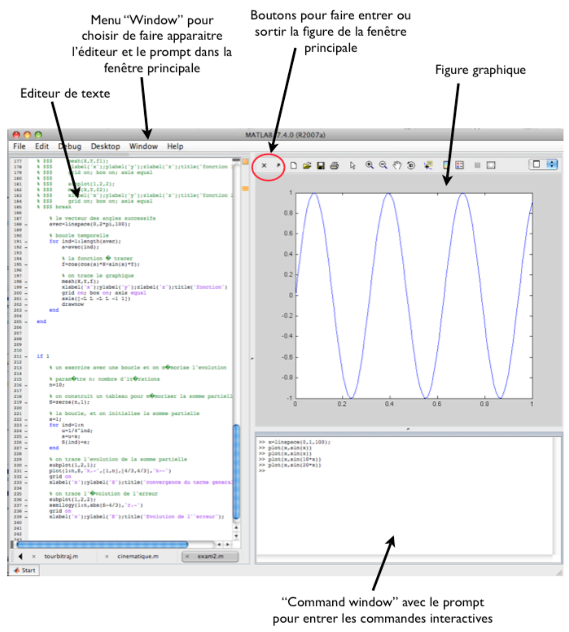
3) Making complex operation with short commands: manipulation of arrays, vectorization, use of arrays of logicals. Avoiding the mistakes which make a slow code, learning the basic ideas towards concision.

4) Linking heavy/efficient code to light/quick analyses: inputs and outputs, interfaces. Computations in parallel. Simulating systems and models.

1.2 Basic Matlab

Matlab offers an interface through which one can enter commands, this is the ”prompt” , the line which starts with `>>`. You can write command lines to do all your needed operations: creation of arrays, arithmetic evaluations, draw graphs. Once the command is written, you strike ”enter” and the computer evaluates this command, check that everything is correct. If there is a problem: calling a variable which does not exist... Matlab gives you an error message. It is important to read the error message and try to understand.

Matlab comes with a graphical interface with a text editor, the command window with the prompt. You can chose the configuration of the window from the ”window” menu. I advise you to chose the following configuration, which allows you to see at the same time your script, the command window to see the error messages and to test simple things, as well as your graphic window.


Menu "Window" pour choisir de faire apparaitre l'éditeur et le prompt dans la fenêtre principale
Boutons pour faire entrer ou sortir la figure de la fenêtre principale
Figure graphique
Editeur de texte
MATLAB 7.4.0 (R2007a)
File Edit Debug Desktop Window Help
tourbitraj.m
cinematique.m
exam2.m
Start
"Command window" avec le prompt pour entrer les commandes interactives

```
% $$$      mesh(X,Y,f1);
% $$$      xlabel('x');ylabel('y');zlabel('z');title('fonction
% $$$      grid on; box on; axis equal
% $$$
% $$$      subplot(1,2,2);
% $$$      mesh(X,Y,f2);
% $$$      xlabel('x');ylabel('y');zlabel('z');title('fonction
% $$$      grid on; box on; axis equal
% $$$ break

    % le vecteur des angles successifs
    avec=linspace(0,2*pi,100);

    % boucle temporelle
    for ind=1:length(avec);
        a=avec(ind);

        % la fonction � tracer
        f=cos(cos(a)*X+sin(a)*Y);

        % on trace le graphique
        mesh(X,Y,f);
        xlabel('x');ylabel('y');zlabel('z');title('fonction')
        grid on; box on; axis equal
        axis([-1 1 -1 1 -1 1])
        drawnow
    end

end




if 1

    % un exercice avec une boucle et on m�morise l'evolution

    % param�tre n: nombre d'it�rations
    n=10;

    % on construit un tableau pour m�moriser la somme partiell
    S=zeros(n,1);

    % la boucle, et on initialise la somme partielle
    s=1;
    for ind=1:n
        u=1/4^ind;
        s=u+s;
        S(ind)=s;
    end

    % on trace l'evolution de la somme partielle
    subplot(1,2,1);
    plot(1:n,S,'k.-',[1,n],[4/3,4/3],'k--')
    grid on
    xlabel('n');ylabel('S');title('convergence du terme general

    % on trace l'�volution de l'erreur
    subplot(1,2,2);
    semilogy(1:n,abs(S-4/3),'r.-')
    grid on
    xlabel('n');ylabel('E');title('Evolution de l''erreur');
```

```
>> x=linspace(0,1,100);
>> plot(x,sin(x))
>> plot(x,sin(x))
>> plot(x,sin(10*x))
>> plot(x,sin(20*x))
>>
```

If you create an array, for instance

```
>> A=[0.1,5,-2];
```

this is a line array (one line and three columns), then this array is now accessible in your "workspace" , which is the memory space of Matlab. You can use from now on this array by calling its name, for instance here we create a new array B in which we put twice the value of elements of the array A:

```
>> B=2*A;
```

```
>> B=2*A
0.2 10 -4
```

Which is indeed what we expected

I can ask Matlab to tell me the names of all existing variables with the function `who`

```
>> who
A ...
B ...
```

These are the two arrays that I have created. If I want to remove an array from the workspace, I use the function `clear` by calling the array by its name. This operation of memory management is useful to free memory space and it is especially important to think about that because in matlab it is very easy to create new variables.

```
>> clear A
```

To remove all variables, use `clear`.

Note that Matlab is a case sensitive language: `A` is thus a different variable from `a`.

1.2.1 Graphs

With the command line, you can draw graphs which represent (visualize) arrays. For instance with the function "plot" for a line graph:

```
>> a=[2, 5, 6];
>> b=[ 3.2, 3.5, 5];
>> plot(a,b)
```

Here I create two arrays called `a` and `b`, and I draw the values of `b` as ordinate and the values of `a` as abscissa. This graph appears in a new window. You can use functions which add details to these graphs:

```
>> xlabel('a'); ylabel('b'); xlim([0,8]); ylim([0,6])
```

Try these commands for yourself to see what they do.

1.2.2 Scripts

If you want to keep a series of commands, instead of writing them every time at the prompt, you can write them in a text file. This is a "script". You can use your favorite text editor to edit your scripts, but the matlab interface offers an editor which has some useful functionalities for syntax highlighting and execution. I use Xemacs, for which there is a matlab mode with equivalent functionalities. Matlab scripts have the extension `.m`.

If I type at the prompt the name of my text file (without the extension) then Matlab will look in its current directory a text file with this name, and execute (interpret) one often the other the commands which are written in this script, just as if I had written them at the prompt. Here is a very simple script:

```
% voici mon script

a=[2, 5, 6];
b=[ 3.2, 3.5, 5];

plot(a,b) % je trace le graphique
```

The characters which follow `%` are not interpreted, this is a comment. These comments are useful to remember what a script or block of commands were thought to do, or to help a user which has not coded the script himself.

1.2.3 Current directory

To be able to execute the script, it must be in your current directory, just as in the linux/Unix shell. The commands associated to the notion of current directory are just as in linux:

- `pwd`: "Print Working Directory" print on screen the address of the current directory.
- `ls`: "List Directory" prints on screen the list of files and directories in your current directory.
- `cd`: "Change Directory" to change the current directory.

If you use the matlab editor and your script is not in the current directory, then a dialog window will offer you to change the current directory.

1.2.4 Loops and tests

To realize repetitive actions, use the `for` loop. For instance:

```
a=7
for ind=1:10
  a*ind
end
```

I create an array called a, and for the index ind going from 1 to 10 with steps of 1, I print on screen the value of a times ind. We will see later that the syntax 1:10 is in fact a shortcut to declare linearly space arrays. The previous list of command is equivalent to

```
a=7
vec=1:10
for ind=vec
  a*ind
end
```

I can use the test structure `if` to execute a block of commands only if a certain condition is true:

```
a=2; b=3;
if a>b
  disp('a is larger than b')
else
  disp('a is smaller than b')
end
```

You must realize that `a>b` is a binary value: true or false. in matlab, true is 1 and false is 0. here disp is a function which prints to screen the value of its argument. Its argument can be a numeric variable, or as here a string of characters, denoted with the special quote character `' '`. Just as for the loop above, the value of the logical on which the test is made can be stored in a variable:

```
a=2; b=3;
test=a>b
if test
  disp('a is larger than b')
else
  disp('a is smaller b')
end
```

Here the variable "test" is a scalar, that is a one line one column array whose value is 1 or 0. Note that the expression `if test` means "if `test` is true" and is a shortcut for the more precise expression `if test==1`. The operator`==` is the equality test, just as `>` is the "large then" test and `<` is the "smaller than" test. Please do not mix up the equality test `==` with the assignment operator `=` which assigns a value to a variable. The test "is not equal is written `~=`, since `~` is the negation operator: if `a` is true, then `~a` is false.

To kill a process: It happens that a command takes much time to process because you made a mistake in the coding, or you might change your mind before completion of the computation to change a parameter. To stop the execution and get the prompt back, strike CTRL-c, just as for the Linux shell.

1.2.5 Simple functions of Matlab

There are many functions in Matlab to which you give "input arguments" and which give back to us "output arguments". What happens inside the function we are not much interested in and can be extremely complicated, but we are just interested in the function output. We have the basic mathematical functions:

```
cos: cosinus
tan: tangente
exp: exponentielle
sqrt: racine carrée
log: logarithme
abs: valeur absolue
sinh: sinus hyperbolique
cosh: cosinus hyperbolique
tanh: tangente hyperbolique
erf: fonction erreur
...
```

and so on, for the time being we have given scalar input arguments and got back scalar output arguments, but we will see later that if you give array input arguments, then matlab will apply the function to each of the element of the array and give you back an array of same size. These sounds trivial at first sight, but this in fact is a key to opening the door to bright perspectives of possibilities of quick and efficient coding.

You have other kinds of functions, as for instance

```
a=num2str(2.3)
```

which transform the numerical value of the input argument into a string of character, which will be useful to build screen outputs of your liking. You can easily imagine what the function `str2num` will do for you.

1.2.6 Create your own functions

You can create your own functions, for this you create a text file with the desired function name, with a ".m" extension as for a script, and with a special header. here is an example of text file to create the fonction "testfunction"

```
function [s1,s2]=testfunction(a,b,c)
s1=a+b;
s2=s1+c;
```

This function has three input arguments `a`, `b`, and `c`, and gives back two output arguments `s1` and `s2`. The type and number of argument is arbitrary. they can be arrays or whatever else. No need to worry for this as in fortran for instance. Here is an example of the use of your new function

```
toto=1
pilou=3
[p,r]=testfunction(toto,pilou,10);
disp(p)
disp(r)
```

1.2.7 Matlab Help

If you have forgotten the details of the use of an operator, use the "help" function, for instance

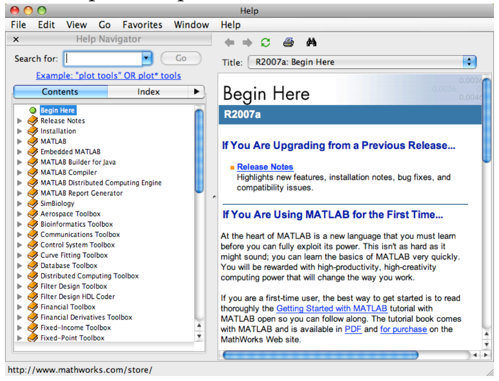
```
>> help for
 FOR    Repeat statements a specific number of times.
    The general form of a FOR statement is:

       FOR variable = expr, statement, ..., statement END

    The columns of the expression are stored one at a time in
    the variable and then the following statements, up to the
    END, are executed. The expression is often of the form X:Y,
    in which case its columns are simply scalars. Some examples
    (assume N has already been assigned a value).

         for R = 1:N
             for C = 1:N
                 A(R,C) = 1/(R+C-1);
             end
         end

    Step S with increments of -0.1
         for S = 1.0: -0.1: 0.0, do_some_task(S), end

    Set E to the unit N-vectors
         for E = eye(N), do_some_task(E), end

    Long loops are more memory efficient when the colon expression appears
    in the FOR statement since the index vector is never created.

    The BREAK statement can be used to terminate the loop prematurely.

    See also if, while, switch, break, continue, end, colon.

    Reference page in Help browser
       doc for
```

One of the most useful aspect of this help, is that it offers in "see also" at the bottom of its output, a list of other functions of Matlab which are related to the help that you have asked for. This is the way you will learn the very large possibilities offered by the Matlab library. This large library and the efficient help which leads you through, together with the easy syntax is really what makes Matlab something special and useful.

If you wonder how a special operator works, for instance verb!:! (”colon”), type

```
>> help colon
help colon
 :  Colon.
    J:K  is the same as [J, J+1, ..., K].
    J:K  is empty if J > K.
    J:D:K  is the same as [J, J+D, ..., J+m*D] where m = fix((K-J)/D).
    J:D:K  is empty if D == 0, if D > 0 and J > K, or if D < 0 and J < K.

    COLON(J,K) is the same as J:K and COLON(J,D,K) is the same as J:D:K.

    The colon notation can be used to pick out selected rows, columns
    and elements of vectors, matrices, and arrays.  A(:) is all the
    elements of A, regarded as a single column. On the left side of an
    assignment statement, A(:) fills A, preserving its shape from before.
    A(:,J) is the J-th column of A.  A(J:K) is [A(J),A(J+1),...,A(K)].
    A(:,J:K) is [A(:,J),A(:,J+1),...,A(:,K)] and so on.

    The colon notation can be used with a cell array to produce a comma-
    separated list.  C{:} is the same as C{1},C{2},...,C{end}.  The comma
    separated list syntax is valid inside () for function calls, [] for
    concatenation and function return arguments, and inside {} to produce
    a cell array.  Expressions such as S(:).name produce the comma separated
    list S(1).name,S(2).name,...,S(end).name for the structure S.

    For the use of the colon in the FOR statement, See FOR.
    For the use of the colon in a comma separated list, See VARARGIN.

    Overloaded functions or methods (ones with the same name in other directories)
       help sym/colon.m

    Reference page in Help browser
       doc colon
```

If instead of help, what you need is ”documentation”, use the command `doc` which open a separate window.



Begin Here

R2007a

If You Are Upgrading from a Previous Release...

- Release Notes
  Highlights new features, installation notes, bug fixes, and compatibility issues.

If You Are Using MATLAB for the First Time...

At the heart of MATLAB is a new language that you must learn before you can fully exploit its power. This isn't as hard as it might sound; you can learn the basics of MATLAB very quickly. You will be rewarded with high-productivity, high-creativity computing power that will change the way you work.

If you are a first-time user, the best way to get started is to read thoroughly the Getting Started with MATLAB tutorial with MATLAB open so you can follow along. The tutorial book comes with MATLAB and is available in PDF and for purchase on the MathWorks Web site.

There are many things in Matlab; it is when you will master its help that you will master it really, and truly use it as a ”high level programming language”.

1.2.8 Special characters

These are the characters which play a special role in the syntax, here is a short list:

- `[ ]` the square brackets to concatenate arrays.
- `;` semicolon, to put at the end of a command whose result you don’t want printed on screen. Also for the array concatenation: going to the next line.
- `:` ”colon”, to define a equispaced array (geometric progression): `3:10` is the same array as the line array `[3,4,5,6,7,8,9,10]` similarly, `0:0.1:1` is the same array as `[0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]`
- `,` The comma, to put at the end of a command if you have several commands on one line of your text file, if you want the result to be printed on screen. Also for array concatenation: to separate the elements of the concatenation without going to the next line.
- `==` logical operator of equality.
- `=` to be pronounced ”receive” and not ”equal”, to assign a value to a variable.
- `' '` Single quotes to define a string a characters.
- `*` Matrix multiplication.
- `.*` Array multiplication: element by element, and not matrix multiplication.
- `./` Array division, element by element.
- `/` Matrix division: `A/B` for two square matrices A and B is equivalent to `A*inv(B)`.

1.3 Arrays

In this section, we will see that we can do many things with arrays, with very short commands. These will be useful in the following when manipulation interesting data: analyze and graphs.

1.3.1 Build arrays by concatenation

The simplest array has one line and one column, it is a ”scalar”

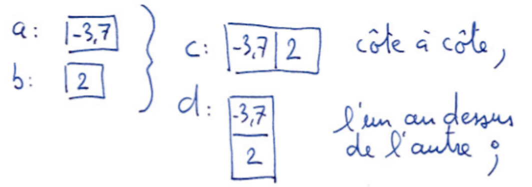
```
a=-3.7;
b=2;
```

I have now two variables, `a` and `b` which have the same size and both contain a real number. I can concatenate these two arrays to get a larger array

```
c=[a,b];
```

the brackets `[ ]` are the concatenation symbols. This command puts in c an array by putting a and b ”side by side”, this being specified by the character. If I want to put `a` et `b` ”one above the other”, I use the brackets with a semi-colon

```
d=[a;b];
```

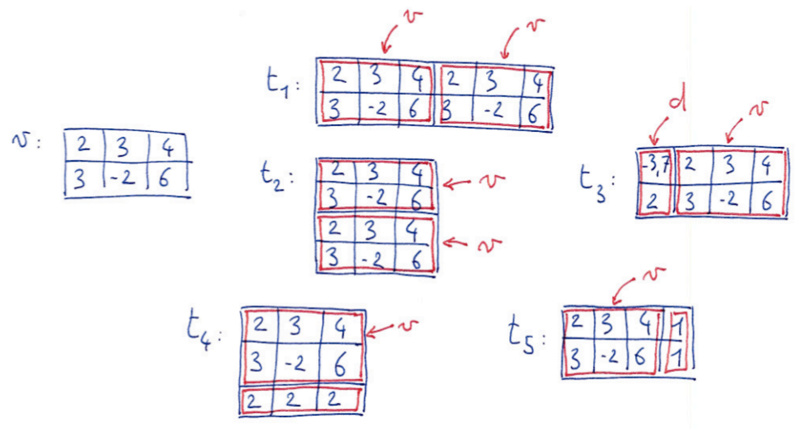


I can build an array by directly concatenating numbers

```
v=[2, 3, 4; 3, -2, 6];
```

I thus have in `v` an array with two lines and one column. I can do many other manipulations of that sort, you just have to think of the blocks you can put next to each others, either side by side or one above the other. The only constraint is that the blocks must be rectangles. see the commands below and the graphical representation of what is going on.

```
t1=[v,v]
t2=[v;v]
t3=[d,v]
t4=[v; 2, 2, 2]
t5=[v,[1;1]]
```



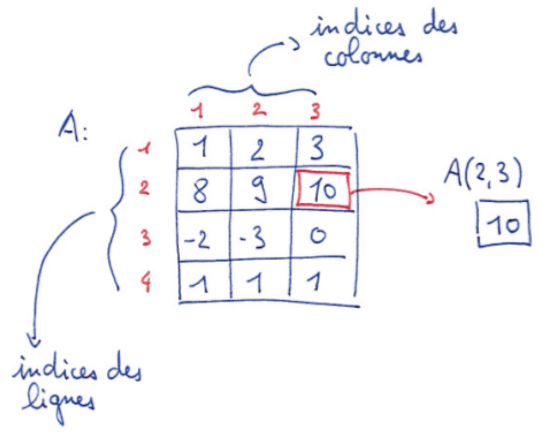
1.3.2 Access to subarrays

I the previous section, we have seen how to create arrays by concatenating smaller arrays, here we will create smaller arrays by selecting subarrays, or directly change the values that are memorized in the subarrays of existing arrays of your workspace.

Suppose we have the array

```
A=[1, 2, 3; 8, 9, 10; -2, -3, 0; 1, 1, 1];
```

which is an array with 4 lines and 3 columns. The simplest sub-array is a scalar element, you can access it by its line and column index, for instance `A(2,3)` is an array of one line and one column which contains the value 10. here the first index is the index for the line, and the second index is the index for the column:



I can do:

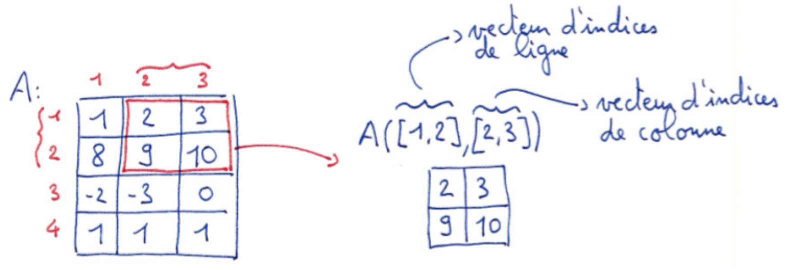
```
b=A(2,3)+A(1,1)
```

here I store in a new variable "b" the sum of the (2,3) and (1,1) elements of A.

You can just the same way access subarrays of A, by putting now arrays of indices:

```
c=A([1,2],[2,3])
```

now c is an array with two lines and one column:



I can as well put the array of indices in a variable instead than explicitly if it fits my needs:

```
vl=[1,2];
vc=[2,3];
c=A(vl,vc);
```

is an equivalent sequence to the previous one. it will prove sometime very useful, for instance when the selection of the indices is subtile, or when extracting the same sub-array of many arrays.

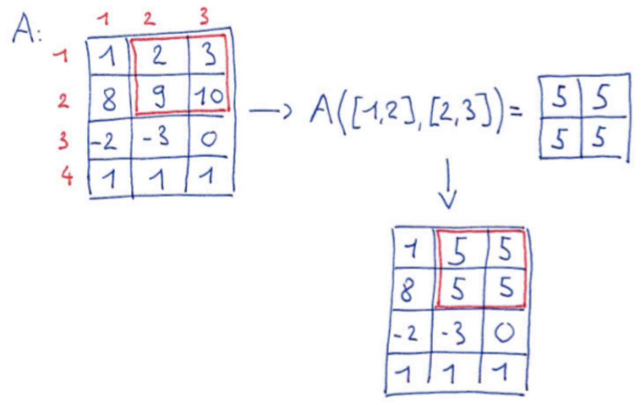
One interesting thing is that you can directly change the values of sub-arrays. First here for just one element:

```
A(2,3)=2;
```

Here we change only the element of the second line third row, but you can change directly a larger sub-array

```
A([1,2],[2,3])=[5, 5; 5, 5]
```

for this to work, the array on the right must be the same size than the array on the left.



1.3.3 Array operations

All the arithmetic operations that we routinely apply to scalars can be extended to arrays just by thinking that these are applied "element by element". We will just have to be careful with operation which naturally have a "matrix" meaning, as for instance array multiplication (element by element) is different from matrix multiplication which is not an element by element operation, but something more complex..

Suppose we have two arrays

```
A=[1, 2; 4, -1];
B=[0, -5; 2, -2];
```

then

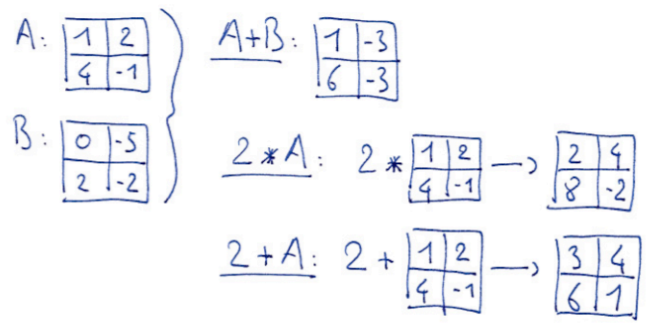
```
C=A+B
```

consist in storing in the `C(i,j)` element, the sum of the elements`A(i,j)` and `B(i,j)`. For instance now `C(2,2)` stores -3.

The two following operations

```
D=2*A
E=2+A
```

consist in storing in `D` the elements of `A` multiplied by two, and to store in `E` the elements of `A` to which we add 2 (to all of them).



The array multiplication is also an operation element by element and it is written `.*` and not `*`.The command

```
C=A.*B
```

stores `A(i,j)*B(i,j)` into `C(i,j)`. It is equivalent to the following:

```
C=[0, 0; 0, 0];
for i=[1,2]
  for j=[1,2]
    C(i,j)=A(i,j)*B(i,j)
  end
end
```

Which should not be mixed-up with the matrix multiplication, `*`:

```
C=A*B
```

which considers `A` and `B` as element of the matrix algebra. this instruction is equivalent to

```
C=[0, 0; 0, 0];

for i=1:2
  for j=1:2
    for k=1:2
      C(i,j)=C(i,j)+A(i,k)*B(k,j);
    end
  end
end
```

We will see that Matlab is very useful to manipulate matrices, to solve linear equations, calculate eigenmodes... after all, matlab means ”MATrix LABoratory”...

You can as well apply mathematical functions to arrays

```
C=sin(A)
D=exp(B)
E=cos(A)+tanh(B)
```

Which is a large gain of coding effort, since one single command applies the operation to all the elements. For instance the third command above is equivalent to

```
E=[0, 0; 0, 0];

for i=[1,2]
  for j=[1,2]
    E(i,j)=cos(A(i,j))+tanh(B(i,j))
  end
end
```

Without this possibility, we have seen in the previous examples that you would need to use several nested `for` loops to go through all indices, which would make your coding considerably heavier.

1.3.4 Special array functions

After having seen functions which apply element by element, and which thus affect arrays just as if they were scalars, we will now see functions which are used to create and manipulate arrays.

Creating arrays

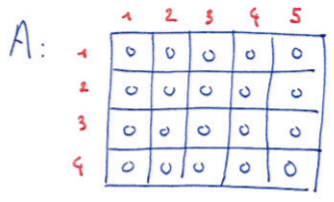
to create an array filled with zeros, use the function `zeros`

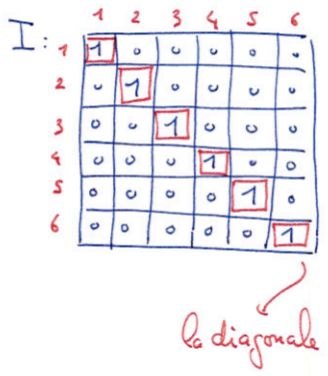
```
A=zeros(4,5)
```

here, `A` becomes an array with four lines and five columns, filled with zeros. You will easily guess what the function `ones` does. We also have the `eye` function

```
I=eye(6)
```

which builds an identity matrix: filled with zeros except for the diagonal elements which are ones. ”Eye” for ”eye-dentity”.





here is a function which will be useful to vary the physical parameters in a model, the `linspace` function, which creates line arrays

```
v=linspace(0,1,65)
```

here , v is a line array of 65 equidistant elements from 0 to 1. This command is equivalent to

```
v=0:1/64:1
```

which is still an equivalent of the explicit assignment

```
v=zeros(1,65);
for i=1:65
  v(i)=(i-1)/64
end
```

The function ”zeros” will be useful to create new variable with the desired size: fill it with zeros and you will later one use it for your own purpose.

Extracting information from arrays

For an array `A`, the `max` function gives as output a line array which stores the largest element of each column of `A`

```
>> A=[2, 3, 4; 1, 4, 4]
>> v=max(A)
v= 2 4 4
```

to have the largest element of `A`, just do:

```
m=max(max(A))
```

you can as well get the index of this largest element, for this look i the matlab help for the output arguments of ”max”. The `min` function is similar. We find this often in matlab: special array function work column-wise.

You can compute the sum and the product of the elements of an array

```
v=sum(A)
r=prod(A)
```

here too, the dimension of the output argument is equal to the the dimension of the input argument minus one because the function acts column-wise. Thus for the sum of all the elements of the array, just do `sum(sum(A))`.

1.4 Vectorization

Vectorization—in fact here we should rather say*array-ization*—is a philosophy of programming which consist in avoiding loops and scalar manipulations. In this section, we will see a few directions and examples, you will soon realize that we have already used many vectorized formulations, which are very natural in Matlab.

here a series of example. Creation of an array of zeros, scalarwise:

```
x=[];
for i=1:20
  x=[x, 0];
end
```

vectorized:

```
x=zeros(1,20);
```

by using a predefined function `zeros`. Construction of an array of element linearly spaced between 0 and 2π, scalarwise

```
for i=1:20
  x(i)=2*pi*(i-1)*1/19;
end
```

vectorized:

```
x=linspace(0,2*pi,20);
```

by using a predefined function `linspace`. Computation of the sinus of these values, scalarwise:

```
for i=1:20
  f(i)=sin(x(i));
end
```

vectorized:

```
f=sin(x)
```

graph of these array x, scalarwise:

```
for i=1:19
  line([x(i),x(i+1)],[f(i),f(i+1)]);
  hold on
end
hold off
```

where we have drawn one by one all the segments to link the points of consecutive coordinates. Vectorized:

```
plot(x,f)
```

by using the predefined function `plot`. Computation of the largest value in the array `f`, and checking where this largest value is stored, scalarwise:

```
maxval=-inf;
indloc=0;
for i=1:20
  if f(i)>maxval;
    maxval=f(i);
    indloc=i;
  end
end
```

here, `inf` is the infinite value, larger than any other. Vectorized:

```
[maxval,indloc]=max(f);
```

using the predefined function `max`. We could go on with this list for a long while. For instance, think how you could code by yourself what the `sort` function does.

Until now, to vectorize, we have had to know many matlab functions. Matlab is a "high level" programming language, not because you need to be very intelligent to use it, but because it allows to skip elementary manipulations, said as "low level" operations. Why, because these low level operation are already coded for you in the most efficient manner in a very large library of functions. We will discuss in a later section how we can improve as well the computation speed by using predefined functions. Now a few examples which do not involve necessarily predefined functions.

We seek to count the number of element which are equal to π in a given array `v`. Scalarwise:

```
n=0;
for i=1:length(v)
  if v(i)==pi;
    n=n+1;
  end
end
```

in this example, `n` is a variable which we use to count. Vectorized:

```
n=sum(v==pi);
```

here, `v==pi` is a logical array with zeros in the elements for which `v` is not equal to π and ones for the elements where `v` is equal to π. The sum of the elements of this array, ones and zeros with `sum` is just nothing else than the count of elements which are equal to π. Now a little more subtle, we would like to know the indices of the elements which are equal to π. Scalarwise:

```
indlist=[]
for i=1:length(v)
  if v(i)==pi;
    indlist=[indlist, i];
  end
end
```

vectorized:

```
k=1:length(v);
indlist=k(v==pi);
```

in this example, we started by building an auxiliary array `k` which is the array of all the indices $(1, 2, 3, ...)$, and we have selected in this array only the elements corresponding to the value π in `v`. If we take down this command in successive small steps, we get this:

```
>> v=[0,2,4,pi,0,pi]
v =
         0    2.0000    4.0000    3.1416         0    3.1416
>> v==pi
ans =
     0     0     0     1     0     1
>> v(v==pi)
ans =
    3.1416    3.1416
>> k=1:length(v)
k =
     1     2     3     4     5     6
>> k(v==pi)
ans =
     4     6
```

I first built my array `v`, and storing π in position four and six. I check at the prompt what the `v==pi` array looks like, this is a binary array with zeros everywhere except at position four and six... I test `v(v==pi)`, that is, I check which elements of `v` are egual to π, the result is good. I then build my auxiliary array `k`, and I check which elements of `k` are are corresponding to the elements of `v` which are equal to π, and I find what I expected.

in fact, for all these, we could have used the predefined `find` function, which gives directly the indices of nonzero values of the input argument `v==p`.

```
>> find(v==pi)
ans =
     4     6
```

1.4.1 More advanced vectorization

Here is an other example for computing the velocity of a point whose successive positions are stored in an array, say `x`, with one line and n columns. We use centered finite difference, and the time step value is `dt`. Scalarwise

```
v=zeros(1,n);
for ind=2:n-1
  v(ind)=(x(ind+1)-x(ind-1))/(2*dt);
end
```

where we go through a loop for every value of the index; and vectorized:

```
v=(x(3:end)-x(1:end-2))/(2*dt);
```

where we have subtracted two shifted subarrays of `x`. We note that if we had wanted to have the velocity for the first and last indices, we would have had to use uncentered finite differences.

An other related example, here to compute the integral of a function whose values are stored in the array `f` with one line and `n` columns, with linearly spaced grid points with spacing `dx`. We use the trapezoid rule. Scalarwise:

```
s=0;
for ind=1:n-1
  s=s+dx*(f(ind)+f(ind+1))/2;
end
```

and vectorized:

```
s=dx*sum(f(1:end-1)+f(2:end))/2;
```

where we use again the shifting of subarrays and the `sum` function. This we may code in a subtly different manner, trough building a new array which shall do the weighting of the different values of the array `f`. Indeed the integral is

$$s = (f_1 + 2f_2 + 2f_3 + \cdots + 2f_{n-1} + f_n)dx/2,$$

which can be computed as

```
S=[1,2*ones(1,n-2),1]*dx/2;
s=S*f;
```

where `S` is a line array and we suppose `f` is a column array, thus the summing of the integration coefficients times each of the element of `f` is done through the matrix multiplication, `s` is thus a scalar. In this example we have built an *auxiliary array* `S`. The same way we can build an auxiliary array to compute the derivative of a function:

```
D=zeros(n,n);
D(1,1:3)=[-3/2, 2, -1/2]/dx;
for ind=2:n-1
    D(ind,ind-1:ind+1)=[-1/2, 0, 1/2]/dx;
end
D(end,end-2:end)=[1/2, -2, 3/2]/dx;
```

now it is straightforward to compute an approximation of the derivative of f by a matrix-vector multiplication `fx=D*f`. In this example we have use decentered finite differences at the two end grid points. Note that we have used a `for` loop to build our differentiation matrix `D`, where we could have used the `toeplitz` function of matlab, which builds matrices by setting values to its diagonals, see this example for instance (or see the help for more details)

```
>> toeplitz([1 2 0 0 0],[1,-2,0,0,0])
ans =
     1    -2     0     0     0
     2     1    -2     0     0
     0     2     1    -2     0
     0     0     2     1    -2
     0     0     0     2     1
```

Thus our differentiation matrix built without a loop is

```
col=[0,-1/2,zeros(1,n-2)]/dx;
lin=[0,1/2,zeros(1,n-2)]/dx;
D=toeplitz(col,lin);
D(1,1:3)=[-3/2, 2, -1/2]/dx;
D(end,end-2:end)=[1/2, -2, 3/2]/dx;
```

This code is not much shorter, especially since we need to take care of the special decentered finite difference on the first and last line, but it shall be faster if the matrices are big (many grid points).

1.4.2 Even more advanced

Here is a graphical example which makes use of the very nice function `kron`. This function is the top of the hill for mastering vectorization, it will do all the things you have no other idea how to do without loops. How it works is simple:

```
>> help kron
 KRON   Kronecker tensor product.
    KRON(X,Y) is the Kronecker tensor product of X and Y.
    The result is a large matrix formed by taking all possible
    products between the elements of X and those of Y.   For
    example, if X is 2 by 3, then KRON(X,Y) is

       [ X(1,1)*Y  X(1,2)*Y  X(1,3)*Y
         X(2,1)*Y  X(2,2)*Y  X(2,3)*Y ]
```

for instance:

```
>> kron([1,2],[1,1;1,1])
ans =
     1     1     2     2
     1     1     2     2
```

Our problem is the following, you want to draw many circles with varying radius and varying center position. We will use this in one of our courses when looking at the flow of sand grains down an incline. Here 1000 circles with random position and radius

```
n=1000;
x=rand(n,1);
y=rand(n,1);
r=0.01*randn(n,1).^2;
```

we can code it simply with a loop:

```
theta=linspace(0,2*pi,20)';
xc=cos(theta);
yc=sin(theta);
for ind=1:n;
  plot(xc*r(ind)+x(ind),yc*r(ind)+y(ind));
  hold on
end
```

we first have built the coordinates along a circle of radius 1 and centered at $(0, 0)$, then at each iteration scaled the circle and translated it, and plotting. Now we will use the `plot` command once for all, but at first building the x and y coordinates for all the circles:

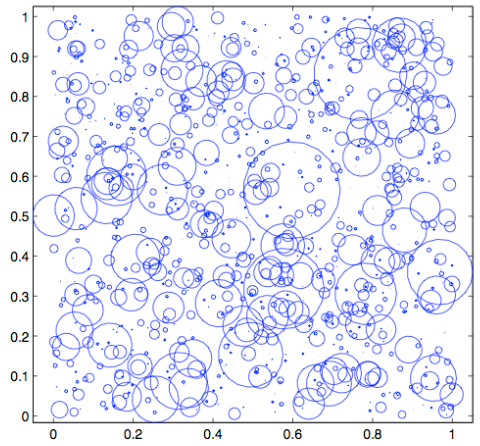
```
X=[];
Y=[];
for ind=1:n
  X=[X;xc*r(ind)+x(ind);NaN];
  Y=[Y;yc*r(ind)+y(ind);NaN];
end
plot(X,Y);
```

Here we have initiated `X` and `Y` with empty arrays `[]`, and added at each iteration new elements (reshaping the arrays). Note in passing that this is not a good idea for speed, see in dedicated section. Also we have used a nice trick, by inserting NaN in the coordinates after each circle. The idea behind is that the `plot` function links with straight segments the successive points in the coordinate arrays, except if the coordinate is an NaN ("Not a Number"). Thus these NaNs can be very useful sometimes!

Now we do the same thing with `kron` instead of a loop:

```
X=kron(r,[xc;NaN])+kron(x,[ones(20,1);NaN]);
Y=kron(r,[yc;NaN])+kron(y,[ones(20,1);NaN]);
plot(X,Y);
```

The first `kron` takes care of multiplying the coordinates of each circle with its radius, the second `kron` takes care of the translation. Take ten minutes for yourself, and draw the arrays on a piece of paper to understand that these lines do precisely the same thing as the previous block of command (but much faster on computer time). Once you are done with this, you have made a definitive step forward.



Vectorization is good for several reasons:

- Avoiding `for` loops. This helps to skip command lines. This is an economy of time and space, this is an economy of potential coding errors and mistakes.

- in Matlab, which is an interpreted language, it is much faster in computing time. See the related section.

1.5 Computing faster

Matlab is an interpreted language. Interpretation is time consuming. But: most commands in matlab are compiled functions written in `c`. For instance the linear algebra functions are those from BLAS and LAPACK, highly optimized to take advantage of your machine architecture and using all possible speed enhancement tricks. So all the computing time spent inside these functions is well spent, in a way as well as possible. This is why it is a bad idea to code in Matlab a linear equation solver (bad idea for speed, not necessarily for learning...). Here is an example, multiplication of two matrices of size 1000:

```
n=1000;
A=randn(n,n);
B=randn(n,n);
```

```
disp('Using compiled function:')
tic
C=A*B;
toc

disp('Using your own nested loops:')
tic
D=zeros(n,n);
for i=1:n
    for j=1:n
        for k=1:n
            D(i,j)=C(i,j)+A(i,k)*B(k,j);
        end
    end
end
toc
```

and the output is:

```
Using compiled function:
Elapsed time is 0.119633 seconds.
Using your own nested loops:
Elapsed time is 18.722617 seconds.
```

the result is clear.

An other time consuming task in Matlab consist in creating a new variable: Matlab will look for a consecutive memory space, create this consecutive space it by manipulating the memory if necessary (big arrays). Every variable is an array in Matlab, this is a complex variable type, so there is much overhead information: size type and so on coming with it. So having many scalar arrays (one line and one column) is a bad use of memory space.

An other bad idea consist in changing the size of an existing array. here is a simple example, both ways with a loop, but we allocate first the memory for the second method:

```
disp('reshaping')
n=10000;
tic
f=[];
```

```
for ind=1:n;
    f=[f, ind];
end
toc

disp('allocating')
tic
f=zeros(n,1);
for ind=1:n
    f(ind)=ind;
end
toc
```

and the output is

```
reshaping
Elapsed time is 0.491824 seconds.
allocating
Elapsed time is 0.000159 seconds.
```

An other time consuming thing is to write numbers on the screen, and draw plots in the figure windows.

To conclude: use as much as possible precompiled functions. Build arrays first and do all the treatment at once then. For all this, the previous section on vectorization is useful. Allocate your arrays at proper size first. Restrict graphical/text output to minimum. Also, when manipulating large arrays, think to clear memory space for unused variables, using the function `clear`.

1.6 Graphs

Data must be *seen*. A good programming skill in graphics is key to accessing much more information than otherwise. This is how you can learn new knowledge from your data. Also a good graphic skill is key to transmission of the information. The main functions are `plot` for line plots, and mesh `mesh` for surfaces. We will see then other functions related to contour plots, vector plots and so on. Also, animation is a good way to see how things behave, we will see how to use the function `drawnow` to see your data moving.

1.6.1 Lines

We will visualize the mathematical formula

$$f(x) = sin(x)e^{\frac{-x^2}{10}}$$

It is composed of an undulating sinus, multiplied by a gaussian envelope e^{-x^2} .

Let us first define the array of the abscissa in x, and compute the value of the function at these abscissa. We will avoid loops by using vectorized expressions.

```
n=200
x=linspace(-10,10,n);
f=sin(x).*exp(-x.^2/10);
```

with n the number of points in this graph. When I type

```
plot(f)
```

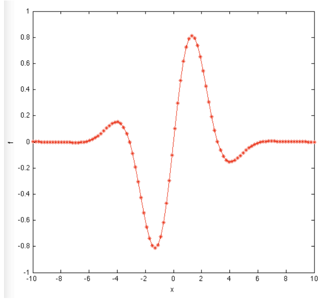
I did not specify the values of the abscissa, I only gave the ordinates. In this case, matlab supposes that the array of the abscissa is simply $(1, 2, 3, 4...)$, until n. To get the proper representation of the formula, you should type:

```
plot(x,f)
```

I can chose the color and the line style

```
plot(x,f,'r*--')
```

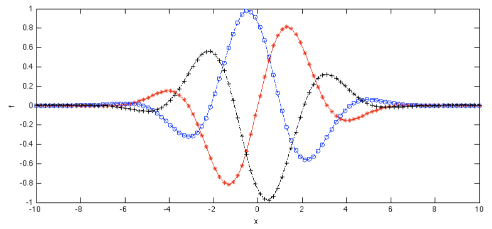
here I ask for a line in red (`r`), with star symbols (`*`), and a dashed line (`--`).



A few colors: "b" for blue, "k" for black, "m" for magenta, "c" for cyan... and for line styles "-" for continuous line, "–" for dash, "-." for dash-dot... For more information, call for help `help plot`.

I can as well superimpose several curves, for instance by putting several triple abcissa/ordinate/line specification

```
f1=sin(x).*exp(-x.^2/10);
f2=sin(x+2*pi/3).*exp(-x.^2/10);
f3=sin(x+4*pi/3).*exp(-x.^2/10);
plot(x,f1,'r*-',x,f2,'bo--',x,f3,'k+-.')
```



here I put three formulas by having a phase shift of $2\pi/3$ and $4\pi/3$ on the sinus. now you stat to see what the "enveloppe" means.

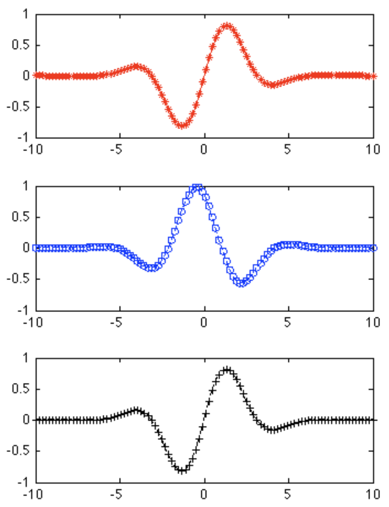
We can use an alternative way to superimpose lines, with the function `hold on` and `hold off` with the following sequence

```
plot(x,f1,'r*-');
hold on
plot(x,f2,'bo--');
plot(x,f3,'k+-.');
hold off
```

which is equivalent to the former one. This will be particularly useful when we draw graphs in loops `for`. The "hold on" means that the graphic commands following will not erase the previous graphs.

if you want to see several curves at the same time but not on the same graph, you can divide the graphic figure into sub graphics using the functionsubplot

```
subplot(3,1,1); plot(x,f1,'r*-');
subplot(3,1,2); plot(x,f2,'bo--');
subplot(3,1,3); plot(x,f3,'k+-.');
```



`subplot(nl,nc,n)` divides the graphical window into *nl* lines and *nc* columns and draw the commands into the subgraph number *n*, counting from left to right and from top to bottom. in the above example there are three lines and one column.

If you want to see your data in the form of an animation, it is convenient to use a `for` loop

```
for t=linspace(0,20,300)
  plot(x,sin(x+t).*exp(-x.^2/10),'r*-')
  ylim([-1,1])
  drawnow
end
```

We have used the command `drawnow`, which as its name indicates, force Matlab to draw at the time when it reads the command and not after the complete execution of the script. By default the graphic treatment is*asynchronous*, that is, matlab takes care of the graphs once all the rest is done. For an animation this is off course not the good choice.

In this example we give a continuous variation of the phase shift by adding t to x. This gives the feeling of a wave moving to the left. The enveloppe is fixed, but the undulations are moving. We can as well have the enveloppe moving, for instance to the right, by adding a t dependency in the formula for the enveloppe:

```
for t=linspace(0,20,300)
  plot(x,sin(x+t).*exp(-(x-t).^2/10),'r*-')
  ylim([-1,1])
  drawnow
end
```

in fact, you may change whatever you like in the course of the loop, for instance in the following example, I have the line color changing from white to black, by using the `'color'` property. The color code by default is "rgb", that is, "red-green-blue": `[1,0,0]` is pure red, `[0,1,0]` is pure green, and `[0,0,1]` is pure blue. Also, `[1,1,1]` is black and `[0,0,0]` is white. I can also have the line width varying with the property `'linewidth'`, here from 0 to 20:

```
for t=linspace(0,20,300)
  plot(x,sin(x+t).*exp(-(x).^2/10),'color',1-[1,1,1]*t/20, ...
              'linewidth',20*t/20+1)
  ylim([-1,1])
  drawnow
end
```

I have used here a usual functionality of the input arguments in matlab, by inserting "property-value" couples , where the name of the property is a string, here `'color'` for the line color and `'linewidth'` for the width of the line.

1.6.2 surfaces

We shall now draw surfaces. Let us consider the formula $sin(x)e^{-y^2}$ which behaves like a sinus along x and like a gaussian bell in y. We saw in the previous section that before to draw a formula, we must "build it", that is, create arrays of coordinates.

To calculate the value of a mathematical formula in two dimensions, we first create the arrays of the x and y points.

```
n=20;
x=linspace(-5,5,n);
y=linspace(-4,4,n);
```

And now we could calculate the value of the formula for every couple $x(i), y(j)$ by implementing two `for` loops

```
f=zeros(n,n);
for i=1:n
  for j=1:n
    f(i,j)=sin(x(i))*exp(-y(j)^2);
  end
end
```

in fact, we should better use a practical trick to vectorize this operation by using the function `meshgrid`

```
[X,Y]=meshgrid(x,y);
f=sin(X).*exp(-Y.^2);
```

To understand what it does, meditate on the following example:

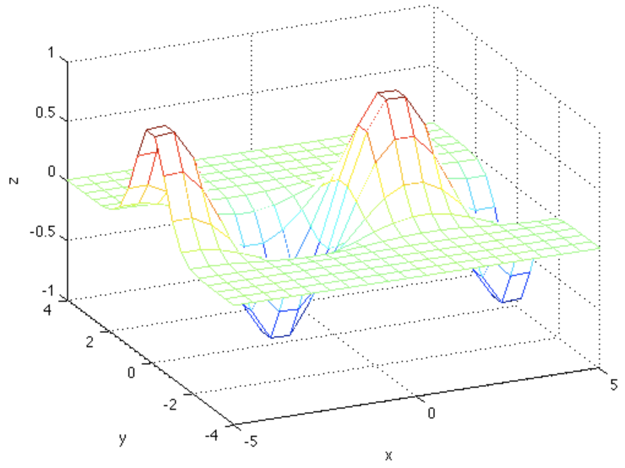
```
>> [X,Y]=meshgrid([1,2,3],[4,5,6])
X =
     1     2     3
     1     2     3
     1     2     3
Y =
     4     4     4
     5     5     5
     6     6     6
```

Thus, `sin(X)` is an array of the same size as `X`, with as values the values of the sinus of the array `X`. Similarly, `Y.^2` is an array of same size as `Y` with for values the square of the values of `Y`, and again in a similar way, `exp(-Y.^2)` is the array which has as elements the exponential of the square of the elements of `Y`. This way, `sin(X).*exp(-Y.^2)` is actually what we are trying to build. The use of `mesgrid` is typical of vectorization: instead of having loops, we manage to build convenient arrays. Now you maybe start to have a feel for the real subtlety of vectorization: having the ideas of the auxiliary arrays that you will need for your own purpose needs good understanding, skill and creativity. It is a good investment to take the time to understand properly what `meshgrid` does and visualize for yourself the arrays that are created.

Once this made, we can draw our graph f

```
mesh(X,Y,f);
xlabel('x');  ylabel('y'); zlabel('z');
title('sin(x)exp(-y^2)');
```

We have used the `mesh` function. if you wish to change the limits of the axes, you can use the `xlim, ylim, zlim` commands as before.



We will now generate a little animation, by as previously adding in our code a loop and a time dependency parameter

```
for t=linspace(0,2*pi,30)
  f=sin(X+t).*exp(-Y.^2);
  mesh(X,Y,f)
  drawnow
end
```

We can change whatever we like during this animation, for instance here we have the camera orbiting

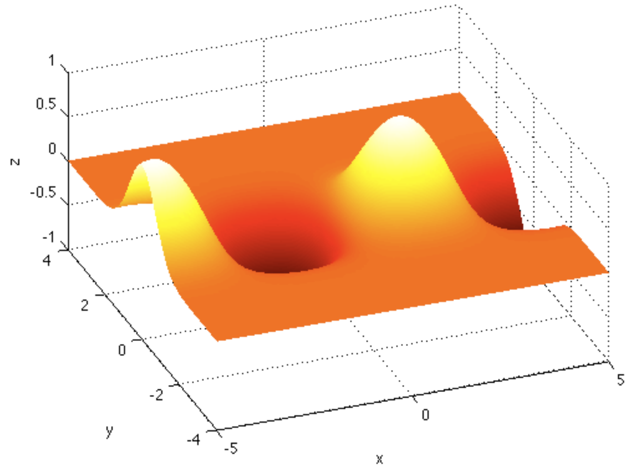
```
for t=linspace(0,2*pi,30)
  mesh(X,Y,sin(X+t).*exp(-Y.^2));
  camorbit(380*t/(2*pi),0);
  drawnow
end
```

The `camorbit` function makes the camera (the point of view) turn of a given angle. here it makes a full turn (from 0 to 380 degrees) while t goes from 0 to 2π.

If you like a continuous surface rather than a mesh, use `surf`

```
surf(X,Y,sin(X+t).*exp(-Y.^2));
shading interp;
colormap(jet(400))
```

I used the function `shading interp`, which interpolate the colors between the surface vertices, and I changed the colors with `colormap`, by using the "jet" colormap and using 400 shades of color instead of the default 64 (I have 100 grid points in x and y).



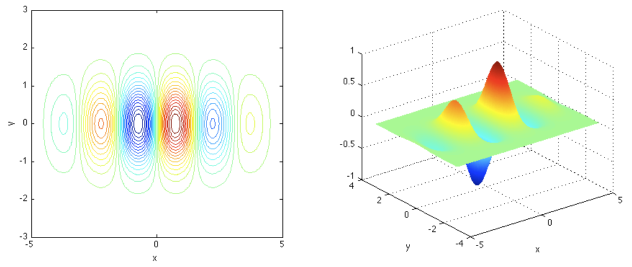
1.6.3 Isovalues

It is sometimes useful to draw isovalues. For instance the isovalues of the streamfunction in fluid mechanics corresponds to streamlines. We continue with the same 2D function as above

```
n=200;
x=linspace(-5,5,n);
y=linspace(-3,3,n);
[X,Y]=meshgrid(x,y);
f=sin(2*X).*exp(-Y.^2).*exp(-X.^2/8);

subplot(1,2,1)
contour(X,Y,f,30)
xlabel('x'); ylabel('y')
```

```
subplot(1,2,2)
surf(X,Y,f); shading interp
xlabel('x'); ylabel('y')
```

Here we have in the left subplot 30 linearly (by default) spaced contours of the function f between the largest value and the smallest value of f. And on the right subplot, the graph of f with `surf`.

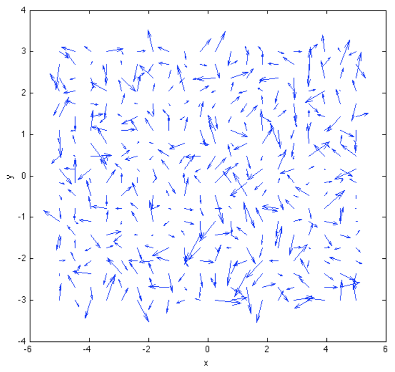


1.6.4 Vector fields

The `quiver` function take as input arguments the arrays of (x, y) coordinates of the start of the vector arrow and the coordinates in (u, v) in x and y of the vectors. Let us take as example a random vector field

```
n=20;
x=linspace(-5,5,n);
y=linspace(-3,3,n);
[X,Y]=meshgrid(x,y);
u=randn(n,n);
v=randn(n,n);
quiver(X,Y,u,v);
```

Here, the `randn` function builds as output argument an array of size (n, n), whose elements are chosen randomly such that the mean is zero and the variance is unity, with a Gaussian probability density function.



We can make a simple animation

```
for t=linspace(0,10,30)
  quiver(X,Y,u+t,v);
  drawnow
end
```

here at each time step, we add t at the horizontal component of the vector field and we draw: order appears in chaos... The `quiver` function by default normalizes the length of the arrow such that they do not overlap. To avoid this normalization and have the true size of the arrows, write `quiver(X,Y,u+t,v,0)`.

1.6.5 The current graph

When you have several graphic windows and/or several subgraphs, the graphic command are by default directed to the "current axes": This is the axis in which you have drawn the last graph, or the axis in which you have clicked last with the mouse, or the axis which was last made active by the `figure` of `subplot` command.

1.6.6 Saving figures

You can save your figure in two ways: you can store it in the format "matlab figure", with file extension `.fig`, or you can save them in an image format. With the fig format, you can open these figures again later on and get them just as you had them when they were created, so you can continue to add graphics and edit them. The image format is the one you need to use the figures in articles/presentation.

To save a figure, in the "file" menu, chose "save" or "save as..." and chose your format. You can also save figures from commands. Here is a little function which I use to save my figures in several formats: .fig .eps and .jpg at the same time. t s very useful to have the .fig figure somewhere if you wish at the last minute to change the font size or any other detail of the figure.

```
function ppp(filename)

% if the last charater of the filename is *
% open the figure from disc instead of saving it
if filename(end)=='*'; filename=filename(1:end-1);
  open([filename '.fig'])

else

% check if the fig allready exists
if exist([filename '.fig'])==2;
  resp=input('replace file? (n?) >>','s');
  if isequal(resp,'n');return;end
end

% so that the figure aspect ratio on screen will be respected
set(gcf,'paperpositionmode','auto');

disp(['saving ' filename]);
saveas(gcf,[filename '.fig' ]);
print('-depsc',[filename '.eps']);
print('-djpeg',[filename '.jpg']);

end
```

There is a little more functionality here. If I call this function and I put a star at the end of the file name, then the function will open the figure from the disc instead of saving to the disc my active figure. Also I check wether there is already a figure with that filename, and ask wether the user wants to overwrite it.

1.7 Set and get

The objects in Matlab have *properties* which sets how they behave and how they look like. This is the case for instance for all graphical objects: points, lines, surfaces and so on. All these objects are identified by their *handle*. For instance most graphical commands which create new objects accept an output argument which is this handle:

```
x=linspace(0,1); h=plot(x,x.^2,)
```

`h` is now the handle of the line on the graph. You can get to know all the properties of this line object through the `get` function:

```
>> get(h)
                   Color: [0 0 1]
               EraseMode: 'normal'
               LineStyle: '-'
               LineWidth: 0.5000
                  Marker: 'none'
              MarkerSize: 6
         MarkerEdgeColor: 'auto'
         MarkerFaceColor: 'none'
                   XData: [1x100 double]
                   YData: [1x100 double]
                   ZData: [1x0 double]
            BeingDeleted: 'off'
           ButtonDownFcn: []
                Children: [0x1 double]
                Clipping: 'on'
               CreateFcn: []
               DeleteFcn: []
              BusyAction: 'queue'
        HandleVisibility: 'on'
                 HitTest: 'on'
           Interruptible: 'on'
                Selected: 'off'
      SelectionHighlight: 'on'
                     Tag: ''
                    Type: 'line'
           UIContextMenu: []
                UserData: []
                 Visible: 'on'
                  Parent: 158.0016
             DisplayName: ''
               XDataMode: 'manual'
             XDataSource: ''
             YDataSource: ''
             ZDataSource: ''
```

you can alter afterward any of these properties through the `set` function. here I change the linewidth to a thicker line:

```
set(h,'linewidth',3)
```

Most of the properties are intuitive. here I give an example with the property `buttondownfcn` which allows you to specify in a string of characters a command which shall be executed every time you click with the mouse on this graphical object. For instance, I draw in a loop a large number of points with random position, and when I click on the point, I want its coordinates to be set as the title of the graphical figure:

```
n=5;
x=rand(n,1);
y=rand(n,1);
for ind=1:n
    h=plot(x(ind),y(ind),'k.'); hold on
    st=['title('' x=' num2str(x(ind)) ', y=' num2str(y(ind)) ' '')
    set(h,'buttondownfcn',st);
end
```

at each iteration of the loop, I build a new string with the coordinates of the current point, which uses the `title` function of matlab. To make a string out of the x and y coordinates, I use the `num2str` function. Note that it is slightly tricky here since I build a string (the command to be executed when clicking), which itself contains a string (the title of the figure). Think about it for a few minutes. In the code, I did not put semicolons at the end of the command which builds the string, so we can see it on screen:

```
st =
title(' x=0.29228, y=0.82265 ');
st =
title(' x=0.29168, y=0.80619 ');
st =
title(' x=0.2455, y=0.25377 ');
st =
title(' x=0.70435, y=0.8005 ');
st =
title(' x=0.77461, y=0.62971 ');
```

This functionality can be extremely useful. you are only limited by your imagination. The commands to be executed can be as complex as you wish. you can also call scripts and functions.

1.8 Your report

it is a good practice to develop your presentation skills at the same time as you develop your programming skills. Especially since the figures that you generate as the result of your coding will later on be inserted in slide presentations in conferences and articles, it is natural to edit your figures at the same time as you generate them. Also, gathering your scripts and figure and comments in a single documents provides you with a device to take time to think about your results, sheltered from the technicalities which led you to obtaining them.

Also, those of you who take this course as a part of their doctoral education will need to give me the reports made during the course. Below in these lecture notes, I include a few of the practical sessions of coding which I teach in the second year program of my university with typical reports. you can get inspiration from them.

Index

,, 15
[, 15
`min`, 22

animation, 35
array multiplication, 19
array of indices, 17
Arrays, 14
asynchronous, 35

camorbit, 38
cd, 9
clear, 8
color, 33, 36
colormap, 39
column, 16
column-wise, 22
concatenation, 15
contour, 39
current directory, 9

dashed line, 33
drawnow, 35

element by element, 18
enveloppe, 33
eye, 21

for, 9
function, 11

high level, 13, 24
hold off, 34
hold on, 34

if, 10
index, 16
inf, 24
iso-lines, 39
isovalues, 39

kron, 27

line, 16
line style, 33
linewidth, 36
linspace, 21
Low level, 24
ls, 9

Matlab functions, 11
matrix, 18
matrix multiplication, 19
max, 22
meshgrid, 37

NaN, 28
num2str, 11

ones, 21

prod, 22
prompt, 6
property-value, 36
pwd, 9

quiver, 40

randn, 40

random, 40

script, 8
shading interp, 39
sub-graphics, 34
subarrays, 16
subplot, 34
sum, 22
surf, 38

title, 37

vector fields, 40
vectorization, 22

who, 7
workspace, 7

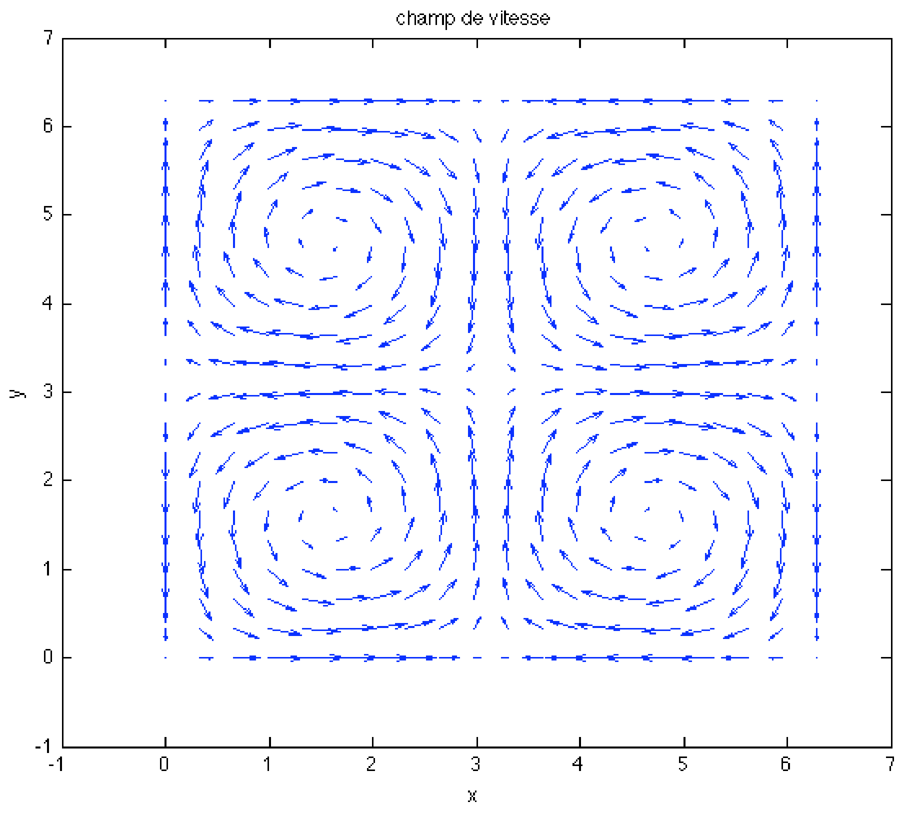
xlabel, 8
xlim, 8

ylabel, 8
ylim, 8

zeros, 21

Matlab: a hands-on course
Vortex dynamics

0) Manipulations

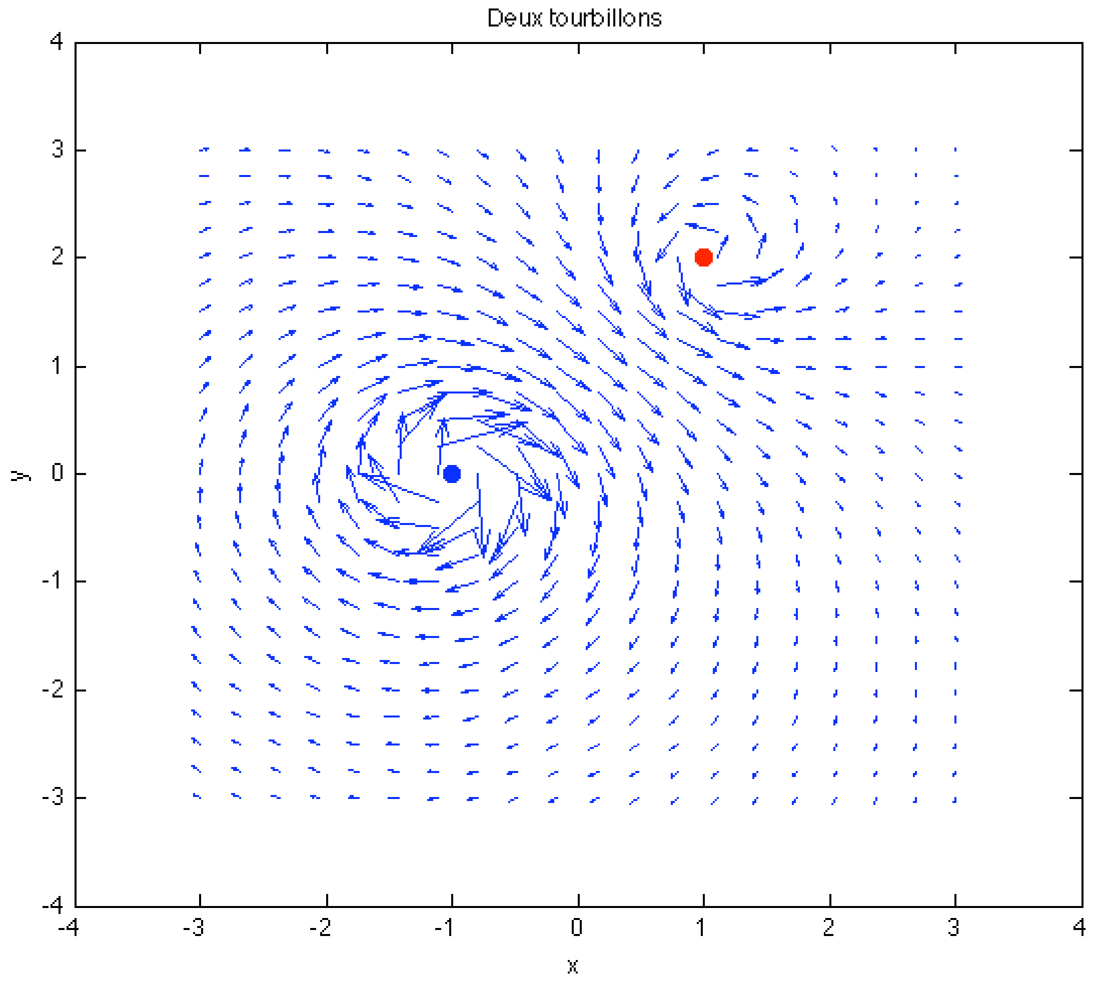


Script

```
% manipulations:
x=linspace(0,2*pi,20);
y=linspace(0,2*pi,20);
[X,Y]=meshgrid(x,y);

n=100;
tvec=linspace(0,4*pi,n);
for ind=1:n
    t=tvec(ind);
    u=sin(X).*cos(Y-t);
    v=-cos(X).*sin(Y-t);

    quiver(X,Y,u,v,'b');
    xlabel('x');
ylabel('y');
    title('champ de
vitesse')
    drawnow
end
```

We draw the vector field u=sin(x)cos(y), v=-cos(x)sin(y) with quiver, and we do as well the animation of the time evolution of this field (see script).

1) Velocity field induced by two vortices



Script

```
% 2D grid
x=linspace(-3,3,20);
y=linspace(-3,3,25);
[X,Y]=meshgrid(x,y);

% vortex positions
x1=1; y1=2; g1=1;
x2=-1; y2=0; g2=-2;

% the velocity field for each vortex
u1=-g1*(Y-y1)./(2*pi*((X-x1).^2+(Y-y1).^2+0.05));
v1=g1*(X-x1)./(2*pi*((X-x1).^2+(Y-y1).^2+0.05));

u2=-g2*(Y-y2)./(2*pi*((X-x2).^2+(Y-y2).^2+0.05));
v2=g2*(X-x2)./(2*pi*((X-x2).^2+(Y-y2).^2+0.05));

% sum of the velocity fields
u=u1+u2;
v=v1+v2;

% graph
quiver(X,Y,u,v,2); hold on
plot(x1,y1,'r.',x2,y2,'b.','markersize',20)
hold off

xlabel('x'); ylabel('y');
title('Deux tourbillons');
```

The velocity field induced by two vortices is the sum of the field induced by each of them. The sign of the intensity of the vortex (gamma) inicates the direction of the roation.

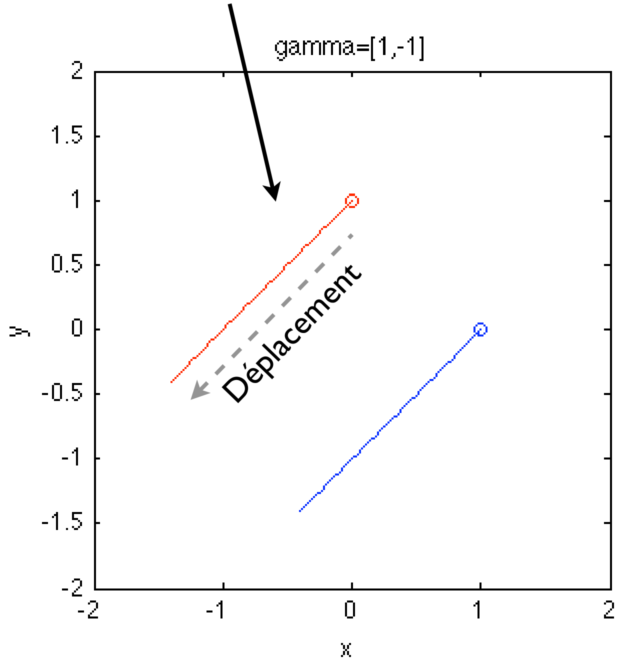
2) Motion of the vortices

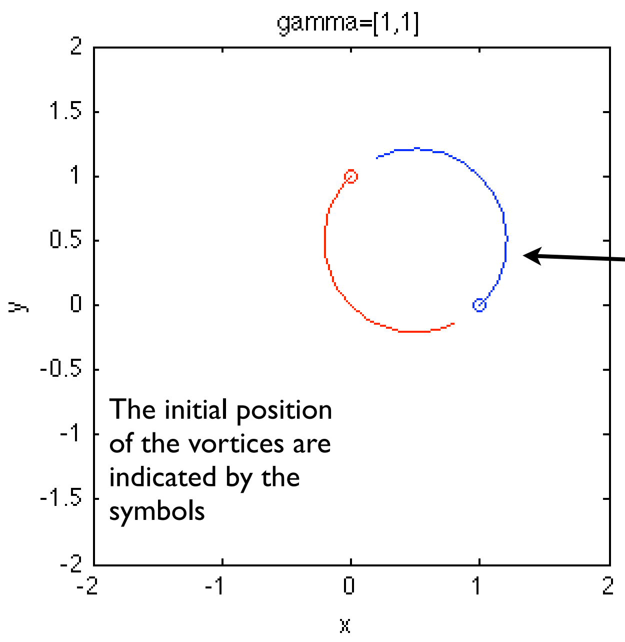
Script

```
% initial position
xpos=[1,0];
ypos=[0,1];
tmax=18;

% contrarotating vortices
subplot(1,3,1)
gamma=[1,-1];
[xt,yt,t]=tourbitraj(xpos,ypos,gamma,tmax);

plot(xt(:,1),yt(:,1),'b',xt(:,2),yt(:,2),'r'); hold on;
plot(xpos(1),ypos(1),'bo',xpos(2),ypos(2),'ro'); hold off
xlabel('x'); ylabel('y');
title('gamma=[1,-1]');
axis equal;   axis([-2,2,-2,2])

% corotating vortices
subplot(1,3,2)
gamma=[1,1];
[xt,yt,t]=tourbitraj(xpos,ypos,gamma,tmax);

plot(xt(:,1),yt(:,1),'b',xt(:,2),yt(:,2),'r'); hold on;
plot(xpos(1),ypos(1),'bo',xpos(2),ypos(2),'ro'); hold off
xlabel('x'); ylabel('y');
title('gamma=[1,1]');
axis equal;   axis([-2,2,-2,2])
```

Each vortex induces a velocity field which advects the other vortices

For two contrarotating vortices (turning in the same direction) the trajectory of the vortices are two parrallel lines





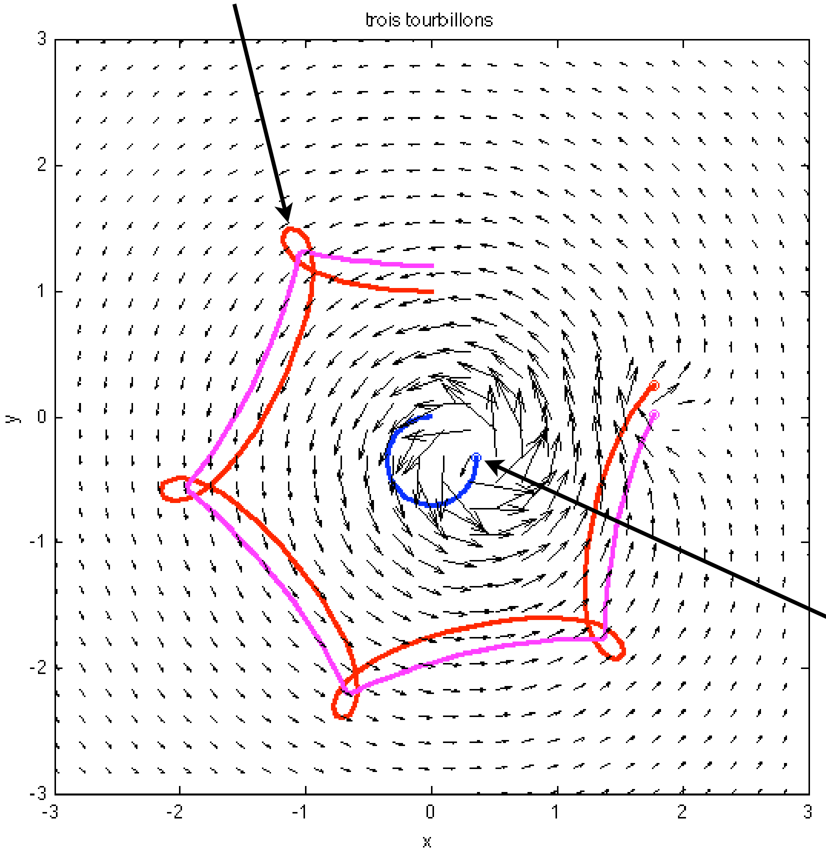
The initial position of the vortices are indicated by the symbols

For two corotating vortices: same intensity and same sign of the intensity, the trajectories are concentric circles.

3) Vortex motion

Animation of the trajectories and also of the velocity fields. here we have made the animation for three vortices instead of two.

Complex trajectory of the two small «orbiting» vortices. The trajectory of three interacting vortices is pseudo-periodic: it repeats itslef in time but does not come back to the initial position.



The central vortex is the most intense (blus trajectory). We see clearly its induces velocity field, which moves arrounds the two smaller vortices.

Script

```
% initial position
xpos=[0,0,0];
ypos=[0,1,1.2];
gamma=[0.5,0.1,-0.2];
tmax=150;

% trajectories
[xt,yt,t]=tourbitraj(xpos,ypos,gamma,tmax);

% mesh for the velocity field
x=linspace(-3,3,30);
y=linspace(-3,3,30);
[X,Y]=meshgrid(x,y);

% animation
for ind=1:length(t)
    % trajectories
    sel=1:ind;
    plot(xt(sel,1),yt(sel,1),'b',xt(sel,2),yt(sel,2),'r',xt(sel,
3),yt(sel,3),'m','linewidth',2)
    hold on;
    plot(xt(ind,1),yt(ind,1),'bo',xt(ind,2),yt(ind,2),'ro',xt(ind,
3),yt(ind,3),'mo')

    % velocity field
    u=0*X; v=0*X;
    for gre=1:3
        x0=xt(ind,gre); y0=yt(ind,gre);
        uu=-gamma(gre)*(Y-y0)./(2*pi*((X-x0).^2+(Y-y0).^2+0.05));
        vv=gamma(gre)*(X-x0)./(2*pi*((X-x0).^2+(Y-y0).^2+0.05));
        u=u+uu; v=v+vv;
    end
    quiver(X,Y,u,v,2,'k');

    xlabel('x'); ylabel('y');
    title('trois tourbillons');
    axis equal; axis([-3,3,-3,3])
    hold off; drawnow;
end
```

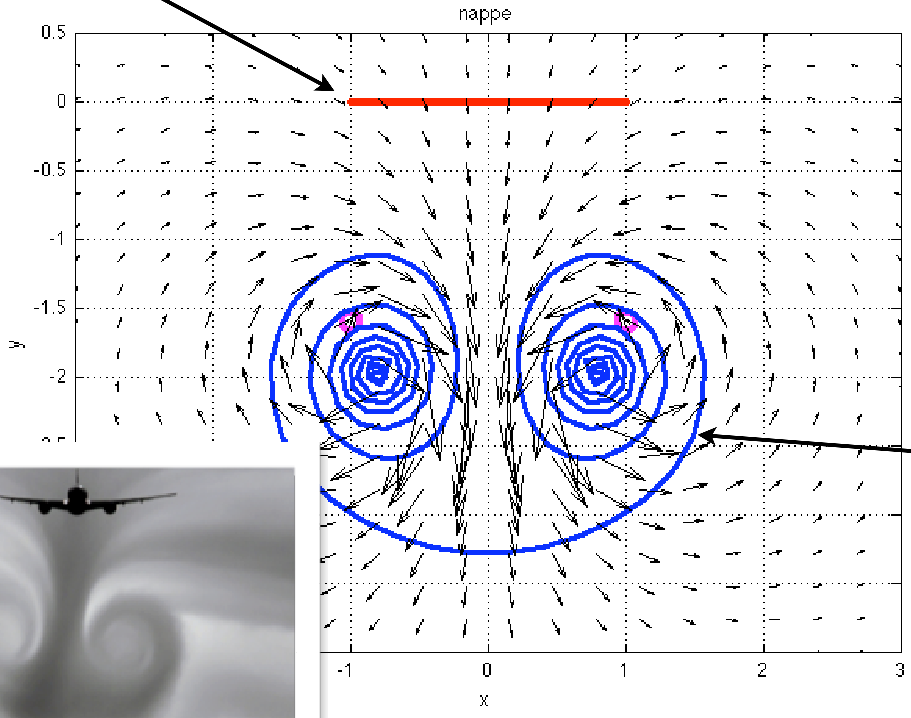
4) Vortex sheet

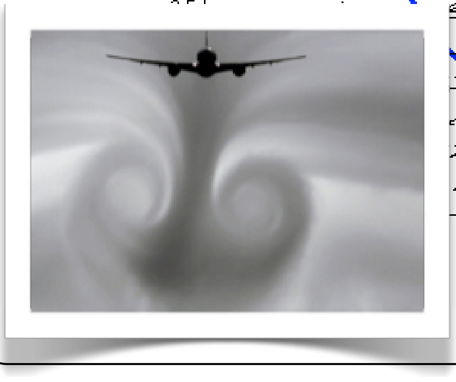
A model for the behavior of the flow around an airplane wing: a sheet of vorticity, modeled here by a great number of point vortices initially aligned along a straight line.

Script

```
n=200; % number of vortices
x=linspace(-1,1,n); % positions in x
y=zeros(1,n); % positions in y
gamma=8*x.^3/n; % the vortical intensities

% marching in time
[xt,yt,t]=tourbitraj(x,y,gamma,20);

% mes for the velocity field
x=linspace(-3,3,20);
y=linspace(-4,0.5,20);
[X,Y]=meshgrid(x,y);

for ind=1:length(t);
    % initial position in red
    plot(xt(1,:),yt(1,:),'r.-','markersize',10); hold on

    % present position in blue
    plot(xt(ind,:),yt(ind,:),'b.-','markersize',10);

    % vorticity field
    u=0*X; v=0*X;
    for gre=1:n
        x0=xt(ind,gre); y0=yt(ind,gre);
        uu=-gamma(gre)*(Y-y0)./(2*pi*((X-x0).^2+(Y-y0).^2+0.05));
        vv=gamma(gre)*(X-x0)./(2*pi*((X-x0).^2+(Y-y0).^2+0.05));
        u=u+uu; v=v+vv;
    end
    quiver(X,Y,u,v,2,'k'); hold on

    hold off; axis equal;    axis([-3,3,-4,0.5]);
    xlabel('x'); ylabel('y'); title('nappe');grid on
    drawnow
end
```

Initial configuration of the vorticity sheet





The sheet rolled up into two large wing-tip vortices. We can clearly see the velocity field resulting from these two large contrarotating vortices, and we also see that they are advecting eachother downwards. In airports the landing and take off frequency is reduced to avoid this large vortical wake.

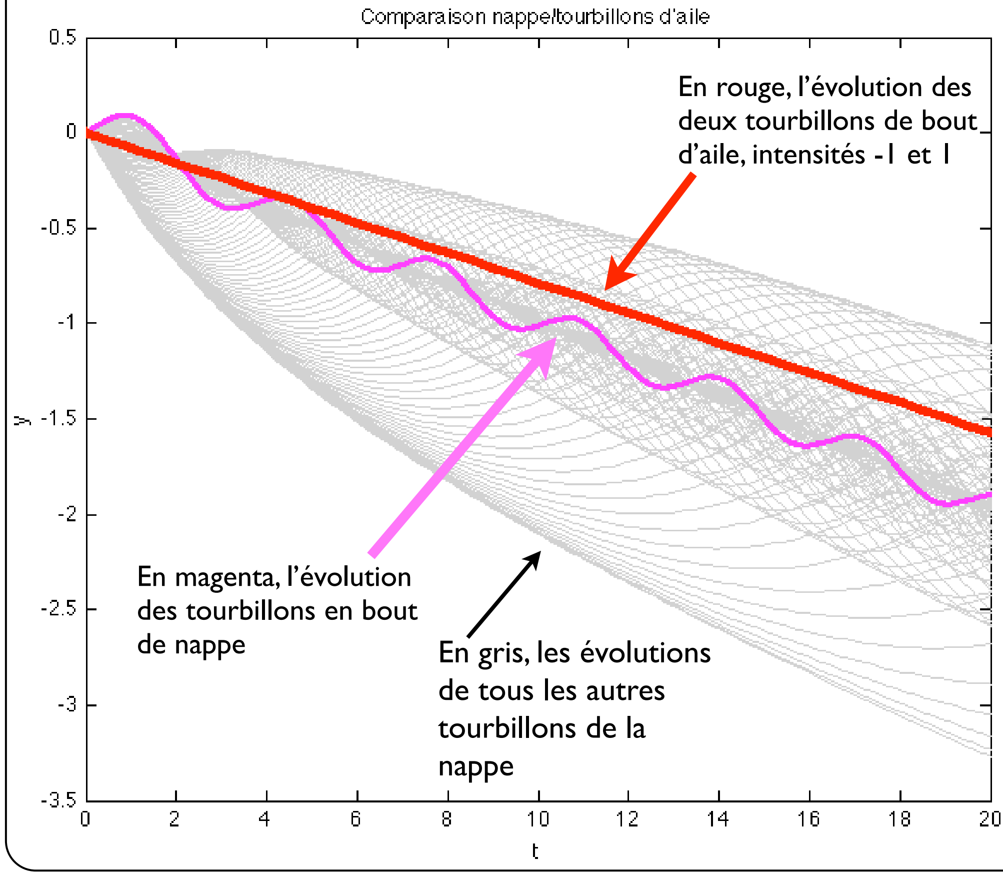
5) Two wing-tip vortices

We compare the rollup of the sheet with the evolution of two contrarotating vortices initially located at the tips of the wing. With intensity -1 on the left ad +1 on the right.

On the graph, we draw the evolution of the y position of all the vortices: how they move down in time.

Script

```
% comparison with two vortices

n=200; % number of vortices
x=linspace(-1,1,n); % x positions
y=zeros(1,n); % y positions
gamma=8*x.^3/n; % circulations
tmax=20; % final time

% march in time
[xt,yt,t]=tourbitraj(x,y,gamma,tmax);

% the two wing-tip vortices
[xtt,ytt,tt]=tourbitraj([-1,1],[0,0],[-1,1],tmax);

% y position of all vortices of the sheet
plot(t,yt,'k'); hold on

% just for the first and last vortces of the sheet
plot(t,yt(:,[1,end]),'m','linewidth',2);

% the two large vortices
plot(tt,ytt,'r','linewidth',4);

xlabel('t'); ylabel('y');
title('Comparaison nappe/tourbillons d''aile')
```



The total intensity of the sheet is equal to the sum of the intensity of each of its vortex. The integral of the intensity on each side of the wing is equal to -1 for the left and +1 for the right (intégrale of $\Gamma=4x^3$ betwen -1 et 0 then between 0 et 1).

Thus when all the vortices on each side have rolled up, we have an equivalent with two large vortices.

6) The tourbitraj function

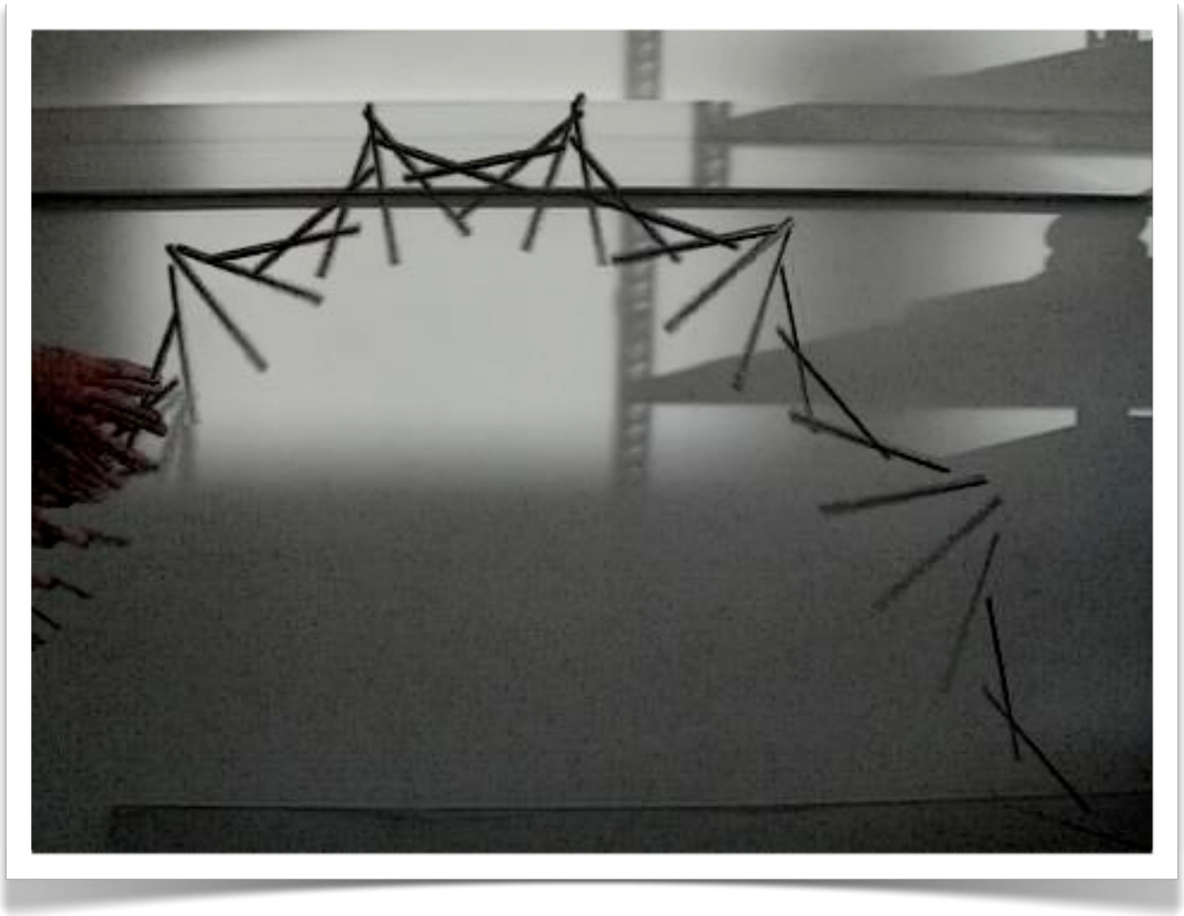
```matlab
function [xtraj,ytraj,tvec]=tourbitraj(x,y,g,tmax);
% calcule la trajectoire de pleins de tourbillons en interaction
global gamma; gamma=g(:);

% utilise ode45 pour la marche en temps
% avec la fonction tourbiv
[tvec,sol]=ode45(@tourbiv,[0,tmax],[x(:);y(:)]);

% extraction du resultat sous le bon format
n=length(x);
xtraj=sol(:,1:n);
ytraj=sol(:,n+1:2*n);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function speed=tourbiv(t,pos)
% cette fonction donne la vitesse induite aux points x,y
% par des tourbillons de circulation gamma en ces positions

global gamma
delta=0.05; % paramètre de régularisation

n=length(pos)/2;
x=pos(1:n); y=pos(n+1:2*n);

% allocation des tableaux pour les vitesses
u=zeros(n,1);
v=zeros(n,1);

% boucle sur les tourbillons
for ind=1:n
    sel=1:n; sel(ind)=[];
    % le denominateur (régularisé)
    d=2*pi*((x(ind)-x(sel)).^2+(y(ind)-y(sel)).^2+delta);

    % vitesses selon x et y
    u(ind)=sum(-gamma(sel).*(y(ind)-y(sel))./d);
    v(ind)=sum(gamma(sel).*(x(ind)-x(sel))./d);
end
speed=[u;v];
```

Function

This function computes the trajectories of a family of vortices with given initial position and intensity. it uses the Matlab function ode45 to perform the time marching.

Here the velocity field induced by each vortex is described in this subfunction.

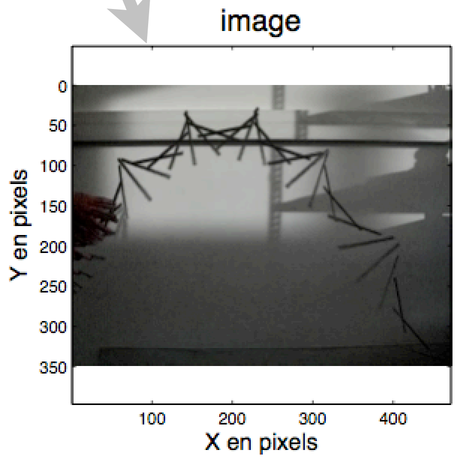
Matlab: a hands-on course
The throw of the stick



The image shows the successive positions of a stick (you can see on the left the hand which trew it). Draw several graphs which study the movement of this object. The length of the stick is 20 centimeters, and the time interval between the images is 0.023 seconds

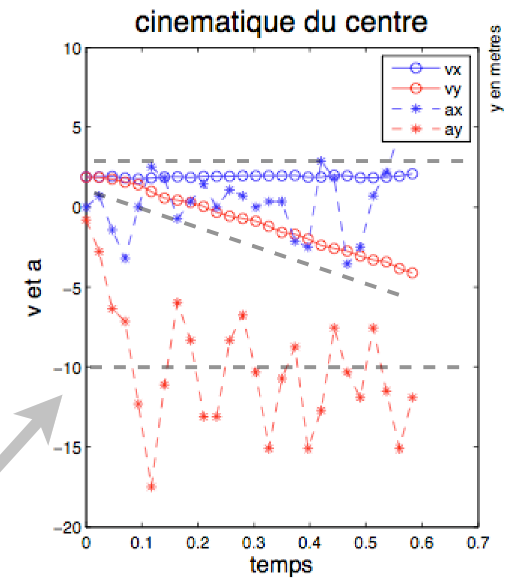
You can ask yourself: what is the trajectory of this stick? How does the rotation speed vary? What are the physical laws at play? What are the more abstract quantities I can measure from this image (energy...)

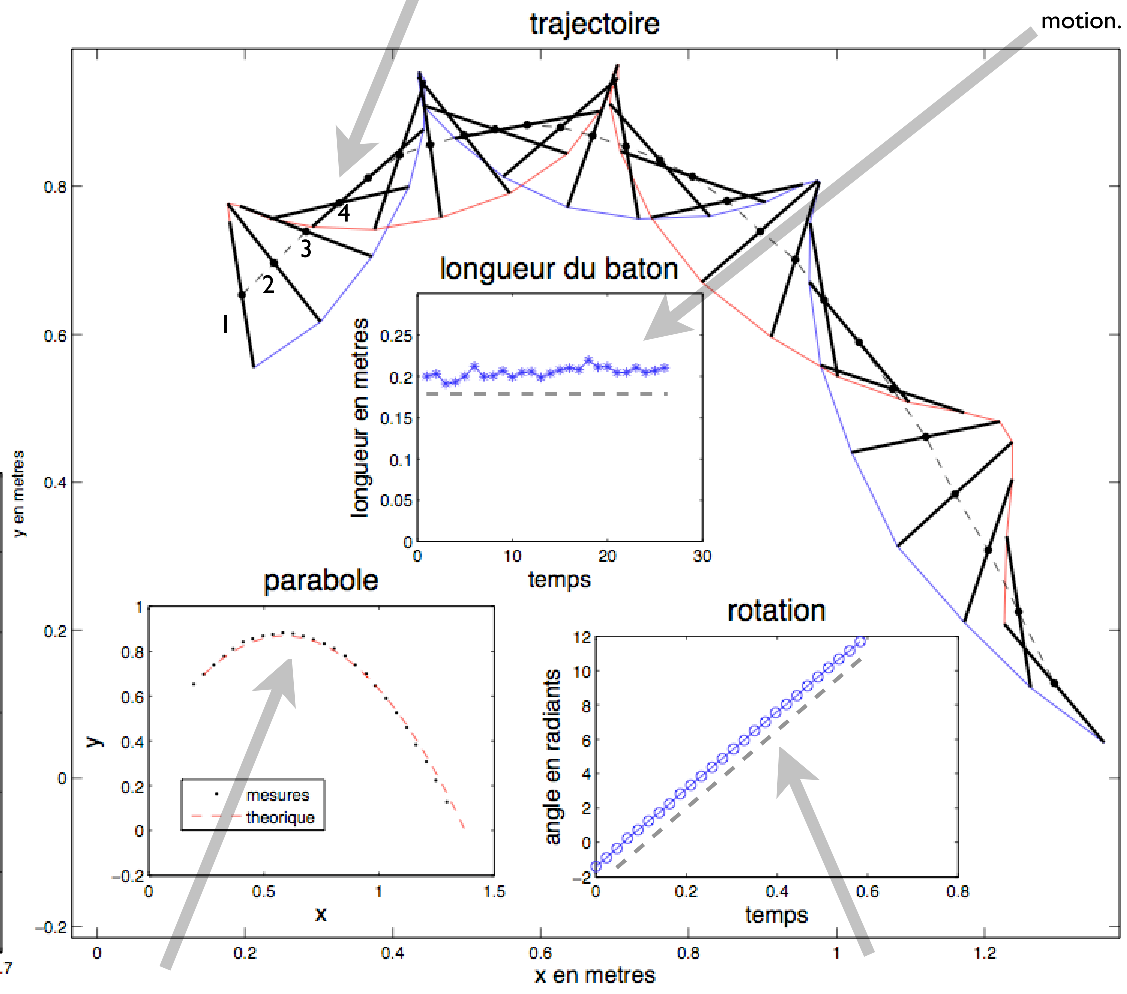
This is the original image. The poins are in the reference frame of the pixels: the origo is in the top left corner.



We have measured using the ginput function the positions (x1,y1) and (x2,y2) of the two ends of the stick for each time. From this we get the coordinates of the center of mass of the stick. The rotating motion happens arround this center of mass. We have visualized the stick at each time, and drawn the trajectory of the center with a dash line, and in red and blue the trajectory of each extremity of the stick.

We have used the length of the stick as a reference for the pixel size. We check here that the size of the stick does not change during the motion.





We calculate the speed and the accelleration of the center of mass. The speed is constant along the horizontal direction, and the speed along the vertical direction decreases linearly due to gravity. The signal has much more noise for the accelleration, but we still can see that the vertical accelleration is about -10.

Along with Newton's law, the trajectory of the center of mass should be a parabola. We compare here the measured trajectory with a theoretical trajectory which has for initial position and speed that of the measured stick.

For a solid, the rotation speed is constant if there is no external moment applied. This is what we observe here. We can see that the angle of the stick with the horizontal inscreases linearly with time.

1

```matlab
% testing the ideas for the flyinf
stick
clear all; clf

a=imread('baton.jpg');
subplot(2,3,1);
image(a);
title('image'); xlabel('X en
pixels'); ylabel('Y en pixels');


d=[
  69.5698  164.9133
  58.8653  100.1702
  99.1347  144.4681
  57.8458   92.2193
 122.0731  115.5041
  63.4529   93.3552
 138.3847   84.8363
  76.7062   99.0344
 145.0114   59.2799
  95.5666  102.4419
 145.5211   37.6989
 123.0925  103.5777
 142.9724   33.7234
 152.6575   98.4664
 142.4627   35.9951
 183.2419   87.6759
 144.5016   48.4894
 208.7289   70.0704
 158.7744   63.2553
 223.0016   51.3290
 180.1834   80.2930
 231.1575   36.5630
 208.7289   93.9231
 231.1575   30.3159
 240.3328   99.0344
 229.1185   34.8592
 271.9367   97.8985
 227.5893   47.9214
 296.4042   91.6514
 232.1769   68.9345
 313.2256   83.7005
 245.9399   98.4664
 319.8523   81.9967
 268.3685  126.8625
 320.3620   82.5646
 298.9529  151.2831
 316.2841  100.7381
 328.5179  168.8887
 315.7744  126.8625
 360.1218  180.2471
 320.8718  163.7774
 384.5893  184.7905
 334.6347  202.3961
 400.3912  188.7660
 355.0244  244.4223
 405.9984  197.8527
 384.5893  277.9296
 405.9984  214.3224
 414.1542  306.8936
 403.4497  239.8789
 446.7776  331.3142
 402.4302  278.4975];

x1=d(1:2:end,1); x2=d(2:2:end,1);
y1=d(1:2:end,2); y2=d(2:2:end,2);
```

2

```matlab
% set the physical origo
y1=-(y1-347); y2=-(y2-347);

% insert the pysical scale in meters
nl=sqrt((x1(1)-x2(1))^2+(y1(1)-y2(1))^2);
pix=0.2/nl;
x1=x1*pix; x2=x2*pix;
y1=y1*pix; y2=y2*pix;

% the center of the stick
x=(x1+x2)/2;
y=(y1+y2)/2;

% the time array
n=length(x1);
dt=1/(300/7);
tvec=(0:1:n-1)*dt;

% We draw all the positions of the stick
subplot(2,3,2);

plot(x,y,'k.-','markersize',15); hold on
plot(x1,y1,'b',x2,y2,'r');
for ind=1:n
    plot([x1(ind) x2(ind)],[y1(ind) y2(ind)],'k','linewidth',
2);
end
title('trajectoire'); xlabel('x en metres'); ylabel('y en
metres');


% measure and draw the length of the stick for each time
lon=zeros(n,1);
for ind=1:n
    lon(ind)=sqrt((x1(ind)-x2(ind))^2+(y1(ind)-y2(ind))^2);
end
subplot(2,3,3); plot(lon,'b*-'); ylim([0,0.3])
title('longueur du baton'); xlabel('temps'); ylabel('longueur
en metres');


% speed and accelleration of the center of mass
vx=zeros(n,1);
vy=zeros(n,1);
vx(1)=(x(2)-x(1))/dt;
vy(1)=(y(2)-y(1))/dt;
for ind=2:n-1;
    vx(ind)=(x(ind+1)-x(ind-1))/(2*dt);
    vy(ind)=(y(ind+1)-y(ind-1))/(2*dt);
end
vx(n)=(x(n)-x(n-1))/dt;
vy(n)=(y(n)-y(n-1))/dt;
```

3

```matlab
ax=zeros(n,1);
ay=zeros(n,1);
ax(1)=(vx(2)-vx(1))/dt;
ay(1)=(vy(2)-vy(1))/dt;
for ind=2:n-1;
    ax(ind)=(vx(ind+1)-vx(ind-1))/(2*dt);
    ay(ind)=(vy(ind+1)-vy(ind-1))/(2*dt);
end
ax(n)=(vx(n)-vx(n-1))/dt;
ay(n)=(vy(n)-vy(n-1))/dt;

subplot(2,3,4);
plot(tvec,vx,'b',tvec,vy,'r',tvec,ax,'b--',tvec,ay,'r--')
title('cinematique du centre'); xlabel('temps');
ylabel('v et a');
legend('vx','vy','ax','ay')

% measure the angle with the horizontal
an=zeros(n,1);
g=-10;
add=0;
for ind=1:n
    c=g;
    g=atan((y2(ind)-y1(ind))/(x2(ind)-x1(ind)));
    if g<c; add=add+pi; end
    an(ind)=g+add;
end

subplot(2,3,5);
plot(tvec,an,'bo-');
title('rotation'); xlabel('temps'); ylabel('angle en
radiants');



% compare the center of mass trajectory with a parabola.
loc=2; g=9.8;
tmax=(-vy(loc)-sqrt(vy(loc)^2+4*y(loc)*g/2))/(-g);
t=linspace(0,tmax,100);
xx=x(loc)+vx(loc)*t;
yy=y(loc)+vy(loc)*t-g*t.^2/2;
subplot(2,3,6);
plot(x,y,'k.',xx,yy,'r--')
title('parabole'); xlabel('x'); ylabel('y');
legend('mesures','theorique')
```