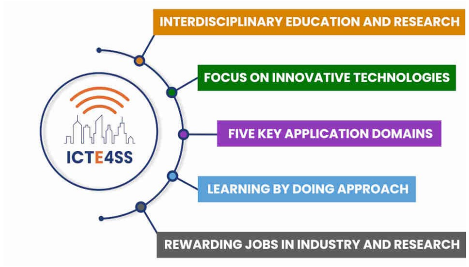
ICT for Health
Laboratory # 0: Python

Prof. Monica Visintin - Politecnico di Torino


ICTE4SS
INTERDISCIPLINARY EDUCATION AND RESEARCH
FOCUS ON INNOVATIVE TECHNOLOGIES
FIVE KEY APPLICATION DOMAINS
LEARNING BY DOING APPROACH
REWARDING JOBS IN INDUSTRY AND RESEARCH

2025/26

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**.

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). , It does not allow you to inspect variables unless you print them, and the autocompletion of commands is not so nice.
- There are **two versions of Python**: Python 2 (old version) and **Python 3**. We will use Python3.
- **Linux distributions** include Python.You simply need to install Pycharm Edu or Spyder or Jupyter notebook.
- Installation on Windows PC: follow the instructions given in the separate file `short_guide_installation.pdf` in DropBox. For Apple computers, search the web.

Python [4]

- 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. Loading files with data to be processed might be complicated.
- 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** or **Gemini**.

Table of Contents

The problem [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. The Linear Least Squares (LLS) gives the optimum solution as

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

3. We need to write the Python script that implements Linear Least Squares.

The initial script [1]

- **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.

The initial script [2]

- 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

  ```
  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.

The initial script [3]

- 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]
```

The initial script [4]

- 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

The initial script [5]

- 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
```

The initial script [6]

- .

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.

The initial script [7]

- 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`

The initial script [8]

- 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)`

The initial script [9]

- 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)
```

The initial script [10]

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

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

The initial script [11]

- 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 $\|\mathbf{y} - \mathbf{A}\hat{\mathbf{w}}\|^2$ should be close to zero if $\hat{\mathbf{w}} = \mathbf{w}$ (numerical errors occur); if it does not, then there is an error somewhere.

```python
Np = 5 # number of rows
Nf = 4 # number of columns
A = np.random.randn(Np,Nf) # matrix with data
w = np.random.randn(Nf,1) # vector with weights
y = A@w # shape of y is (Np,1)
```

The initial script [12]

- Let us now apply the linear least square method:

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

```
ATA = A.T@A # A transpose times A
ATAinv = np.linalg.inv(ATA) # inverse of A transpose A
ATy = A.T@y # A transpose times y
w_hat = ATAinv@ATy # LLS found solution
```

or you can write, in just one line,

```
w_hat = np.linalg.inv(A.T@A)@A.T@y # LLS found solution
```

or you can use `np.linalg.pinv` which directly generates the pseudo-inverse (i.e. $\left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T$):

```
w_hat=np.linalg.pinv(A)@y # LLS found solution
```

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. In this class we do not want to optimize CPU time or memory (this is a topic for a course in Compuer Science) and the three above solutions are equivalent.

The initial script [13]

- Now we want to print the $\mathbf{w}$ and $\hat{\mathbf{w}}$ to see if they are equal:

```
print(w.T)
print(w_hat.T)
```

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

```
print('Value of vector w')
print(w.T)
print('Value of vector w_hat')
print(w_hat.T)
print('estimation error e=y-A*w_hat')
e=y-A@w_hat # error vector
print(e.T)
print('Square norm of the error')
print(np.linalg.norm(e)**2)
```

The initial script [14]

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

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

Note: We import the matplotlib library in line 10, which is not “good writing”; all the required libraries should be imported in the first lines of the script.
Search `https://matplotlib.org/` for the details about plots (really many possibilities).
WARNING: whatever is the plot you generate, remember to **always 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)

Python is an Object-Oriented programming language.

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?

- 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 subclasses `SolveLLS` (LLS method), `SolveGrad` (gradient algorithm), 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).
- We start by writing in a separate file `minimization.py` the lines in the next slide.

Class SolveMinProbl [1]

Import the libraries and define the initialization of the class.

```
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 (known)
        self.y = y # column vector y (known)
        self.Np = y.shape[0] # number of rows
        self.Nf = A.shape[1] # number of columns
        self.what = np.zeros((self.Nf,1),dtype=float) # column vector w_hat to be found
        self.min = 0 # square norm of the error (initially set to zero)
        return
```

Class SolveMinProbl [2]

Define the methods to print and plot the optimum vector $\hat{\mathbf{w}}$. Note the indentation on the left: if you use it, then method `plot_what` is a method of class `SolveMinProbl`, otherwise it is a method on its own.

```
    def plot_what(self,title='Solution'): # method to plot what
        what=self.what # retrieve what
        n=np.arange(self.Nf) # instantiate the x-axis
        plt.figure() # instantiate the figure
        plt.plot(n,what) # generate the plot what versus n
        plt.xlabel('n') # set the x-axis label
        plt.ylabel('w_hat(n)') # set the y-axis label
        plt.title(title) # set the title of the figure
        plt.grid() # include grids
        plt.show() # show the picture
        return
    def print_result(self,title): # method to print the result
        print(title,' :')
        print('the optimum weight vector is: ')
        print(self.what.T)
        return
```

Subclass SolveLLS

Now define method `SolveLLS` that implements the LLS method (same lines as in the previous script)

```
class SolveLLS(SolveMinProbl):# subclass SolveLLS belongs to class SolveMinProb
    """
    Add here your comments...
    """
    def run(self):
        A=self.matr # retrieve the known matrix A
        y=self.y # retrieve the known vector y
        w_hat=np.linalg.inv(A.T@A)@(A.T@y) # evaluate what (the solution)
        self.what=w_hat # store the solution in self.what, so that it can be used by other methods
        self.min=np.linalg.norm(A@w_hat-y)**2 # find the error square norm
        return
```

Note that we just define method `run`, not the initialization method `__init__`, nor the methods `plot_what` and `print_result` since these methods are all inherited from class `SolveMinProbl` which `SolveLLS` belongs to.
Save file `minimization.py`.

Main script [1]

Open a new file `lab0.py` in which we define the values of matrix **A**, of vector **y** and then we call subclass `SolveLLS`:

```
import minimization as mymin # import the file with the (sub)classes
import numpy as np
Np = 100 #number of rows
Nf = 4 #number of columns
A = np.random.randn(Np,Nf) # matrix/Ndarray A
w = np.random.randn(Nf,1) # true vector w
y = A@w# column vector y
m = SolveLLS(y,A) # instantiate the object
m.run() #run LLS
m.print_result('LLS') # print the results (inherited method)
m.plot_w_hat('LLS')# plot w_hat (inherited method)
```

Note that we instantiate the object `SolveLLS`, we call it `m` with arguments `y,A` as defined in the initialization of class `SolveMinProbl`. Then we use the methods defined for subclass `SolveLLS` using the dot notation: `m.run()` (without arguments, because it has none), `m.print_result('LLS')` (with a string as argument), etc.
In the definition of the method `run` we have instead one argument, which is `self`: the method `run` needs to access the parameters initialized in the class itself (`self`).

Main script [2]

If lab0.py and minimization.py are in the same folder, then the script runs as it is.
If lab0.py is in folder lab0 and minimization.py is in folder lab0/sub, then use the syntax:

```
import sub.minimization as mymin # import the file with the (sub)classes
```

Subclass SolveGrad [1]

Now we want to add in file `minimization.py` the subclass `SolveGrad` that implements the gradient algorithm.

```python
class SolveGrad(SolveMinProbl):
    """ Comments...
    """
    def run(self,gamma=1e-3,Nit=100):# hyperparameters gamma and number of iterations
        self.err = np.empty((0,2), dtype=float) # empty array with two columns
        self.gamma = gamma # learning rate
        self.Nit = Nit # number of iterations
        A = self.matr # retrieve A
        y = self.y # retrieve y
        w = np.random.rand(self.Nf,1)# random #initialization of the weight vector
        for i in range(Nit):
            grad = 2*A.T@(A@w-y) # gradient at current value of w
            w = w-gamma*grad # update of w
            sqerr = np.linalg.norm(A@w-y)**2 # square norm of the error
            self.err=np.append(self.err,np.array([[i,sqerr]]), axis=0)
        self.what = w # store w in what
        self.min = sqerr # store sqerr in self.min
```

Subclass SolveGrad [2]

In the above implementation the stopping condition is the number of iterations (`Nit`). In the lecture about minimization other stopping conditions were discussed (and you can implement them).
If everything is correct, the error $\|\mathbf{y} - \mathbf{A}\mathbf{w}(i)\|^2$ should decrease as the iteration step i 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.
As a consequence we add method `plot_err` to class `SolveMinProbl` (it will be useful also for other subclasses).

Subclass SolveGrad [3]

```
    def plot_err(self,title='Square error',logy=0,logx=0):
        """        """
        err=self.err # get the matrix with the error values
        plt.figure()
        if (logy==0) & (logx==0):
            plt.plot(err[:,0],err[:,1]) # nat-nat scales
        if (logy==1) & (logx==0):
            plt.semilogy(err[:,0],err[:,1]) # nat-log scales
        if (logy==0) & (logx==1):
            plt.semilogx(err[:,0],err[:,1]) # log-nat scales
        if (logy==1) & (logx==1):
            plt.loglog(err[:,0],err[:,1]) # log-log scales
        plt.xlabel('n')
        plt.ylabel('e(n)')
        plt.title(title)
        plt.margins(0.01,0.1)# leave some space
        plt.grid()
        plt.show()
        return
```

Subclass SolveGrad [4]

In `lab0.py` we add the following lines

```python
Nit = 1000 # number of steps for the gradient algorithm
gamma = 1e-5 # learning rate for the gradient algorithm
g=mymin.SolveGrad(y,A) # instantiate SolveGrad
g.run(gamma,Nit) # run SolveGrad
g.print_result('Gradient algorithm') # inherited method
logx = 0 # we want a natural scale on the x-axis
logy = 1 # we want a logarithmic scale on the y-axis
g.plot_err('Gradient algorithm: square error',logy,logx) # inherited method
g.plot_what('Gradient algorithm') # inherited method
```

By running the script, you can notice that `g.what` is different from `m.what`. Why? which parameter(s) should you change to make them (almost) equal?

Table of Contents

To do (before next lab)

- Write the Python code in your file/files.
- **Set the seed** so that every time you run your script, the random matrices/vectors are always equal.
- **Write subclass `SolveSteepDesc` that implements the steepest descent algorithm**. Note that, if the gradient is zero or very small, then the optimum learning rate might be NAN (division by zero). Write your script taking care of this problem.
- **Compare the results you get with LLS, gradient algorithm, steepest descent algorithm**. Make them equal.
- **Change the stopping condition for the gradient algorithm as you like** and run again the script.
- Lab0 will be used to solve the regression problem of Lab 1, so have the software ready for the next lab (in two weeks).

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".