Contents

Introduction to Python : Numpy and Matplotlib

This chapter, and especially section 2 about `Graphic with matplotlib` *has been proofread and upgraded by Naomi*

1 Linear algebra with numpy

Throughout this section we will be using the script file `f31numpy.py`

```
# First note that constants such as "pi" are not defined in basic Python
# and must be imported from numpy
import numpy as np
np.pi    # 3.141592653589793
np.e     # 2.718281828459045
```

1.1 Vectors and matrices

```
# f31numpy.py

"""________________________________________________________________________
LINEAR ALGEBRA with numpy
VECTORS and MATRICES : ARRAYS
________________________________________________________________________"""
import numpy as np
"""__________________________________________________
1) Vectors  = one-dimensional numpy.arrays
   Matrices = two-dimensional numpy.arrays
__________________________________________________"""
# a vector (one-dimensional)
v = np.array([-4,17,0,11])
print(v)

# a matrix (two-dimensional array)
m = np.array([[1,  2],[3,4],[5,6]])
print(m)

"""__________________________________________________
2) Some useful functions : type, np.size, np.shape
__________________________________________________"""
type(v)
```

```
# compare with:
type([-4,17,0,11])

type(m)

sizev = np.size(v)  # 4
sizem = np.size(m)  # 6

shapev = np.shape(v)  # (4,)
shapem = np.shape(m)  # (3, 2)

type(shapev) # tuple
type(shapem) # tuple

"""________________________________________
3) Some particular matrices:
        np.zeros, np.ones, np.eye, np.diag
        np.arange, np.linspace
____________________________________________"""
z1 = np.zeros(5)
z2 = np.zeros((2,3))

o1 = np.ones(6)
o2 = np.ones((3,4))

e1 = np.eye(6)

m = np.array([[1, 2, 3],[4,5,6],[7,8,9]]) # a square matrix
d = np.diag(m)        # the main diagonal : array([1, 5, 9])
np.diag(m, 1)         # the first upper diagonal
np.diag(m, -1)        # the first lower diagonal
A = np.diag(d)        # a matrix with main diagonal = d
B = np.diag(d,1)      # a matrix with first upper diagonal = d
C = np.diag(d,-2)     # a matrix with second lower diagonal = d

a = np.arange(5)
# array([0, 1, 2, 3, 4])
b = np.arange(2,8)
# array([2, 3, 4, 5, 6, 7])
c = np.arange(5,2,-0.5)
# array([ 5. , 4.5, 4. , 3.5, 3. , 2.5])
t = np.linspace(-2,3,10)
# 10 values uniformly distributed between -2 and 3

"""________________________________________
4) Shape and reshape
        it is possible to change the shape of an array
____________________________________________"""
import numpy as np
U = np.arange(1,25)
type(U)
np.shape(U)
```

```

# shape
U.shape = (3,8)      # modify the shape of U
type(U)
np.shape(U)

# reshape
V = U.reshape(6,4)  # modify the shape of U

"""_____________________________________________
5) Access and slicing:
        access to elements of an array
        recovering part of an array
_____________________________________________"""
v = np.array([-4, 17, 0, 11, -4, 8])
u = v[3]    # element at position 3 (=> fourth element)
w = v[1:4]  # elements from position 1 to position 3 (= 4-1)

M = np.array([[4, 1, 2, -4],[8, 3, 4, -5],[5, 6, -4, 2]])
e = M[1,2]
N = M[1:3,1:3]

# warning :
U = M[:,2]  #column 2 (writen as a one-dimensional array ==> horizontally)
V = M[0,:]  #line 0
```

1.2 Matrices concatenation

```
"""_____________________________________________
6) Concatenation of matrices: two methods
        function concatenate()
        functions hstack() & vstack() : simpler and more intuitive
_____________________________________________"""
import numpy as np
# definition of matrices (arrays) for the tests
A = np.array([[4, 1, 2, -4],[8, 3, 4, -5]])
B = np.array([[-1, -1],[-1,-1]])
C1 = np.array([4,4,4,4])     #--> be careful, C1 is a vector !!
C2 = np.array([[4,4,4,4]])  # two dimensional array
D = np.array([[5,5,5,5],[6,6,6,6],[7,7,7,7]])

# we check the shape (important to concatenate)
np.shape(A)   #(2, 4)
np.shape(B)   #(2, 2)
np.shape(C1)  #(4,)       #--> C1 is a vector
np.shape(C2)  #(1, 4)
np.shape(D)   #(3, 4)

"""
function concatenate
"""
```

```
# concatenation along axis=0 ("vertically")
#                   or axis=1 ("horizontally")
# the input array dimensions must agree along the axis
E1 = np.concatenate((A,C1),axis=0)  # wrong : not the same shape
E2 = np.concatenate((A,C2),axis=0)  # OK
F = np.concatenate((A,C2,D),axis=0) # OK
G = np.concatenate((A,C2,D),axis=1) # wrong
H = np.concatenate((A,B,A),axis=1)  # OK

"""
function append
"""
# concatenation of vectors (one dimensional array)
# C1 is a vector
C3 = np.append(C1,2*C1)      # array([4, 4, 4, 4, 8, 8, 8, 8])
C4 = np.append(-23,3*C1)     # array([-23,  12,  12,  12,  12])

"""
functions vstack and hstack
"""
FF = np.vstack((A,C2,D)) # stack vertically
HH = np.hstack((A,B,A))  # stack horizontally

EE = np.vstack((A,C1))   # stack vertically array and vector
CC = np.hstack((C1,C1))  # stack horizontally vectors
```

1.3 Operations on matrices

```
"""_______________________________________________
7) Basic operations:
        sum,
        product by a scalar,
        term by term product
_______________________________________________"""
import numpy as np
# operation on vectors
v1 = np.array([-4,17,0,11])
v2 = np.array([7,-14,3,-8])
w1 = 5 * v1      # product by a scalar
w2 = v1 + v2     # sum
w3 = v1 * v2     # term by term product

# operation on matrices
M1 = np.array([[1, 2],[3,4],[5,6]])
M2 = np.array([[5, 0],[1,2],[-1,3]])
U1 = -2 * M1     # product by a scalar
U2 = M1 + M2     # sum
U3 = M1 * M2     # term by term product

# more on term by term operations :
v0 = np.array([1,2,3,4.])
```

```
v02 = v0**2             # square of each term
v03 = v0**3             # cube of each term
w0 = np.array([[1, 2],[3,4],[5,6.]])
w02 = w0**2             # square of each term
        # Out[]:
        # array([[  1.,   4.],
        #        [  9.,  16.],
        #        [ 25.,  36.]])


"""________________________________________
8) Matricial product:
   with numpy.dot()
________________________________________"""
A = np.array([[4, 1], [2, -3],[-1, 3], [4, -2]])
B = np.array([[2, 1],[3, -1]])

C = np.dot(A,B)
D = np.dot(B,A) # produces an error, dimensions doesn't agree

"""________________________________________
9) Transposition of a matrix:
________________________________________"""
A = np.array([[4, 1], [2, -3],[-1, 3], [4, -2]])
        # [[ 4  1]
        #  [ 2 -3]
        #  [-1  3]
        #  [ 4 -2]]
AT = A.T
print(AT)
        # [[ 4  2 -1  4]
        #  [ 1 -3  3 -2]]
```

1.4 Action of a function on a vector or a matrix

This item is a consequence (or similar to) of the term by term product (point 7).
Consider a function f that evaluate the value of an input parameter x and return $y = f(x)$.
If the input parameter x is an `array`, the output parameter y is an array of the same shape as x.

Precisely, if
`x = [x[0], x[1],..., x[n]]`
we will get
`y = f(x) = [f(x[0]), f(x[1]),..., f(x[n])]`
and similarly for a matrix.

```
"""________________________________________
10) Action of a function on a vector or a matrix:
________________________________________"""
# this item is a consequence of the term by term product (point 7)

import numpy as np
x0 = np.array([-1,2,-3,4.,-5])
```

```
y0 = abs(x0)              # array([1., 2., 3., 4., 5.])

x1 = np.linspace(0, np.pi/2, 4)
        # array([0.        , 0.52359878, 1.04719755, 1.57079633])
y1 = np.sin(x1)
        # array([0.        , 0.5       , 0.8660254, 1.        ])

def f0(x):
        return 1 + x + x**2
t = np.array([1,2,3,4])
y = f0(t)   # array([ 3,  7, 13, 21])

# Note that numpy functions usually work with a list (of numerical values)
u = [2., 4., -3.]
abs(u)      # bad operand type for abs(): 'list'
np.cos(u)   # OK
f0(u)       # unsupported operand type(s) for +: 'int' and 'list'
```

1.5 Complex matrices

```
"""________________________________________
11) Complex matrix and conjugate:
________________________________________"""
import numpy as np
B1 = np.array([[2-1j, 1+2j], [3-2j, -1]])
B2 = np.conj(B1)
B = B1 + B2
C = np.conj(B1).T
```

1.6 Copy and Hard copy

```
"""________________________________________
12) Notes about copy and Hard copy:
________________________________________"""
import numpy as np
A = np.array([1,2,3,4])
A   # array([1, 2, 3, 4])

# Copy (in fact, this is just a link copy) :
B = A   # create a new name B to the existing object already named A
B       # array([1, 2, 3, 4])
A[2] = -7
A   # array([ 1,  2, -7,  4])
B   # array([ 1,  2, -7,  4])  ==> B remains equal to A
B[1] = 56
A   # array([ 1, 56, -7,  4])  ==> A and B remains equal
B   # array([ 1, 56, -7,  4])

# Hard copy (creates a new array and so uses more memory)
B = np.copy(A)
```

```
A[2] = -9
A   # array([ 1, 56, -9,  4])
B   # array([ 1, 56, -7,  4])
B[0] = 3
A   # array([ 1, 56, -9,  4]
B   # array([ 3, 56, -7,  4])
```

1.7 Array display

```
"""_______________________________________________
13) Array display
_______________________________________________"""
# help(np.set_printoptions)
#
# precision : int or None, optional
#     Number of digits of precision for floating point output (default 8)
# threshold : int, optional
#     Total number of array elts which trigger summarization (default 1000)
#     To always use the full repr without summarization :
#         import sys
#         np.set_printoptions(threshold=sys.maxsize)
import numpy as np

# precision :
x = np.linspace(0,1,4)
print(x)    # [0.         0.33333333 0.66666667 1.        ]
np.set_printoptions(precision=2)
print(x)    # [0.   0.33 0.67 1.  ]

# threshold :
np.set_printoptions(threshold=5)
u = np.arange(10)
print(u)    # [0 1 2 ... 7 8 9

np.set_printoptions(threshold=10)
print(u)    # [0 1 2 3 4 5 6 7 8 9]

import sys
np.set_printoptions(threshold=sys.maxsize)

u = np.arange(5000)
print(u)    # all values are displayed
```

1.8 Save an array into a text file

```
"""_______________________________________________
14) Save (and then load) an array to a text file:
_______________________________________________"""
import os   # import operating system
myPath = "C:/Users/Luke/Documents/Luc/Myanmar/NumericalMaths"
myPath = "C:/Users/Luke/Documents/Numerical Analysis with Python"
myWorkingPath = myPath + "/01 Python introduction/My Working Space"
os.chdir(myWorkingPath)

import numpy as np
x = np.array([ -1.5,  4.,  -2.5, 3, 8, 5])
y = x**2
# Creation of a data file "NEWdata.txt"
# and writting the arrays x and y in this text file :
U = (x,y)
np.savetxt('NEWdata.txt', U, fmt='%1.8e')
# Loading the arrays x and y from the file "NEWdata.txt"
(x,y) = np.loadtxt('NEWdata.txt')
```

1.9 Determinant and Inverse

```
"""_______________________________________________________________
LINEAR ALGEBRA with numpy.linalg
_______________________________________________________________"""

"""__________________________________________
15) Determinant:
with numpy.linalg.det()
__________________________________________"""
import numpy as np
A = np.array([[4, 5], [2, 3]])
dt = np.linalg.det(A)
# or:
from numpy.linalg import det
dt = det(A)

"""__________________________________________
16) Inverse:
with numpy.linalg.inv()
__________________________________________"""
A = np.array([[4, 5], [2, 3]])
A1 = np.linalg.inv(A)
# or :
from numpy.linalg import inv
A1 = inv(A)

# then check:
np.dot(A,A1)
```

1.10 Resolution of a linear system

```
"""__________________________________________
17) Resolution of a linear system:
        with numpy.linalg.solve()
__________________________________________"""
```

```
import numpy as np
A = np.array([[4, 5], [2, 3]])
b = np.array([14, 8])
x = np.linalg.solve(A,b)
# or:
from numpy.linalg import solve
x = solve(A,b)
```

1.11 Eigenvalues

```
"""_________________________________________________
18) Eigenvalues:
        with numpy.linalg.eig()
_____________________________________________________"""
import numpy as np
A = np.array([[1, 1, -2], [-1, 2, 1], [0, 1, -1]])
D, V = np.linalg.eig(A)
#or:
from numpy.linalg import eig
D, V = eig(A)

# eigenvalues of the matrix A are in D
# associate eigen vectors are the columns of V
print(D)
        # Out[1]: array([ 2.,  1., -1.])
print(V)
        # Out[2]:
        #     array([[  3.01511345e-01,  -8.01783726e-01,   7.07106781e-01],
        #            [  9.04534034e-01,  -5.34522484e-01,  -1.92296269e-16],
        #            [  3.01511345e-01,  -2.67261242e-01,   7.07106781e-01]])

# WE WANT TO CHECK ... :
# eigen vectors :
v0 = V[:,0]
v1 = V[:,1]
v2 = V[:,2]
# do we have  A * v = lambda * v ?
np.dot(A,v0) - 2*v0
np.dot(A,v1) - v1
np.dot(A,v2) + v2
```

1.12 The matrix class of numpy

```
"""_________________________________________________
19) the class matrix of numpy
        A matrix is a specialized 2-D array
        Data : array_like or string
        If the input is a string, it is interpreted as a matrix
        with commas or spaces separating columns,
        and semicolons separating rows.
```

```
——————————————————————————————————————————————————"""
import numpy as np
# Data = array_like
# np.matrix is similar as np.array
A1 = np.array([[4, 1, 2, −4],[8, 3, 4, −5]])
A2 = np.matrix([[4, 1, 2, −4],[8, 3, 4, −5]])
A3 = np.matrix([8, 3, 4, −5]) # matrix([[ 8,  3,  4, −5]]) : double brackets
type(A1) # numpy.ndarray
type(A2) # numpy.matrix
np.shape(A1) # (2, 4)
np.shape(A2) # (2, 4)

# Data = string
# commas or spaces separate columns, semicolons separate rows
A = np.matrix('1 2; 3 4')
B = np.matrix('5.0 6.0')
C = np.matrix("-1 ;-2 ;-3")
D = np.matrix('1.0 2.0; 3.0 4.0; 5.0 6.0')
E = np.matrix('-1 -2; 1 2; 3 -2')
type(A)      # numpy.matrix
np.shape(A) # (2, 2)

# sum and product of matrices
2*D − 3*E
A*B      # dimensions do not agree
B*A      # OK
D*A      # OK

# power, determinant and inverse of a square matrix
A**3
from numpy.linalg import det
dt = det(A)
from numpy.linalg import inv
U = inv(A)
# or
U = A**(−1)

# Concatenation
np.concatenate((A, B))
np.concatenate((C, D), axis=1)
# or with vstack and hstack :
np.vstack((A,B))         # stack them vertically
np.hstack((C,D,C,D))     # stack them horizontally
```

1.13 Nan, Inf

```
"""——————————————————————————————————————————————————
20) Miscellaneous:
        Non real "floating values"
——————————————————————————————————————————————————"""
a = np.arange(3,dtype=float)
```

```
a[0] =  np.nan  # Not A Number
a[1] =  np.inf
a[2] = -np.inf
print(a)        # is now [nan,inf,-inf]
np.isnan(a[0])  # True
np.isinf(a[1])  # True
np.isinf(a[2])  # True
```

1.14 Random : uniform and normal distributions

```
"""_____________________________________________
21) Random
    uniform and normal distribution of N values
_________________________________________________"""
import numpy as np

# EXAMPLE 1
N = 15
xu = np.random.uniform(2,10,N)
     # uniform distribution of N values
     # in the interval [2,10[ (10 is excluded)

xn = np.random.normal(5,2,N)
     # normal (Gaussian) distribution of N values
     # with Mean ("centre") = 5
     # and Standard deviation = 2

# EXAMPLE 2
# we use matplotlib to display the histogram of each distribution
import matplotlib.pyplot as plt

plt.figure("uniform distribution")
xu = np.random.uniform(-2,4,1000)
hu = plt.hist(xu,bins=100) # bins is the number of "rectangles"

plt.figure("normal distribution")
xn = np.random.normal(3,2,1000)
hn = plt.hist(xn,bins=100)
```

1.15 The module time

In order to compare the efficiency of some algorithms, it may be convenient to use the `module time`

```
"""__________________________________________
22) the module Time
    here, we are no more working with numpy...
______________________________________________"""
import time

start = time.time() # number of seconds from
```

```
                        # January 1st 1970, 00:00:00
print("start =",start)
for i in range(2000):
    for j in range(2000):
        x = np.sqrt(i+j)
end = time.time()
print("end =",end)
print("Duration =",end-start)
```

Exercise 0.1

Write in the simplest way the following vectors and matrices.

$$v1 = \begin{pmatrix} 5 & 6 & 7 & 8 & 9 & 10 \end{pmatrix} \quad v2 = \begin{pmatrix} 0 & 0 & 0 & 5 & 6 & 7 & 8 & 9 & 10 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$v3 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 9 & 7 & 5 & 3 & 1 \end{pmatrix}$$

$$M1 = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix} \quad M2 = \begin{pmatrix} 1 & 3 & 5 & 7 & 9 \\ 8 & 6 & 4 & 2 & 0 \\ 8 & 6 & 4 & 2 & 0 \end{pmatrix} \quad M3 = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

Exercise 0.2

Write in the simplest way the following matrix.

$$A = \begin{pmatrix} 2. & 1. & 0. & 0. & 0. & 0. & 0. & 0. \\ 1. & 4. & 1. & 0. & 0. & 0. & 0. & 0. \\ 0. & 1. & 4. & 1. & 0. & 0. & 0. & 0. \\ 0. & 0. & 1. & 4. & 1. & 0. & 0. & 0. \\ 0. & 0. & 0. & 1. & 4. & 1. & 0. & 0. \\ 0. & 0. & 0. & 0. & 1. & 4. & 1. & 0. \\ 0. & 0. & 0. & 0. & 0. & 1. & 4. & 1. \\ 0. & 0. & 0. & 0. & 0. & 0. & 1. & 2. \end{pmatrix}$$

Exercise 0.3

Write in the simplest way the following matrices.

$$\texttt{M4} = \begin{pmatrix} 1. & 1. & 2. & 2. & 3. & 3. \\ 1. & 1. & 2. & 2. & 3. & 3. \\ 2. & 2. & 3. & 3. & 4. & 4. \\ 2. & 2. & 3. & 3. & 4. & 4. \\ 3. & 3. & 4. & 4. & 5. & 5. \\ 3. & 3. & 4. & 4. & 5. & 5. \end{pmatrix} \qquad \texttt{M5} = \begin{pmatrix} 1. & 0. & 2. & 0. & 4. & 0. \\ 0. & 2. & 0. & 4. & 0. & 8. \\ 2. & 0. & 4. & 0. & 8. & 0. \\ 0. & 4. & 0. & 8. & 0. & 16. \\ 4. & 0. & 8. & 0. & 16. & 0. \\ 0. & 8. & 0. & 16. & 0. & 32. \end{pmatrix}$$

Exercise 0.4

Pascal's triangle.
Write a Python script computing each line "n" of Pascal's triangle for $0 \leq n \leq N_{\max}$ as in the example given below.

```
Enter N_max : 7
n = 0 :   [ 1.]
n = 1 :   [ 1.   1.]
n = 2 :   [ 1.   2.   1.]
n = 3 :   [ 1.   3.   3.   1.]
n = 4 :   [ 1.   4.   6.   4.   1.]
n = 5 :   [  1.    5.   10.   10.    5.    1.]
n = 6 :   [  1.    6.   15.   20.   15.    6.    1.]
```

```
n = 7 :   [   1.     7.    21.    35.    35.    21.     7.     1.]
```

Exercise 0.5 (*Delete duplicates*)

Consider the following process that you will implement in Python.

1. *We first choose N integers between 0 and 9, with $N > 10$, so that two integers (at least) are identical.*
   These integers are stored in the array `r0 = [r0[0], r0[1],...,r0[N-1]`
2. *We sort these integers in increasing order with the function* `sorted()` *of basic Python. We thus get a list* `r1` *of sorted integers in increasing order.*
   Use the instruction `help(sorted)` *in the console.*
3. *We delete duplicates.*
   Precisely, we define a function that copy all the different elements of `r1` *in a new array* `r2`*, as follows.*
   - initialization : `r2 = [r1[0]`
   - for each other element `r1[i]` *of the initial array* `r1` *:*
   - if `r1[i]` *is different from its predecessor, we add this element in the array* `r2`
   - otherwise, we do nothing and go to the next element in the array `r1`

   At the end of this process, the array `r2` *contains all the distinct elements of* `r1` *in increasing order*

You can take advantage of the following script.

```
r0 = np.random.uniform(0,10,N) # uniform distribution of N values
                               # in the interval [0,10[ (10 is excluded)

r0 = r0.astype(int)  # each floating number of the array r0
                     # is converted into an integer
                     # we thus get an array of integers
```

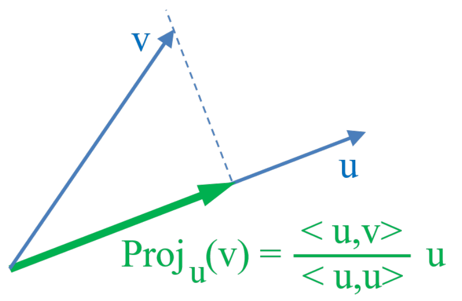
Exercise 0.6 (*Gram-Schmidt algorithm and orthogonal matrices*)

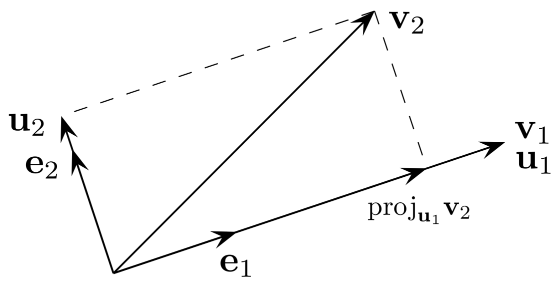
As a reminder, consider the usual Euclidean space $\mathbb{R}^2$ with the canonical orthogonal basis $e_1 = (1,0)$ and $e_2 = (0,1)$, and let $v = (x,y)$ be a vector of $\mathbb{R}^2$. We have

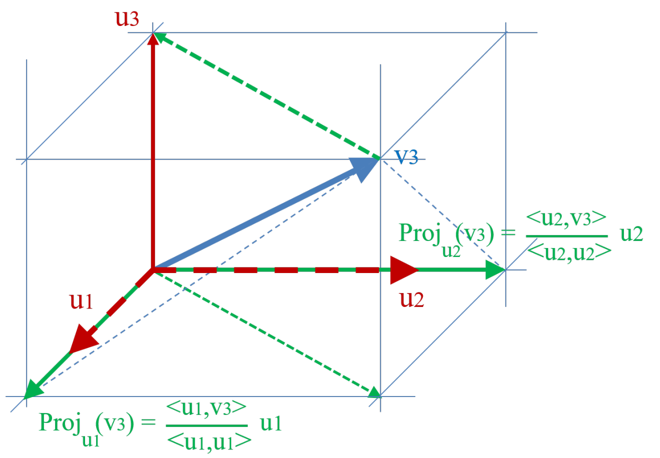
$$\texttt{Proj}_{e_1}(v) = \langle e_1, v\rangle\, e_1 = \begin{pmatrix}1\\0\end{pmatrix}\cdot\begin{pmatrix}x\\y\end{pmatrix}\; e_1 = x\, e_1 \quad \text{and} \quad \texttt{Proj}_{e_2}(v) = \langle e_2, v\rangle\, e_2 = \begin{pmatrix}0\\1\end{pmatrix}\cdot\begin{pmatrix}x\\y\end{pmatrix}\; e_2 = y\, e_2$$

More generally, the projection of a vector v on the line defined by a non zero vector u in $\mathbb{R}^n$ is defined thanks to the unit vector $\frac{u}{||u||}$ associated with u

$$\texttt{Proj}_u(v) \;=\; \langle \frac{u}{||u||}, v\rangle\, \frac{u}{||u||} \;=\; \frac{1}{||u||^2}\,\langle u, v\rangle\, u \;=\; \frac{\langle u, v\rangle}{\langle u, u\rangle}\, u$$






u3
v3
Proj_u2(v3) = <u2,v3> / <u2,u2> u2
u1
u2
Proj_u1(v3) = <u1,v3> / <u1,u1> u1

first steps of GS algorithm

Gram-Schmidt algorithm —
Given p linearly independant vectors $v_1, v_2, \ldots, v_p \in \mathbb{R}^n \quad (n \geq p)$, the Gram-Schmidt algorithm produces an orthonormal family of p vectors $e_1, e_2, \ldots, e_p$ such that

$$\begin{array}{lcl}
\texttt{Vect}(e_1) & = & \texttt{Vect}(v_1) \\
\texttt{Vect}(e_1, e_2) & = & \texttt{Vect}(v_1, v_2) \\
\texttt{Vect}(e_1, e_2, e_3) & = & \texttt{Vect}(v_1, v_2, v_3) \\
\vdots & = & \vdots \\
\texttt{Vect}(e_1, e_2, \ldots, e_p) & = & \texttt{Vect}(v_1, v_2, \ldots, v_p)
\end{array}$$

and of course :

$$\langle e_i, e_j \rangle = \delta_{ij}, \quad 1 \leq i, j \leq p$$

The **algorithm** *is as follows*

$$\begin{array}{ll}
u_1 = v_1 & e_1 = \frac{u_1}{||u_1||} \\
u_2 = v_2 - \texttt{Proj}_{u_1}(v_2) & e_2 = \frac{u_2}{||u_2||} \\
u_3 = v_3 - \texttt{Proj}_{u_1}(v_3) - \texttt{Proj}_{u_2}(v_3) & e_3 = \frac{u_3}{||u_3||} \\
\vdots & \vdots \\
u_p = v_p - \sum_{i=1}^{p-1} \texttt{Proj}_{u_i}(v_p) & e_p = \frac{u_p}{||u_p||}
\end{array}$$

The objectives of this exercise are the following.

1. *Implement the Gram-Schmidt algorithm.*
2. *If $p = n$, the family $e_1, e_2, \ldots, e_n$ is an orthogonal basis of $\mathbb{R}^n$ and the matrix P of the components of these vectors is an orthogonal matrix, which means that $P^T\,P = I_n$.*
   The objective here is to generate randomly orthogonal matrices of order n (e.g., with $n = 5$).

2 Graphic with matplotlib

If not already done, it is recommended to configure the Spyder environment so the graphics appear in a separate window, as indicated in section ??, or as recalled here.

1. Go to *Tools/Preferences/IPython console*
2. Select *Graphics* on the right side of the window's menu
3. In *Graphics backend*, choose *Automatic* (if it is not already enabled)
4. You may have to re-launch Spyder

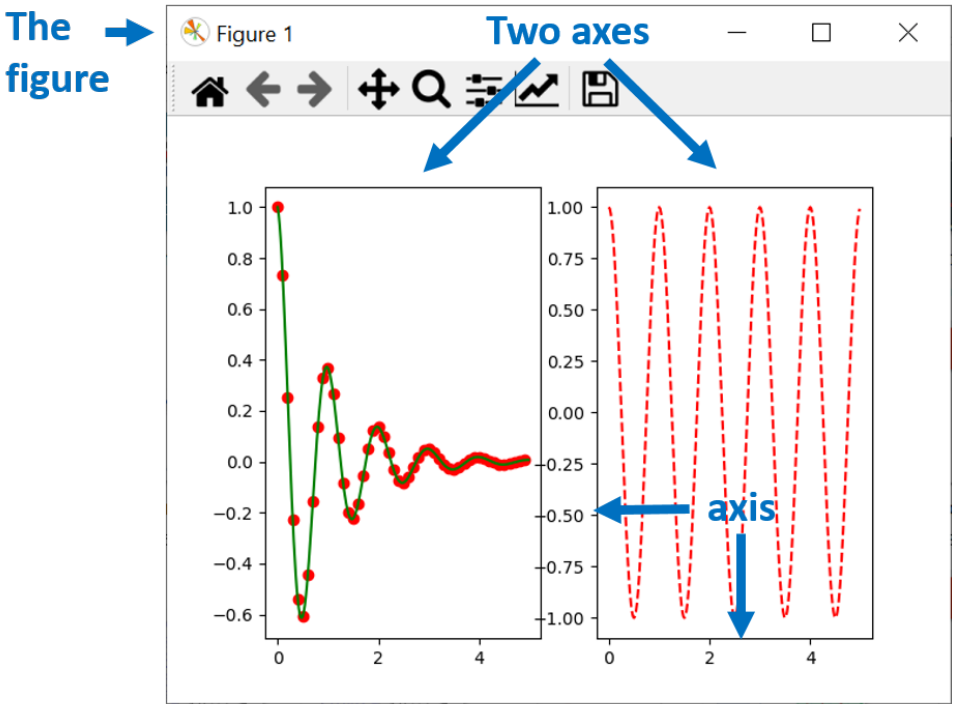
```
"""________________________________________________
--> matplotlib (graphic library) :
        matplotlib.pyplot is a collection of command style functions
        that make matplotlib work like MATLAB
--> pyplot, have the concept of the current figure and the current axes :
        All plotting commands apply to the current axes
____________________________________________________"""
```

`http://matplotlib.org/users/pyplot_tutorial.html`

2.1 Figure and Axes

We present some basic instructions and useful tools for working with the Matplotlib graphics window. In particular, Matplotlib distinguishes the *figure* from the *axes*.

- The `figure` is the graphic window.
- The `axes` is a part of a figure devoted to the display (the current plot).
  Note that a figure can contains several axes, with instruction `subplot` (see section 2.5)
- Note that an `axis` is a coordinate axis and is part of an axes.



Here are some useful Python instructions which will be used often in the subsections below :

```
"""________________________________________________
First , some usefull instructions
___________________________________________________"""
import matplotlib.pyplot as plt
plt.figure(17)  # create figure with number 17
plt.figure(3)   # create figure with number 3
plt.figure()    # which number of figure...? (number 18 here)
plt.figure('This is a new figure')  # which number ?
# number 19 here, but not displayed

plt.cla()       # Clear the current axis
plt.clf()       # Clear the current figure

plt.close()     # Close the current figure window
plt.close(17)   # Close figure window number 17
plt.close("all") # close all figure windows

plt.gcf()       # returns the current figure
plt.gca()       # returns the current axes
```

2.2 2D plot of points and polylines

Here we present the main tools for plotting points and polylines as well as some important graphics options that will be essential for displaying the graph of real functions and parametric curves.

Configuration of figure (example)

```
"""________________________________________________
--> 2D plot : points and polylines
       We need first to configure
       the graphic window and the axes
___________________________________________________"""
import matplotlib.pyplot as plt

# Each of these 3 instructions will automatically open a figure
plt.title("First example of figure")
plt.xlabel('x-axis: abscissas')
plt.ylabel('y-axis: ordinates')

# to change some labels
ax = plt.gca()                 # first , recover the current axes
ax.set_xlabel('u-axis')        # new label for xaxis
ax.set_ylabel('v-axis')        # new label for yaxis
ax.set_title('So, now second example....') # new title

# and change again :
ax.set_xlabel('XLabel', loc='right')    # on the right side
ax.set_ylabel('YLabel', loc='top')      # at the top
```

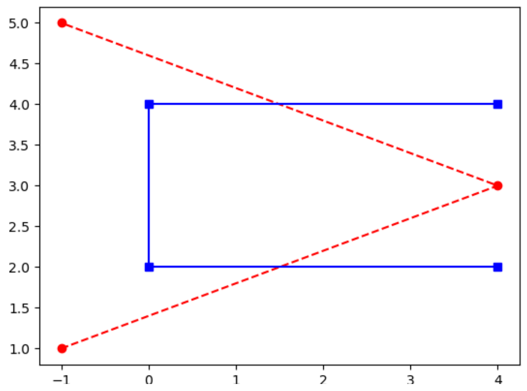
Plot (for lines)

All graphic functions such as the function `plot` are functions from `matplotlib.pyplot` and thus must be used as follows :

`plt.plot(...)` to apply to the curent figure
`ax.plot(...)` to apply to the axes `ax`

The instruction `plt.plot([x0,x1,x2,...,xn],[y0,y1,y2,...,yn],'options')` plots the polyline joining points $(x0, y0)$, $(x1, y1)$, $(x2, y2)$,..., $(xn, yn)$ with `'options'` display.

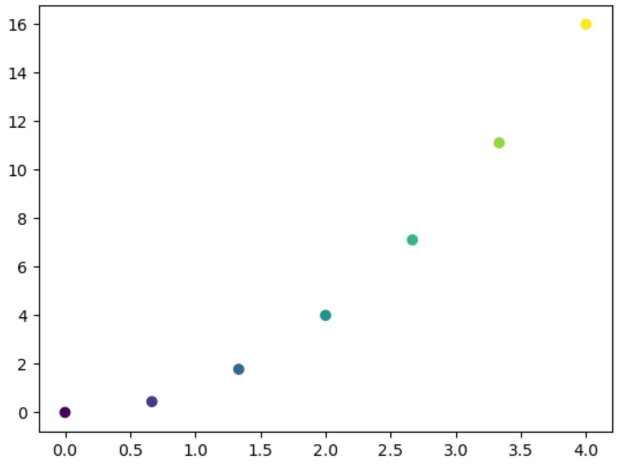
```
"""__________
plot
______________"""
# The instruction plt.plot([x0,x1,x2,...,xn],[y0,y1,y2,...,yn],'options')
# plots the polyline joining points
# $(x0,y0)$, $(x1,y1)$, $(x2,y2)$.., $(xn,yn)$ with 'options' display.
plt.plot([-1,4,-1], [1,3,5], 'ro--') # points in red, joined by dot line
ax = plt.gca()
ax.plot([4,0,0,4], [4,4,2,2], 'bs-') # blue squares joined by solid line
plt.show()                           # optional (most of the time)
```



Scatter (for points)

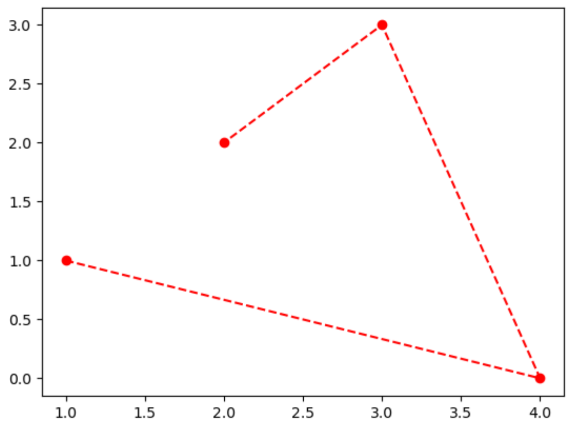
The instruction `scatter` only displays isolated points.

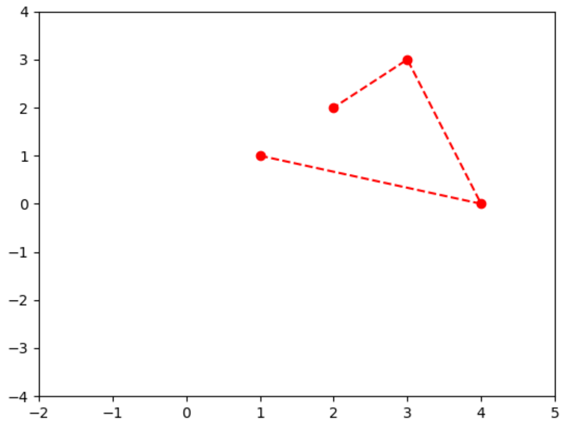
```
"""__________
scatter
______________"""
# plot only points
plt.figure()
ax = plt.gca()
ax.scatter([0, 1], [1, -1], c='r')  # plot the two points in red

import numpy as np
plt.figure()
ax = plt.gca()
t = np.linspace(0,4,7)
ax.scatter(t, t**2, c = t)
                    # c is a list of colors of same size as data points
                    # here c is defined by the array t
```



Replot

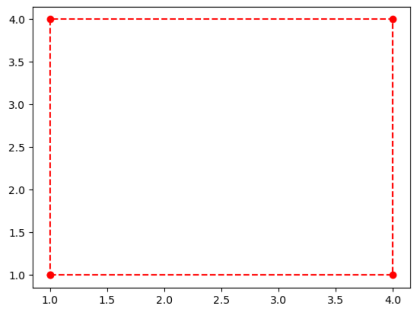
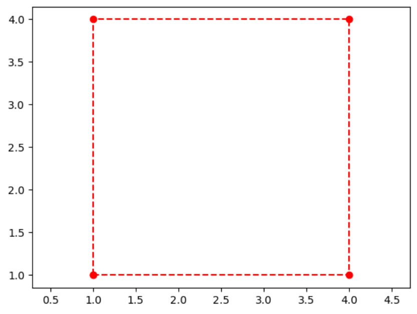
```
"""_________
replot
_____________"""
# example of a polyline with 4 points
plt.close('all')
plt.plot([1, 4, 3 , 2], [1, 0, 3, 2], 'ro--')
ax = plt.gca()
ax.axis([−2, 5, −4, 4]) # we replot the axes with different bounds
```





Axis equal

```
"""_________
axis equal
_____________"""
plt.close('all')
# look at this square ...
plt.plot([1, 4, 4, 1, 1], [1, 1, 4, 4, 1], 'ro--')
ax = plt.gca()              # now, it's better
ax.axis("equal")
```

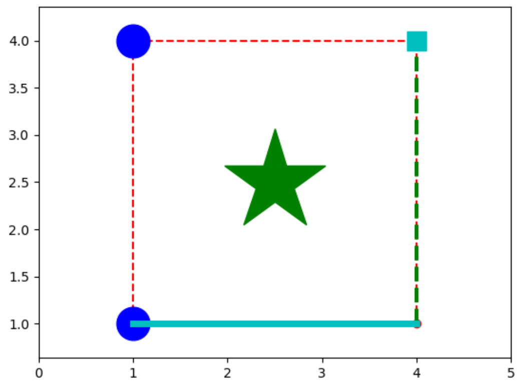
2.3 Graphic options

The instruction `plot` can be completed by adding parameters which are graphic options in the form of a string representing the color and display style. These options also make it possible to choose the size of the points and the segment connecting these points as well as the legend label of the plotted polyline.
If not specified, matplotlib automatically chooses and updates the colors of your different displays.
`https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html`

Examples are provided below and throughout this section.

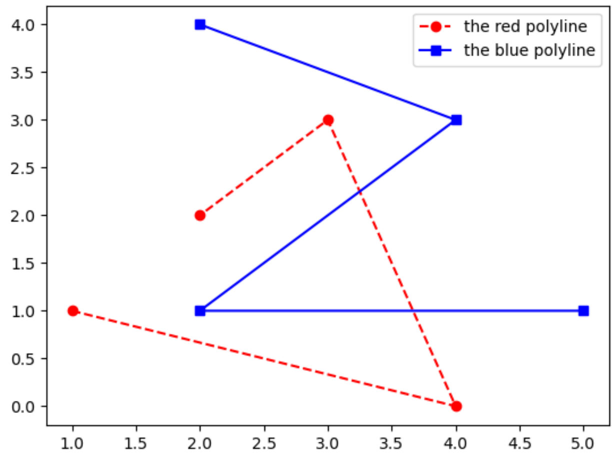
Linewidth, MarkerSize

```
"""__________
linewidth, markersize
______________"""
# we continue from the square above...
plt.plot([4, 4], [1, 4],'g--', linewidth=3)
plt.plot([1, 1], [1, 4],'bo', markersize=25)
plt.plot([1, 4], [1, 1],'c', lw=5)  # lw for linewidth
plt.plot(4, 4,'cs', ms=15)          # ms for markersize

plt.plot([2.5], [2.5], 'g*', ms=80) # * is a star marker

# replot
ax2 = plt.gca()
ax2.axis([0, 5, 0, 5])
```



Legend

```
"""___________
Legend
______________"""
plt.close('all')
# for a legend we need to label each polyline (each curve, each graphic)
plt.plot([1, 4, 3 , 2], [1, 0, 3, 2], 'ro--', label='the red polyline')
plt.plot([5, 2, 4 , 2], [1, 1, 3, 4], 'bs-', label='the blue polyline')
plt.legend(loc='best')          # best location of the legend in the axes
```



Axis and frame

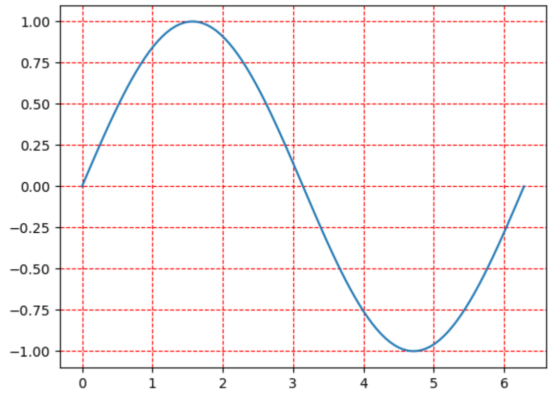
```
"""___________
axis and frame
______________"""
# NOTE : the 4 following instructions have to be used for
# "xlim, ylim", "grid", "get and set axes position"
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0,2*np.pi,100)
plt.plot(x, np.sin(x))

# note the difference between axis and frame :
plt.axis('off')          # current axis is hidden
plt.axis('on')

ax = plt.gca()
ax.set_frame_on(False)   # the frame is hidden
ax.set_frame_on(True)
```

xlim, ylim

```
"""___________
xlim, ylim
______________"""
plt.xlim(-2, 5)           # plt.xlim(x_min,x_max)
plt.ylim(0, 2)            # plt.ylim(y_min,y_max)
```

Grid

```
"""__________
grid
______________"""
plt.grid(True)          # display a grid on the current axes
plt.grid(True, which="both", linestyle='--', color='r')
plt.grid(False)
```

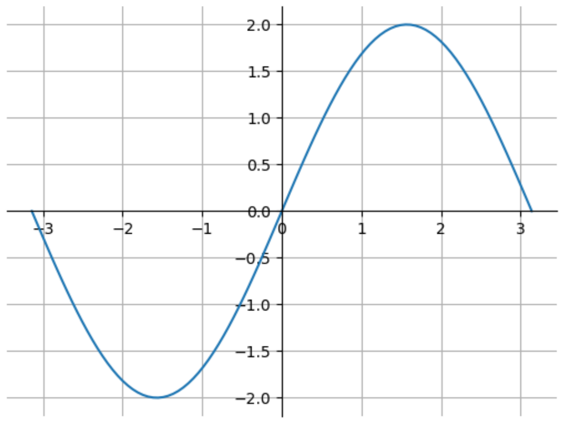


get and set axes position

```
"""__________
get and set axes position
______________"""
ax = plt.gca()
pos1 = ax.get_position() # get the original position of ax in the figure
pos2 = [pos1.x0 + 0.5, pos1.y0 + 0.2,    \
        pos1.width / 5.0,                \
        pos1.height / 2.0]      # new position
ax.set_position(pos2)        # set the new position
```

Put the origin in the center of the figure

```
"""__________
Put the origin in the center of the figure
______________"""
x = np.linspace(-np.pi, np.pi, 100)
y = 2*np.sin(x)
ax = plt.gca()
ax.plot(x, y)
ax.grid(True)
# we put the origin in the center of the figure
ax.spines['left'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('zero')
ax.spines['top'].set_color('none')
```



Save the figure

```
"""____________
Save the figure (in the current folder)
_______________"""
# SOLUTION 1 :
#   click on the button "Save the figure" of the window figure
# SOLUTION 2 :
#   with the instruction :
plt.savefig("MyFigure.png")
#   => the current figure is saved with name "MyFigure.png" in the current folder
```

2.4 Plot of the graph of a function

The display of the graph of a function $f : x \in [a, b] \mapsto f(x) \in \mathbb{R}$ is quite similar to the display of a polyline.
The graph C_f of f is the 2D curve defined as follows

$$C_f = \{(x, f(x), \ x \in [a, b]\}.$$

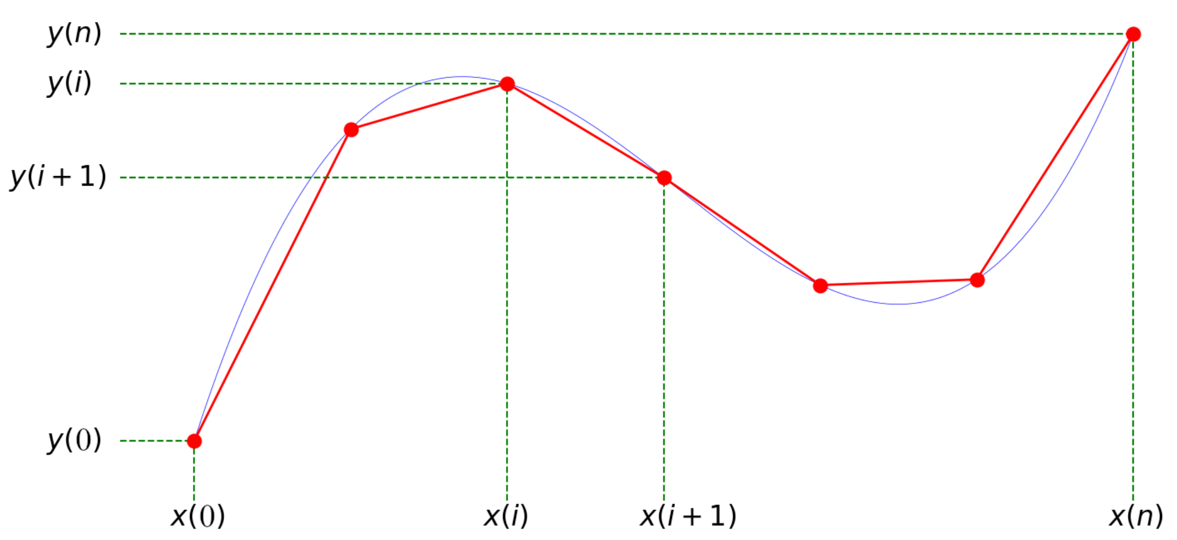
Clearly, all points of this curve C_f can not be represented since this curve has an infinite number of points. Consequently, we can only draw an approximation of the curve C_f, *by choosing a finite number of points on this curve.* Mathematically, we says that "*the curve is approximated by piecewise linear interpolation*".

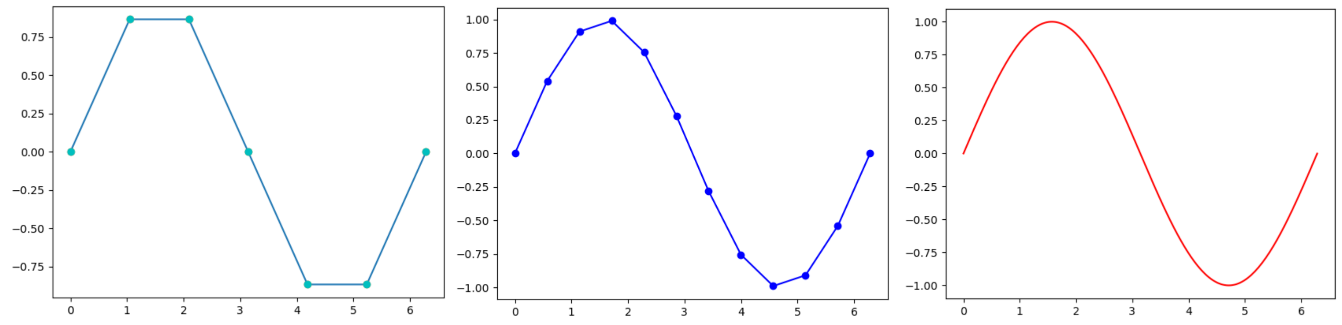
• We proceed as follows (see next figure).

1. The interval $[a, b]$ is sampled by a sequence of $n+1$ increasing abscissae, in the form of a Python vector (a one dimensional array) :
   `x =` $[a = x_0, x_1, ..., x_i, x_{i+1}, ..., x_n = b]$
   Typically : `x = np.linspace(a,b,n+1)` or `x = np.arange(a, b, step)`

2. We define a sequence of $n + 1$ ordinates which are the images of the previous abscissae by the function f :
   `y =` $[y_0 = f(x_0), ..., y_i = f(x_i), y_{i+1} = f(x_{i+1}), ..., y_n = f(x_n)]$
   Typically : `y = f(x)` if the function f is defined as a Python function.

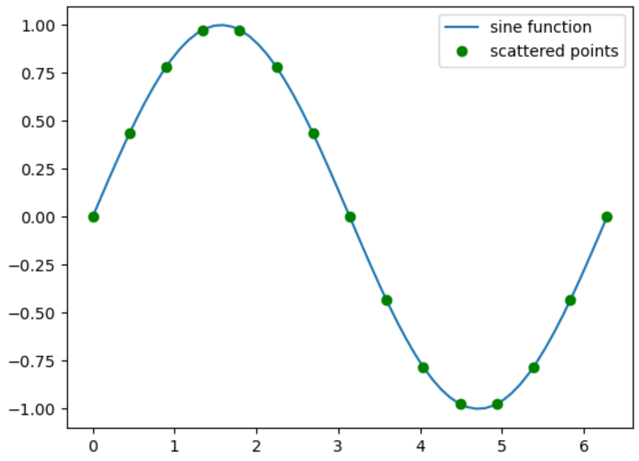
3. We then plot the sequence of points defined by the two vectors `x` and `y` as in the previous section.
   Typically : `plt.plot(x,y)`





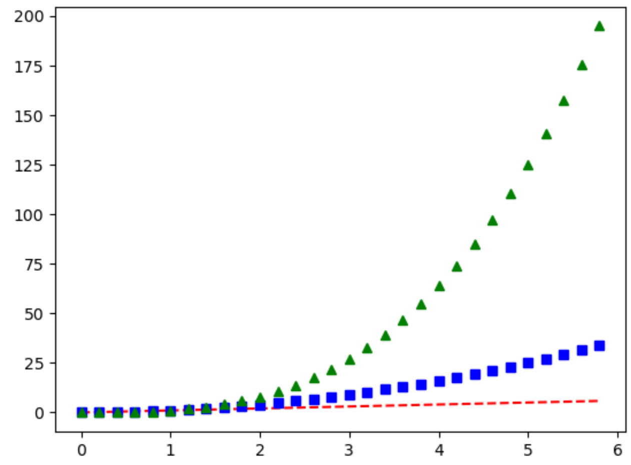
As an example : plot of the sine function with 7, 12 and 200 points

```
"""_______________________________________________
—> Plot of the graph of a function
__________________________________________________"""
import numpy as np
import matplotlib.pyplot as plt

n = 60
x = np.linspace(0,2*np.pi,n)
y = np.sin(x)
plt.plot(x, y, label="sine function") # color is chosen by pyplot

n = 15
x = np.linspace(0,2*np.pi,n)
y = np.sin(x)
plt.plot(x, y, 'go', label="scattered points")
plt.legend(loc='best')
```



One plot, several graphs

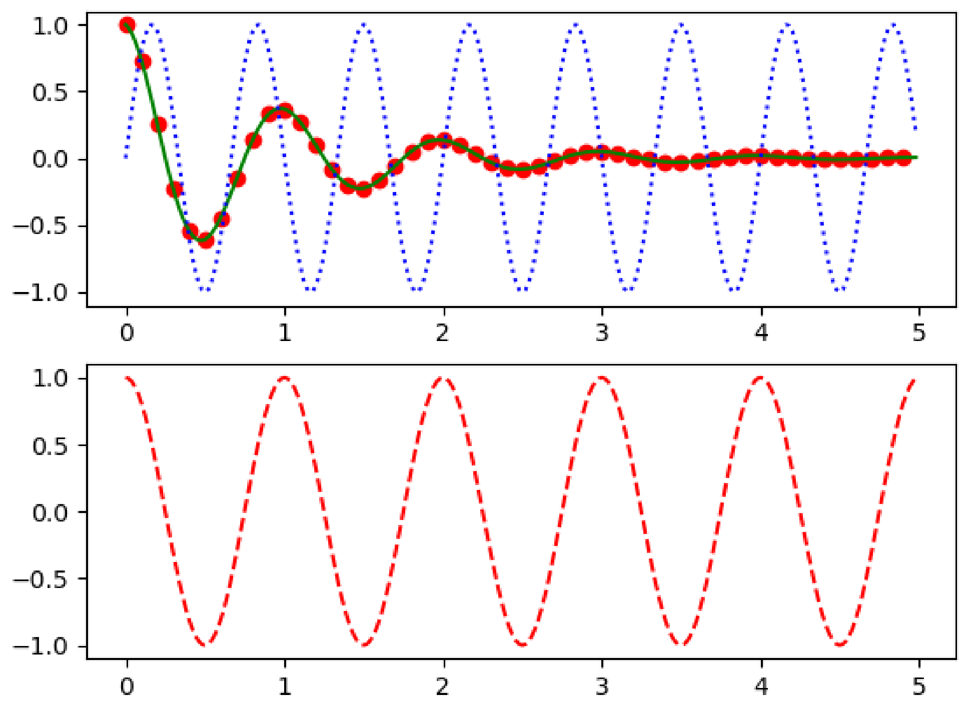
```
"""____________________________________________
—> one plot, several graphs
____________________________________________"""
plt.clf()  #clear the current figure
# evenly sampled abscissa
t = np.arange(0., 6., 0.2)
# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```



2.5 MultiFig and subplot

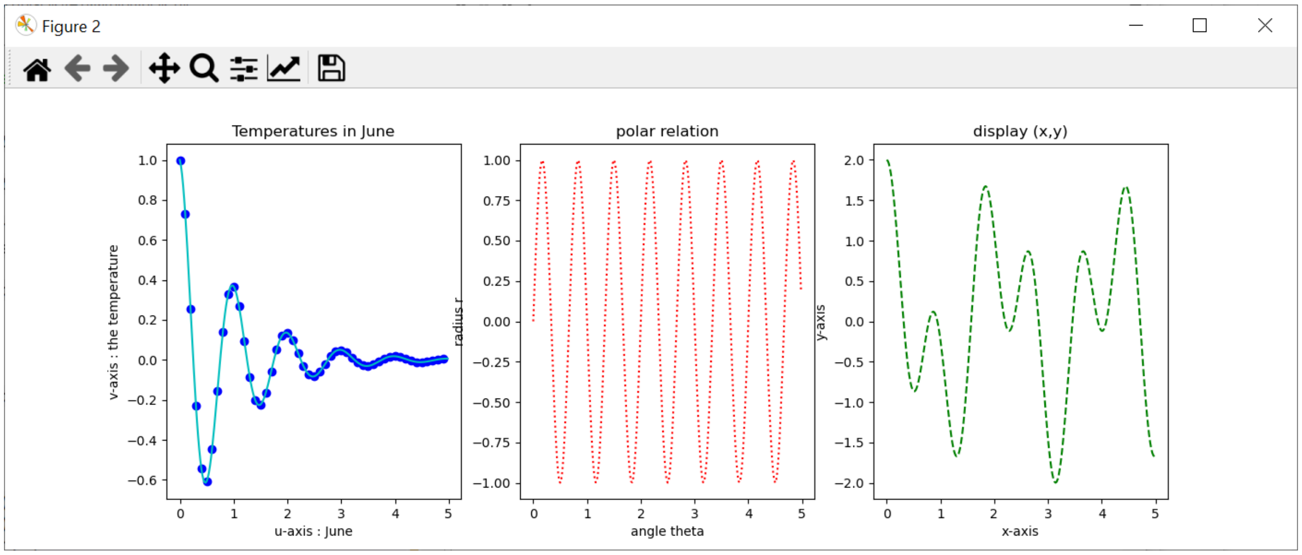
```
"""____________________________________________
—> MultiFig and subplot :
____________________________________________"""
import numpy as np
import matplotlib.pyplot as plt

# some material for future plots
def f(t):
        return np.exp(-t) * np.cos(2*np.pi*t)
```

```
t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

"""
VERSION 1 : with plt.subplot(ijk) to "activate" axes k
"""
plt.figure(1)        # optional
plt.subplot(211)     # subplot(nrows, ncols, plot_number)
plt.plot(t1, f(t1), 'ro', t2, f(t2), 'g')

plt.subplot(212)     # create the second axes (= the current axes)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()           # optional here

plt.subplot(211)     # becomes the current axes
plt.plot(t2, np.sin(3*np.pi*t2), 'b:') # a new plot on this axes

plt.cla()            # clear the current axes --> axes 211  here
plt.clf()            # clear the current figure
plt.close()          # close the current figure window
```



```
"""
VERSION 2 : with a direct link to each axes
"""
plt.close("all")
fig2 = plt.figure(2,(14,5)) # create figure number 2, with size 14 x 5

ax1 = fig2.add_subplot(131) # first axes with link "ax1"
ax1.set_xlabel('u-axis : June')
ax1.set_ylabel('v-axis : the temperature')
ax1.set_title("Temperatures in June")
```

```
ax3 = fig2.add_subplot(133) # new axes with link "ax3"
ax3.set_xlabel('x-axis')
ax3.set_ylabel('y-axis')
ax3.set_title("display (x,y)")

ax2 = fig2.add_subplot(132) # new axes with link "ax2"
ax2.set_xlabel('angle theta')
ax2.set_ylabel('radius r')
ax2.set_title("polar relation")

ax3.plot(t2, np.cos(3*t2) + np.cos(7*t2), 'g--')
ax1.plot(t1, f(t1), 'bo', t2, f(t2), 'c')
ax2.plot(t2, np.sin(3*np.pi*t2), 'r:')

ax2.clear()      # clear axes ax2
plt.clf()        # clear the current figure (all axes)
```

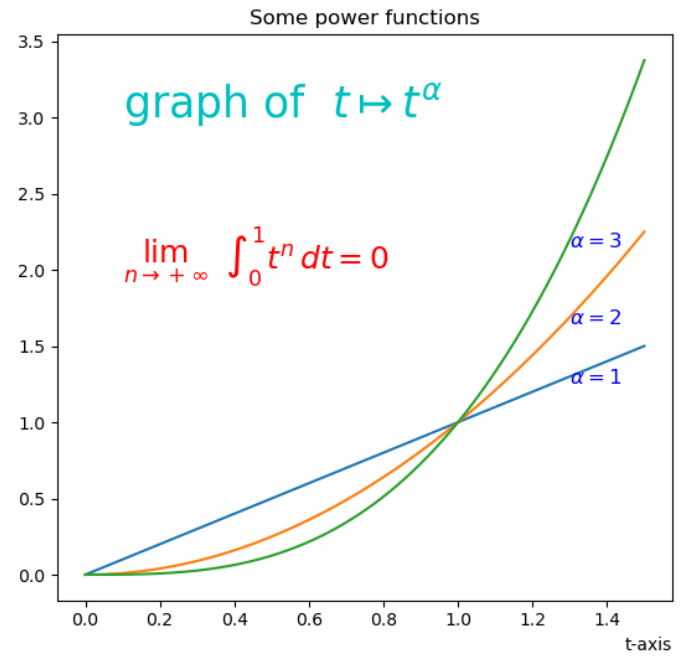


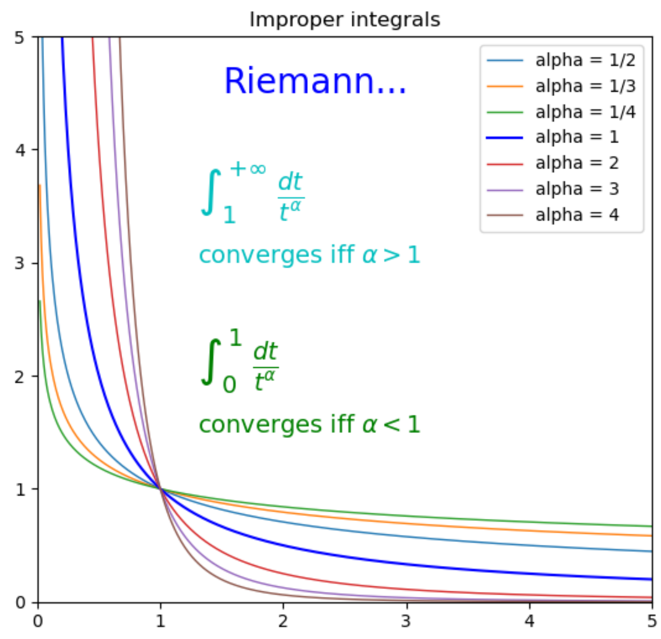
2.6 Working with text

The instruction `text(x0,y0,'Mytext')` add `"Mytext"` at position `(x0,y0)`.
Syntax `r'$...$'` allows to write Latex formulas.

```
"""________________________________________________
—> Working with text
___________________________________________________"""
# instruction text(x0,y0,'Mytext') add "Mytext" at position (x0,y0)
# syntax r'$...$' allows to write Latex formulas
import numpy as np
import matplotlib.pyplot as plt
plt.close("all")
plt.figure('This is a new figure with text')
plt.xlabel('t-axis', loc='right')
plt.ylabel('y(t)',loc='top')
plt.title('Some power functions')
```

```
"""
EXAMPLE 1
"""
x0 = 1.3
t = np.linspace(0,1.5,500)
for p in range(1,4):
        plt.plot(t, t**p, label='power '+str(p))
        y0 = x0**p - 0.05       # (x0,y0) = position of text
        plt.text(x0, y0,\
                r'$\alpha = $'+str(p),\
                color='b', fontsize=12)

plt.text(0.1, 3,'graph of  '\
        r'$t \mapsto t^\alpha$',\
        color='c', fontsize=25)
plt.text(0.1, 2.,\
        r'$\lim_{n\rightarrow +\infty} \;\; \int_0^1 t^n\,dt = 0$',\
        color='r', fontsize=18)

"""
EXAMPLE 2
"""
plt.cla()
plt.title('Improper integrals')
plt.text(1.5, 4.5,'Riemann...', color='b', fontsize=20)
t = np.linspace(0.02,5,500)

for p in range(2,5):
        pp = 1/p
        plt.plot(t, 1./t**pp,lw=1, label='alpha = 1/'+str(p))

plt.plot(t, 1./t, c='b', lw=1.5, label='alpha = 1')

for p in range(2,5):
        plt.plot(t, 1./t**p,lw=1, label='alpha = '+str(p))

plt.legend(loc='best')
plt.axis([0,5,0,5])

x0 = 1.3
plt.text(x0, 3.5, r'$\int_1^{+\infty} \; {\frac{dt}{t^\alpha}}$',\
        color='c', fontsize=20)
plt.text(x0, 3, r'converges iff $\alpha > 1$',\
        color='c', fontsize=14)
plt.text(x0, 2., r'$\int_0^{1} \; {\frac{dt}{t^\alpha}}$',\
        color='g', fontsize=20)
plt.text(x0, 1.5, r'converges iff $\alpha < 1$',\
        color='g', fontsize=14)
```





2.7 Plot of a 2D parametric curve

Consider a 2D parametric curve defined by

$$F \ : \quad t \in [a,b] \ \mapsto \ \begin{cases} x = F_1(t) \\ y = F_2(t) \end{cases}$$

To plot the parametric curve $\Gamma = F([a,b])$, we proceed as follows.

1. The parametric interval $[a,b]$ is sampled by a sequence of $n+1$ increasing parameters, in the form of a Python vector :
   `t =` $[a = t_0, t_1, ..., t_i, t_{i+1}, ..., t_n = b]$
   Typically : `t = np.linspace(a,b,n+1)` or `t = np.arange(a,b,step)`

2. We then define two sequences :

   - a sequence of $n+1$ abscissae which are the images of the parameters t_i by the function F_1 :
     `x =` $[x_0, ..., x_i, ..., x_n] = [F1(t_0), ..., F1(t_i), ..., F1(t_n)]$
     Typically : `x = F1(t)` if the function F_1 is defined as a Python function,
   - a sequence of $n+1$ ordinates which are the images of the parameters t_i by the function F_2 :
     `y =` $[y_0, ..., y_i, ..., y_n] = [F2(t_0), ..., F2(t_i), ..., F2(t_n)]$
     Typically : `y = F2(t)` if the function F_2 is defined as a Python function.

3. We plot the sequence of points defined by the two vectors `x` and `y` as a polyline.
   Typically : `plt.plot(x,y)`

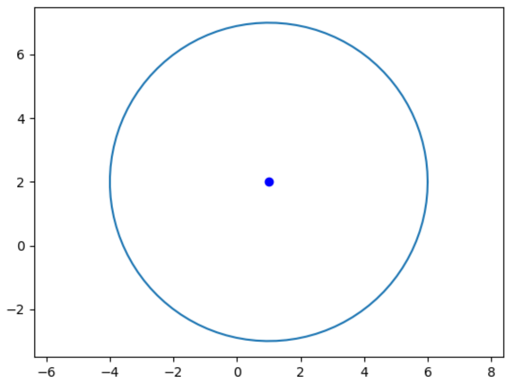
As an example, we plot the following two parametric curves.

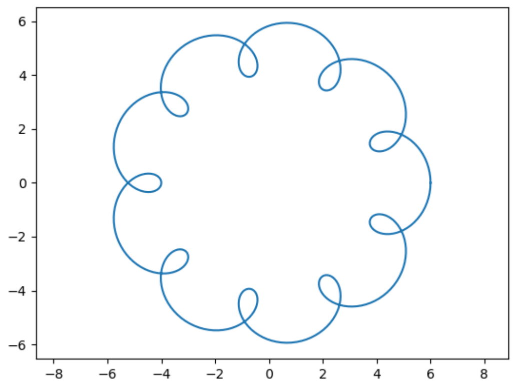
Circle with center (x_0, y_0) and radius R

$$\theta \in [0, 2\,\pi] \ \mapsto \ \begin{cases} x &=& x_0 + R\,\cos(\theta) \\ y &=& y_0 + R\,\sin(\theta) \end{cases}$$

A strange curve....

$$\theta \in [0, 2\,\pi] \ \mapsto \ \begin{cases} x &=& R\,\cos(\theta) + r\,\cos(10\,\theta) \\ y &=& R\,\sin(\theta) + r\,\sin(10\,\theta) \end{cases}$$

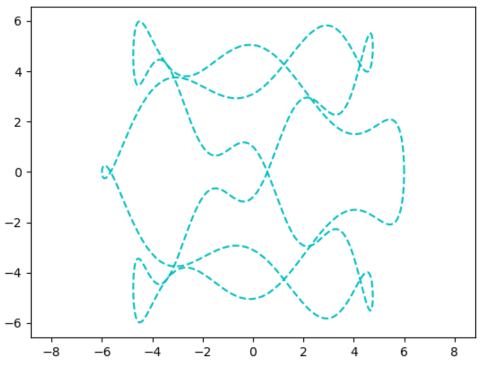




```
"""_______________________________________________
—> Plot of a 2D parametric curve
__________________________________________________"""
# Example 1 : a circle
import numpy as np
import matplotlib.pyplot as plt

theta = np.linspace(0, 2*np.pi, 100)
x0, y0 = 1, 2
R = 5
x = x0 + R * np.cos(theta)
y = y0 + R * np.sin(theta)
plt.figure()
plt.plot(x0,y0,'bo') # we mark the center
plt.plot(x, y)
plt.axis("equal")

# Example 2 : a strange curve...
import numpy as np
import matplotlib.pyplot as plt

theta = np.linspace(0, 2*np.pi, 500)
R = 5
r = 1
x = R * np.cos(theta) + r * np.cos(10*theta)
y = R * np.sin(theta) + r * np.sin(10*theta)
plt.figure()
plt.plot(x,y)
plt.axis("equal")
```

If we change the last 5 lines of the previous script as follows, we get a more strange curve...

```
x = R * np.cos(3*theta) + r * np.cos(5*theta)
y = R * np.sin(2*theta) + r * np.sin(17*theta)
plt.figure()
plt.plot(x,y,'c--')
plt.axis("equal")
```

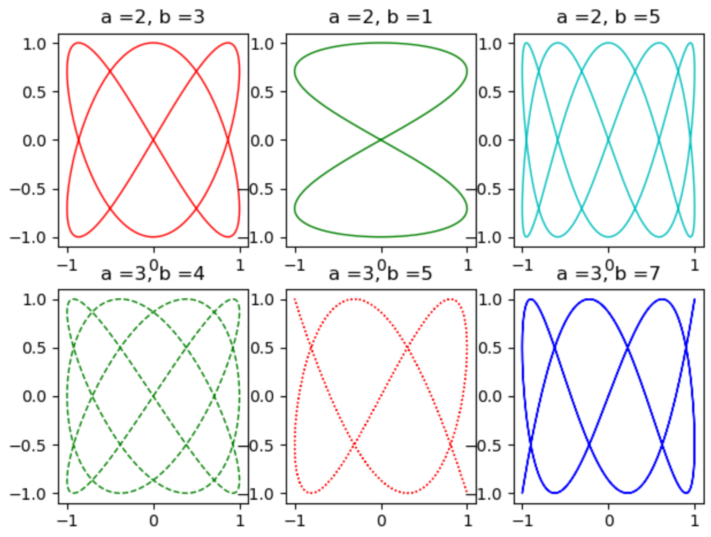


Exercise 0.7

We give the following script.

```
# exof41LissajousA.py
"""
Function for evaluating Lissajous curves
"""
import numpy as np

def sinatsinbt(t,a,b):
        x = np.sin(a*t)
        y = np.sin(b*t)
        return x,y
```

Write another Python script `exof41LissajousB.py` *allowing to display the following figure*



and where each curve is a Lissajous curve with parametric equation

$$t \in [0, 2\,\pi] \quad \mapsto \quad \begin{cases} x &= \sin(a\,t) \\ y &= \sin(b\,t) \end{cases}$$

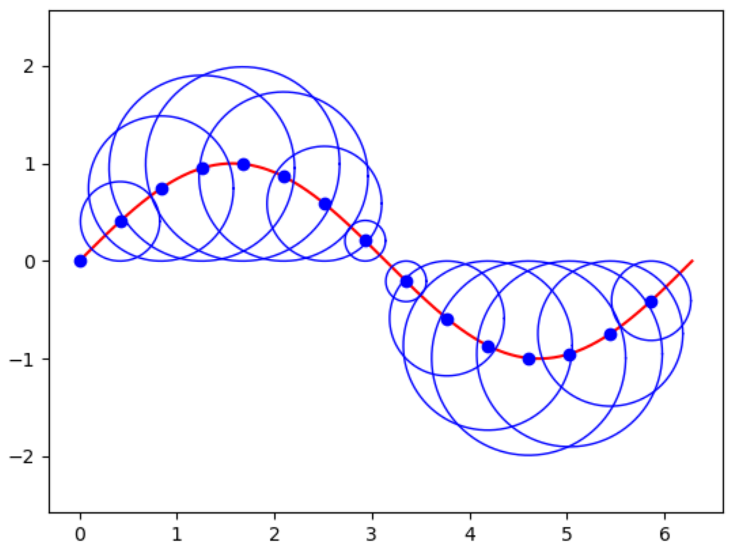
where parameters a and b are given in the figure.

Exercise 0.8

Write a script with name `exof41CirclesOnSine.py` *including a Python function that draws a circle with center* `(Cx,Cy)`*, radius* `R` *and color* `col` *:*

```
# a function that display circles with color "col"
def DrawCircle(Cx,Cy,R,col):
        """
                Cx,Cy   = center of the circle : 2 floating numbers
                R       = radius : float
                col     = color of the circle : string
        """
        .....
```

and that produces the following figure where all circles have center on the graph of the sine *function and are tangent to the x-axis (15 circles in this figure, the first one of radius zero).*

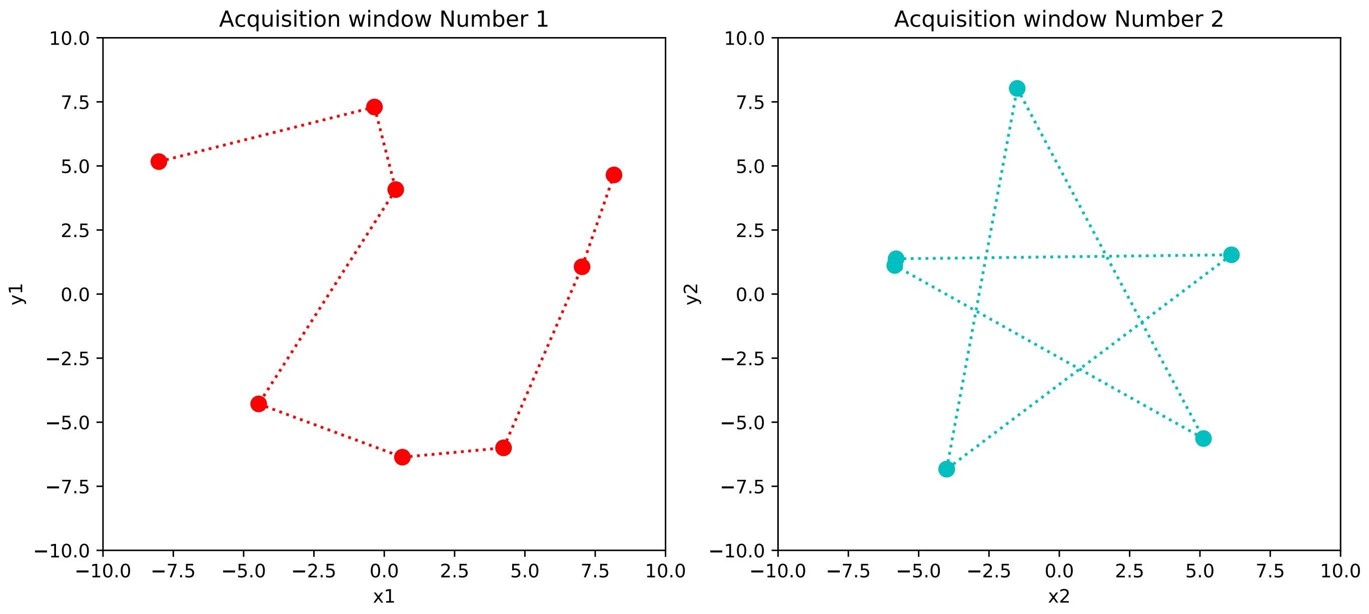


2.8 Mouse acquisition

The following script is used to acquire a polygon in a graphics window with the mouse. It will be very useful for testing interpolation and approximation methods.

```
"""__________________________________________________
—> Mouse acquisition : ginput()
______________________________________________________"""
import numpy as np
import matplotlib.pyplot as plt

def AcquisitionPolygone(ax,color1,color2) :
    """ Mouse acquisition of a polygon in the axes with subplot "ax"
                color1 for points and color2 for segments
                right click to stop
        """
    x = []
    y = []
    coord = 0
    while coord != []:
        coord = plt.ginput(1, mouse_add=1, mouse_stop=3, mouse_pop=2)
        if coord != []:
            xx = coord[0][0]
            yy = coord[0][1]
            ax.plot(xx,yy,color1,ms=8);
            x.append(xx);
            y.append(yy);
            plt.draw()
            if len(x) > 1 :
                ax.plot([x[-2],x[-1]],[y[-2],y[-1]],color2)
    return x,y

# we create a figure with 2 axes
```

```
plt.close()
fig = plt.figure(1,(12,5))
minmax = 10

ax1 = fig.add_subplot(121)
ax1.set_xlim((-minmax,minmax))
ax1.set_ylim((-minmax,minmax))
ax1.set_xlabel('x1')
ax1.set_ylabel('y1')
ax1.set_title("Window #1")

ax2 = fig.add_subplot(122)
ax2.set_xlim((-minmax,minmax))
ax2.set_ylim((-minmax,minmax))
ax2.set_xlabel('x2')
ax2.set_ylabel('y2')
ax2.set_title("Window #2")

# acquisition of a polygon in each axes
xi1, yi1 = AcquisitionPolygone(ax1, 'or',':r')
xi2, yi2 = AcquisitionPolygone(ax2, 'oc',':c')
```

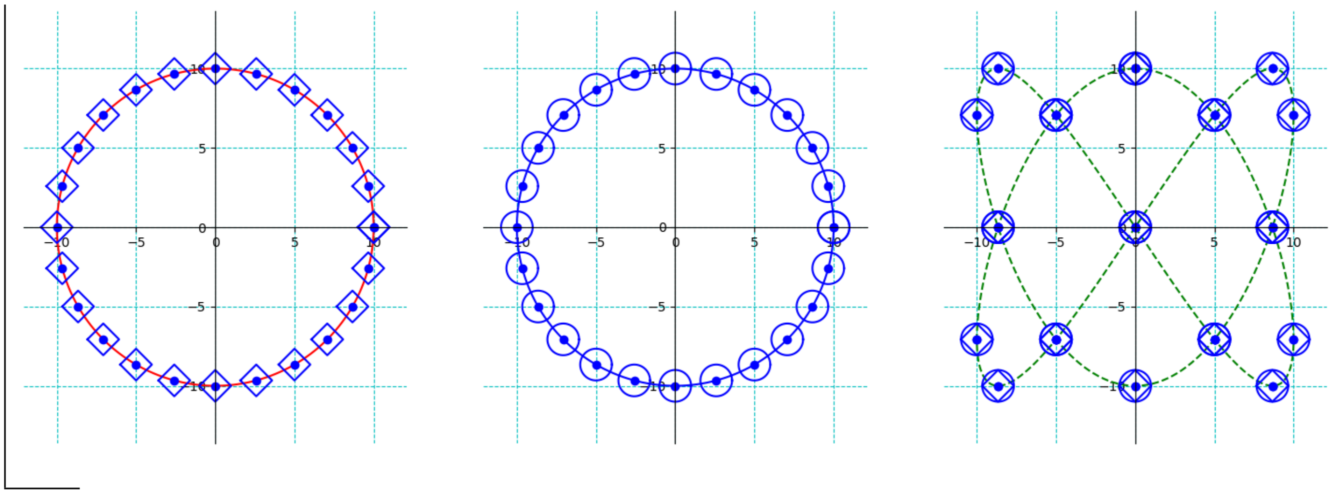


Exercise 0.9 (*squares and circles on a curve*)

Write a Python script (named `exof41SquaresOnCurve.py`*) to plot 25 small squares or small circles on a larger circle of radius $R = 10$ or on the lissajous curve*

$$t \in [0, 2\,\pi] \quad \mapsto \quad \left\{ \begin{array}{rcl} x & = & R\,\sin(2\,t) \\ y & = & R\,\cos(3\,t) \end{array} \right.$$

as shown in the following figure.



Note : from now, we use the Python script `f41Matplotlib3D.py`

2.9 3D display

Three-dimensional plots are enabled by importing the `mplot3d` toolkit. Once this submodule is imported, a 3D axis is created by using the instruction : `projection='3d'` Thus, we will start any 3D display with :

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

fig = plt.figure()
ax = plt.axes(projection='3d')
# or
ax = fig.gca(projection='3d')
```

2.10 Plot of a 3D parametric curve

The method to plot a 3D parametric curve is similar to that of the 2D case.
Consider a 3D parametric curve defined by

$$F \;:\quad t \in [a,b] \;\mapsto \begin{cases} x = F_1(t) \\ y = F_2(t) \\ y = F_3(t) \end{cases}$$

To plot the parametric curve $\Gamma = F([a,b])$, we proceed as follows.

1. The parametric interval $[a,b]$ is sampled by a sequence of $n+1$ increasing parameters, in the form of a Python vector :
   `t =` $[a = t_0, t_1, ..., t_i, t_{i+1}, ..., t_n = b]$
   Typically : `t = np.linspace(a,b,n+1)` or `t = np.arange(a,b,step)`

2. We then define three sequences :
   - a sequence of $n+1$ values x_i which are the images of parameters t_i by the function F_1 : `x =` $[x_0, ..., x_i, ..., x_n] = [F1(t_0), ..., F1(t_i), ..., F1(t_n)]$
   - a sequence of $n+1$ values y_i which are the images of parameters t_i by the function F_2 : `y =` $[y_0, ..., y_i, ..., y_n] = [F2(t_0), ..., F2(t_i), ..., F2(t_n)]$
   - a sequence of $n+1$ values z_i which are the images of parameters t_i by the function F_3 : `z =` $[z_0, ..., z_i, ..., z_n] = [F3(t_0), ..., F3(t_i), ..., F3(t_n)]$

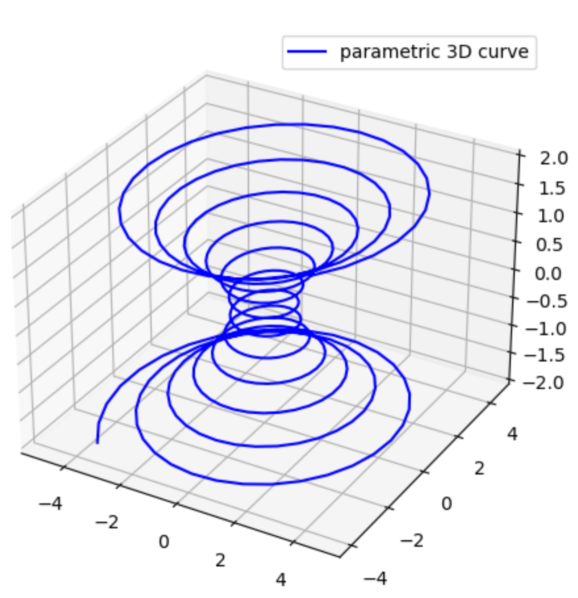
   Typically :
   `x = F1(t)` if the function F_1 is defined as a Python function,
   `y = F2(t)` if the function F_2 is defined as a Python function,
   `z = F3(t)` if the function F_3 is defined as a Python function.

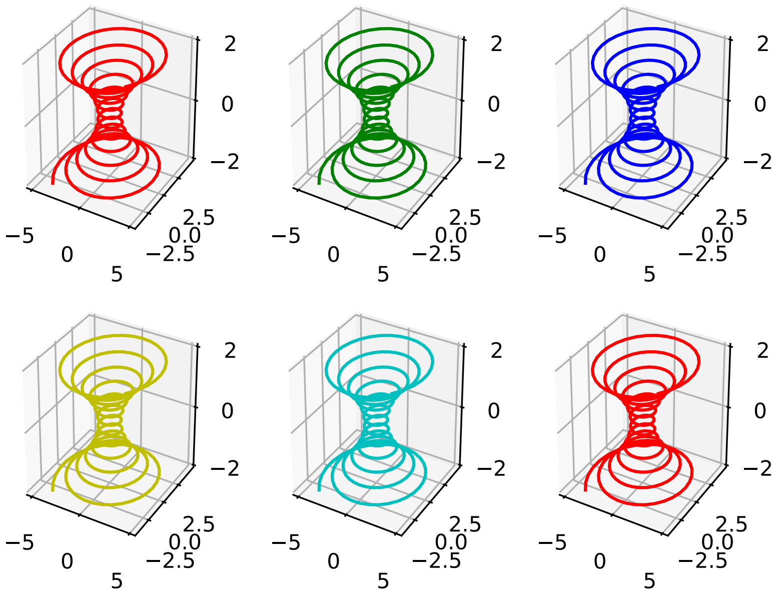
3. Then, we plot the sequence of points defined by the three vectors `x`, `y` and `z` as a 3D polyline with the instruction

```
ax.plot3D(x, y, z,...)
# or
ax.plot(x, y, z,...)
```

As an example consider the following 3D spiral propeller.

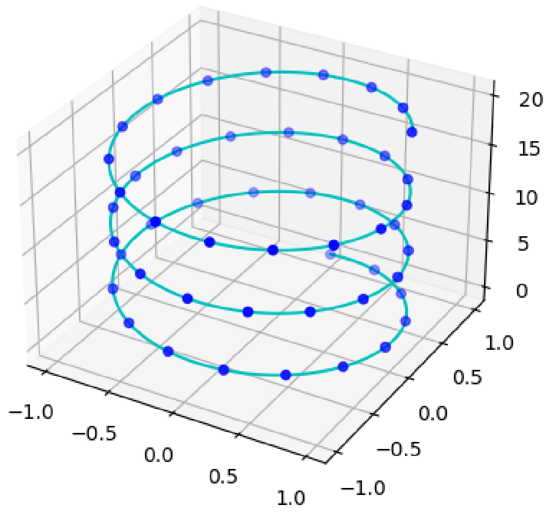
```
"""_________________________________________________
--> Plot of a 3D parametric curve
_________________________________________________"""
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

nbpts = 400
theta = np.linspace(-40, 40, nbpts)
z = theta / 20.
r = z**2 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)

# one plot:
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot3D(x, y, z, 'blue', label='parametric 3D curve')
plt.legend(loc="best")

# multiplotting:
fig = plt.figure()
TabC = ['r','g','b','y','c','r']
for k in range(6):
FigNumber = "23"+str(k+1)
ax = fig.add_subplot(FigNumber, projection='3d')
ax.plot(x, y, z, color=TabC[k])
```





Plot of scattered 3D points

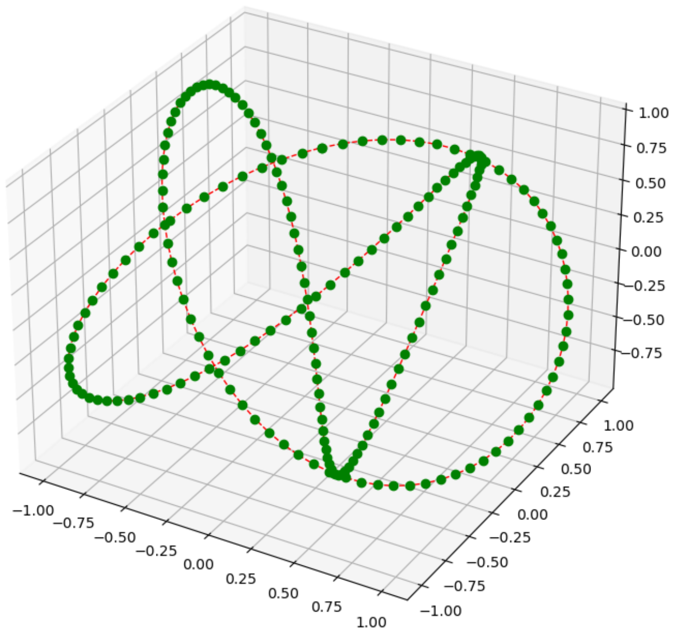
```
"""_______________________________________________
—> Plot of scattered 3D points
_______________________________________________"""
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

fig = plt.figure()
ax = plt.axes(projection='3d')

# first , a 3D line (which contains the points):
z = np.linspace(0, 20, 1000)
x = np.sin(z)
y = np.cos(z)
ax.plot3D(x, y, z, 'c')

# then, 3D scattered points :
zz = np.linspace(0, 20, 50) # uniform distribution
xx = np.sin(zz)
yy = np.cos(zz)
ax.scatter(xx, yy, zz, c='blue')
```



Exercise 0.10 (*Lissajous 3D curve*)

Write a Python script (named `exof41Lissajous3DCurve.py`*) to plot the following Lissajous 3D curve.*

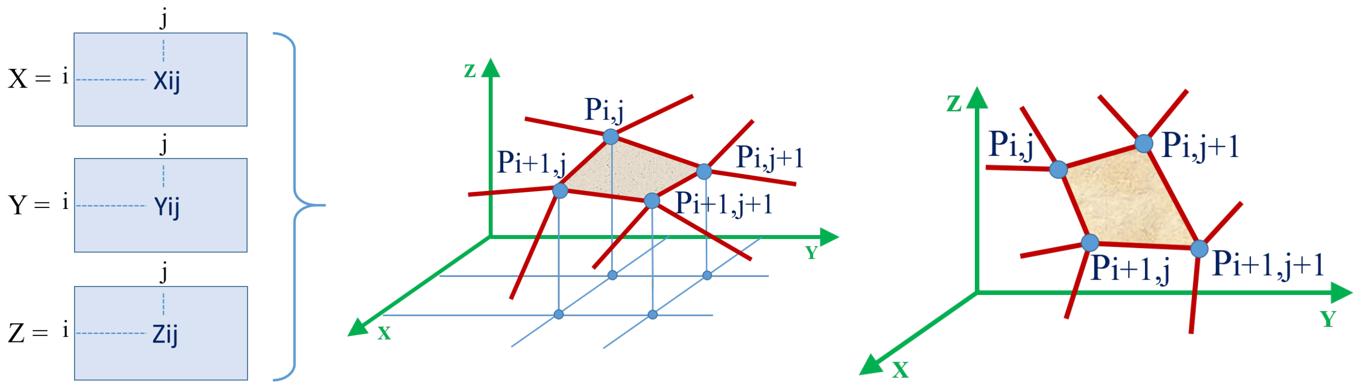
$$t \in [0, 2\,\pi] \;\mapsto\; M(t) = \begin{cases} x &= \sin(2t) \\ y &= \sin(3t) \\ z &= \cos(3t) \end{cases}$$



2.11 Tools for Surface display

For general surface display we can use three different methods :
- wireframe plot
- surface plot
- contour plot

A `wireframe plot` consists in plotting the two-dimensional grid of values of the 3D points $P_{ij} = (X_{ij}, Y_{ij}, Z_{ij})$ (i.e. the red segments in the figure below).
Usually the resulting three-dimensional shape is then quite easy to visualize and understand.



On the left, we can see the 3 two-dimensional matrices of the coordinates of the 3D points $P_{ij} = (X_{ij}, Y_{ij}, Z_{ij})$
The picture in the middle represents the case of an explicit surface : the Z-coordinates are the images of the X and Y coordinates using a function f (see section 2.12).
On the right is a parametric surface : the three coordinates are defined independently using 3 functions with `u` and `v` parameters ((see section 2.13).

A `surface plot` is like a wireframe plot, but each face of the wireframe is a filled polygon.

A `contour plot` consists of displaying level sets (level lines) according to one of the coordinates (e.g., the Z coordinate, as on a topographic map).
Thus `contour plot` is a 2D representation of a 3D surface.

Each of these methods requires all the input data X, Y, Z to be in the form of two-dimensional regular grids. For this purpose, we can use the following two numpy functions :
- np.meshgrid()
- np.outer()

np.meshgrid()

Given two sequences of numerical values :

$$x = [x_0, x_1, \dots, x_i, \dots, x_m]$$
$$y = [y_0, y_1, \dots, y_j, \dots, y_n]$$

the numpy instruction `X, Y = np.meshgrid(x,y)` produces the following arrays with the same shape $(n+1, m+1)$

$$X = \begin{bmatrix} x_0 & x_1 & \dots & \dots & x_m \\ x_0 & x_1 & \dots & \dots & x_m \\ \vdots & \vdots & & & \vdots \\ x_0 & x_1 & \dots & \dots & x_m \end{bmatrix} \quad \text{and} \quad Y = \begin{bmatrix} y_0 & y_0 & \dots & \dots & y_0 \\ y_1 & y_1 & \dots & \dots & y_1 \\ \vdots & \vdots & & & \vdots \\ y_n & y_n & \dots & \dots & y_n \end{bmatrix}$$

np.outer()

Given two sequences of numerical values :

$$x = [x_0, x_1, \ldots, x_i, \ldots, x_m]$$
$$y = [y_0, y_1, \ldots, y_j, \ldots, y_n]$$

the numpy instruction `U = np.outer(x,y)` produces the following array with shape $(m+1, n+1)$

$$U = \begin{bmatrix} x_0y_0 & x_0y_1 & \ldots & x_0y_j & \ldots & x_0y_n \\ x_1y_0 & x_1y_1 & \ldots & x_1y_j & \ldots & x_1y_n \\ \vdots & \vdots & & \vdots & & \vdots \\ x_iy_0 & x_iy_1 & \ldots & x_iy_j & \ldots & x_iy_n \\ \vdots & \vdots & & \vdots & & \vdots \\ x_my_0 & x_my_1 & \ldots & x_my_j & \ldots & x_my_n \end{bmatrix}$$

`Note :` Keep in mind that the array created with np.outer() has a different shape than the one created with np.meshgrid() ! – The shapes are transposed.

2.12 Plot of an Explicit surface

An explicit surface is the graph of a function $f : [a, b] \times [c, d] \to \mathbb{R}$. The graph S_f of this function is precisely defined as

$$S_f = \Big\{ \big(x, y, f(x, y)\big) \in \mathbb{R}^3,\ x \in [a, b],\ y \in [c, d] \Big\}.$$

It might commonly be referred to as the surface $z = f(x, y)$.

To plot it, we proceed as follows.

1. The intervals $[a, b]$ and $[c, d]$ are sampled by two sequences of $m+1$ and $n+1$ increasing values :
   $x = [a = x_0, x_1, \ldots, x_i, \ldots, x_m = b]$
   $y = [c = y_0, y_1, \ldots, y_j, \ldots, y_n = d]$
   `X = np.linspace(a,b,m+1)` and `Y = np.linspace(c,d,n+1)`

2. These input data x, y are duplicated in the form of two-dimensional regular grids,
   `X,Y = np.meshgrid(x,y)`

3. The function f is evaluated at each position (x_i, y_j) to produce a two-dimensional grid `Z` :
   `Z = f(X,Y)`

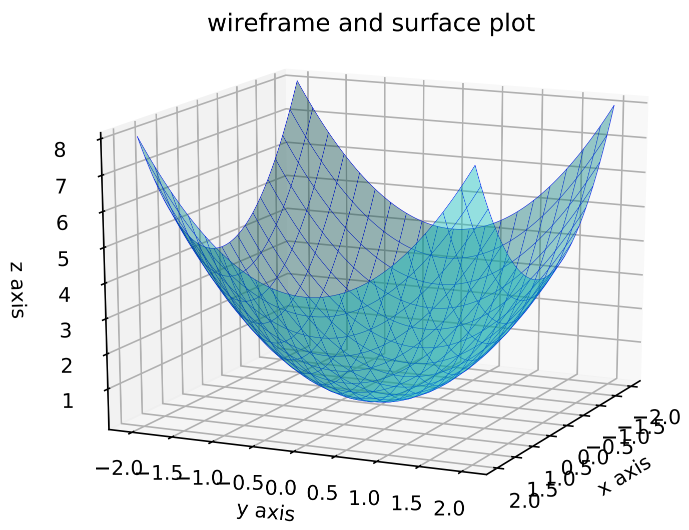
$$\mathtt{Z} = \begin{bmatrix} f(x_0, y_0) & f(x_1, y_0) & \ldots & f(x_i, y_0) & \ldots & f(x_m, y_0) \\ f(x_0, y_1) & & & & & f(x_m, y_1) \\ \vdots & & & \vdots & & \vdots \\ f(x_0, y_j) & & \ldots & f(x_i, y_j) & \ldots & f(x_m, y_j) \\ \vdots & & & \vdots & & \vdots \\ f(x_0, y_n) & f(x_1, y_n) & \ldots & f(x_i, y_n) & \ldots & f(x_m, y_n) \end{bmatrix}$$

4. We then display the two-dimensional grid of surface points P_{ij} defined by the 3 two-dimensional grids X, Y, Z with one of the instructions
   `plot_wireframe(X, Y, Z,...)` or `plot_surface(X, Y, Z,...)` or `contour(Z,...)`

   Note that the x-coordinates x_i are incremented horizontally and the y-coordinates y_j vertically.

Now consider the simple example of a paraboloid. We will use the methods `plot_wireframe` and `plot_surface`. The `contour` method will be considered later at the end of this section.

```
"""____________________________________________
--> EXPLICIT surface
    i.e. graph of a function f : R^2 --> R
                              (x,y) --> f(x,y)
____________________________________________"""
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

# figure & 3D-axes for plotting the surface
fig = plt.figure()
ax = plt.axes(projection='3d')

def f(x,y) :
    return x**2 + y**2

x = np.linspace(-2,2,100)
y = np.linspace(-2,2,100)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)

# wireframe display of the surface :
ax.plot_wireframe(X, Y, Z, color='c', lw=0.5,
                  rstride=4, cstride=4) #row stride & column stride
# surface display :
ax.plot_surface(X, Y, Z, color='c',
                alpha=0.4,            # transparency
                rstride=5, cstride=5, linewidth=0.5)

ax.set_xlabel('x axis')
ax.set_ylabel('y axis')
ax.set_zlabel('z axis')
ax.set_title('wireframe and surface plot')
ax.view_init(15, 30)
        # elevation of 15 degrees (above the x-y plane)
        # azimuth of 30 degrees (counter-clockwise about the z-axis)
```



2.13 Plot of a parametric surface

A 3D parametric surface is the image of a domain of $\mathbb{R}^2$ (typically a rectangle) by a continuous function $S : \mathbb{R}^2 \to \mathbb{R}^3$. Precisely, given a continuous function

$$S \; : \quad (u,v) \in \; [a,b] \times [c,d] \; \mapsto \left\{ \begin{array}{l} x = S_1(u,v) \\ y = S_2(u,v) \\ z = S_3(u,v) \end{array} \right.$$

the parametric surface S will commonly refer to the set $S([a,b] \times [c,d])$.

To plot this parametric surface, we proceed as follows.

1. The parameter intervals $[a,b]$ and $[c,d]$ are sampled by two sequences of m and n increasing values :
   $u = [a = u_0, u_1, \ldots, u_i, \ldots, u_m = b]$
   $v = [c = v_0, v_1, \ldots, v_j, \ldots, v_n = d]$
   `u = np.linspace(a,b,m+1)` and `v = np.linspace(c,d,n+1)`

2. Each of the three functions S_1, S_2, S_3 is evaluated at each position (u_i, v_j) and these values are stored in the 2-dimensional arrays `X`, `Y`, `Z` of dimension $m+1 \times n+1$:
   $$\begin{array}{rcll} X_{i,j} & = & S_1(u_i,v_j), & 0 \le i \le m, \quad 0 \le j \le n \\ Y_{i,j} & = & S_2(u_i,v_j), & 0 \le i \le m, \quad 0 \le j \le n \\ Z_{i,j} & = & S_3(u_i,v_j), & 0 \le i \le m, \quad 0 \le j \le n \end{array}$$

   There are two ways to evaluate the 3 arrays `X`, `Y`, `Z` :

   (1) General case :
   ```
   U,V = np.meshgrid(u,v)
   X = S1(U,V)
   Y = S2(U,V)
   Z = S3(U,V)
   ```

   (2) If each function S_i can be written as the product of a u-function by a v-function, i.e. $S_i(u,v) = f_i(u) \times g_i(v)$:
   ```
   X = np.outer(f1(u),g1(v))
   Y = np.outer(f2(u),g2(v))
   Z = np.outer(f3(u),g3(v))
   ```
   Note : in that case, we say that the parametric surface admits a *separable parametrization.*

3. We then display these three arrays `X`, `Y`, `Z` with the instructions
   `plot_wireframe(X, Y, Z,...)` or `plot_surface(X, Y, Z,...)`

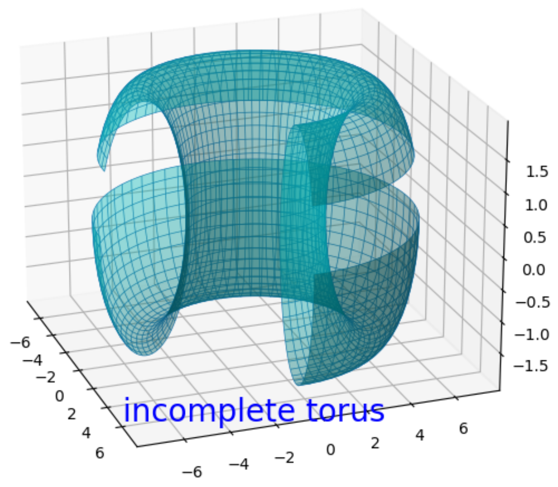
Note that method (1) (with np.meshgrid) leads to three matrices `X,Y,Z` of shape $(n+1 \times m+1)$, while method (2) (with np.outer) leads to matrices of shape $(m+1 \times n+1)$. As an example, we have for matrix `Z` :

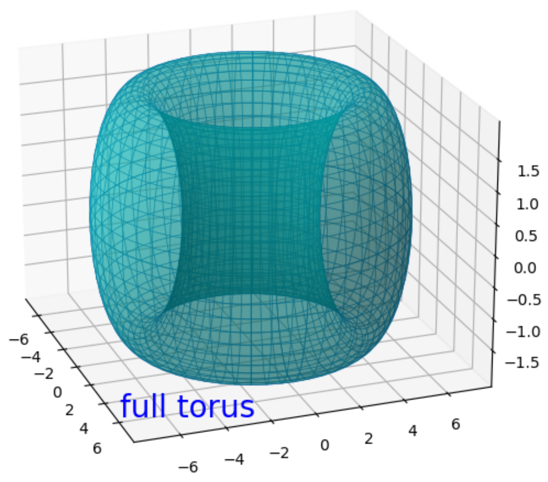
$$\mathtt{Z} = \begin{bmatrix} S_3(u_0,v_0) & S_3(u_1,v_0) & \dots & S_3(u_m,v_0) \\ S_3(u_0,v_1) & & & S_3(u_m,v_1) \\ \vdots & & & \vdots \\ & & & \\ \vdots & & & \vdots \\ S_3(u_0,v_n) & S_3(u_1,v_n) & \dots & S_3(u_m,v_n) \end{bmatrix} \qquad \mathtt{Z} = \begin{bmatrix} S_3(u_0,v_0) & S_3(u_0,v_1) & \dots & S_3(u_0,v_n) \\ S_3(u_1,v_0) & & & S_3(u_1,v_n) \\ \vdots & & & \vdots \\ & & & \\ \vdots & & & \vdots \\ S_3(u_m,v_0) & S_3(u_m,v_1) & \dots & S_3(u_m,v_n) \end{bmatrix}$$

method (1) method (2)

A torus

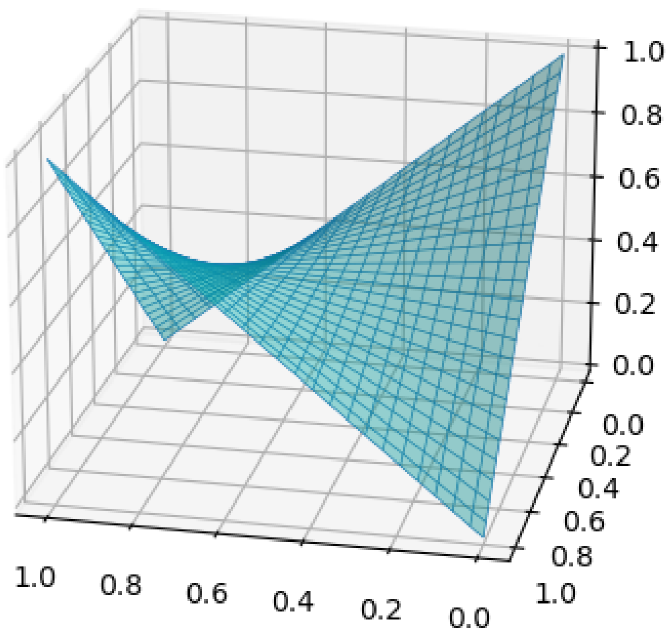
```
"""_______________________________________________
--> Parametric surfaces
    EXAMPLE 1 : torus
    # x = (a + b*np.cos(v)) * np.cos(u)
    # y = (a + b*np.cos(v)) * np.sin(u)
    # z = b*np.sin(v)
___________________________________________________"""
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits import mplot3d

a = 5   # main radius
b = 2   # secondary radius
def f1(u,v):
        return (a + b*np.cos(v)) * np.cos(u)
def f2(u,v):
        return (a + b*np.cos(v)) * np.sin(u)

# u, v parameters for incomplete torus :
u = np.linspace(0, 3*np.pi/2, 200)
v = np.linspace(np.pi/6, 2*np.pi, 200)
    # u = np.linspace(0, 2*np.pi, 200)
    # v = np.linspace(0, 2*np.pi, 200)
U,V = np.meshgrid(u,v)
X = f1(U,V)
Y = f2(U,V)
Z = b*np.sin(V)

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_wireframe(X,Y,Z,rstride=4, cstride=4, linewidth=0.5)
ax.plot_surface(X,Y,Z, rstride=4, cstride=4, color='c', alpha=0.4)

# we add text in the figure, at position (6,-8,-2)
ax.text(6, -8, -2,'incomplete torus', color='b', fontsize=20)
```





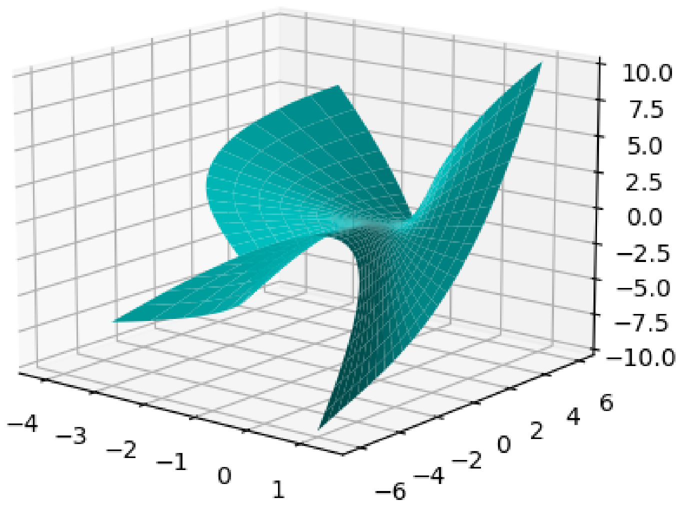
A ruled surface

```
"""_____________________________________________
--> Parametric surfaces
    EXAMPLE 2 : a ruled surface
     [0,1]^2 --> R^3
                 x = (1-u)*(1-v) + u*(1-v)
      (u,v)  --> y = u*(1-v) + u*v
                 z = u*(1-v) + (1-u)*v
_____________________________________________"""
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits import mplot3d

fig = plt.figure()
ax = plt.axes(projection='3d')

u = np.linspace(0, 1, 100)
v = np.linspace(0, 1, 100)
U,V = np.meshgrid(u,v)
X = (1-U)*(1-V) + U*(1-V)
Y = U*(1-V) + U*V
Z = U*(1-V) + (1-U)*V
ax.plot_wireframe(X,Y,Z,rstride=4, cstride=4, linewidth=0.5)
ax.plot_surface(X,Y,Z, rstride=4, cstride=4, color='c', alpha=0.4)
ax.view_init(15, 105)
```



A separable parametric surface

```
"""________________________________________________
--> Separable parametric surfaces
EXAMPLE 3 : a polynomial surface
each component is defined as fi(u) * gi(v)
==> we use np.outer
________________________________________________"""
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits import mplot3d

fig = plt.figure()
ax = plt.axes(projection='3d')

u = np.linspace(-1, 1, 100)
v = np.linspace(-1, 1, 100)
f1 = 1 + u - u**3;
g1 = v - v**2 + v**3
X = np.outer(f1, g1)
f2 = u + u**3
g2 = 1 - v + 2 * v**2 + v**3
Y = np.outer(f2, g2)
f3 = -1 + 2*u + u**2
g3 = 2 * v + 4 *v**2 - v**3
Z = np.outer(f3, g3)
ax.plot_surface(X, Y, Z, rstride=4, cstride=4, color='c')
```

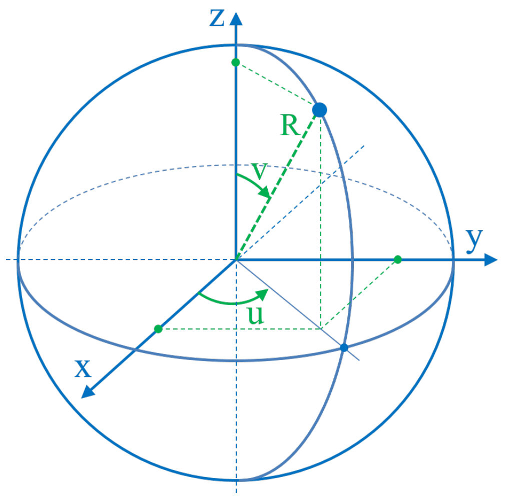


Exercise 0.11 (*incomplete sphere*)

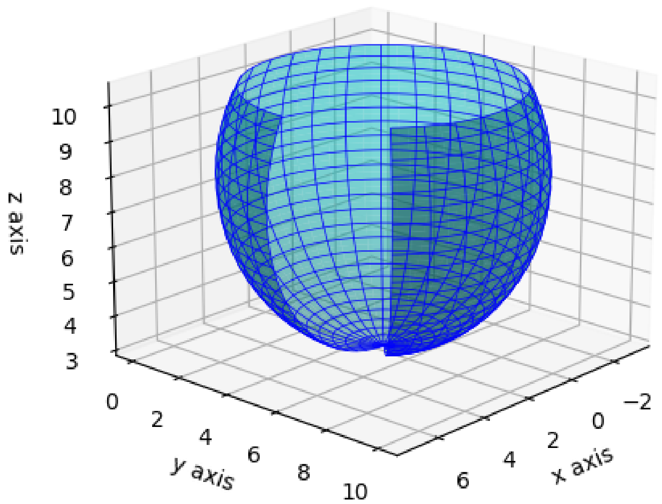
Plot the following incomplete sphere defined by the parametric representation

$$(u,v) \in [\frac{\pi}{4}, 2\,\pi] \times [\frac{\pi}{3}, \pi] \quad \mapsto \quad \begin{cases} x &= Cx + R\,\cos(u)\,\sin(v) \\ y &= Cy + R\,\sin(u)\,\sin(v) \\ z &= Cz + R\,\cos(v) \end{cases}$$

with center $Cx = 2, Cy = 5$, $Cz = 8$ *and radius* $R = 5$.



Spherical coordinates



The incomplete sphere

2.14 Level sets

The function `plt.contour` consists of displaying level sets (level lines), i.e. sections of the surface by parallel planes according to one of the coordinates (for example according to the Z coordinate as in a topographic map). Thus, it is a 2D representation of a 3D surface.
If we consider level sets according to the Z coordinate, the process is as follows.

1. We need first to define the matrix `Z` of the z-coordinates of surface points P_{ij}. Note that this usually requires defining the 3 matrices `X, Y, Z` as in the previous methods.

2. We define a vector of the different `heights` of the `level sets` (the different plane sections) :
   `z_heights` = $[z_0, z_1, \ldots, z_i, \ldots, z_p]$

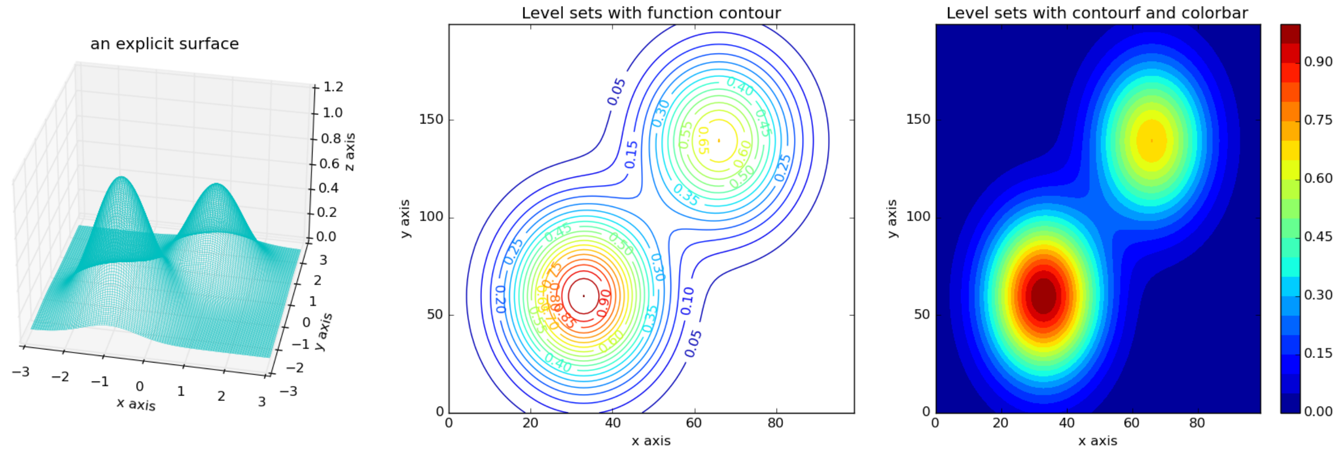
3. Then, the function `plt.contour(Z, z_heights)` displays each level set.
   The function `plt.contour()` returns a variable (e.g. `cont`) that contains all the information about these level sets, allowing the next step ...

4. Finally, for example, the function `plt.clabel(cont, fmt='%.2f')` labels these level sets according to the format `%.2f`

Level sets : sum of 2 Gaussians

```
"""_______________________________________________
--> LEVEL SET with function contour
EXAMPLE : sum of 2 Gaussians
_______________________________________________"""
import matplotlib.pyplot as plt
import numpy as np

# Explicit surface : sum of two Gaussians
def f(x,y) :
y1 =      np.exp(-(x+1)**2 - (y+1)**2)
y2 = 0.7*np.exp(-(x-1)**2 - (y-1)**2)
return y1 + y2

fig = plt.figure(1,figsize=(22,6))
ax1 = fig.add_subplot(131, projection='3d')
ax1.set_xlabel('x axis')
ax1.set_ylabel('y axis')
ax1.set_zlabel('z axis')
ax1.set_title('an explicit surface')

# parametric domain [a,b]x[c,d]
a = -3; b = 3
c = -2.5; d = 2.5
# the surface
u = np.linspace(a,b,100)
v = np.linspace(c,d,200)
X,Y = np.meshgrid(u,v)
Z = f(X,Y)
ax1.plot_wireframe(X, Y, Z, color='c', linewidth=0.3)

# Level set with contour
ax2 = fig.add_subplot(132)
ax2.set_xlabel('x axis')
ax2.set_ylabel('y axis')
ax2.set_title('Level sets with function contour')
# heights of the level sets (surface sections) :
heights = np.arange(0, 1.05, 0.05)
cont = plt.contour(Z, heights) # draws level sets
plt.clabel(cont, fmt='%.2f')   # label of level sets (format fmt)
```

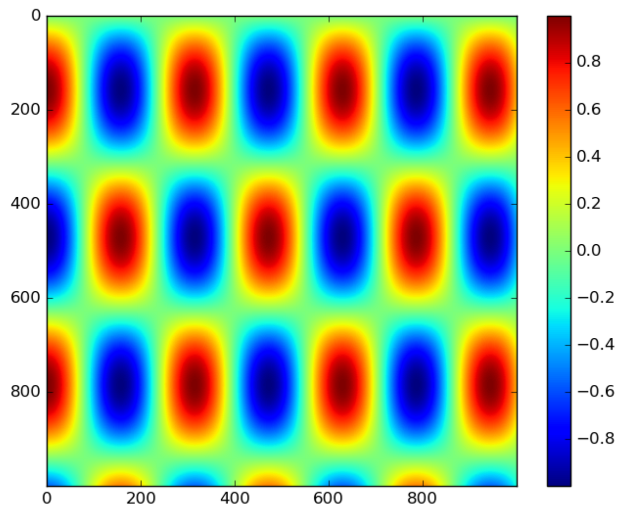
```
# Level set with contourf and colorbar
ax3 = fig.add_subplot(133)
ax3.set_xlabel('x axis')
ax3.set_ylabel('y axis')
ax3.set_title('Level sets with contourf and colorbar')
# heights of the level sets (surface sections) :
heights = np.arange(0, 1.05, 0.05)
cont = plt.contourf(Z, heights) # draw filled level sets
plt.colorbar()                  # like a "continous" legend
```



Colorbar

The function colorbar indicates the color scale, and is drawn into a dedicated axis.
A first example is given above in the right figure. We now give a second example.

```
"""____________________________________________
—> Colorbar and imshow
____________________________________________"""

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 1000)
u = np.sin(x)
v = np.cos(2*x)
z = np.outer(u,v)

plt.imshow(z)  # z is plotted as an image
plt.colorbar() # like a legend
```



Exercise 0.12 (*the Monkey Saddle*)

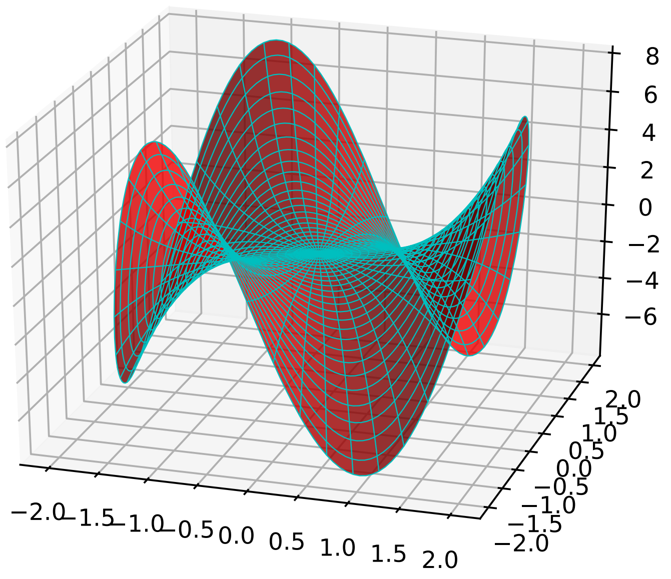
Consider the Monkey saddle defined by the parametric representation

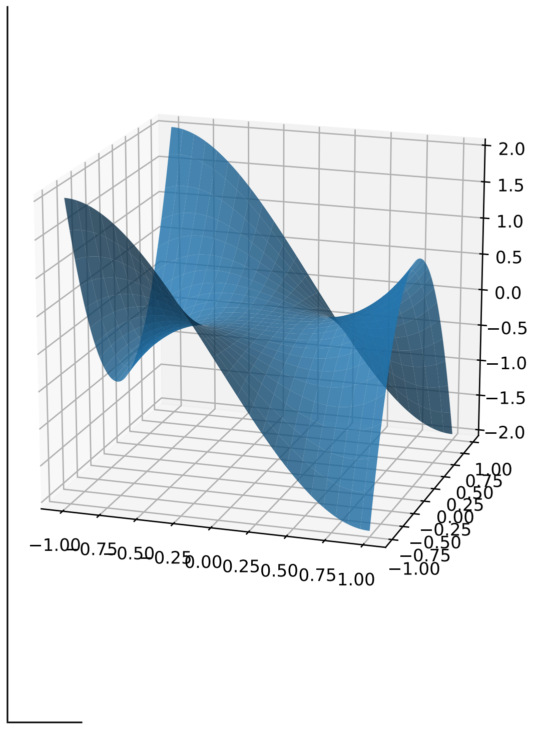
$$(u,v) \in [0,2] \times [0,2\,\pi] \quad \mapsto \quad \left\{ \begin{array}{rcl} x & = & u\,\cos(v) \\ y & = & u\,\sin(v) \\ z & = & u^3\,\cos(3v) \end{array} \right.$$

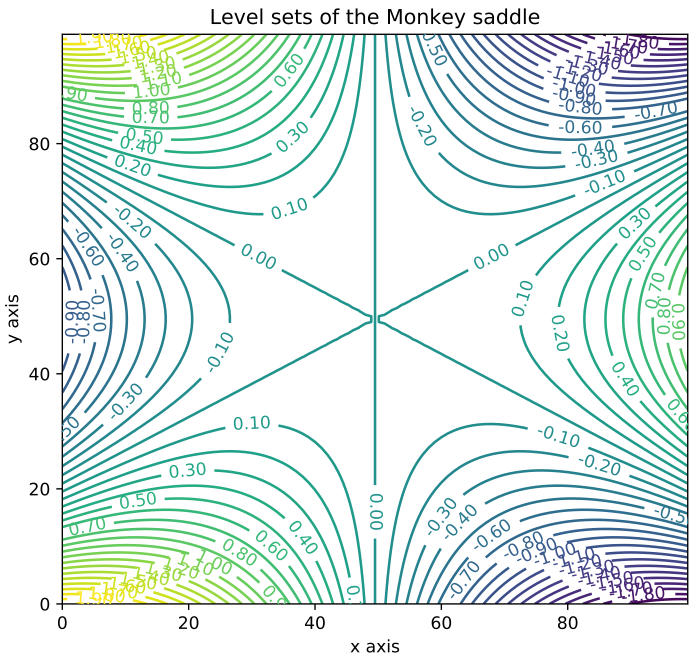
and admitting also the explicit equation : $z = x^3 - 3\,x\,y^2$

1. *Plot this surface using its parametric representation.*

2. *In the same figure with two subplots,*
   a) plot the surface using its explicit equation,
   b) plot its level sets according to the z*-coordinate.*







2.15 Remark : save the current figure

The current figure can be saved in the current folder with the instruction

```
plt.savefig("TheNameOfMyFigure.png")
```

3 Appendix

3.1 Display of floating numbers

```
# f61DisplayFloats.py
"""
DISPLAY of floating numbers
"""
import numpy as np
v = np.pi
format(v, '.12g')   # gives 12 significant digits, return a string
#  '3.14159265359'
print('%.12g' % v)  # similar display but type return = "NoneType"

format(v, '.10f')   # gives 10 digits after the point, return a string
#  '3.1415926536'
print('%.10f' % v)  # similar display but type return = "NoneType"

print("%15.8f" % v) # 15 characters, 8 binary digits after the point
#      3.14159265
# so 5 blanks at the beginning
print("%15.8e" % v) # scientic format : 15 characters all together
#  3.14159265e+00
# just 1 blank at the beginning due to
# the characters e+00

# More about display and precision of floating numbers :
b = 10**100
print("%.17f" %b)
c = 0.3
print("%10.7f"  %c)
print("%20.17f" %c)
print("%40.37f" %c)
```

3.2 Files

We will use the file *f65Files.py*

- Change directory

```
# File f65Files.py
"""________________________________________________
Change directory
________________________________________________"""
"""
Remark : we can use directly some windows commands, such as:
pwd   --> print working directory
ls    --> list directory contents
Directory paths :
Windows uses backslashes (\) to separate directories in file paths:
'C:\Users\Luke\Documents\...'
```

```
whereas Python uses two backslash (\\)
'C:\\Users\\Luke\\Documents\\...'
Here, we can use two backslash (\\) or one slash (/)
"""
import os   # import operating system

os.getcwd() # print the current working directory
# with two backslash (\\): similar to pwd
os.chdir("C:\\Users\\Luke\\Documents\\Luc") # change the directory
os.getcwd()

# --> now, we can use one slash to change the directory:
myPath1 = "C:/Users/Luke/Documents/Luc"
myWorkingPath = myPath1 +\
"/NumericalMaths/01IntroPython"
os.chdir(myWorkingPath)

# --> relative path:
os.chdir('../../NumericalIntegration') # 2 parent directories, 1 child
os.getcwd()
```

- Opening and closure of an existing file

```
"""_______________________________________________
Opening and closure of an existing file
___________________________________________________"""
os.chdir(myWorkingPath)              # contains our testing files
my_file = open("file0.txt", "r")     # opening in mode "r" (read)
type(my_file)
content = my_file.read()             # read the content 'my_file'
type(content)
print(content)
my_file.close()                      # close the file
```

- Modifying (writing in) or Creating an existing file

```
"""_______________________________________________
Modifying (writing in) an existing file
Creating a file
- First, we need to open the existing file
mode "w" overwrites the possible content of the file
mode "a" (append) adds what is written at the end of the file
(\n to skip a line)
- If the file doesn't exist, it will be created
___________________________________________________"""

my_file = open("file1.txt", "w") # Oops, I overwrite everything !
my_file.write("writing in a file via Python !!!")
# return the number of characters
my_file.close()
```

```
my_file = open("file1.txt", "w") # again, we overwrite everything
string = "I am going to write in a file... "
my_file.write(string)
my_file.close()

my_file = open("file1.txt", "a") # to write at the end of the file
my_file.write("\nDoes it work ?\n--> YES, IT WORKS !!")
my_file.close()
# Repeat the last 3 instructions several times
# and then check the text file
```