ICT for Health
Python

Monica Visintin

Politecnico di Torino

2023/24

Table of Contents

Table of Contents

Python [1]

- Python is a high level programming language, somehow similar to Matlab since it allows to add two vectors/arrays `a` and `b` by simply writing `c=a+b`.
- Like Matlab, it is not necessary to declare the variables (which is an advantage and a disadvantage...).
- Python is free (Matlab costs around 10 keuros per year)
- Python scripts run faster than corresponding Matlab scripts, but slightly slower than corresponding C programs, but not all agree on this.
- Python uses extensions/libraries that can be imported, if necessary. For example NumPy is required for working with vectors and matrices, or for using the FFT, matplotlib is an extension for generating plots with a syntax similar to that of Matlab. We will use **Numpy, Matplotlib, Pandas, Scikit-Learn (sklearn)**.

Python [2]

- You can **use/run Python**
  1. Interactively (like Matlab): you simply enter the command `$python` in a console (not so nice, but useful).
  2. By running a Python script `myscript.py` from a console with the command line `$ python3 myscript.py`
  3. Through the *jupyter notebook*, which allows you to write and comment your code and execute portions of it or the entire code.
  4. Through an IDE (Integrated Development Environment) like *Spyder* or *PyCharm Edu*, which allows you to write your code, automatically check the grammar and "good writing", and run portions of code or the entire code.

  We suggest to use Spyder (it is simpler), but you can use whatever you want.

Python [3]

- You can **write a Python script** with
  1. A normal editor (but then you can only run the entire script, and you must use the command `$ python3 myscript.py` in a shell).
  2. An IDE like *Spyder* or *PyCharm Edu*: the advantage is that you can autocomplete your code, get help, inspect your variables and objects.
  3. *Jupyter notebook* (you have autocompletion and help, you can run the entire script, or only portions of it; numerical results, pictures etc are embedded in the file, and, even if you close your session and you reopen it later, you find not only the script, but also the figures and the results).
- There are **two versions of Python**: Python 2 (old version) and **Python 3** (new version, partially incompatible with Python 2). In the future, only Python 3 will be maintained. We will use Python 3. As an example, if you want to print the value of variable `x`, in Python 2 the code line is
  `print x`
  In Python 3 it is `print(x)`.

Python [4]

- **Linux distributions** include Python (it might be Python2 or Python3 or both). If you have Python2, you can install Python3; if you have both installed and you want to use Python3, the command in the shell must be `$ python3 myscript.py`, otherwise Python2 will be used. Pycharm Edu, Spyder and Jupyter notebook ask you which version of Python you want to use.
- Python installation. Installation might be a complex task. One possibility is to first install **Anaconda**, then use Anaconda to install Python and all its extensions/libraries; Anaconda allows you to have different environments and different settings for each environment (for example you can use Python2 in one environment and Python3 in another). Another possibility is to directly install Python3 and use pip3 to add the needed extensions. Search Google to find out how to install Python3 on your specific operating system.
- There is an **online** version of Python provided by Google. If you have a Google email/account, you can use your browser at the link `https://colab.research.google.com/` to generate and run your Python code. Note that in this case you use the CPU/GPU of Google servers, not of your PC. Colab basically is a cloud jupyter version. The main Python libraries are already available in Colab, you do not have to install anything on your PC.

Python [5]

- Python has many libraries, developed by engineers, mathematicians, statisticians, physicists, medical doctors, economists. It is impossible to teach everything. Moreover Python is changing in time: what is available now might not be available any more next year. In this course we will only see and use what is strictly necessary to solve the lab problems with a ”hand on” approach. In this set of slides we will see some basics on Python, Numpy, Matplotlib; later something about Pandas and Scikit-learn. You are invited to search the web to deepen your knowledge. This course is NOT a course that teaches you Python, we will simply use it.
- If you do not know how to do something you want to do, search the web or ask **ChatGPT**.

Table of Contents

Example of a Python-3 script [1]

Goal:

1. We want to solve the minimization program

$$\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{A}\mathbf{w}\|^2$$

   where $\mathbf{A} \in \mathbb{R}^{N_p \times N_f}$, $\mathbf{w} \in \mathbb{R}^{N_f \times 1}$, $\mathbf{y} \in \mathbb{R}^{N_p \times 1}$

2. We write Python-3 class that implements Linear Least Squares (LLS) and a main script to test the algorithm.

3. The LLS gives the optimum solution as

$$\hat{\mathbf{w}} = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T\mathbf{y}$$

Example of a Python-3 script [2]

- **Comments** in Python are preceded by #. If a comment takes more than one line, you must use # at the beginning of each line. In Spyder, the shortcut to comment a selected range of rows is CTRL 1.
- Python uses **indentations** (tabs) to separate "sections" and identify the part of code that must be run inside a **for** loop or an **if** statement. If blanks are not correctly placed, you get an error and the code does not run. Spyder automatically sets the required spaces, or gives you a warning as you type if they are not correct.

Example of a Python-3 script [3]

- In the problem we have vectors and matrices, that are managed by the Python extension/library called **Numpy**. The first line of the Python script is

```
1 import numpy as np
```

  to import the Numpy library. If you get an error, you probably forgot to download and install Numpy, do it!
- In Numpy vectors and matrices are called Ndarrays (Nd stands for N-dimension). We will focus on one dimension arrays (i.e. vectors) and two dimension arrays (i.e. matrices), but Numpy manages arrays with more than two dimensions.

Example of a Python-3 script [4]

- How do you create an Ndarray?

```
y=np.ones((Np,1),dtype=float)# column vector of Np floats all equal to 1
y=np.zeros((1,Np),dtype=int)# row vector of Np integers all equal to 0
A=np.eye(4)# 4x4 identity matrix
y=np.array([1,2,3])# vector (neither column nor row) #with shape (3,) and values 1,2,3
A=np.array([[1,2,3],[4,5,6]])# 2x3 matrix
z=np.arange(5)# z=[0,1,2,3,4], 5 elements starting from 0
z=np.arange(5,7,dtype=float)# z=[5.0,6.0]
```

Example of a Python-3 script [5]

- Operations on Ndarrays:
  - If `A` and `B` have the same size, `A+B` is the **elementwise sum** of the two Ndarrays.
  - If `A` and `B` have the same size, `A*B` is the **elementwise product** of the two Ndarrays (WARNING: in Matlab this corresponds to `A.*B`).
  - If `A` has shape (N, M) and `B` has shape (M, P), `A@B` is the product of the two Ndarrays, with shape (N, P) (WARNING 1: operator `@` have been only recently included, it is equivalent to `np.matmul(A,B)`) (WARNING 2: in Matlab this corresponds to `A*B`).
  - If `A` is an Ndarray, `M,N=A.shape` gives the shape of the Ndarray: `M` rows and `N` columns.
  - If `A` is an Ndarray with shape `(M1,N1)` and `(M2,N2)` is another possible shape (`M2` and `N2` integers and `M2*N2=M1*N1`), then `B=np.reshape(A,(M2,N2))` gives `B` with the elements of `A` and shape `M2,N2` (check how this is done: by rows or by columns?).
- **Indexing** (i.e. reading/writing some elements of an Ndarray): if `a` is a 1-dimensional Ndarray, then
  `a[0]` is the first element of `a`
  `a[1]` is the second element of `a`
  `a[-1]` is the last element of `a`
  If b is a 2-dimensional Ndarray, then
  `b[0,0]` is the element in the first row, first column

Example of a Python-3 script [6]

- Slicing:

```
a=A[:,0]# A matrix, a is the first column
a=A[1,:]# A matrix, a is the second row
a=A[1:3,0:4]# a is a submatrix, rows from 2 to 4, cols from 1 to 5
```

Example of a Python-3 script [7]

- Useful functions/methods available in Numpy:

```
y=np.sort(x)# y is the sorted version of x (ascending)
x.sort()# in place sorting (ascending)
n=np.argsort(x)# indexes of the sorted x; x[n] is sorted
n=np.where(x>2)# indexes where x is larger than 2
y=x[n]# y is a subset of x, picking only values larger than 2
```

- Sometimes you get arrays with size `(N,)`, that are nor rows nor columns; to change the array into a **column vector** write `a.shape=(N,1)`, to change the array into a row vector write `a.shape=(1,N)`

Example of a Python-3 script [8]

- Numpy has sub-libraries: `linalg` for linear algebra, `random` to generate samples of random variables, etc.
- Useful functions available in the `linalg` sub-library:

```
d=np.linalg.norm(x)# d is the norm of 1D array x
d=np.linalg.det(A)# d is the determinant of matrix A
lam, U = np.linalg.eig(A)# lam stores the eigenvalues of matrix A
# U stores the eigenvectors of matrix A
A=np.linalg.inv(B)# A is the inverse of B (AB=I)
```

Example of a Python-3 script [9]

- Useful functions available in the `random` sub-library:

```
1  np.random.seed(71)# sets the seed used to generate
2  #the random variables to 71
3  X=np.random.rand(M,N)# generates a matrix MxN with random values
4  #unif. distr. in [0,1)
5  X=np.random.randn(M,N)# generates a matrix MxN with Gaussian
6  #distrib. random values (mean 0, var 1)
7  X=np.random.randint(K, size=(M,N))# generates a matrix MxN with
8  #integer random values unif distr. in [0,K-1]
9  np.random.shuffle(x)#randomly permutes/shuffles the elements of
10 #x (in place)
```

Example of a Python-3 script [10]

- Now that we know the basics of Numpy, let us try and write the code to implement Linear Least Squares. We need a matrix `A` and a vector `y`. We generate a **fake** case, in which we know $\mathbf{w}$ (random), we know $\mathbf{A}$ and we generate $\mathbf{y} = \mathbf{A}\mathbf{w}$. Then we pretend we do not know $\mathbf{w}$ and we use the LLS formula to find it; if the LLS result $\hat{\mathbf{w}}$ is equal to $\mathbf{w}$ each time we run the code (and each time the random matrices/vectors change), then the algorithm is correctly implemented. Note that the square error $\|y - A\hat{\mathbf{w}}\|^2$ should be close to zero if $\hat{\mathbf{w}} = \mathbf{w}$ (numerical errors occur).

```
2 Np=5 # Number of rows
3 Nf=4 # Number of columns
4 A=np.random.randn(Np,Nf) # Gauss. random matrix A, shape (Np,Nf)
5 w=np.random.randn(Nf) # Gauss. random vector w_id, shape (Nf,)
6 y=A@w  # shape (Np,)
```

Example of a Python-3 script [11]

- Let us now apply the linear least square method:

$$\hat{\mathbf{w}} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{y}$$

```
7  ATA=A.T@A # generate AT*A
8  ATAinv=np.linalg.inv(ATA) # generate (AT*A)**(-1)
9  ATy=A.T@y # generate AT*y
10 w_hat=ATAinv@ATy # generate (AT*A)**(-1)*AT*y
```

or you can use `np.linalg.pinv` which directly generates the pseudo-inverse:

```
7  w_hat=np.dot(np.linalg.pinv(A),y) # generate w
```

Carefully search the web when you have to do an operation: maybe it is already included in a Python library and you simply need to call it.

Example of a Python-3 script [12]

- Now we want to print the result:

```
8 print(w_hat)
```

If you want something nicer, you can use this code:

```
8  print('Result obtained by applying the LLS method:')
9  print('estimated vector w_hat is: ',w_hat)
10 print('true vector w is: ',w)
11 e=y-A@w_hat # error vector, shape (Np,)
12 print('square error |y-A*what|^2: ',np.linalg.norm(e)**2)
```

Example of a Python-3 script [13]

- Now we want to plot the result. We need Python library matplotlib

```
13 import matplotlib.pyplot as plt
14 plt.figure()# create a new figure
15 plt.plot(w_hat, label='w_hat')# plot w_hat with a line
16 plt.plot(w,'o', label='w')# plot w with markers (circle)
17 plt.xlabel('n')# label on the x-axis
18 plt.ylabel('w(n)')# label on the y-axis
19 plt.legend()# show the legend
20 plt.grid() # set the grid
21 plt.title('Comparison between w and w_hat')# set the title
22 plt.show()# show the figure on the screen
```

Search `https://matplotlib.org/` for the details about plots (really many possibilities). WARNING: whatever is the plot you generate, remember to specify the labels on the x and y axes. You will get a lower grade if you forget this.

Table of Contents

Object Oriented Programming (OOP)

Very briefly: Python allows to define "**class**es" for which you can define "**method**s"; once you "**instantiate**" an "**object**" belonging to a given "class", the object "**inherits**" the "methods" of that "class". An example will show the meaning of this.

What do we want to do in the end?

- We want to use **more than one optimization technique** apart from LLS, and **for all** these techniques we want to **print and plot** the resulting solution vector `w_hat`. We want the printing and plotting functions to be in common to all the optimization techniques: we define them once, we use them in all the cases.
- We first define a **class** `SolveMinProbl` in which we define the methods/functions to print and to plot vector `w_hat` and maybe other methods. Then we define the classes `SolveLLS`, `SolveGrad`, etc as **belonging to class** `SolveMinProbl` so that all **inherit** the methods to plot and to print `w_hat` (so you do not have to repeat the methods for each of the classes). Classes `SolveLLS`, `SolveGrad`, etc. are actually called **subclasses** of `SolveMinProbl`.
- We start by writing in a separate file `minimization.py` the lines in the next slide.

Class SolveMinProbl (1)

```
import numpy as np
import matplotlib.pyplot as plt
class SolveMinProbl:
    def __init__(self,y=np.ones((3,)),A=np.eye(3)): #initialization
        self.matr=A # matrix A
        self.Np=y.shape[0] # number of rows
        self.Nf=A.shape[1] # number of columns
        self.vect=y # column vector y
        self.sol=np.zeros((self.Nf,),dtype=float) # column vector w_hat
        return
    def plot_w_hat(self,title='Solution'): # method to plot
        w_hat=self.sol
        n=np.arange(self.Nf)
        plt.figure()
            plt.plot(n,w)
        plt.xlabel('n')
        plt.ylabel('w_hat(n)')
        plt.title(title)
        plt.grid()
        plt.show()
        return
    def print_result(self,title): # method to print the result
        print(title,': ')
        print('the optimum weight vector is: ')
        print(self.sol)
        return
```

Class SolveLLS (2)

```
27 class SolveLLS(SolveMinProbl):# class SolveLLS belongs to class SolveMinProb
28     """
29     Comments...
30     """
31     def run(self):
32         A=self.matr
33         y=self.vect
34         w_hat=np.linalg.inv(A.T@A)@(A.T@y)
35         self.sol=w_hat
36         self.min=np.linalg.norm(A@w_hat-y)**2
37         return
```

Note that initialization is inherited from class `SolveMinProbl` and therefore the construct `def __init__(self,...):` is missing. On the other side we introduce method `run` (without parameters) that sets the values of `self.sol` (`w_hat`) and `self.min` (the error square norm).

Use of the defined classes in the main script [1]

- We have just one file `minimization.py` that already contains classes `SolveMinProbl` and `SolveLLS`, we have now to tell Python what it has to do if you write in the shell `$ python3 minimization.py`: we add (at the bottom of file `minimization.py`) the following lines:

```
38 if __name__ == "__main__":
39     Np=100 #number of rows
40     Nf=4 #number of columns
41     A=np.random.randn(Np,Nf) # matrix/Ndarray A
42     w=np.random.randn(Nf,) # true vector w
43     y=A@w# column vector y
44     m=SolveLLS(y,A) # instantiate the object
45     m.run() #run LLS
46     m.print_result('LLS') # print the results (inherited method)
47     m.plot_w_hat('LLS')# plot w_hat (inherited method)
```

This solution thus only requires one file, in which you write the entire code (both the main and the called classes/methods).

Use of the defined classes in the main script [2]

- You might want to have one file with the classes/methods (like a library, in folder `./sub`) and a separate file for the main script. You write another Python file `lab0.py`, the main file, that uses `SolveLLS` but you have to import file `./sub/minimization.py` that includes `SolveLLS`. Your file `lab0.py` will be:

```
1  import sub.minimization as mymin # import (entirely) file minimization.py
2  import numpy as np
3  Np=100 #number of rows
4  Nf=4 #number of columns
5  A=np.random.randn(Np,Nf) # matrix/Ndarray A
6  w=np.random.randn(Nf,) # true vector w
7  y=A@w# column vector y
8  m=mymin.SolveLLS(y,A) # instantiate LLS. Note mymin in front of SolveLLS
9  m.run() #run LLS
10 m.print_result('LLS') # print the results (inherited method)
11 m.plot_w_hat('LLS')# plot w_hat (inherited method)
```

Use of the defined classes in the main script [3]

Note that, even if you import `minimization.py` which includes the lines `if __name__ == "__main__":` etc., these last lines will not be executed (they are executed only if you write in the shell `$ python3 minimization.py`), and therefore you do not have to remove these last lines from your file `minimization.py`.
The last part of texttttminimization.py with the lines `if __name__ == "__main__":` etc. is typically used to **test the correct working of a library**: if you can run successfully `minimization.py` (alone) the classes defined in it are correct.

Class SolveGrad [1]

Now we add the class that implements the gradient algorithm:

```
37 class SolveGrad(SolveMinProb):
38     """ Comments...
39     """
40
41     def run(self,gamma=1e-3,Nit=100):
42         self.err=np.zeros((Nit,2),dtype=float)
43         self.gamma=gamma
44         self.Nit=Nit
45         A=self.matr
46         y=self.vect
47         w=np.random.rand(self.Nf,)# random #initialization of the weight vector
48         for it in range(Nit):
49         grad=2*A.T@(A@w-y)
50         w=w-gamma*grad
51         self.err[it,0]=it
52         self.err[it,1]=np.linalg.norm(A@w-y)**2
53         self.sol=w
54         self.min=self.err[it,1]
```

Class SolveGrad [2]

- Class `SolveGrad` must be added to file `minimization.py`.
- If everything is correct, the error $\|\mathbf{y} - \mathbf{A}\mathbf{w}(k)\|^2$ should decrease as the iteration step k increases, and it is convenient to check it is like that, because this is a way to understand if the learning coefficient is too small or too large.

Class SolveGrad [3]

- Then we want to plot the error square norm versus k. Since also other algorithms iteratively estimate the solution and show a decreasing error, it is convenient to add method `plot_err` in class `SolveMinProb`:

```
27      def plot_err(self,title='Square error',logy=0,logx=0):
28          """            """
29          err=self.err
30          plt.figure()
31          if (logy==0) & (logx==0):
32             plt.plot(err[:,0],err[:,1])
33          if (logy==1) & (logx==0):
34              plt.semilogy(err[:,0],err[:,1])
35          if (logy==0) & (logx==1):
36              plt.semilogx(err[:,0],err[:,1])
37          if (logy==1) & (logx==1):
38              plt.loglog(err[:,0],err[:,1])
39          plt.xlabel('n')
40          plt.ylabel('e(n)')
41          plt.title(title)
42          plt.margins(0.01,0.1)# leave some space
43          plt.grid()
44          plt.show()
45          return
```

Class SolveGrad [4]

- In `lab0.py` we add the following lines

```
12 Nit=1000
13 gamma=1e-5
14 g=mymin.SolveGrad(y,A)
15 g.run(gamma,Nit)
16 g.print_result('Gradient algorithm') # inherited method
17 logx=0
18 logy=1
19 g.plot_err('Gradient algorithm: square error',logy,logx) # inherited method
20 g.plot_w_hat('Gradient algorithm') # inherited method
```

Table of Contents

Data types in Python

In the previous scripts we have used variables, strings, arrays, Ndarrays. Of course other data types exist, which might be useful in the future:

- lists
- tuples
- dictionaries

List

Example:
`list=['akn',27.4,3]`
As you see, a list can contain different kinds of variables (in the example: string, float number, integer number), but is can also contain lists (you have a list of lists), Ndarrays (list of Ndarrays, maybe with different shapes), etc.
`list[0]`
is the element in position 0 in the list; for the example above, `list[0]` is the string `'akn'`.
You can initialize a list as empty: `list=[]`
and then append items to it:
`list.append('abcd')`
The difference between a list and an Ndarray is that an Ndarray should contain the same type of variables in all its elements (i.e. only float numbers, or only strings, but not strings and numbers).
Check the web if you want to remove specific elements from a list, or add an element in a specific position.

Tuple

Example:
`tup=('akn',27.4,3)`
You use parentheses (round brackets) instead of brackets, but it seems that the tuple is equivalent to a list. Actually the difference is that, once you have generated a tuple, you cannot modify its content, i.e. it is not possible to write `tup[2]=4`; the tuple is **immutable**. On the contrary, you can read the content of a tuple: `a=tup[2]`.

Dictionary

Eaxmple 1:

```
dict = { }
dict['one'] = "This is one"
dict[2] = "This is two"
```

Eaxmple 2:

```
dict = {
'one' :  "This is one",
2 :  "This is two"}
```

In this example the keys of the dictionary are 'one' and 2 (you get them by writing `dict.keys`); the corresponding values are "This is one" and "This is two" (you get them by writing `dict.values`). `a=dict['one']` makes `a` equal to "This is one".

Table of Contents

Ndarrays in NumPy [1]

- In Python, a **list** is an array of numbers or characters or other more complex objects; you cannot perform math operations on lists; the objects in the list can be arrays, not necessarily all with the same length
- In NumPy, an **Ndarray** (**N-d**imensional **array**) is an `N`-dimensional array (a linear algebra vector if `N`=1, a linear algebra matrix if `N`=2, stacked matrices if `N`=3, etc.), and you can perform mathematical operations on these structures. A 2-dimensional Ndarray has `Nr` rows and `Nc` columns, and all the columns have `Nr` elements, all the rows have `Nc` elements (lists do not require the same lengths, on the contrary).
- If `x` is a NumPy Ndarray, `x.ndim` is its dimension (1 for linear algebra vectors, 2 for linear algebra matrices, etc), `x.shape` is `(Nr,Nc)` for a 2-dimensional Ndarray, `x.size` is the number of elements of the Ndarray (for dimension 2, it is equal to `Nc*Nr`), `x.dtype` is the type of the elements (integers, float, etc). Moreover `type(x)` gives you the answer `'NumPy.ndarray'`.

Ndarrays in NumPy [2]

- If, in Python, you write
  `a=[1, 2, 3, 4]`
  then `a` is a **list**. If you want an **Ndarray**, you must write
  `x=np.array([1, 2, 3, 4])`
  (essentially, you ask Python to change the list [1,2,3,4] into a NumPy Ndarray, that is why you need the square brackets within the round brackets).

Ndarrays in NumPy [3]

- .

VERY IMPORTANT:

If in Python `A` is an Ndarray (assume it has dimension 1, for simplicity) and you write `B=A`, then **the address of the first memory cell of the Ndarray `A` is copied in `B`** (Python does not copy matrix `A` into a new portion of memory to create the new matrix `B`); this means that, if you write:

```
A=B
B[1]=3
```

then `A[1]` is also equal to 3. This property may be useful when you want to change a portion of an Ndarray; assume for example that `A` is a (3,5) Ndarray and you want the second column to have all ones, then you can write

```
s=A[:,1]
s[:]=1
```

or you can write

```
A[:,1]=1
```

When the Ndarray has many dimensions, the possibility of writing simply `s` instead of `A[:,1]` might be useful.

Ndarrays in NumPy [4]

- If you want to **copy** Ndarray A into a new Ndarray B (i.e. a new portion of memory), then you must write
  `B=A.copy()`
  or `B=1*A`
- To create an Ndarray with **linearly increasing values**, you can use:
  `c1=np.arange(i1,i2,istep)# from i1 to i2 with step istep`
  `c2=np.linspace(i1,i2,Nitem)# Nitem values from i1 to i2, included`
- Other operations: if `a` is an Ndarray with shape (6,2), then
  `a.reshape(3,4)` is a new Ndarray with **shape** (3,4) (by rows, but check!)
  `a.ravel()` is a new Ndarray with shape (1,12) (by rows, but check!)
  `a.resize(3,4)` changes `a` (it does not generate a new Ndarray in new memory)
- If `A` is a NumPy Ndarray with 10 rows and 8 columns and you want to **remove the fourth column**, you can write
  `Ad=np.delete(A,3,1)`
  If you want to remove the fourth row, you can write
  `Ad=np.delete(A,3,0)`

Table of Contents

To do (before next lab)

- Write the Python code in your file/files (you can copy and paste from this pdf file) and run it. Check that you have a correct Python installation, with all the required libraries.
- **Set the seed** so that every time you run your script, the random matrices/vectors are always equal.
- **Compare the results you get with the LLS and the gradient algorithm**. Play with the **learning coefficient**: make it very big, make it very small.
- **Change the stopping condition for the gradient algorithm**.
- Answer these questions:
  1. From a theoretical point of view, should the solution you find with the gradient algorithm be equal to the solution you find with LLS?
  2. From a theoretical point of view, should the solution you find with the steepest descent algorithm be equal to the solution you find with LLS?
  3. Do you get numerically equal solutions with the three techniques? why?
- This part will be used to solve the regression problem of Lab 1 (in two weeks).