4 Brevi lezioni di python

Francesco Zeno Costanzo *

(Do you remember the) 21(st night of) September 2022

I think, it's time we blow this scene.
Get everybody and the stuff together.
Ok three, two, one, let's jam.
Seatbelts, Tank! (1999)

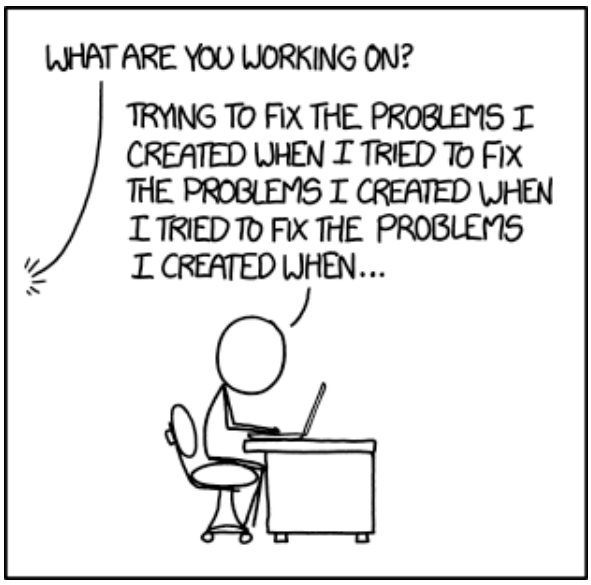
*Ringraziamenti agli autori del materiale di cui queste note sono una trascrizione, revisione e ampliamento: Antonio D'Abbruzzo, Maria Domenica Galati, Francesco Maio, Damiano Lucarelli, Giulio Carotta

Indice

1 Introduzione

Lo scopo di queste note è fornire una breve introduzione al linguaggio di programmazione Python per il corso: "Quattro brevi lezioni di python", organizzato dal comitato locale aisf pisa.

Python è un linguaggio di programmazione generalista noto per essere semplice da utilizzare per noi poveri umani, ovvero la fase di scrittura del codice è molto più leggera e scorrevole, rispetto ad esempio ad un codice in linguaggio C. In oltre, a differenza di altri linguaggi, esso è interpretato e non compilato; questo porta dei vantaggi, ad esempio se si verifica un errore a tempo di esecuzione la shell ci avverte indicandoci le righe di codice da noi scritte dove l'errore è avvenuto. In linguaggi compilati, come C, fortran o altri, il compilatore crea un file chiamato eseguibile dal quale però non può risalire al codice scritto da noi e quindi ciò che causa un errore a tempo di esecuzione (e.g. il famoso segmentation fault) è difficile da ritrovare. Ovviamente a causa della conservazione della massa, o si ha la botte piena o la moglie ubriaca, aut aut terzium non datur; nella fattispecie un esempio di svantaggio che possiede un linguaggio interpretato rispetto ad uno compilato è nelle prestazioni: Python è molto più lento di C o fortran, anche se un buon uso delle molte e vaste librerie che Python possiede può migliorare un po' le cose.



1.1 Notazioni

Nel seguito delle note saranno presenti codici in dei riquadri e, per completezza, dopo la riga [Output] viene presentato anche il risultato degli stessi nel caso ci fossero (i.e. ciò che viene stampato su shell).

1.2 Primo comandamento dell'informatica

Se funziona quanto basta
non toccare che si guasta.

1.3 Secondo comandamento dell'informatica

RTFM: Read The Fucking Manual.
La documentazione on-line è il miglior posto dove trovare risposte.

1.4 Terzo comandamento dell'informatica

Non dite che non funziona finché non
avete provato a spegnere e riaccendere.

1.5 Quarto comandamento dell'informatica

Il computer fa esattamente quello che voi gli
dite di fare non quello che volete che faccia.
La bravura è far coincidere le due cose.

2 Lezione Zero: Installazione

2.1 Installazione dell'ambiente: Pyzo

Il primo passo è procurarsi l'ambiente software tramite il quale è possibile scrivere, gestire e compilare il codice. La scelta su quale ambiente utilizzare è chiaramente arbitraria e soggetta al gusto del singolo. Un buon ambiente che si consiglia è Pyzo. Alla pagina `https://pyzo.org/start.html` è possibile trovare i link per scaricare l'opportuno installer a seconda del sistema operativo che si usa (quelli indicati sotto lo Step 1). Si faccia anche attenzione alla differenza tra gli installer per sistemi a 32 o 64 bit[1]. Nel caso in cui vi piaccia smanettare con i sistemi Linux, consigliamo come procedura alternativa (e più immediata) accedere al terminale e digitare i seguenti comandi:

```
$ sudo apt -get install python3 -pip python3 - pyqt5
$ sudo python3 -m pip install pyzo -- upgrade
$ pyzo
```

Tramite l'ultimo comando si accede alla schermata dell'ambiente Pyzo. A seconda della distribuzione che si utilizza potrebbe essere necessario utilizzare il comando yum al posto di apt-get, in particolare se utilizzate Fedora e derivati invece di Debian/Ubuntu.

2.2 Installazione dell'interprete: Anaconda

Ora che abbiamo l'ambiente bisogna munirisi di un interprete. Tra i tanti, si consiglia Anaconda, che porta in automatico tutti i pacchetti necessari per il lavoro scientifico. Esso è reperibile al seguente indirizzo: `https://www.anaconda.com/download/`. Allo scopo di mantenere la compatibilità con il sistema Pyzo si raccomanda di scaricare la versione corrispondente a Python 3 e non Python 2. Alternativamente è possibile procurarsi Miniconda, che è una versione ridotta e più leggera di Anaconda che arriva con molti meno pacchetti, ma occupa chiaramente meno spazio in memoria. Esso è reperibile al seguente indirizzo: `https://conda.io/miniconda.html`. È fortemente consigliato installare l'interprete nella cartella di default, in modo da rendere più semplice il lavoro di riconoscimento del programma da parte di Pyzo. Una volta installato l'interprete, aprendo Pyzo dovreste essere in grado di riconoscere sulla sinistra un editor di testo e sulla destra, uno sopra l'altro, una console per l'inserimento dei comandi e un file browser per accedere in modo più immediato alle cartelle del computer. Una volta aperto Pyzo, quest'ultimo dovrebbe riconoscere automaticamente l'interprete appena installato (Anaconda, Miniconda o altro) e potrebbe chiedervi di confermare questa scelta. Nel caso invece non riesca a trovare da solo l'interprete, magari perché installato in una cartella diversa da quella di default o perché ne avete installato più di una versione, bisogna selezionarlo manualmente tramite la procedura seguente. Dalla schermata principale di Pyzo, selezionate il menu "Shell" in alto, scegliendo quindi "Edit shell configurations". Nella finestra che viene aperta, selezionate dal menu a tendina del campo "exe" la versione di Python (ad esempio, anaconda3) che avete appena installato. Cliccate sul pulsante "Done" e poi riavviate Pyzo per terminare questa procedura. Se invece non vedete l'interprete appena installato tra le opzioni del menu a tendina, occorre specificare manualmente il percorso intero dove è stato installato l'interprete. Dato che sono stati registrati numerosi problemi nella ricerca del percorso da indicare per quanto riguarda Anaconda su Mac OS, di seguito è riportato un template del percorso dove avviene l'installazione di default, da indicare per intero.

```
/Users/nome_utente/opt/anaconda3/bin/python
```

oppure

```
/Users/nome_utente/anaconda3/bin/python
```

2.3 Installazione dei pacchetti

Python, come tanti altri linguaggi di programmazione, dispone di pacchetti di funzioni già pronte e direttamente utilizzabili da parte del programmatore. Anaconda contiene già tutti i pacchetti che ci serviranno, nel caso in cui abbiate optato per Miniconda, è probabile che abbiate bisogno di scaricare alcuni pacchetti aggiuntivi. L'operazione può essere effettuata accedendo alla console di Pyzo e digitando semplicemente:

```
install <nome_del_pacchetto >
```

oppure

```
pip install <nome_del_pacchetto >
```

Per essere sicuri che sia andato tutto bene provate a scrivere:

```
import <nome_del_pacchetto >
```

se non succede nulla siete apposto

[1] Al seguente link potete trovare informazioni per scoprire, nel caso in cui non lo sapeste, se l'architettura del vostro computer è a 32 o 64 bit: `https://support.microsoft.com/it-it/help/15056/windows-7-32-64-bit-faq`

3 Prima lezione

3.1 Funzione print

Se un macchina fosse senziente e gentile, quini non un'intelligenza artificiale cresciuta su twitter, forse come prima cosa saluterebbe tutti e il modo per comunicare è la funzione print, vediamo quindi il più classico degli esempi:

```
print('Hello world!')

[Output]
Hello world!
```

Questa funzione può stampare sia valori che espressioni:

```
print('Hello world!')
print('42')

print('Hello world!', 42)
print('Adesso vado \n a capo')

[Output]
Hello world!
42
Hello world! 42
Adesso vado
 a capo
```

3.2 Commenti

Al fine di rendere fruibile ad altri o anche al voi stesso del futuro il codice è opportuno inserire i commenti, ovvero frasi che non vengono lette dall'interprete (o dal compilatore) che spiegano cosa voi stiate facendo; altrimenti vi ritroverete nella scomoda situazione in cui solo Dio saprebbe spiegarvi il funzionamento del vostro codice.

```
#per i commenti che occupano una singola linea di codice si una il cancelletto
#stampo hello world
print('Hello world!')

"""
per un commento di maggiori linee di codice

vanno usate tre virgole per racchiuderlo
"""
'''
ma van bene

anche tre apici
'''
[Output]
Hello world!
```

3.3 Variabili

Una variabile è un nome, un simbolo, che si da ad un certo valore. In Python non è necessario né definire le variabili prima di utilizzarle, né specificare il loro tipo, esse si creano usando il comando di assegnazione '='. Facciamo un esempio con variabile numeriche:

```
numerointero= 13
numeroavirgolamobile= 13.

print('Numero intero:', numerointero, 'Numero in virgola mobile:', numeroavirgolamobile)
#oppure possiamo stampare in questo modo:
print(f'Numero intero: {numerointero}, Numero in virgola mobile: {numeroavirgolamobile}')

[Output]
Numero intero: 13 Numero in virgola mobile: 13.0
Numero intero: 13 Numero in virgola mobile: 13.0
```

Ovviamente le variabili possono essere non solo numeri ma anche molto altro, e possiamo verificarne il tipo grazie alla funzione 'type()':

```
#inizializziamo delle variabili
n = 7
```

```
x = 7.
stringa = 'kebab'
lista = [1, 2., 'cane']
tupla = (42, 'balena')
dizionario = {'calza': 0, 'stampante': 0.5}

#stampiamole e stampiamone il tipo
print(n, type(n))
print(x, type(x))
print(stringa, type(stringa))
print(lista, type(lista))
print(tupla, type(tupla))
print(dizionario, type(dizionario))

[Output]
7 <class 'int'>
7.0 <class 'float'>
kebab <class 'str'>
[1, 2.0, 'cane'] <class 'list'>
(42, 'balena') <class 'tuple'>
{'calza': 0, 'stampante': 0.5} <class 'dict'>
```

Vediamo ora come fare le classiche operazioni matematiche:

```
x1 = 3
y1 = 4

somma = x1 + y1
prodotto = x1*y1
differenza = x1 - y1
rapporto = x1/y1
potenza = x1**y1

print(somma, prodotto, differenza, rapporto, potenza)
[Output]
7 12 -1 0.75 81
```

Vale la pena dilungarsi un attimo su una piccola questione: l'aritmetica in virgola mobile è intrinsecamente sbagliata poiché giustamente il computer ha uno spazio di memoria finita e quindi non può tenere infinite cifre decimali:

```
x = 0.1
y = 0.2
z = 0.3

print(x + y)
print(z)

[Output]
0.30000000000000004
0.3
```

Il precedente è solo uno tra i molti esempi che si potrebbero fare per far notare come l'aritmetica dei numeri in virgola mobile possa dare problemi; Il lettore provi a vedere se è vero che $a + (b + c) = (a + b) + c$ con a, b, c numeri in virgola mobile, probabilmente il computer non sarà d'accordo. Ai computer i numeri in virgola mobile, i numeri reali, non piacciono molto, preferiscono i numeri interi e quelli razionali (e ovviamente adorano le potenze di due, grazie codice binario):

```
#variabili
x = 0.1
y = 0.2
z = 0.3
#sommo le prime due
t = x + y

"""
applico una funzione che mi fornisce
una tupla contenente due numeri interi
il cui rapporto restituisce il numero iniziale.
output del tipo: (numeratore, denominatore)
"""
print(t.as_integer_ratio())
print(z.as_integer_ratio())

[Output]
(1351079888211149, 4503599627370496)
(5404319552844595, 18014398509481984)
```

Per quanto riguarda i numeri in virgola mobile, possiamo scegliere quante cifre dopo la virgola stampare, vediamolo con un esempio:

```
#definiamo una variabile
c = 3.141592653589793

#stampa come intero
print('%d' %c)

#stampa come reale
print('%f' %c) #di default stampa solo prime 6 cifre
print(f'{c}')  #di default stampa tutte le cifre

#per scegliere il numero di cifre, ad esempio sette cifre
print('%.7f' %c)
print(f'{c:.7f}')

[Output]
3
3.141593
3.141592653589793
3.1415927
3.1415927
```

Notare che il computer esegue un arrotondamento.
Una variabile può essere ridefinita e cambiare valore, il computer userà l'assegnazione più recente:

```
#definiamo una variabile
x = 30

"""

operazioni varie


"""

#ridefiniamo la variabile
x = 18

print('x=', x)

"""
E' possibile anche sovrascrivere una variabile
con un numero che dipende dal suo valore precedente:
"""
x = x + 1 #incrementiamo di uno
#Oppure:
x += 1
print('x=', x)

x = x * 2 #moltiplichiamo per due
#Oppure:
x *= 2
print('x=', x)

[Output]
x= 18
x= 20
x= 80
```

Come si vede i vari comandi $x = x$ *operazione numero* possono essere abbreviati con x *operazione*$=$ *numero*.

3.4 Librerie

Le librerie sono luoghi mistici create dagli sviluppatori, esse contengono molte funzioni, costanti e strutture dati predefinite; in generale se volete fare qualcosa esisterà una libreria con una funzione che implementa quel qualcosa o che comunque vi può aiutare in maniera non indifferente. Prima di poter accedere ai contenuti di una libreria, è necessario importarla. Per farlo, si usa il comando import. Solitamente è buona abitudine importare tutte le librerie che servono all'inizio del file. Ecco un paio di esempi:

```
import numpy

#per usare un contenuto di questa libreria basta scrivere numpy.contenuto
pigreco = numpy.pi
print(pigreco)
```

```

#Possiamo anche ribattezzare le librerie in questo modo
import numpy as np
#da ora all'interno del codice numpy si chiama np

eulero = np.e
print(eulero)

[Output]
3.141592653589793
2.718281828459045
```

```
import math

coseno=math.cos(0)
seno = math.sin(np.pi/2) #python usa di default gli angoli in radianti!!!
senosbagliato = math.sin(90)

print('Coseno di 0=', coseno, "\nSeno di pi/2=", seno, "\nSeno di 90=", senosbagliato)

#bisogna quindi stare attenti ad avere tutti gli angoli in radianti
angoloingradi = 45
#questa funzione converte gli angoli da gradi a radianti
angoloinradianti = math.radians(angoloingradi)

print("Angolo in gradi:", angoloingradi, "Angolo in radianti:", angoloinradianti)

[Output]
Coseno di 0= 1.0
Seno di pi/2= 1.0
Seno di 90= 0.8939966636005579
Angolo in gradi: 45 Angolo in radianti: 0.7853981633974483
```

Le due librerie qui usate contengono funzioni simili, ad esempio il seno è implementato sia in numpy che in math, cambia il fatto che math può calcolare il seno di un solo valore, mentre numpy, come vedremo, può calcolare il seno di una sequenza di elementi. Come sapere tutte le possibili funzioni contenute nelle librerie e come usarle? "Leggetevi il cazzo di manga" (n.d.r. Leggetevi la documentazione disponibile tranquillamente online)

3.5 Gestione errori

È molto facile scrivere codice che produca errore, magari perchè distrattamente ci siamo dimenticati qualcosa o magri qualcosa è stato implementato male. Esiste un costrutto che ci permette di gestire gli errori in maniera tranquilla diciamo. Facciamo un semplice esempio:

```
a = 0
b = 1/a
print(b)

[Output]
Traceback (most recent call last):
  File "<tmp 1>", line 5, in <module>
    b = 1/a
ZeroDivisionError: division by zero
```

Abbiamo fatto una cosa molto brutta, nemmeno Dio può dividere per zero (cosa non proprio vera infatti si hanno ancora teorie fisiche con divergenze purtroppo) e quindi il computer ci da errore. Possiamo aggirare il problema, evitando così il second impact, in due modi diciamo:

3.5.1 Try e except

Possiamo utilizzare il costrutto try except dicendo al computer: prova a fare la divisione e, se questa da errore, e l'errore è "ZeroDivisionError" allora assegna a b un altro valore. in questo modo eventuali istruzioni presenti dopo vengono eseguite e il codice non si arresta.

```
a = 0
try :
    b = 1/a
except ZeroDivisionError:
    b = 1

print(b)

[Output]
1
```

3.5.2 Raise Exception

Mettiamo il caso in cui ci siano operazioni da fare in cui il valore della variabile "b" è importante, quindi sarebbe meglio interrompere il flusso del codice perché con un dato valore il risultato finale sarebbe poco sensato. Si può fare il controllo del valore e sollevare un'eccezione per fermare il codice.

```
"""
leggo un valore da shell
uso del comando try per evitare che venga letto
qualcosa che non sia un numero: e.g. una stringa
"""
try:
    b = int(input('scegliere un valore:'))
except ValueError:
    print('hai digitato qualcosa diverso da un numero, per favore ridigitare')
    b = int(input('scegliere un valore:'))

#se si sbaglia a digitare di nuovo il codice si arresta per ValueError

#controllo se e' possibile proseguire
if b > 7 :
    #se vero si blocca il codice sollevando l'eccezione
    messaggio_di_errore = 'il valore scelto risulta insensato in quanto nulla supera 7, misura massima di ogni cosa'
    raise Exception(messaggio_di_errore)

[Output]
8
Traceback (most recent call last):
  File "<tmp 1>", line 18, in <module>
    raise Exception(messaggio_di_errore)
Exception: il valore scelto risulta insensato in quanto nulla supera 7, misura massima di ogni cosa
```

4 Seconda lezione

Ripetiamo tutti insieme: Python conta da zero.

4.1 Gli array

Un array unidimensionale è semplicemente una sequenza ordinata di numeri; è, possiamo dire, un vettore. Utilizzeremo la libreria numpy. Per alcuni aspetti essi sono simili alle liste native di Python ma le differenze sono molte, in seguito ne vedremo alcune. Cominciamo con qualche esempio:

```
import numpy as np

#Creiamo un array di 5 elementi
array1 = np.array([1.0, 2.0, 4.0, 8.0, 16.0]) #scrivere 2.0 equivale a scrivere 2.

print(array1)

#per accedere a un singolo elemento dell'array basta fare come segue:
elem = array1[1]

#ATTENZIONE! Gli indici, per Python, partono da 0, non da 1!
print(elem)

[Output]
[ 1.  2.  4.  8. 16.]
2.0
```

ora, avendo creato il nostro array potremmo volendo aggiungere o togliere degli elementi:

```
import numpy as np

array1=np.array([1.0, 2.0, 4.0, 8.0, 16.0])

#Aggiungiamo ora un numero in una certa posizione dell'array:
array1 = np.insert(array1, 4, 18)
'''
abbiamo aggiunto il numero 18 in quarta posizione, la sintassi e' :
np.insert(array a cui vogliamo aggiungere un numero, posizione dove aggiungerlo, numero)
'''
print(array1)

#Per aggiungere elementi in fondo ad un array esiste anche il comando append della libreria
    numpy:
array2 = np.append(array1, -4.)
print(array2)
#Mentre per togliere un elemento basta indicare il suo indice alla funzione remove di numpy:
array2 = np.delete(array2, 0)
print(array2)

[Output]
[ 1.  2.  4.  8. 18. 16.]
[ 1.  2.  4.  8. 18. 16. -4.]
[ 2.  4.  8. 18. 16. -4.]
```

4.2 Tipi di array

Come le variabile numeriche sopra anche gli array posseggono i tipi e qui viene la prima differenza con le liste, se ad un array di numeri provassimo ad aggiungere un elemento che sia una stringa avremmo un errore; questo perché ogni array di numpy ha un suo tipo ben definito, che viene fissato, implicitamente o esplicitamente, al momento della creazione. Possiamo sì creare un array di tipo misto ma con tale array non si potrebbero fare le classiche operazioni matematiche.

```
import numpy as np

array1 = np.array([1.0, 2.0, 4.0, 8.0, 16.0])

tipoarray1 = array1.dtype
print(tipoarray1)

a = np.array([0, 1, 2])
#abbiamo scritto solo numeri interi => array di interi

b = np.array([0., 1., 2.])
#abbiamo scritto solo numeri con la virgola => array di numeri float
```

```
13
14 """
15 #nota: anche se si dice "numero con la virgola",
16 vanno scritti sempre col punto!
17 La virgola separa gli argomenti
18 """
19
20 c = np.array([0, 3.14, 'giallo'])
21 #quest'array e' misto. Ci sono sia numeri interi che float che stringhe
22
23
24 #ora invece il tipo viene definito in maniera esplicita:
25 d = np.array([0., 1., 2.], 'int')
26 e = np.array([0, 1, 2], 'float')
27
28 print(a, a.dtype)
29 print(b, b.dtype)
30 print(c, c.dtype)
31 print(d, d.dtype)
32 print(e, e.dtype)
33
34
35 [Output]
36 float64
37 [0 1 2] int32
38 [0. 1. 2.] float64
39 ['0' '3.14' 'giallo'] <U32
40 [0 1 2] int32
41 [0. 1. 2.] float64
```

4.3 Array predefiniti

Vediamo brevemente alcuni tipi di array già definiti e di uso comune:

```
1 import numpy as np
2
3 #array contenente tutti zero
4 arraydizeri_0 = np.zeros(3)#il numero specificato e' la lunghezza
5 arraydizeri_1 = np.zeros(3, 'int')
6
7 #array contenente tutti uno
8 arraydiuni_0 = np.ones(5)#il numero specificato e' la lunghezza
9 arraydiuni_1 = np.ones(5, 'int')
10
11 print(arraydizeri_0, arraydizeri_1)
12 print(arraydiuni_0, arraydiuni_1)
13
14 """
15 questo invece e' un array il cui primo elemento e' zero
16 e l'ultimo elemento e' 1, lungo 10 e i cui elementi sono
17 equispaziati in maniera lineare tra i due estremi
18 """
19 equi_lin = np.linspace(0, 1, 10)
20 print(equi_lin)
21
22
23 """
24 questo invece e' un array il cui primo elemento e' 10^1
25 e l'ultimo elemento e' 10^2, lungo 10 e i cui elementi sono
26 equispaziati in maniera logaritmica tra i due estremi
27 """
28 equi_log = np.logspace(1, 2, 10)
29 print(equi_log)
30
31 [Output]
32 [0. 0. 0.] [0 0 0]
33 [1. 1. 1. 1. 1.] [1 1 1 1 1]
34 [0.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
35  0.66666667 0.77777778 0.88888889 1.        ]
36 [ 10.          12.91549665  16.68100537  21.5443469   27.82559402
37   35.93813664  46.41588834  59.94842503  77.42636827 100.        ]
```

4.4 Operazioni con gli array

Vediamo ora un po' di cose che si possono fare con gli array:

```
import numpy as np

array1 = np.array([1.0, 2.0, 4.0, 8.0, 16.0])

primi_tre = array1[0:3]
print('primi_tre = ', primi_tre)
"""
Questa sintassi seleziona gli elementi di array1
dall'indice 0 incluso all'indice 3 escluso.
Il risultato e' ancora un array.
"""

esempio = array1[1:-1]
print(esempio)
esempio = array1[-2:5]
print(esempio)
#Questo metodo accetta anche valori negativi, con effetti curiosi


elementi_pari = array1[0::2]
print('elementi_pari = ', elementi_pari)
"""
In questo esempio invece, usando invece due volte il simbolo :
intendiamo prendere solo gli elementi dall'indice 0 saltando di 2 in 2.
Il risultato e' un array dei soli elementi di indice pari
"""

rewind = array1[::-1]
print('rewind = ', rewind)
"""
Anche qui possiamo usare valori negativi.
In particolare questo ci permette di saltare "all'indietro"
e, ad esempio, di invertire l'ordine di un'array con un solo comando
"""

[Output]
primi_tre =  [1. 2. 4.]
[2. 4. 8.]
[ 8. 16.]
elementi_pari =  [ 1.  4. 16.]
rewind =  [16.  8.  4.  2.  1.]
```

Grande comodità sono le operazione matematiche che possono essere fatte direttamente senza considerare i singoli valori, o meglio Python ci pensa da sé a fare le operazioni elemento per elemento. Gli array devono avere la stessa dimensione altrimenti avremmo errore, infatti potrebbe esserci un elemento spaiato.

```
import math
import numpy as np

v = np.array([4, 5, 6])
w = np.array([1.2, 3.4, 5.8])

#classiche operazioni
somma = v + w
sottr = v - w
molt = v * w
div = v / w

print(v, w)
print()
print(somma, sottr, molt, div)
print()
#altri esempi
print(v**2)
print(np.log10(w))

"""
come dicevamo prima qui' otterremmo errore poiche'
math lavora solo con numeri o, volendo,
array unidimensionali lunghi uno
"""
print(math.log10(w))

[Output]
[4 5 6] [1.2 3.4 5.8]

[ 5.2  8.4 11.8] [2.8 1.6 0.2] [ 4.8 17.  34.8] [3.33333333 1.47058824 1.03448276]
```

```
[16 25 36]
[0.07918125 0.53147892 0.76342799]
Traceback (most recent call last):
  File "<tmp 1>", line 26, in <module>
    print(math.log10(w))
TypeError: only size-1 arrays can be converted to Python scalars
```

Se provassimo le stesse con delle liste solo la somma non darebbe errore, ma il risultato non sarebbe comunque lo stesso che otteniamo con gli array. Anche moltiplicare un array o una lista per un numero intero produce risultati diversi se provate vi sarà facile capire perché si è specificato che il numero deve essere intero.

4.5 Matrici

Se un array unidimensionale lungo n è un vettore ad n componenti allora un array bidimensionale sarà una matrice.

```
import numpy as np

#esiste la funzione apposita di numpy per scrivere matrici.
matrice1 = np.matrix('1 2; 3 4; 5 6')
#Si scrivono essenzialmente i vettori riga della matrice separati da ;

#equivalente a:
matrice2 = np.matrix([[1, 2], [3, 4], [5,6]])

print(matrice1)
print(matrice2)


matricedizeri = np.zeros((3, 2)) #tre righe, due colonne: matrice 3x2
print('Matrice di zeri:\n', matricedizeri, '\n')
matrice diuni = np.ones((3,2))
print('Matrice di uni:\n', matrice diuni, '\n')

[Output]
[[1 2]
 [3 4]
 [5 6]]
[[1 2]
 [3 4]
 [5 6]]
Matrice di zeri:
 [[0. 0.]
 [0. 0.]
 [0. 0.]]

Matrice di uni:
 [[1. 1.]
 [1. 1.]
 [1. 1.]]
```

E ovviamente anche qui possiamo fare le varie operazioni matematiche:

```
import numpy as np

matrice1 = np.matrix('1 2; 3 4; 5 6')
matrice diuni = np.ones((3,2))

sommadimatrici = matrice1 + matrice diuni
print('Somma di matrici:\n', sommadimatrici)

matrice3 = np.matrix('3 4 5; 6 7 8') #matrice 2x3
prodottodimatrici = matrice1 * matrice3  #matrice 3x(2x2)x3
#alternativamente si potrebbe scrivere: prodottodimatrici = matrice1 @ matrice3

print('\nProdotto di matrici:\n', prodottodimatrici)

[Output]
Somma di matrici:
 [[2. 3.]
 [4. 5.]
 [6. 7.]]

Prodotto di matrici:
 [[15 18 21]
```

```
[33 40 47]
[51 62 73]]
```

5 Terza lezione

5.1 Le funzioni

Don't repeat yourself: è questa la logica delle funzioni. Le funzioni sono frammenti di codici, atti a ripetere sempre lo stesso tipo di operazioni con diversi valori dei parametri in input a seconda delle esigenze. Come al solito vediamo degli esempi:

```
def area(a, b):
    """
    restituisce l'area del rettangolo
    di lati a e b
    """
    A = a*b #calcolo dell'area
    return A

#chiamiamo la funzione e stampiamo subito il risultato
print(area(3, 4))
print(area(2, 5))

"""
Se la funzione non restituisce nulla
ma esegue solo un pezzo di codice,
si parla propriamente di procedura
e il valore restituito e' None.
"""
def procedura(a):
    a = a+1

print(procedura(2))

"""
Volendo si possono creare anche funzioni
che non hanno valori in ingresso:
"""
def pigreco():
    return 3.14
print(pigreco())

[Output]
12
10
None
3.14
```

Portiamo all'attenzione due fatti importati:

- È fondamentale in Python che il corpo della funzione sia indentato, per seguire un raggruppamento logico del codice.
- Definendo degli argomenti per una funzione si creano delle variabili 'locali', il cui nome non influenza tutto quello che c'è fuori dalla funzione stessa. Ad esempio, per la funzione area abbiamo definito una variabile b, ma posso tranquillamente definire una nuova variabile b al di fuori della funzione.

Abbiamo visto che le funzioni possono prendere dei parametri o anche nessun parametro, quindi la domanda che sorge spontanea è: ne possono prendere infiniti? La risposta è sì ma prima di vederlo facciamo una piccola deviazione e parliamo delle istruzioni di controllo.

5.2 Istruzioni di controllo

Per istruzioni di controllo si intendono dei comandi che modificano il flusso di compilazione di un programma in base a determinati confronti e/o controlli su certe variabili. Ci sono casi in cui il computer deve fare cose diverse a seconda degli input o fare la stessa cosa un certo numero di volte fino a che un certa condizione sia o non sia soddisfatta.

5.2.1 Espressioni condizionali: if, else, elif

Tramite l'istruzione if effettuiamo un confronto/controllo. Se il risultato è vero il programma esegue la porzione di codice immediatamente sotto-indentata. In caso contrario, l'istruzione else prende il controllo e il programma esegue la porzione di codice indentata sotto quest'ultima. Se l'istruzione else non è presente e il controllo avvenuto con l'if risultasse falso, il programma semplicemente non fa niente. Vediamo il caso classico del valore assoluto:

```
def assoluto(x):
    """
    restituisce il valore assoluto di un numero
    """
    # se vero restituisci x
    if x >= 0:
        return x
    #altrimenri restituisci -x
    else:
        return -x

print(assoluto(3))
print(assoluto(-3))

[Output]
3
3
```

È possibile aggiungere delle coppie if/else in cascata tramite il comando "elif", che è identico semanticamente a "else if"; per esempio:

```
def segno(x):
    """
    funzione per capire il segno di un numero
    """
    #se vero ....
    if x > 0:
        return 'Positivo'
    #se invece ....
    elif x == 0:
        return 'Nullo'
    #altrimenti ....
    else:
        return 'Negativo'

print(segno(5))
print(segno(0))
print(segno(-4))

[Output]
Positivo
Nullo
Negativo
```

5.2.2 Cicli: while, for

Partiamo con in cicli while: essi sono porzioni di codice che iterano le stesse operazioni fino a che una certa condizione risulta essere verifica:

```
def fattoriale(n):
    """
    Restituisce il fattoriale di un numero
    """
    R = 1
    #finche' e' vero fai ...
    while n > 1:
        R *= n
        n -= 1
    return R

print(fattoriale(5))

[Output]
120
```

Un'accortezza da porre con i cicli while è verificare che effettivamente la condizione inserita si verifichi altrimenti il ciclo non si interrompe e va avanti per sempre, ed è molto molto tempo.

Passando ai cicli for invece essi ripetono una certa azione finché un contatore non raggiunge il massimo. Vediamo come implementare il fattoriale con questo ciclo:

```
def fattoriale(n):
    """
    restituisce il fattoriale di un numero
    """
```

```
    R = 1
    #finche' i non arriva ad n fai ...
    for i in range(1, n+1):
        R = R*i
    return R

print(fattoriale(5))

[Output]
120
```

Abbiamo quindi introdotto una variabile ausiliaria "i" utilizzata in questo contesto come contatore, cioè come variabile che tiene il conto del numero di cicli effettuati. Nel caso in esame, stiamo dicendo tramite l'istruzione for che la variabile "i" deve variare all'interno della lista range(1, n+1) = [1,2,..., n]. Il programma effettua l'operazione R = R*i per tutti i valori possibili che i assume in questa lista, nell'ordine. Da notare il comando range che crea una lista sulla quale iterare, ma noi abbiamo visto già le liste e gli array e abbiamo visto che presentano alcune somiglianze, un'altra somiglianza da far vedere è che entrambi sono 'iterabili' e quindi possiamo iterarci sopra:

```
import numpy as np

def trova_pari(array):
    """
    restituisce un array contenente solo
    i numeri pari dell'array di partenza
    """
    R = np.array([]) #array da riempire
    #per ogni elemento in arrary fai ...
    for elem in array:
        if elem%2 == 0:
            R = np.append(R,elem)
    return R

a = np.array([i for i in range(0, 11)])
"""
il precedente e' un modo piu' conciso di scrivere:
a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
"""
print(a)
print(trova_pari(a))

[Output]
[ 0  1  2  3  4  5  6  7  8  9 10]
[ 0.  2.  4.  6.  8. 10.]
```

In questo esempio abbiamo utilizzato gli array ma si potrebbe senza problemi rifare tutto con le liste. Altri due comandi interessanti per quanto riguarda i cicli sono: enumerate e zip.
enumerate:

```
import numpy as np

#creiamo un array
array = np.linspace(0, 1, 5)

"""
in questo modo posso iterare contemporaneamente
sia sugli indici sia sugli elementi dell'array
"""
for index, elem in enumerate(array):
    print(index, elem)

[Output]
0 0.0
1 0.25
2 0.5
3 0.75
4 1.0
```

zip:

```
import numpy as np

#creiamo tre un array
array1 = np.linspace(0, 1, 5)
array2 = np.linspace(1, 2, 5)
array3 = np.linspace(2, 3, 5)
```

```
"""
in questo modo posso iterare contemporaneamente
sugli elementi di tutti gli array
"""
for a1, a2, a3 in zip(array1, array2, array3):
    print(a1, a2, a3)

[Output]
0.0 1.0 2.0
0.25 1.25 2.25
0.5 1.5 2.5
0.75 1.75 2.75
1.0 2.0 3.0
```

Anche qui come le funzioni è necessario indentare.

5.3 Ancora funzioni

Dopo questa digressione torniamo alle funzioni, abbiamo detto che una funzione può prendere infiniti argomenti, ma dal punto di vista pratico come lo implementiamo, in un modo semi decente? Una risposta sarebbe quella di passare alla funzione non delle singole variabili ma un array o una lista, cosa che si può fare tranquillamente, e lavorare poi all'interno della funzione con gli indici per utilizzare i vari elementi dell'array, o della lista, o ciclarci sopra. Un altro modo per farlo è usare: *args (args è un nome di default, potremmo chiamarlo mimmo):

```
def molt(*numeri):
    """
    restituisce il prodotto di n numeri
    """
    R = 1
    for numero in numeri:
        R *= numero
    return R

print(molt(2, 7, 10, 11, 42))
print(molt(5, 5))
print(molt(10, 10, 2))

[Output]
64680
25
200
```

L'esempio appena visto non è altro che la funzione fattoriale di prima leggermente modificata e che non prende più in input una sequenza crescente di numeri. I parametri vengono passati come una tupla e in questo caso il simbolo "*" viene definito operatore di unpacking proprio perché "spacchetta" tutte le variabili che vengono passate alla funzione.

5.4 Grafici

Fare un grafico è un modo pratico e comodo di visualizzare dei dati, qualsiasi sia la loro provenienza. Capita spesso che i dati siano su dei file (per i nostri scopi in genere file .txt o .csv) e che i file siano organizzati a colonne:

```
#t[s] x[m]
1     1
2     4
3     9
4     16
5     25
6     36
```

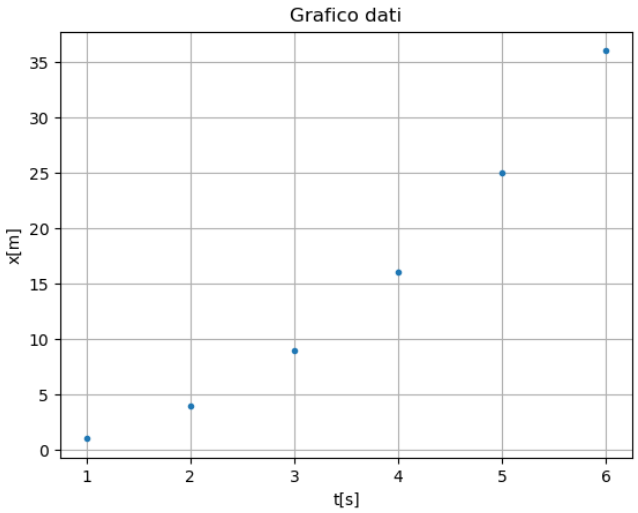
Per leggeri:

```
import numpy as np

#Leggiamo da un file di testo classico
path = 'dati.txt'
dati1, dati2 = np.loadtxt(path, unpack=True)
"""
unpack=True serve proprio a dire che vogliamo che
dati1 contenga la prima colonna e dati2 la seconda
La prima riga avendo il cancelletto verra' saltata
"""

```

```
#se vogliamo invece che venga letto tutto come una matrice scriviamo:
path = 'dati.txt'
dati = np.loadtxt(path, unpack=True)
#dati sara' nella fattispecie una matrice con due colonne e 6 righe


#leggere da file.csv
path = 'dati.csv'
dati1, dati2 = np.loadtxt(path,usecols=[0,1], skiprows=1, delimiter=',',unpack=True)
"""
a differenza di quanto sopra dobbiamo specificare le colonne da usare (contiamo da zero)
a causa poi dell'organizzazione dei .csv dobbiamo dire anche quante righe skippare
"""
```

Creare ora un grafico è semplice grazie all'utilizzo della libreria matplotlib:

```
import numpy as np
import matplotlib.pyplot as plt

#Leggiamo da un file di testo classico
path = 'dati.txt'
dati1, dati2 = np.loadtxt(path, unpack=True)

plt.figure(1) #creiamo la figura

#titolo
plt.title('Grafico dati')
#nomi degli assi
plt.xlabel('t[s]')
plt.ylabel('x[m]')
#plot dei dati
plt.plot(dati1,dati2, marker='.',linestyle='')
#aggiungiamo una griglia
plt.grid()
#comando per mostrare a schermo il grafico
plt.show()
```

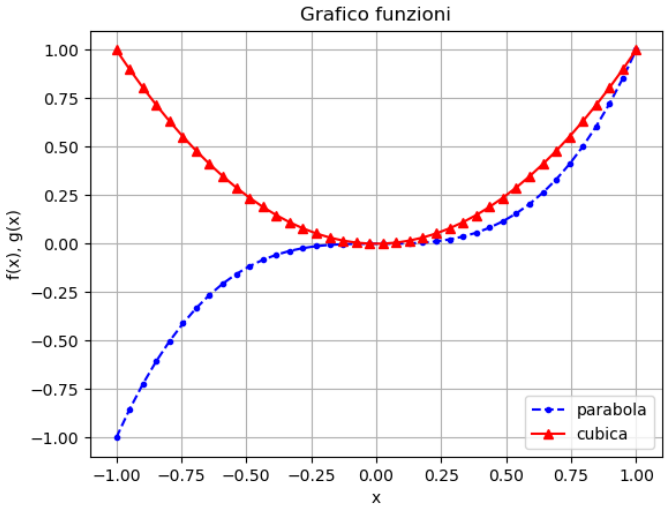


commentiamo un attimo quanto fatto: dopo aver letto i dati abbiamo fatto il grafico mettendo sull'asse delle ascisse la colonna del tempo e su quello delle ordinate la colonna dello spazio; se all'interno del comando "plt.plot(...)" scambiassimo l'ordine di dati1 e dati2 all'ora gli assi si invertirebbero, non avremmo più x(t) ma t(x). Inoltre il comando "marker='.'" sta a significare che il simbolo che rappresenta il dato deve essere un punto; mentre il comando "linestyle="" significa che non vogliamo che i punti siano uniti da una linea (linestyle='-' dà una linea, linestyle='- -' dà una linea tratteggiata).
Se invece volessimo graficare una funzione o più definite da codice? Anche qui i comandi sono analoghi:

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    """
    restituisce il cubo di un numero
    """
    return x**3

def g(x):
    """
    restituisce il quadrato di un numero
    """
    return x**2

#array di numeri equispaziati nel range [-1,1] usiamo:
x = np.linspace(-1, 1, 40)

plt.figure(1) #creiamo la figura

#titolo
plt.title('Grafico funzioni')
#nomi degli assi
plt.xlabel('x')
plt.ylabel('f(x), g(x)')
#plot dei dati
plt.plot(x, f(x), marker='.', linestyle='--', color='blue', label='parabola')
plt.plot(x, g(x), marker='^', linestyle='-', color='red', label='cubica')
```

```
#aggiungiamo una leggenda
plt.legend(loc='best')
#aggiungiamo una griglia
plt.grid()
#comando per mostrare a schermo il grafico
plt.show()
```

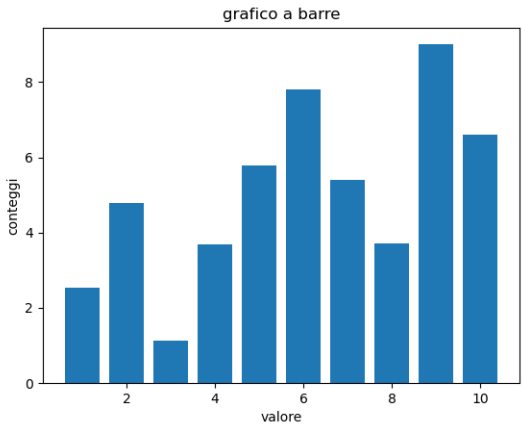
Notare che per distinguere le due funzioni oltre al "marker" e al "linestyle" abbiamo aggiunto il comando "color" per dare un colore e il comando "label" che assegna un'etichetta poi visibile nella legenda (loc='best' indica che Python la mette dove ritiene più consono, in modo che non rischi magari di coprire porzioni di grafico). Ovviamente è consigliata una lettura della documentazione per conosce tutti gli altri comandi possibili per migliorare/abbellire il grafico da adre alle funzioni già presenti. Altre funzioni utili possono essere: "plt.axis(...)" che imposta il range da visualizzare su entrambi gli assi; il comando "plt.xscale(...)" che permette di fare i grafici con una scala, magari logaritimica o altro sull'asse x (analogo sarà sulle y mutatis mutandis).

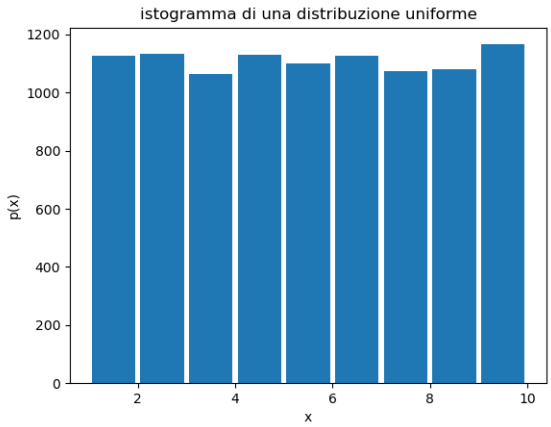


Ultima menzione da fare sono gli istogrammi (il fetish dei pallettari):

```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(1)
plt.title('grafico a barre')
plt.xlabel('valore')
plt.ylabel('conteggi')
# Sull'asse x utilizziamo un array di 10 punti equispaziati.
x = np.linspace(1,10,10)
# Sull'asse y abbiamo, ad esempio, il seguente set di dati:
y = np.array([2.54, 4.78, 1.13, 3.68, 5.79, 7.80, 5.4, 3.7, 9.0, 6.6])

# Il comando per la creazione dell'istogramma corrispondente e':
plt.bar(x, y, align = 'center')

plt.figure(2)
plt.title('istogramma di una distribuzione uniforme')
plt.xlabel('x')
plt.ylabel('p(x)')

"""
lista di numeri distribuiti uniformemente fra 0 e 10
si usa l'underscore nel for poiche' non serve usare
un'altra variabile. Avremmo potuto scrivere for i ...
ma la i non sarebbe comparsa da nessun' altra parte
sarebbe stato uno spreco
"""
z = [np.random.uniform(10) for _ in range(10000)]
plt.hist(z, bins=10, rwidth=0.9)


plt.show()
```

Piccolo appunto che bisogna fare, nel caso di "plt.hist()" bisogna stare attenti perché il numero di bin va scelto con cura (qui abbiamo scritto nove sulla fiducia).





5.5 Esercizio riassuntivo

Vogliamo provare a fare un breve esercizio che riassuma quanto fatto, proviamo a calcolare le aree di tre poligoni regolari di lati rispettivamente $3, 4, 5$ e numero di lato generici. Questo scrivendo una funzione "Area(l, n)" per poi confrontare il valore di "Area(3, 4) + Area(4, n)" con quello di "Area(5, n)" per ogni valore di n.

```
import numpy as np

def Area(l, n):
    """
    calcolo area di un poligono
    regolare di lato l e numero lati n
    """
    a = l/2 * 1/np.tan(np.pi/n) #apotema
    p = n*l                     #perimetro
    A = p*a/2                   #area
    return A

l = [3, 4, 5] #dimesioni dei lati, deve essere una terna pitagorica
n = np.arange(4, 12) #numero di lati dei poligoni

A3 = np.array([]) #array in cui ci saranno le aree dei poligoni di lato 3
A4 = np.array([]) #array in cui ci saranno le aree dei poligoni di lato 4
A5 = np.array([]) #array in cui ci saranno le aree dei poligoni di lato 5

for nn in n: #loop sul numero di lati
    for ll in l: #lup sulla dimesione, quindi sul triangolo
        A0 = Area(ll, nn) #calcolo dell'area

        if ll == 3:
            A3 = np.append(A3, A0)
        elif ll == 4:
            A4 = np.append(A4, A0)
        elif ll == 5:
            A5 = np.append(A5, A0)


for i in range(len(n)):
    print(f'A3 + A4 = {A3[i]+A4[i]:.3f}')
    print(f'   A5   = {A5[i]:.3f}')

[Output]
A3 + A4 = 25.000
   A5   = 25.000
A3 + A4 = 43.012
   A5   = 43.012
A3 + A4 = 64.952
   A5   = 64.952
A3 + A4 = 90.848
   A5   = 90.848
A3 + A4 = 120.711
   A5   = 120.711
A3 + A4 = 154.546
   A5   = 154.546
A3 + A4 = 192.355
   A5   = 192.355
```

```
A3 + A4 = 234.141
   A5   = 234.141
```

Scopriamo quindi piacevolmente che il teorema di Pitagora vale non solo per i quadrati ma per ogni poligono regolare.

5.6 Prestazioni

Avevamo accennato al fatto che Python fosse lento ma che utilizzando le librerie si potesse un po' migliorare le prestazioni, vediamo un esempio:

```
import time
import numpy as np

#inizio a misurare il tempo
start = time.time()


a1 = 0       #variabile che conterra' il risultato
N = int(5e6) # numero di iterazioni da fare = 5 x 10**6

#faccio il conto a 'mano'
for i in range(N):
    a1 += np.sqrt(i)

#finisco di misurare il tempo
end = time.time()-start

print(end)

#inizio a misurare il tempo
start = time.time()

#stesso conto ma fatto tramite le librerie di python
a2 = sum(np.sqrt(np.arange(N)))

#finisco di misurare il tempo
end = time.time()-start

#sperabilmente sara' minore del tempo impiegato prima
print(end)

[Output]
11.588378429412842
0.8475463390350342
```

Vediamo che quindi usando le funzioni di numpy, (np.arange) e le funzioni della libreria standard di Python (sum), è possibile fare lo steso conto in un tempo molto minore che tramite un ciclo for. Questo perché le librerie non sono totalmente in Python ma in molta parte in C e/o fortran.

6 Quarta lezione

6.1 Importare file Python

Abbiamo visto come utilizzare le librerie, tutto a partire dal comando import. Oltre alle librerie possiamo importare anche altri file Python scritti da noi, magari perchè in quel file è implementata una funzione che ci serve. Facciamo un esempio:

```
def f(x, n):
    """
    restituisce la potenza n-esima di un numero x
    Parametri
    ---------
    x, n : float

    Return
    ---------
    v : float
        x**n
    """

    v = x**n

    return v

if __name__ == '__main__':
    #test
    print(f(5, 2))

[Output]
25
```

Abbiamo questo codice che chiamiamo "elevamento.py" che ha implementato la funzione di elevamento a potenza e supponiamo di voler utilizzare questa funzione in un altro codice, possiamo farlo grazie ad import:

```
import elevamento

print(elevamento.f(3, 3))

[Output]
27
```

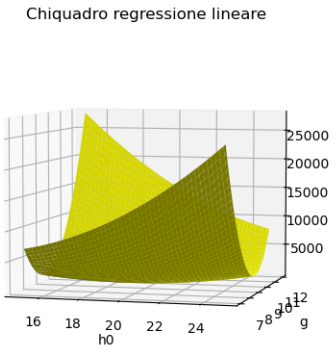
Notiamo nel codice iniziale la presenza dell' if, esso serve per far si che tutto ciò che sia scritto sotto venga eseguito solo se il codice viene lanciato come 'main' appunto e non importato come modulo su un altro codice. In genere l'utilizzo di questa istruzione è buona norma quando si vuol scrivere un codice da importare altrove.

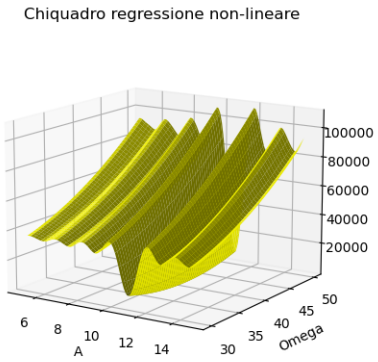
6.2 Fit

Nell'ambito della statistica un fit, cioè una regressione lineare o non che sia (dove la linearità è riferita ai parametri della funzione), è un metodo per trovare la funzione che meglio descrive l'andamento di alcuni dati. Nel caso di regressione lineare la procedura da eseguire non è troppo complicata, mentre per la regressione non lineare le cose si fanno parecchio complicate e si utilizzano algoritmi di ottimizzazione. Se noi abbiamo quindi un modello teorico che ci dice che un corpo cade con una legge oraria della forma $y(t) = h_0 - \frac{1}{2}gt^2$, grazie al fit possiamo trovare i valori dei parametri della leggere oraria, h_0 e g, che meglio adattano la curva ai dati (nella speranza che escano valori fisicamente sensati, dato che in genere i dati sono di origine sperimentale o simulativa). Nella nostra pigrizia deleghiamo tutto il da fare alla funzione "scipy.optimaze.curve_fit()". In ogni caso comunque l'idea di ciò che va fatto è trovare il minimo della seguente funzione:

$$S^2(\{\theta_i\}) = \sum_i \frac{(y_i - f(x_i; \{\theta_i\}))^2}{\sigma_{y_i}^2}$$

che nel caso in chi il termine dentro la somma sia distribuito in modo gaussiano allora la quantità S^2 è distribuita come un chiquadro, e da qui si potrebbe fare tutta una discussione sulla significatività statistica di quello che andiamo a fare, che ovviamente noi non facciamo. Il problema della non linearità fondamentalmente si può esprimere nell'esistenza di minimi locali che potrebbero bloccare il fit dando valori per i parametri θ_i non realistici; mentre per una regressione lineare il minimo è solo uno e assoluto. Prima di vedere il codice vediamo brevemente due grafici della quantità S^2, che con un po' di abuso di notazione chiamiamo chiquadro, nel caso di regressione lineare e non:



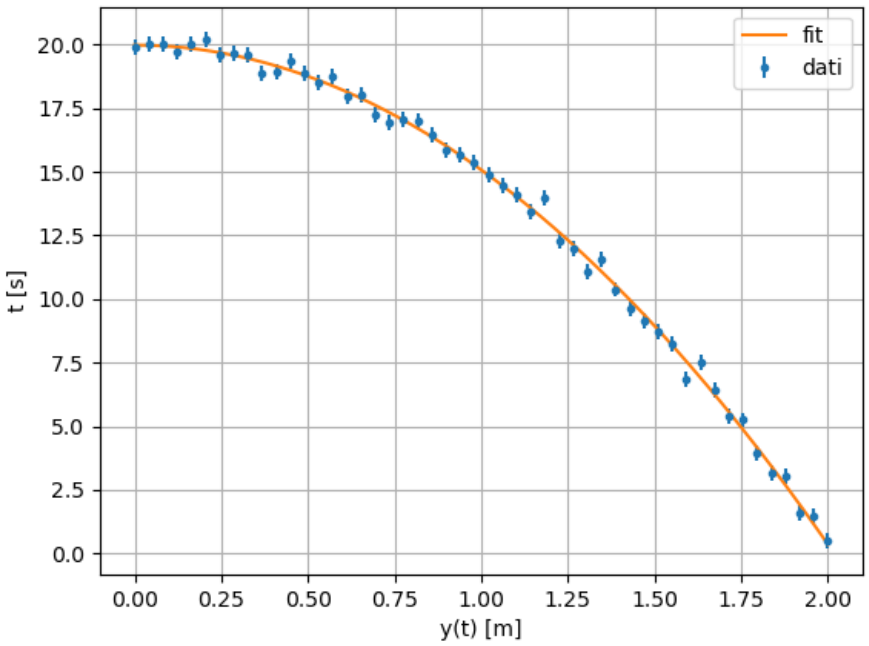


modello: $y(t) = h_0 - \frac{1}{2}gt^2$ modello:$y(t) = Acos(\omega t)$

Vediamo come effettivamente siano presenti nel caso non lineare una serie di minimi locali che sarebbe meglio evitare (i codici per la realizzazione di grafici sono riportati a fine sezione). Vediamo ora un semplice esempio di codice:

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def Legge_oraria(t, h0, g):
    """
    Restituisce la legge oraria di caduta
    di un corpo che parte da altezza h0 e
    con una velocita' inziale nulla
    """
    return h0 - 0.5*g*t**2

"""
dati misurati:
xdata : fisicamemnte i tempi a cui osservo
        la caduta del corpo non affetti da
        errore
ydata : fisicamente la posizione del corpo
        misurata a dati tempi xdata afetta
        da errore
"""

#misuro 50 tempi tra 0 e 2 secondi
xdata = np.linspace(0, 2, 50)

#legge di caduta del corpo
y = Legge_oraria(xdata, 20, 9.81)
rng = np.random.default_rng()
y_noise = 0.3 * rng.normal(size=xdata.size)
#dati misurati afferri da errore
ydata = y + y_noise
dydata = np.array(len(ydata)*[0.3])

#funzione che mi permette di vedere anche le barre d'errore
plt.errorbar(xdata, ydata, dydata, fmt='.', label='dati')

#array dei valori che mi aspetto, circa, di ottenere
init = np.array([15, 10])
#eseguo il fit
popt, pcov = curve_fit(Legge_oraria, xdata, ydata, init, sigma=dydata, absolute_sigma=False)

h0, g = popt
dh0, dg = np.sqrt(pcov.diagonal())
print(f'Altezza inziale h0 = {h0:.3f} +- {dh0:.3f}')
print(f"Accelerazione di gravita' g = {g:.3f} +- {dg:.3f}")

#garfico del fit
t = np.linspace(np.min(xdata), np.max(xdata), 1000)
plt.plot(t, Legge_oraria(t, *popt), label='fit')

plt.grid()
plt.xlabel('y(t) [m]')
```

```
plt.ylabel('t [s]')
plt.legend(loc='best')
plt.show()

[Output]
Altezza inziale h0 = 19.975 +- 0.064
Accelerazione di gravita' g = 9.810 +- 0.070
```



L'utilizzo dell'array init ci aiuta a trovare il minimo assoluto in modo che il codice vada a cercare intorno a quei valori, evitando che il codice si incastri altrove; anche se in questo caso non era necessario in quanto regressione lineare, è comunque buona norma utilizzarlo. Di seguito riportiamo i codici usati per costruire i grafici del chiquadro mostrati sopra:

Caso lineare

```
import numpy as np
import matplotlib.pyplot as plt

def Legge_oraria(t, h0, g):
    """
    Restituisce la legge oraria di caduta
    di un corpo che parte da altezza h0 e
    con una velocita' inziale nulla
    """
    return h0 - 0.5*g*t**2

"""
dati misurati:
xdata : fisicamemnte i tempi a cui osservo
        la caduta del corpo non affetti da
        errore
ydata : fisicamente la posizione del corpo
        misurata a dati tempi xdata afetta
        da errore
"""

#misuro 50 tempi tra 0 e 2 secondi
xdata = np.linspace(0, 2, 50)

#legge di caduta del corpo
y = Legge_oraria(xdata, 20, 9.81)
rng = np.random.default_rng()
y_noise = 0.3 * rng.normal(size=xdata.size)
#dati misurati afferri da errore
```

```
ydata = y + y_noise
dydata = np.array(ydata.size*[0.3])

N = 100
S2 = np.zeros((N, N))
h0 = np.linspace(15, 25, N)
g = np.linspace(7, 12, N)
for i in range(N):
    for j in range(N):
        S2[i, j] = (((ydata - Legge_oraria(xdata, h0[i], g[j]))/dydata)**2).sum()

#grafico del chi quadro
fig = plt.figure(1)
gridx, gridy = np.meshgrid(h0, g)
ax = fig.add_subplot(projection='3d')
ax.plot_surface(gridx, gridy, S2, color='yellow')
ax.set_title('Chiquadro regressione lineare')
ax.set_xlabel('h0')
ax.set_ylabel('g')
plt.show()
```

Caso non lineare

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def Legge_oraria(t, A, omega):
    """
    Restituisce la legge oraria di un corpo che
    oscilla con ampiezza A e frequenza omega
    """
    return A*np.cos(omega*t)

"""""
dati misurati:
xdata : fisicamemnte i tempi a cui osservo
        l'osscilazione del corpo non
        affetti da errore
ydata : fisicamente la posizione del corpo
        misurata a dati tempi xdata afetta
        da errore
"""

#misuro 50 tempi tra 0 e 2 secondi
xdata = np.linspace(0, 2, 50)

#legge di oscillazione del corpo
y = Legge_oraria(xdata, 10, 42)
rng = np.random.default_rng()
y_noise = 0.3 * rng.normal(size=xdata.size)
#dati misurati afferri da errore
ydata = y + y_noise
dydata = np.array(ydata.size*[0.3])

N = 100
S2 = np.zeros((N, N))
A = np.linspace(5, 15, N)
O = np.linspace(30, 50, N)
for i in range(N):
    for j in range(N):
        S2[i, j] = (((ydata - Legge_oraria(xdata, A[i], O[j]))/dydata)**2).sum()

#grafico chiquadro
fig = plt.figure(1)
gridx, gridy = np.meshgrid(A, O)
ax = fig.add_subplot(projection='3d')
ax.plot_surface(gridx, gridy, S2, color='yellow')
ax.set_title('Chiquadro regressione non-lineare')
ax.set_xlabel('A')
ax.set_ylabel('Omega')
plt.show()
```

6.3 Risolvere numericamente le ODE

In fisica è prassi che spuntino fuori equazioni differenziali che non ammettano soluzione analitica; piuttosto che lamentarci di questo ringraziamo quando ciò capita con le ODE, cioè le equazione differenziali ordinarie, perché spesso e volentieri madre natura preferisce l'utilizzo delle equazioni differenziali alle derivate parziali(dette PDE) che in genere da risolvere sono abbastanza più complicate. Qui vedremo semplici esempi per risolvere un'ode. I metodi mostrati saranno per brevità solo due: l'utilizzo delle funzione "odeint()" di scipy e il metodo di eulero, basato sulla definizione di derivata, che mostriamo brevemente: sia $\frac{df(t)}{dt} = g(t, f(t))$ l'equazione da risolvere, allora:

$$\frac{df}{dt} \xrightarrow{discretizzando} \frac{f(t+dt) - f(t)}{dt}$$

Dove la forma ottenuta discretizzando non è altro che il rapporto incrementale di $f(t)$, di cui per definizione la derivata ne è il limite per $dt \to 0$. Sapendo la forma funzionale della derivata, ovvero la $g(t, f(t)$, data dall'equazione differenziale, possiamo ottenere la soluzione dell'equazione per passi:

$$f(t+dt) = f(t) + dtg(t, f(t))$$

quindi possiamo trovare la soluzione al tempo t+dt, sapendo quella al tempo t. inoltre nella g non compare la dipendenza da $f(t+dt)$ ma solo da $f(t)$, per questo il metodo è chiamato esplicito.

6.3.1 Esponenziale

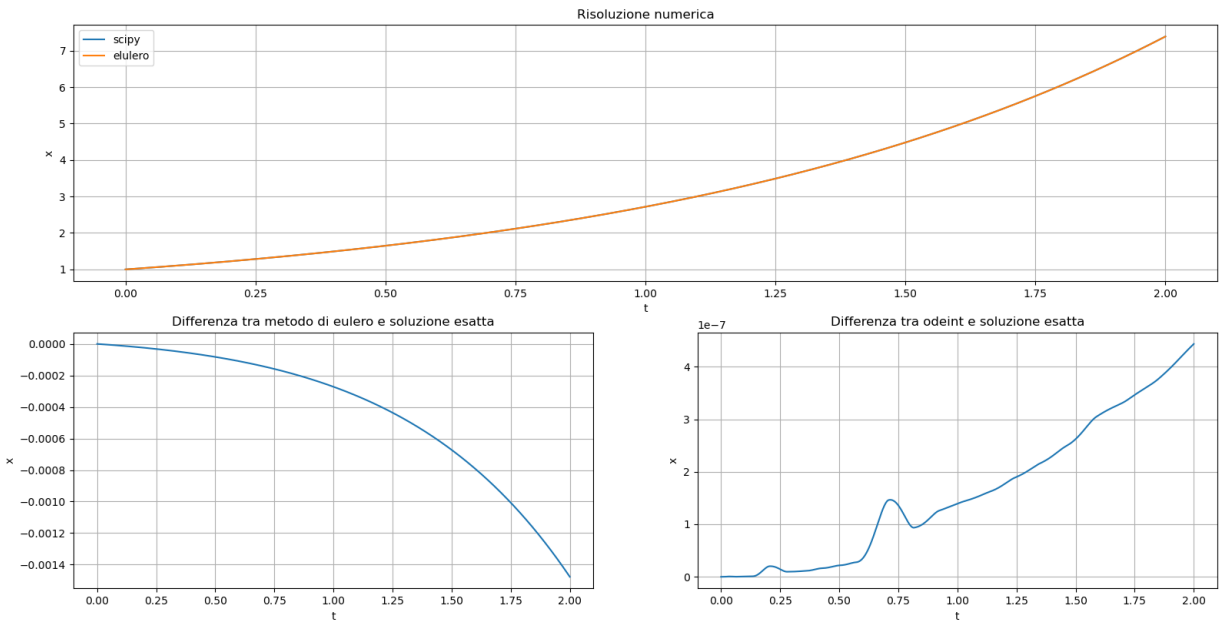
Cominciamo con il problema di Cauchy:

$$\begin{cases} \frac{dx(t)}{dt} = x(t) \\ x(t=0) = 1 \end{cases}$$

Abbiamo una funzione incognita x(t) di cui sapiamo che la derivata è uguale a se stessa e che calcolata in zero restituisce uno. Nella fattispecie la soluzione è semplice, si stratta di un esponenziale crescente, tuttavia vediamo come risolvere numericamente tale equazione.

```
import numpy as np
import scipy.integrate
import  matplotlib.pyplot as plt

#parametri
x0 = 1      #condizione inizale
tf = 2      #fino a dove integrare
N = 10000   #numero di punti

#odeint
def ODE_1(y, t):
    """
    equzione da risolvere per odeint
    """
    x = y
    dydt = x
    return dydt


y0 = [x0] #x(0)
t = np.linspace(0, tf, N+1)
sol = scipy.integrate.odeint(ODE_1, y0, t)

x_scipy = sol[:,0]

#metodo di eulero
def ODE_2(x):
    """
    equzione da risolvere per eulero
    """
    x_dot = x
    return x_dot

def eulero(N, tf, x0):
    """
    si usa che dx/dt = (x[i+1]-x[i])/dt
    che e' praticamente la definizione di rapporto incrementale
    discretizzata la derivata sappiamo a cosa eguagliarla
    perche dx/dt = g(x(t)) nella fattispecie g(x) = x
    quindi discretizzando tutto:
    (x[i+1]-x[i])/dt = x[i]
```

```
    da cui si isola x[i+1]
    """
    dt = tf/N #passo di integrazione
    x = np.zeros(N+1)
    x[0] = x0

    for i in range(N):
        x[i+1] = x[i] + dt*ODE_2(x[i])

    return x

x_eulero = eulero(N, tf, x0)

plt.figure(1)

ax1 = plt.subplot(211)
ax1.set_title('Risoluzione numerica')
ax1.set_xlabel('t')
ax1.set_ylabel('x')
ax1.plot(t, x_scipy, label='scipy')
ax1.plot(t, x_eulero, label='elulero')
ax1.legend(loc='best')
ax1.grid()

ax2 = plt.subplot(223)
ax2.set_title('Differenza tra metodo di eulero e soluzione esatta')
ax2.set_xlabel('t')
ax2.set_ylabel('x')
ax2.plot(t, x_eulero-np.exp(t))
ax2.grid()


ax3 = plt.subplot(224)
ax3.set_title('Differenza tra odeint e soluzione esatta')
ax3.set_xlabel('t')
ax3.set_ylabel('x')
ax3.plot(t, x_scipy-np.exp(t))
ax3.grid()

plt.show()
```



Vediamo che entrambi i metodi sembrano funzionare bene, scipy usa un integratore migliore rispetto ad eulero infatti vediamo che la differenza fra le due soluzioni e dell'ordine di 10^{-7}, ma costruirne uno analogo non è difficile, si può provare con i metodi di Runge-kutta; famoso e molto usato è quello di ordine 4. Abbiamo

risolto un'ode del primo ordine, e per ordine più elevati la cosa è analoga perché con cambi di variabili si può abbassare l'ordine fino ad ottenere un sistema di ode accoppiate di ordine 1;

6.3.2 Pendolo

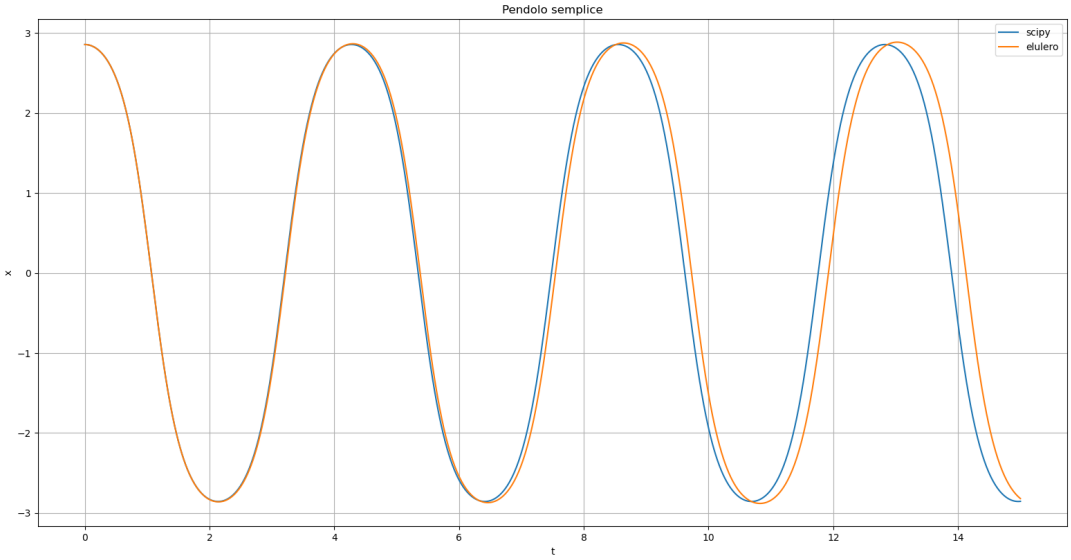
Vediamo un esempio sta volta con un'equazione che non sappiamo risolvere:

$$\begin{cases} \frac{d^2x(t)}{dt^2} = -\frac{l}{g}sin(x(t)) \\ \frac{dx(t)}{dt}|_{t=0} = v_0 \\ x(t=0) = x_0 \end{cases} \quad \Rightarrow \quad \begin{cases} \frac{dx(t)}{dt} = v(t) \\ \frac{dv(t)}{dt} = -\frac{l}{g}sin(x(t)) \\ x(t=0) = x_0 \\ v(t=0) = v_0 \end{cases}$$

È la famosa equazione del pendolo semplice che approssimata dà luogo all'oscillatore armonico ovvero a tutta la fisica. Vediamo come si modifica il codice di sopra ora:

```
import numpy as np
import scipy.integrate
import  matplotlib.pyplot as plt

#parametri
N = 100000      #numero di punti
l = 1           #lunghezza pendolo
g = 9.81        #accellerazione di gravita'
o0 = g/l        #frequenza piccole oscillazioni
v0 = 0          #condizioni iniziali velocita'
x0 = np.pi/1.1  #condizioni iniziali posizione
tf = 15         #fin dove integrare

#odeint
def ODE_1(y, t):
    """
    equzione da risolvere per odeint
    """
    theta, omega = y
    dydt = [omega,  - o0*np.sin(theta)]
    return dydt


y0 = [x0, v0] #x(0), x'(0)
t = np.linspace(0, tf, N+1)
sol = scipy.integrate.odeint(ODE_1, y0, t)

x_scipy = sol[:,0]

#metodo di eulero
def ODE_2(x, v):
    """
    equzione da risolvere per eulero
    """
    x_dot = v
    v_dot = -o0*np.sin(x)
    return x_dot, v_dot

def eulero(N, tf, x0, v0):
    """
    si usa che dx/dt = (x[i+1]-x[i])/dt
    che e' praticamente la definizione di rapporto incrementale
    discretizzata la derivata sappiamo a cosa eguagliarla
    perche dx/dt = g(x(t)) nella fattispecie g(x) = x
    quindi discretizzando tutto:
    (x[i+1]-x[i])/dt = x[i]
    da cui si isola x[i+1]
    """
    dt = tf/N #passo di integrazione
    x = np.zeros(N+1)
    v = np.zeros(N+1)
    x[0], v[0] = x0, v0

    for i in range(N):
        dx, dv = ODE_2(x[i], v[i])
        x[i+1] = x[i] + dt*dx
        v[i+1] = v[i] + dt*dv

    return x, v
```

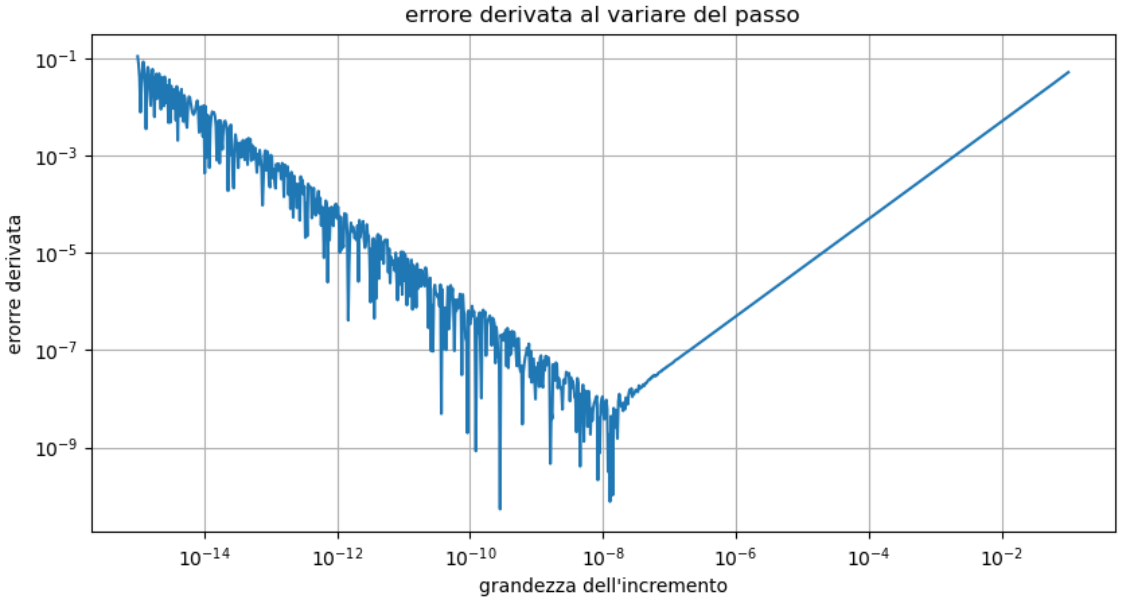
```
60
61 x_eulero, _ = eulero(N, tf, x0, v0)
62
63
64 plt.figure(1)
65
66 plt.title('Pendolo semplice')
67 plt.xlabel('t')
68 plt.ylabel('x')
69 plt.plot(t, x_scipy, label='scipy')
70 plt.plot(t, x_eulero, label='elulero')
71 plt.legend(loc='best')
72 plt.grid()
73
74 plt.show()
```



Dal grafico vediamo le due soluzione distaccarsi, questo è dovuto al fatto che l'integrazione con il metodo di eulero non è delle migliori perché è un metodo del primo ordine e il passo di integrazione non è sufficientemente piccolo; si potrebbe fare tutta una trattazione su come scegliere il passo di integrazione ma va oltre i nostri scopi. Vale la pena sottolineare che come non è buono un passo di integrazione troppo grande nemmeno un troppo piccolo lo è (esistono poi algoritmi adattivi in cui il valore del passo può cambiare durante l'integrazione). Vediamo un esempio di come varia l'errore nel calcolo di una derivata numerica al variare dell'incremento:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def f(x):
5     '''
6     funzione di cui calcolare la derivata
7     '''
8     return np.exp(x)
9
10 def df(f, x, h):
11     """
12     derivata di f
13     """
14     dy = (f(x+h) - f(x))/h
15     return dy
16
17 #array del passo di discretizzazione
18 h = np.logspace(-15, -1, 1000)
19
20 plt.figure(1)
21 plt.title('errore derivata al variare del passo')
22 plt.ylabel('erorre derivata')
23 plt.xlabel("grandezza dell'incremento")
24
```

```
plt.plot(h, abs(df(f, 0, h)-f(0)))

plt.xscale('log')
plt.yscale('log')
plt.grid()
plt.show()
```



vediamo quindi come un passo di 10^{-14} che intuitivamente potremmo credere migliore da lo stesso errore di un passo di 10^{-2}; è un argomento delicato.

6.3.3 Animazione

Abbiamo simulato il movimento del pendolo semplice e abbiamo visto il grafico dell'ampiezza in funzione del tempo ma sarebbe carino riprodurre il movimento del pendolo e creare un'animazione semplice ma comunque realistica. Grazie a matplotlib possiamo farlo senza troppi problemi. Per quanto visto sopra useremo come integratore la funzione "odeint()".

```
import numpy as np
import scipy.integrate
from matplotlib import animation
import  matplotlib.pyplot as plt

#parametri
N = 10000        #numero di punti
l = 1            #lunghezza pendolo
g = 9.81         #accellerazione di gravita'
o0 = g/l         #frequenza piccole oscillazioni
v0 = 0           #condizioni iniziali velocita'
x0 = np.pi/1.1   #condizioni iniziali posizione
tf = 15          #fin dove integrare

#odeint
def ODE_1(y, t):
    """
    equzione da risolvere per odeint
    """
    theta, omega = y
    dydt = [omega,  - o0*np.sin(theta)]
    return dydt


y0 = [x0, v0] #x(0), x'(0)
t = np.linspace(0, tf, N+1)
sol = scipy.integrate.odeint(ODE_1, y0, t)

#passaggio in cartesiane
theta = sol[:,0]
```

```
x = l*np.sin(theta)
y = -l*np.cos(theta)

#grafico e bellurie
fig = plt.figure(1, figsize=(10, 6))
plt.suptitle('Pendolo semplice')
ax = fig.add_subplot(121)
time_template = 'time = %.1fs'
time_text = ax.text(0.05, 0.9, '', transform=ax.transAxes)
plt.xlim(-2, 2)
plt.ylim(-2, 2)
plt.gca().set_aspect('equal', adjustable='box')

#coordinate del perno e della pallina
xf, yf = [0,x[0]],[0,y[0]]

line1, = plt.plot(xf, yf, linestyle='-', marker='o',color='k')

plt.grid()

def animate(i):
    """
    funzione che a ogni i aggiorna le corrdinate della pallina
    """
    xf[1] = x[i]
    yf[1] = y[i]
    line1.set_data(xf, yf)
    time_text.set_text(time_template % (i*t[1]))

    return line1, time_text

#funzione che fa l'animazione vera e propria
anim = animation.FuncAnimation(fig, animate, frames=range(0, len(t), 5), interval=1, blit=True
    , repeat=True)

plt.subplot(122)
plt.ylabel(r'$\theta$(t) [rad]')
plt.xlabel('t [s]')
plt.plot(t, theta)
plt.grid()
plt.show()
```

Provate da voi ad eseguire il codice e vedrete il pendolo oscillare.

A Zeri di una funzione

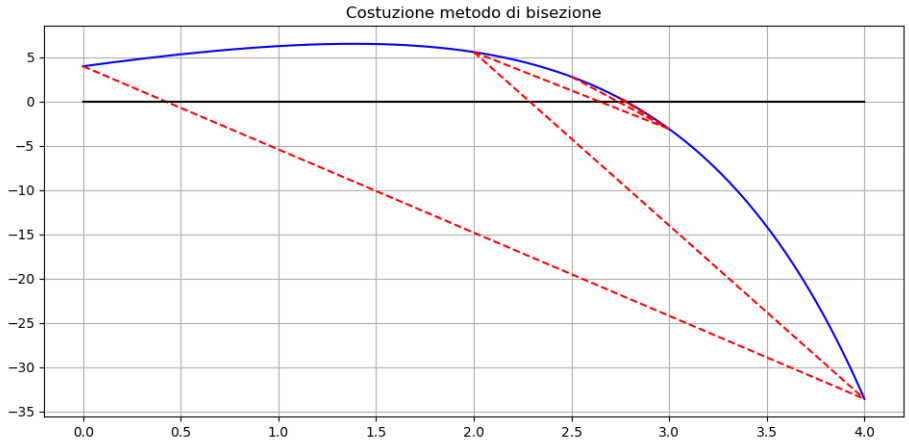
Capita spesso la necessità di trovare gli zeri di una funzione, o più, per risolvere un'equazione o un sistema di equazioni. Brevemente vedremo due metodi per la risoluzione di un' equazione, quindi per trovare lo zero, o gli zeri, di una funzione: il metodo di bisezione e il metodo di Newton, o delle tangenti. Ovviamente tutto ciò può essere fatto con la libreria "scipy.optimize" ma qui vogliamo fare le cose a mano.

A.1 Bisezione

L'algoritmo di bisezione è fondamentalmente una ricerca binaria, e si basa sul teorema degli zeri, ovvero se una funzione è buona quanto basta allora esiste uno zero. Chiaramente quindi dobbiamo più o meno sapere dove cercare perché è necessario che la regione selezionata contenga lo zero. Scelto un intervallo si cerca il punto medio e si valuta in quel punto la funzione, a seconda di una condizione il punto medio diventa il nuovo estremo dell'intervallo, e così via l'intervallo va riducendosi (praticamente la funzione calcolata in un estremo deve avere segno opposto rispetto alla stessa calcolata nell'altro estremo):

```
import numpy as np
import matplotlib.pyplot as plt


def f(x) :
    """
    funzione di cui trovare lo zero
    """
    return 5.0+4.0*x-np.exp(x)

a = 0.0 #estremo sinistro dell'intervallo
b = 4.0 #estremo destro dell'intervallo
t = 1.0e-15 #tolleranza

x=np.linspace(a, b, 1000)
#plot per vedere come scegliere gli estremi
plt.figure(1)
plt.plot(x, f(x))
plt.grid()
plt.show()

##metodo bisezione
fa = f(a)
fb = f(b)
if fa*fb>0:
    print("protrebbero esserci piu' soluzioni" , fa , fb)
"""
Potrebbero esserci piu' zeri anche se la condizione non fosse verificata
Ma se la condizione e' verificata allora di certo ci sono piu' soluzioni
non e' un se e solo se
"""

iter = 1
#fai finche' l'intervallo e' piu' grande della tolleranza
while (b-a) > t:
    c = (a+b)/2.0 #punto medio
    fc = f(c)
    #se hanno lo stesso segno allora c e' piu' vicino allo zero che a
    if fc*fa > 0:
        a = c
    #altrimenti e' b ad essere piu' lontano
    else:
        b = c
    iter += 1

print(iter , " iterazioni necessarie:")
print("x0 = " ,c)
print("accuracy = " , '{:.2e}' .format(b-a))
print("f (x0)=" ,f(c))

[Output]
53  iterazioni necessarie:
x0 =  2.780080782051699
accuracy =  8.88e-16
f (x0)= 7.105427357601002e-15
```

Vediamo graficamente cosa succede:



Per generare il grafico precedente si può fare così:

```
import numpy as np
import matplotlib.pyplot as plt

a = 0.0 #estemo sinistro dell'intervallo
b = 4.0 #estremo destro dell'intervallo
t = 1.0e-15 #tolleranza

plt.figure(2)
plt.title('Costuzione metodo di bisezione')
plt.plot(x, f(x), 'b')
plt.plot([a, b],[f(a), f(b)], linestyle='--', c='r')
plt.plot(x, x*0, 'k')

iter = 1
#fai finche' l'intervallo e' piu' grande della tolleranza
while (b-a) > t:
    c = (a+b)/2.0 #punto medio
    fc = f(c)
    #se hanno lo stesso segno allora c e' piu' vicino allo zero che a
    if fc*fa > 0:
        a = c
    #altrimenti e' b che e' piu' lontano
    else:
        b = c
    iter += 1
    plt.plot([a, b],[f(a), f(b)], linestyle='--', c='r')

plt.grid()
plt.show()
```

Il metodo di bisezione non è il migliore in genere per questo tipo di cose però checché se ne dica funziona sempre, quindi in caso non sappiate che pesci pigliare...

A.2 Metodo di Newton

Se si considera un x_0 molto vicino alla soluzione possiamo espandere in serie di taylor e ottenere:

$$f(s) = 0 = f(x_0) + (x_0 - s)\frac{df}{dx}(x_0) \quad \text{da cui} \quad s = x_0 + \frac{f(x_0)}{\frac{df}{dx}(x_0)}$$

che conduce quini al metodo iterativo:

$$x_{n+1} = x_n + \frac{f(x_n)}{\frac{df}{dx}(x_n)}$$

Nel seguente codice utilizzeremo la libreria sympy che permette di eseguire calcoli analitici. Ovviamente qual ora non sia fattibile la derivata va calcolata numericamente.

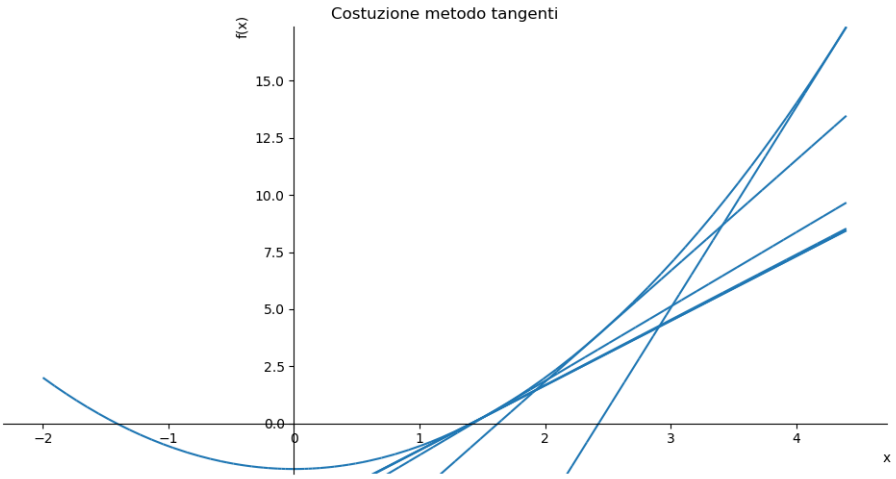
```
import sympy as sp

x = sp.Symbol('x')
f = sp.tan(x)-x #funzione di cui trovare gli zeri
df = sp.diff(f, x) #derivata della funzione f
t = 1e-13 #tolleranza
```

```

def tangenti(x0, t):
    iter = 1
    while abs(f.subs(x, x0))>=t:
        x0 = x0 - ( f.subs(x,x0) / df.subs(x,x0) )
        iter += 1
        if iter > 10000 or abs(f.subs(x, x0))>500:
            if iter > 10000:
                raise Exception('troppe iterazioni')

            if abs(f.subs(x, x0))>500:
                raise Exception('la soluzione sta divergendo\nscegliere meglio il punto di partenza')

    return x0, iter


#valore iniziale da cui partire
init = 4.4


xs, iter = tangenti(init, t)

print(iter , " iterazioni necessarie")


print("xs= %.15f" %xs)

print("|f(xs)|= %e" %abs(f.subs(x,xs)))

[Output]
7  iterazioni necessarie
xs= 4.493409457909064
|f(xs)|= 8.881784e-16
```

Per vedere graficamente cosa succede, cambiamo funzione dato che la tangente è troppo ripida e costruiamo come prima il grafico delle iterazioni. Il codice è il seguente:

```
import sympy as sp
from sympy.plotting import plot

x = sp.Symbol('x')
f = x**2 -2 #funzione di cui trovare gli zeri
df = sp.diff(f, x) #derivata della funzione f
t = 1e-13 #tolleranza
init = 4.4
P1 = plot(f, (x, -2, init), ylim=(-2.2, f.subs(x,init)), show=False, title='Costuzione metodo tangenti')

def tangenti(x0, t):
    iter = 1
    while abs(f.subs(x, x0))>=t:
        P2 = plot(f.subs(x,x0)+(x-x0)*df.subs(x,x0), (x, -2, init), ylim=(-2.2, f.subs(x,init)), show=False)
        P1.extend(P2)
        x0 = x0 - ( f.subs(x,x0) / df.subs(x,x0) )
        iter += 1
        if iter > 10000 or abs(f.subs(x, x0))>500:
            if iter > 10000:
                raise Exception('troppe iterazioni')

            if abs(f.subs(x, x0))>500:
                raise Exception('la soluzione sta divergendo\nscegliere meglio il punto di partenza')

    return x0, iter


#valore iniziale da cui partire
xs, iter = tangenti(init, t)

print(iter , " iterazioni necessarie")
print("xs= %.15f" %xs)
print("|f(xs)|= %e" %abs(f.subs(x,xs)))

P1.show()

```

```
[Output]
7  iterazioni necessarie
xs= 1.414213562373095
|f(xs)|= 4.440892e-16
```



Purtroppo questo metodo può presentare problemi, provate a risolvere l'equazione $x^3 - 2x + 2 = 0$ vi accorgerete che a seconda di dove partite succedono cose strane...

A.3 Zeri in più dimensioni

Ovviamente oltre agli zeri di una singola funzione possiamo anche risolvere un sistema; vedremo sia un 'implementazione manuale, che è il newton raphson, sia un paio di funzioni di scipy. Fondamentalmente newton raphson è come la regola di newton vista sopra, solo che ora x è un vettore e invece della derivata dobbiamo calcolare la matrice delle derivate e invertirla.

$$\mathbf{x}_{n+1} = \mathbf{x}_n - J(\mathbf{x}_n)^{-1} F(\mathbf{x}_n)$$

Dove J è definito come:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \qquad J_{ij} = \frac{\partial f_i(\mathbf{x})}{\partial x_j}.$$

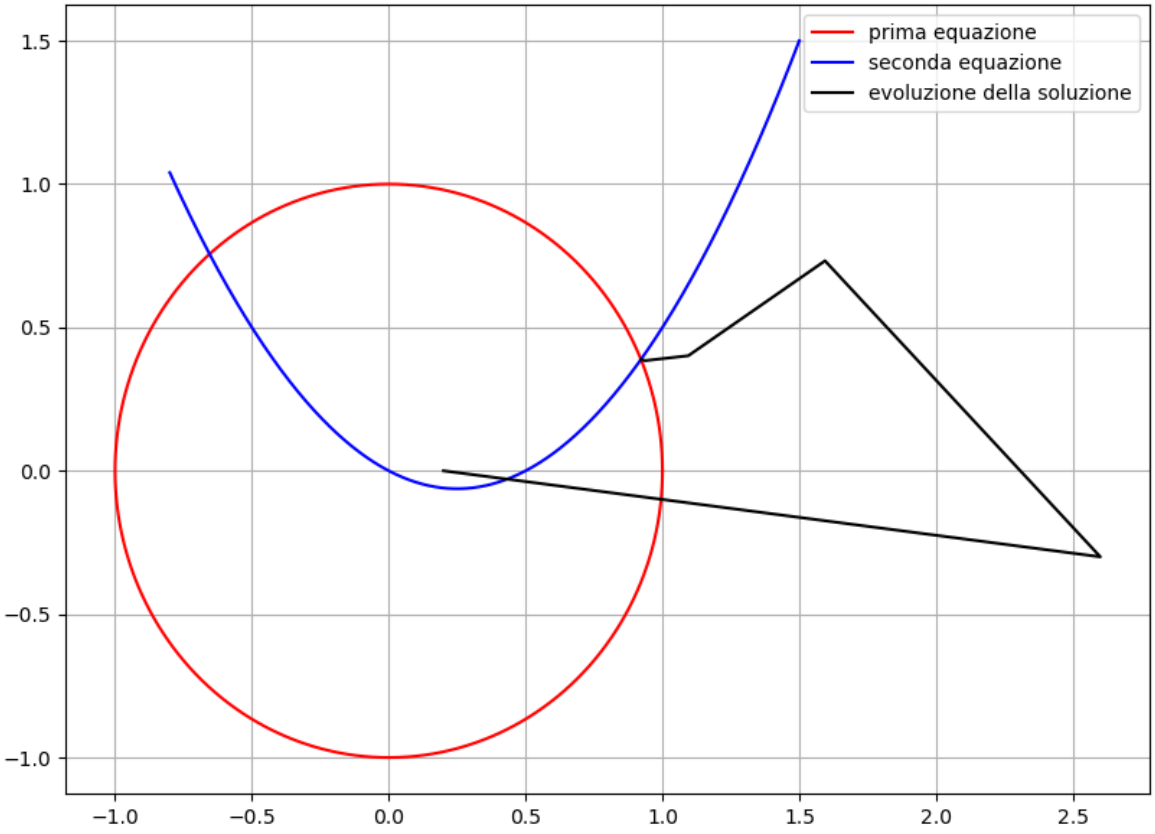
Proviamo un caso semplice in cui invertire la matrice a mano sia facile, quindi una matrice due per due, quindi solo due equazioni:

$$\begin{cases} x^2 + y^2 - 1 = 0 \\ y - x^2 + x/2 = 0 \end{cases}$$

Vediamo il codice con sia implementazione manuale che tramite scipy:

```

import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve, root

## risoluzione "manuale"
x = sp.Symbol('x')
y = sp.Symbol('y')

f1 = x**2 + y**2 - 1
f2 = y - x**2 + x/2

f1x = sp.diff(f1, x)
f1y = sp.diff(f1, y)
f2x = sp.diff(f2, x)
f2y = sp.diff(f2, y)

#valori iniziali da cui partire che vengono
```

```
#ongi volta aggiornati fino ad arrivare alla soluzione
x0 = 0.2
y0 = 0.0

tau = 1e-10
iter = 0
xs = np.array([x0])
ys = np.array([y0])

while (abs(f1.subs(x, x0).subs(y, y0))>tau and abs(f2.subs(x, x0).subs(y, y0)) > tau):
    #calcolo jacobiano
    a11 = f1x.subs(x, x0).subs(y, y0)
    a12 = f2x.subs(x, x0).subs(y, y0)
    a21 = f1y.subs(x, x0).subs(y, y0)
    a22 = f2y.subs(x, x0).subs(y, y0)

    det_a = a11*a22 - a12*a21
    #calcolo inverso
    b11 = a22/det_a
    b12 = -a21/det_a
    b21 = -a12/det_a
    b22 = a11/det_a

    f1_0 = f1.subs(x, x0).subs(y, y0)
    f2_0 = f2.subs(x, x0).subs(y, y0)
    #risolvo
    d1 = -(b11*f1_0 + b12*f2_0)
    d2 = -(b21*f1_0 + b22*f2_0)
    #aggiorno le coordinate
    x0 = x0 + d1
    y0 = y0 + d2
    #conservo per fare il plot
    xs = np.insert(xs, len(xs), x0)
    ys = np.insert(ys, len(ys), y0)

    iter += 1

print(f"x_0: {xs[-1]} e y_0: {ys[-1]} raggiunti in {iter} iterazioni")

t = np.linspace(0, 2*np.pi, 1000)
z = np.linspace(-0.8, 1.5, 1000)

plt.figure(1)
plt.grid()
plt.plot(np.cos(t), np.sin(t), 'r', label='prima equazione')
plt.plot(z, z**2- z/2, 'b', label='seconda equazione')
plt.plot(xs, ys, 'k', label='evoluzione della soluzione')
plt.legend(loc='best')
plt.show()

##risoluzione con fsolve di scipy

def sistema(V):
    x1, x2 = V
    r1 = x1**2 + x2**2 - 1
    r2 = x2 - x1**2 + x1/2
    return[r1 , r2]

start = (0.2, 0.0)
sol = fsolve(sistema , start, xtol=1e-10)
print("soluzione con fsolve:", sol)

##risoluzione con root di scipy

def sistema(V):
    x1, x2 = V
    r1 = x1**2 + x2**2 - 1
    r2 = x2 - x1**2 + x1/2
    return[r1 , r2]

start = (0.2, 0.0)
sol = root(sistema, start, method='hybr')
print("soluzione con root:", sol.x)
```


prima equazione
seconda equazione
evoluzione della soluzione
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5

B Risolvere numericamente le PDE

Come si diceva sopra sono tantissimi i fenomeni fisici che sono descritti da un'equazione differenziale alle derivate parziali, e per non dilungarci nella trattazione, tratteremo solo due esempi: equazione del trasporto ed equazione del calore (per dimensioni 1+1), che possono essere viste come casi specifici della stessa equazione. I metodi che vedremmo, sempre per dare giusto un'infarinatura e lasciare molto all'approfondimento personale, sono l'FCTS (forward time centered space) e il metodo di Lax. I metodi sono entrambi espliciti, ovvero non richiedono la risoluzione di un'equazione algebrica, o di un sistema di equazioni. Sono presenti nei codici delle animazioni per meglio visualizzare la soluzione:

B.1 Equazione del trasporto

L'equazione di nostro interesse è:

$$\frac{\partial u}{\partial t} + v\frac{\partial u}{\partial x} = 0$$

ovviamente ci serve una condizione iniziale $u(x, t = 0)$ per far evolvere il sistema. Per risolverlo si potrebbe pensare il metodo di eulero (notazione: gli apici indicano la dipendenza temporale i pedici quella spaziale) :

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -v\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}$$

dove per approssimare la derivata nello spazio si è utilizzata il metodo delle differenze centrali, più preciso, poiché date le condizioni iniziali sappiamo la soluzione per ogni x ad un dato tempo. Se però per le ode il metodo di Eulero, detto per le PDE: FCTS, funziona praticamente sempre, già in questo esempio il metodo fallisce. Per vederlo si esegue quella che è un'analisi di stabilità, ovvero si sostituisce nella formula di sopra una soluzione del tipo $u_j^n = \xi^n \exp ikj\Delta x$ e si vede che l'ampiezza ξ diverge per ogni scelta di Δt e Δx per risolvere si può usare il metodo di Lax nel quale in termine u_j^n viene sostituito dalla media dei punti spaziali immediatamente accanto:

$$u_j^{n+1} = \frac{u_{j+1}^n + u_{j-1}^n}{2} - v\Delta t\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}$$

Ora il metodo è stabile se $\frac{v\Delta t}{\Delta x} < 1$

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

N = 100     #numero punti sulle x
T = 400     #numero di punti nel tempo
v = 1       #velocita' di propagazione
dt = 0.001  #passo temporale
dx = 0.01   #passo spaziale

alpha = v*dt/dx #<1
print(alpha)

Sol = np.zeros((N+1, T))
sol_v = np.zeros(N+1)
sol_n = np.zeros(N+1)

#condizione iniziale
q = 2*np.pi
x = np.linspace(0, (N+1)*dx, N+1)
sol_v = np.sin(q*1*x)
Sol[:, 0] = sol_v

#evoluzione temporale con lax
for time in range(1, T):
    for j in range(1, N):
        sol_n[j] = 0.5*(sol_v[j+1]*(1 - alpha)) + 0.5*(sol_v[j-1]*(1 + alpha))

    #condizione periodiche al bordo
    sol_n[0] = sol_n[N-1]
    sol_n[N] = sol_n[1]

    #aggiorno la soluzione
    sol_v = sol_n

    #conservo la soluzione per l'animazione
    Sol[:, time] = sol_v
```

```
38
39
40 fig = plt.figure(1)
41 ax = fig.add_subplot(projection='3d')
42 ax.set_title('Equazione trasporto con Lax')
43 ax.set_ylabel('Distanza')
44 ax.set_xlabel('Tempo')
45 ax.set_zlabel('Ampiezza')
46
47 gridx, gridy = np.meshgrid(range(T), x)
48 ax.plot_surface(gridx, gridy, Sol)
49
50 plt.figure(2)
51
52 plt.title('Animazione soluzione', fontsize=15)
53 plt.xlabel('distanza')
54 plt.ylabel('ampiezza')
55 plt.grid()
56 plt.xlim(np.min(x), np.max(x))
57 plt.ylim(np.min(Sol[:,0]) - 0.1, np.max(Sol[:,0]) + 0.1)
58
59 line, = plt.plot([], [], 'b-')
60
61 def animate(i):
62
63     line.set_data(x, Sol[:, i])
64     return line,
65
66 anim = animation.FuncAnimation(fig, animate, frames=np.arange(0, T, 1) ,interval=10, blit=True
       , repeat=True)
67
68 plt.show()
```

Eseguendo il codice è possibile vedere che l'ampiezza dell'onda iniziale va diminuendo, cosa che guardando l'equazione non ci aspetteremmo; ciò è dovuto al fatto che il metodo di Lax può essere visto come un FCTS di un'equazione con un termine diffusivo, ovvero un termine di derivata seconda stile equazione del calore. Vi è quindi un problema di diffusione numerica. Possiamo però risolverlo utilizzando un altro metodo, quello di Lax-Wendroff.

B.2 Equazione del calore

L'equazione del calore è:

$$\frac{\partial u}{\partial t} - D\frac{\partial^2 u}{\partial x^2} = 0$$

Questa volta si può vedere che lo schema FCTS è stabile:

$$u_j^{n+1} = u_j^n + \frac{D\Delta t}{2\Delta x^2}(u_{j+1}^n - 2u_j^n + u_{j-1}^n)$$

La condizione di stabilità è: $\frac{D\Delta t}{\Delta x^2} < \frac{1}{2}$

```
1 import numpy as np
2 import matplotlib as mp
3 import matplotlib.pyplot as plt
4 import matplotlib.animation as animation
5
6 N = 100 #punti sulle x
7 x = np.linspace(0, N, N)
8 tstep = 5000 #punti sul tempo
9 T = np.zeros((N,tstep))
10
11 #Profilo di temperatura iniziale
12 T[0:N,0] = 500*np.exp(-((50-x)/20)**2)
13
14 D = 0.5
15 dx = 0.01
16 dt = 1e-4
17 r = D*dt/dx**2
18 #r < 1/2 affinche integri bene
19 print(r)
20
21 for time in range(1,tstep):
22     for i in range(1,N-1):
23         T[i,time]=T[i,time-1] + r*(T[i-1,time-1]+T[i+1,time-1]-2*T[i,time-1])
```

```
#    T[0,time]=T[1,time] #per avere bordi non fissi
#    T[N-1,time]=T[N-2,time]

fig = plt.figure(1)
ax = fig.gca(projection='3d')
gridx, gridy = np.meshgrid(range(tstep), range(N))
ax.plot_surface(gridx,gridy,T, cmap=mp.cm.coolwarm,vmax=250,linewidth=0,rstride=2, cstride
    =100)
ax.set_title('Diffusione del calore')
ax.set_xlabel('Tempo')
ax.set_ylabel('Lunghezza')
ax.set_zlabel('Temperatura')

fig = plt.figure(2)
plt.xlim(np.min(x), np.max(x))
plt.ylim(np.min(T), np.max(T))

line, = plt.plot([], [], 'b')
def animate(i):
    line.set_data(x, T[:,i])
    return line,


anim = animation.FuncAnimation(fig, animate, frames=tstep, interval=10, blit=True, repeat=True
    )

plt.grid()
plt.title('Diffusione del calore')
plt.xlabel('Distanza')
plt.ylabel('Temperatura')

#anim.save('calore.mp4', fps=30, extra_args=['-vcodec', 'libx264'])

plt.show()
```

C Presa dati da foto

Può capitare che sia interessante prendere dei dati da analizzare, in un qualche modo o maniera, da una foto. Riportiamo quindi un semplice codice che permettere di aprire una foto e salvare su file.txt le coordinate dei pixel, tutto ciò semplicemente cliccando sulla foto (Ogni click che si effettua sulla foto vengono lette e salvate le coordinate del pixel cliccato).

```
import matplotlib as mp
import matplotlib.pyplot as plt


#il file txt su cui scrivere se non esiste viene creato automaticamente

path_dati = "C:\\Users\\franc\\Desktop\\dati0.txt"
path_img = "C:\\Users\\franc\\Documents\\DatiL\\datiL3\\FIS2\\e0verm\\DSC_0005.jpg"

fig, ax = plt.subplots()

img = mp.image.imread(path_img)

ax.imshow(img)


def onclick(event):
    #apre file, il permesso e' a altrimenti sovrascriverebbe i dati
    file= open(path_dati, "a")

    x=event.xdata
    y=event.ydata
    print('x=%f, y=%f' %(x, y)) #stampa i dati sulla shell

    #scrive i dati sul file belli pronti per essere letti da codice del fit
    file.write(str(x))
    file.write('\t')
    file.write(str(y))
    file.write('\n')
    file.close() #chiude il file


fig.canvas.mpl_connect('button_press_event', onclick)


plt.show()
```

D Fit

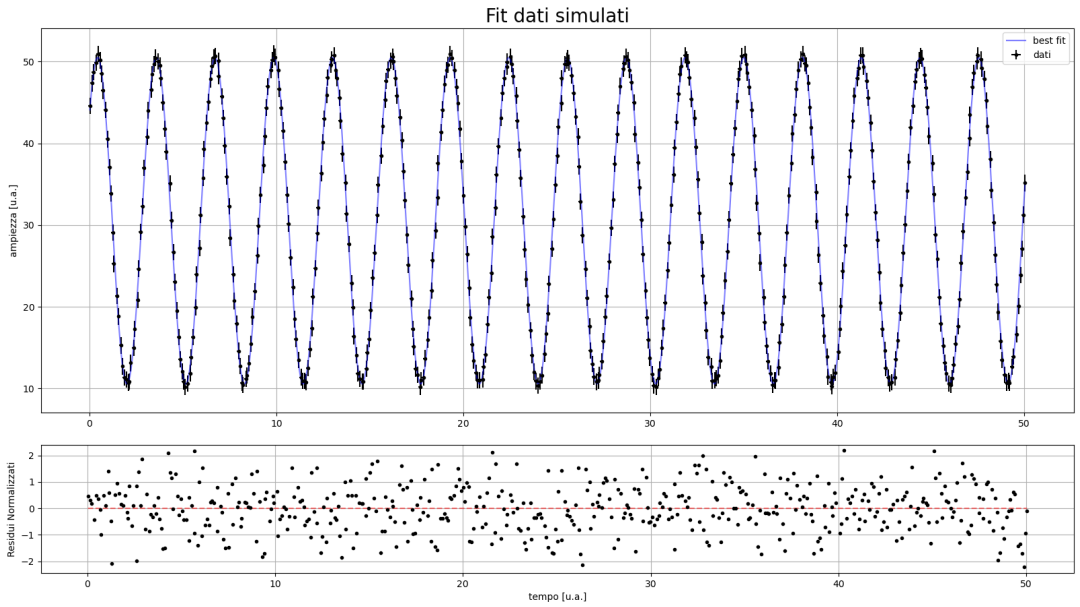
Illustriamo brevemente alcuni modi di eseguire dei fit numerici, cosa in generale in fisica molto utile poiché ci permette di determinare se i dati seguano o meno un certo andamento predetto dalla teoria.

D.1 Fit con scipy

La libreria scipy grazie alla funzione "curve_fit()" ci permette di eseguire un gran numero di fit; riportiamo sotto un esempio di codice e relativi risultati e grafico:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

#Importiamo i dati (va inserito il path assoluto per permettere di trovare) e definiamo la
    funzione di fit:
#x, y= np.loadtxt(r'C:\Users\franc\Desktop\datiL\DatiL2\onda.txt', unpack = True)
N = 500
ex, ey = 0.1, 1
dy = np.array(N*[ey])
dx = np.array(N*[ex])
x = np.linspace(0, 50, N)

A1 = 20
o1 = 2
v1 = 30
phi = np.pi/4

y = A1*np.sin(o1*x + phi) + v1
k = np.random.uniform(0, ey, N)
l = np.random.uniform(0, ex, N)
y = y + k #aggiungo errore
x = x + l

def f(x, A, o, f, v):
    '''funzione modello
    '''
    return A*np.sin(o*x + f) + v

"""
definiamo un array di parametri iniziali contenente
i volori numerici che ci si aspetta il fit restituisca,
per aiutare la convergenza dello stesso:
init = np.array([A, o, f, v])
"""
init = np.array([25, 2.1, 3, 29])


#Eseguiamo il fit e stampiamo i risultati:
pars, covm = curve_fit(f, x, y, init, sigma=dy, absolute_sigma=False)
print('A  = %.5f +- %.5f ' % (pars[0], np.sqrt(covm.diagonal()[0])))
print('o  = %.5f +- %.5f ' % (pars[1], np.sqrt(covm.diagonal()[1])))
print('f  = %.5f +- %.5f ' % (pars[2], np.sqrt(covm.diagonal()[2])))
print('v  = %.5f +- %.5f ' % (pars[3], np.sqrt(covm.diagonal()[3])))

#Calcoliamo il chi quadro,indice ,per quanto possibile, della bonta' del fit:
chisq = sum(((y - f(x, *pars))/dy)**2.)
ndof = len(y) - len(pars)
print(f'chi quadro = {chisq:.3f} ({ndof:d} dof)')


#Definiamo un matrice di zeri che divvera' la matrice di correlazione:
c=np.zeros((len(pars),len(pars)))
#Calcoliamo le correlazioni e le inseriamo nella matrice:
for i in range(0, len(pars)):
    for j in range(0, len(pars)):
       c[i][j] = (covm[i][j])/(np.sqrt(covm.diagonal()[i])*np.sqrt(covm.diagonal()[j]))
print(c) #matrice di correlazione


#Grafichiamo il risultato
fig1 = plt.figure(1)
#Parte superiore contenetnte il fit:
frame1=fig1.add_axes((.1,.35,.8,.6))
#frame1=fig1.add_axes((trasla lateralmente, trasla verticamente, larghezza, altezza))
```

```
frame1.set_title('Fit dati simulati',fontsize=20)
plt.ylabel('ampiezza [u.a.]',fontsize=10)
#plt.ticklabel_format(axis = 'both', style = 'sci', scilimits = (0,0))#notazione scientifica
    sugliassi
plt.grid()

#grafichimao i punti e relative barre d'erroe
plt.errorbar(x, y, dy, dx, fmt='.', color='black', label='dati')
t = np.linspace(np.min(x),np.max(x), 10000)
s = f(t, *pars)
plt.plot(t,s, color='blue', alpha=0.5, label='best fit') #grafico del best fit
plt.legend(loc='best')#inserisce la legenda nel posto migliorte


#Parte inferiore contenente i residui
frame2=fig1.add_axes((.1,.1,.8,.2))

#Calcolo i residui normalizzari
ff = (y-f(x, *pars))/dy
frame2.set_ylabel('Residui Normalizzati')
plt.xlabel('tempo [u.a.]',fontsize=10)
#plt.ticklabel_format(axis = 'both', style = 'sci', scilimits = (0,0))


plt.plot(t, 0*t, color='red', linestyle='--', alpha=0.5) #grafico la retta costantemente zero
plt.plot(x, ff, '.', color='black') #grafico i residui normalizzati
plt.grid()

plt.show()

[Output]
A   = 19.95561 +- 0.05547
o   = 1.99991 +- 0.00019
f   = 6.96973 +- 0.00565
v   = 30.54562 +- 0.03930
chi quadro = 382.795 (496 dof)
[[ 1.          0.00576834 -0.00302643  0.00468354]
 [ 0.00576834  1.         -0.86984711 -0.02110156]
 [-0.00302643 -0.86984711  1.          0.02174393]
 [ 0.00468354 -0.02110156  0.02174393  1.        ]]
```

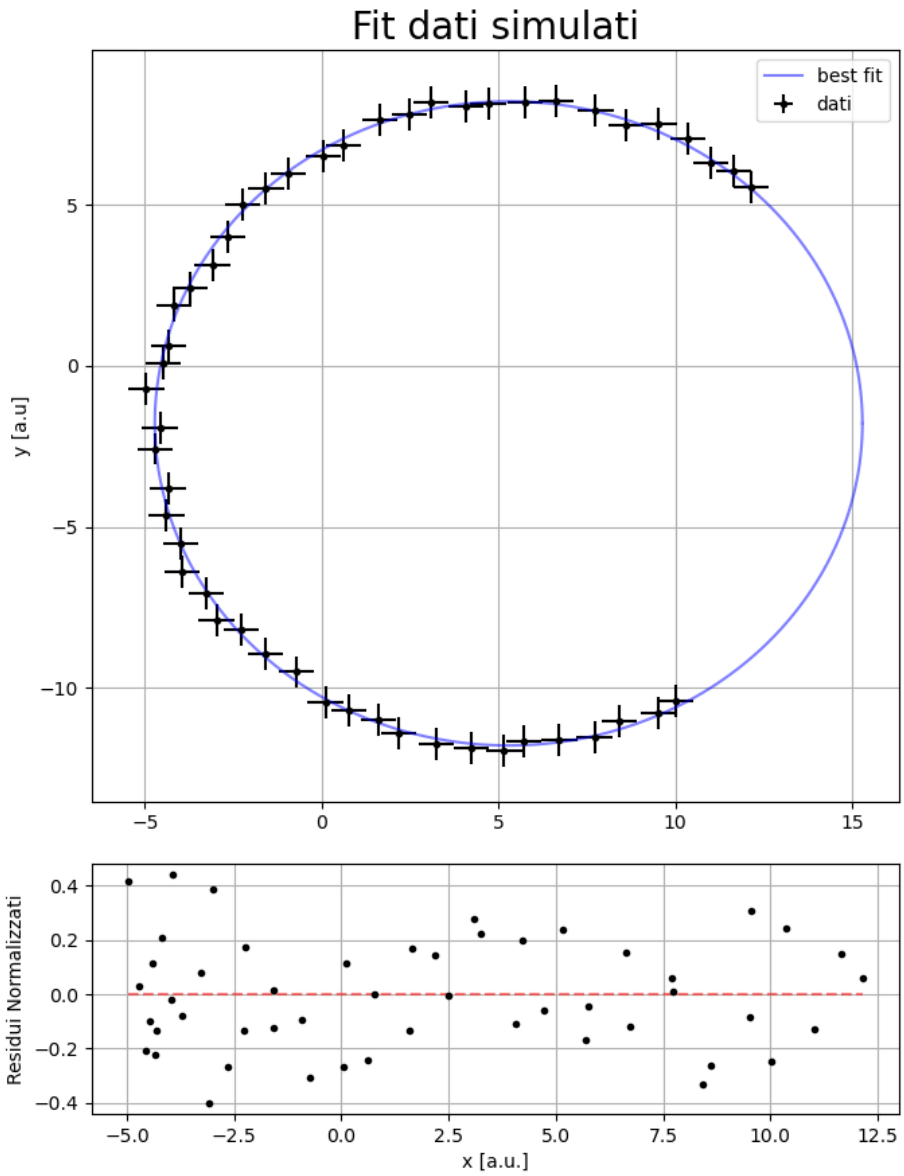


D.2 Fit circolare, metodo di Coope

Ci sono casi in cui, come per un circonferenza o un'ellisse, curve fit non è comodo da usare, in quanto non si tratta di vere e proprie funzioni. Mostraimo un esempio di fit circolare seguito con il metodo di Coope e riportiamo qui il link all'articolo originale: `https://core.ac.uk/download/pdf/35472611.pdf`

```
import numpy as np
import matplotlib.pyplot as plt


def cerchio(xc, yc, r, N, phi_min=0, phi_max=2*np.pi):
    """
    Restituisce un cerchio di centro (xc, yc) e di raggio r
    phi e' il parametro di "percorrenza" del cerchio
    """

    phi = np.linspace(phi_min, phi_max, N)

    x = xc + r*np.cos(phi)
    y = yc + r*np.sin(phi)

    return x, y



def fitcerchio(pt):
    '''
    fit di un cerchio con metodo di coope
    Parameters
    ----------
    pt : 2Darray
        contiene le coordinate del cerchio

    Returns
    -----------
    c : 1Darray
        array con le coordinate del centro del cerchio
    r : float
        raggio del cerchio
    d : 1Darray
        array con gli errori associati a c ed r
    A1 : 2Darray
        matrice di covarianza
    '''
    npt = len(pt[0])
    S = np.column_stack((pt.T, np.ones(npt)))
    y = (pt**2).sum(axis=0)

    A = S.T @ S #@ e' il prodotto matriciale
    b = S.T @ y
    sol = np.linalg.solve(A, b)

    c = 0.5*sol[:-1]
    r = np.sqrt(sol[-1] + c.T @ c)

    d = np.zeros(3)
    A1 = np.linalg.inv(A)

    for i in range(3):
        d[i] = np.sqrt(A1[i,i])
    return c, r, d, A1


if __name__ == "__main__":
    #numero di punti
    N = 50
    #paramentri cerchio
    xc, yc, r1 = 5, -2, 10
    #errori
    ex, ey= 0.5, 0.5
    dy = np.array(N*[ey])
    dx = np.array(N*[ex])
    dr = np.sqrt(dx**2 + dy**2)
    k = np.random.uniform(0, ex, N)
    l = np.random.uniform(0, ey, N)
    #creiamo il cerchio
    x, y = cerchio(xc, yc, r1, N, np.pi/4, 5/3*np.pi)
```

```
    x = x + k #aggiungo errore
    y = y + l

    a = np.array([x, y])
    c, r, d , A = fitcerchio(a) #fit

    print(f'x_c = {c[0]:.5f} +- {d[0]:.5f}; valore esatto={xc:.5f}')
    print(f'y_c = {c[1]:.5f} +- {d[1]:.5f}; valore esatto={yc:.5f}')
    print(f'r   = {r:.5f} +- {d[2]:.5f}; valore esatto={r1:.5f}')


    chisq = sum(((np.sqrt((x-c[0])**2 + (y-c[1])**2) - r)/dr)**2.)
    ndof = N - 3
    print(f'chi quadro = {chisq:.3f} ({ndof:d} dof)')

    corr=np.zeros((3,3))
    for i in range(0, 3):
        for j in range(0, 3):
            corr[i][j]=(A[i][j])/(np.sqrt(A.diagonal()[i])*np.sqrt(A.diagonal()[j]))
    print(corr)

    #plot
    fig1 = plt.figure(1, figsize=(7.5,9.3))
    frame1=fig1.add_axes((.1,.35,.8,.6))
    #frame1=fig1.add_axes((trasla lateralmente, trasla verticamente, larghezza, altezza))
    frame1.set_title('Fit dati simulati',fontsize=20)
    plt.ylabel('y [a.u]',fontsize=10)
    plt.grid()

    plt.errorbar(x, y, dy, dx, fmt='.', color='black', label='dati')
    xx, yy = cerchio(c[0], c[1], r, 10000)
    plt.plot(xx, yy, color='blue', alpha=0.5, label='best fit')
    plt.legend(loc='best')


    frame2=fig1.add_axes((.1,.1,.8,.2))
    frame2.set_ylabel('Residui Normalizzati')
    plt.xlabel('x [a.u.]',fontsize=10)

    ff=(np.sqrt((x-c[0])**2 + (y-c[1])**2) - r)/dr
    x1=np.linspace(np.min(x),np.max(x), 1000)
    plt.plot(x1, 0*x1, color='red', linestyle='--', alpha=0.5)
    plt.plot(x, ff, '.', color='black')
    plt.grid()

    plt.show()

[Output]
x_c = 5.28657 +- 0.02684; valore esatto=5.00000
y_c = -1.78606 +- 0.01830; valore esatto=-2.00000
r   = 10.00627 +- 0.15355; valore esatto=10.00000
chi quadro = 2.075 (47 dof)
[[ 1.          -0.11448697 -0.35569543]
 [-0.11448697  1.          0.19860027]
 [-0.35569543  0.19860027  1.        ]]
```


Fit dati simulati
best fit
dati
y [a.u]
Residui Normalizzati
x [a.u.]

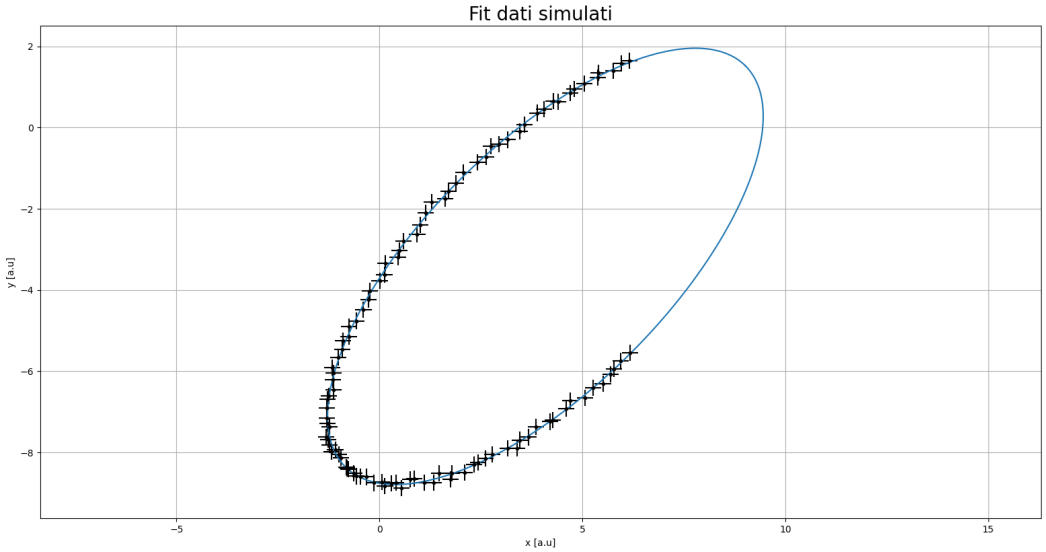
D.3 Fit di un'elissi, metodo di Halir e Flusser

Riportiamo anche un esempio di fit di ellisse basato sull'articolo di Halir e Flusser: `http://autotrace.sourceforge.net/WSCG98.pdf` (n.d.r. è consigliato leggere l'articolo per i vedere i caveat del metodo). Non è riportato il calcolo degli errori sui parametri perché nemmeno nell'articolo è trattato.

```
import numpy as np
import matplotlib.pyplot as plt


def ellisse(parametri, n, tmin=0, tmax=2*np.pi):
    """
    Resistuisce un'ellisse di centro (x0, y0),
    di semiassi maggiore e minore (semi_M, semi_m)
    inclinata di un anglo (phi) rispetto all'asse x
    t e' il parametro di "percorrenza" dell'ellisse
    """

    x0, y0, semi_M, semi_m, phi = parametri
    t = np.linspace(tmin, tmax, n)

    x = x0 + semi_M*np.cos(t)*np.cos(phi) - semi_m*np.sin(t)*np.sin(phi)
    y = y0 + semi_M*np.cos(t)*np.sin(phi) + semi_m*np.sin(t)*np.cos(phi)

    return x, y


def cartesiano_a_polari(coef):
    """
    Converte i coefficenti di: ax^2 + bxy + cy^2 + dx + fy + g = 0
    nei coefficeniti polari: centro, semiassi, inclinazione ed eccentricita'
    Per dubbi sulla geometria: https://mathworld.wolfram.com/Ellipse.html
    """
    #i termini misti presentano un 2 nella forma piu' generale
    a = coef[0]
    b = coef[1]/2
    c = coef[2]
    d = coef[3]/2
    f = coef[4]/2
    g = coef[5]

    #Controlliamo sia un ellisse (i.e. il fit sia venuto bene, forse)
    den = b**2 - a*c
    if den > 0:
        Error = 'I coefficenti passati non sono un ellisse: b^2 - 4ac deve essere negativo'
        raise ValueError(Error)

    #Troviamo il centro dell'ellisse
    x0, y0 = (c*d - b*f)/den, (a*f - b*d)/den

    num = 2*(a*f**2 + c*d**2 + g*b**2 - 2*b*d*f - a*c*g)
    fac = np.sqrt((a - c)**2 + 4*b**2)
    #Troviamo i semiassi maggiori e minori
    semi_M = np.sqrt(num/den/(fac - a - c))
    semi_m = np.sqrt(num/den/(-fac - a - c))

    #Controlliamo che il semiasse maggiore sia maggiore
    M_gt_m = True
    if semi_M < semi_m:
        M_gt_m = False
        semi_M, semi_m = semi_m, semi_M

    #Troviamo l'eccentricita'
    r = (semi_m/semi_M)**2
    if r > 1:
        r = 1/r
    e = np.sqrt(1 - r)

    #Troviamo l'angolo di inclinazione del semiasse maggiore dall'asse x
    #l'angolo come solito e misurato in senso antiorario
    if b == 0:
        if a < c :
            phi = 0
        else:
            phi = np.pi/2

    else:
```

```
        phi = np.arctan((2*b)/(a - c))/2
        if a > c:
            phi += np.pi/2

    if not M_gt_m :
        phi += np.pi/2

    #periodicita' della rotazione
    phi = phi % np.pi

    return x0, y0, semi_M, semi_m, e, phi


def fit_ellisse(x, y):
    """
    Basato sull'articolo di Halir and Flusser,
    "Numerically stable direct
    least squares fitting of ellipses".
    """

    D1 = np.vstack([x**2, x*y, y**2]).T
    D2 = np.vstack([x, y, np.ones(len(x))]).T

    S1 = D1.T @ D1
    S2 = D1.T @ D2
    S3 = D2.T @ D2

    T = -np.linalg.inv(S3) @ S2.T
    M = S1 + S2 @ T
    C = np.array(((0, 0, 2), (0, -1, 0), (2, 0, 0)), dtype=float)
    M = np.linalg.inv(C) @ M

    eigval, eigvec = np.linalg.eig(M)
    cond = 4*eigvec[0]*eigvec[2] - eigvec[1]**2
    ak = eigvec[:, cond > 0]

    return np.concatenate((ak, T @ ak)).ravel()


if __name__ == "__main__":

    #numero di punti
    N = 100
    #parametri dell'ellissi
    x0, y0 = 4, -3.5
    semi_M, semi_m = 7, 3
    phi = np.pi/4
    #eccentricita' non fondamentale per la creazione
    r = (semi_m/semi_M)**2
    if r > 1:
        r = 1/r
    e = np.sqrt(1 - r)

    #errori
    ex, ey = 0.2, 0.2
    dy = np.array(N*[ey])
    dx = np.array(N*[ex])
    #creiamo l'ellisse
    x, y = ellisse((x0, y0, semi_M, semi_m, phi), N, np.pi/4, 3/2*np.pi)
    k = np.random.uniform(0, ex, N)
    l = np.random.uniform(0, ey, N)
    x = x + k #aggiungo errore
    y = y + l

    coef_cart = fit_ellisse(x, y) #fit

    print('valori esatti:')
    print(f'x0:{x0:.4f}, y0:{y0:.4f}, semi_M:{semi_M:.4f}, semi_m:{semi_m:.4f}, phi:{phi:.4f},
     e:{e:.4f}')
    x0, y0, semi_M, semi_m, e, phi = cartesiano_a_polari(coef_cart)
    print('valori fittati')
    print(f'x0:{x0:.4f}, y0:{y0:.4f}, semi_M:{semi_M:.4f}, semi_m:{semi_m:.4f}, phi:{phi:.4f},
     e:{e:.4f}')

    #plot
    plt.figure(1)
```

```
    plt.title('Fit dati simulati',fontsize=20)
    plt.ylabel('y [a.u]',fontsize=10)
    plt.xlabel('x [a.u]',fontsize=10)
    plt.axis('equal')
    plt.errorbar(x, y, dy, dx, fmt='.', color='black', label='dati')
    x, y = ellisse((x0, y0, semi_M, semi_m, phi), 1000)
    plt.plot(x, y)
    plt.grid()
    plt.show()

[Output]
valori esatti:
x0:4.0000, y0:-3.5000, semi_M:7.0000, semi_m:3.0000, phi:0.7854, e:0.9035
valori fittati
x0:4.0888, y0:-3.4169, semi_M:6.9755, semi_m:2.9975, phi:0.7859, e:0.9030
```



E Metodi Montecarlo

In molte simulazioni di interesse fisico è necessario dover generare numeri casuali, o quanto meno pseudo casuali. La generazione di numeri casuali è effettivamente sempre argomento di ricerca per riuscire a raggiungere sempre un livello di casualità maggiore; esistono poi in letteratura esempi di buoni generatori che però in certe simulazioni falliscono, dando risultati fisicamente molto poco sensati. Insomma è un'argomento abbastanza delicato. Non potendone parlare nel dettaglio vedremmo brevemente un esempio di generatore di numeri casuali e poi una simulazione vera e propria con l'utilizzo però di librerie di python apposite (sia la libreria numpy che la libreria random, sono molto utili nella generazione di numeri random).

E.1 Generatori numeri pseudo-casuali

Uno dei modi più famosi di costruire un generatore è secondo uno schema che ha in nome di: generatore congruenziale lineare. Dato un certo seme x_0 posso generare il numero x_1 e da questo x_2 e via seguendo. Lo schema generale è:

$$x_{n+1} = (ax_n + c) \mod M$$

dove a, c e M, detti rispettivamente: moltiplicatore, incremento e modulo sono dei numeri scelti con più o meno cura. Vediamo un esempio:

```
import numpy as np

def GEN(r0, n=1, M=2**64, a=6364136223846793005, c=1442695040888963407, norm=True):
    """
    generatore conguenziale lineare
    Parametri
    ---------
    r0 : int
        seed della generazione
    n : int, opzionale
        dimensione lista da generare, di default e' 1
    M : int, opzionale
        periodo del generaltore di default e' 2**64
    a : int, opzionale
        moltiplicatore del generatore, di default e' 6364136223846793005
    c : int, opzionale
        incremento del generatore, di default e' 1442695040888963407
    norm : bool, opzionale
        se True il numero restituito e' fra zero ed 1

    Returns
    ---------
    r : list
        lista con numeri distribuiti casualmente
    """
    if n==1:
        r = (a*r0 + c)%M
    else:
        r = []
        x = r0
        for i in range(1, n):
            x = (a*x + c)%M
            r.append(x)

    if norm :
        if n==1:
            return float(r)/(M-1)
        else :
            return [float(el)/(M-1) for el in r]
    else :
        return r

if __name__ == '__main__':
    seed = 42

    print(GEN(seed, n=5))
    momenti1 = [np.mean(np.array(GEN(seed, n=int(5e5)))**i) for i in range(1, 10)]
    momenti2 = [1/(1+p) for p in range(1, 10)]

    for M1, M2 in zip(momenti1, momenti2):
        print(f'{M1:.3f}, {M2:.3f}')

[Output]
```

```
[0.5682303266439077, 0.22546342894775137, 0.41283831882951183, 0.6303980498395979]
0.500, 0.500
0.333, 0.333
0.250, 0.250
0.200, 0.200
0.167, 0.167
0.143, 0.143
0.125, 0.125
0.111, 0.111
0.100, 0.100
```

Quello che si fa a Riga 47 è il calcolo dei primi momenti della distribuzione uniforme con il nostro generatore; alla riga successiva troviamo i momenti analitici, da confrontare con quelli da noi calcolati per fare un piccolo test sulla bontà del generatore.

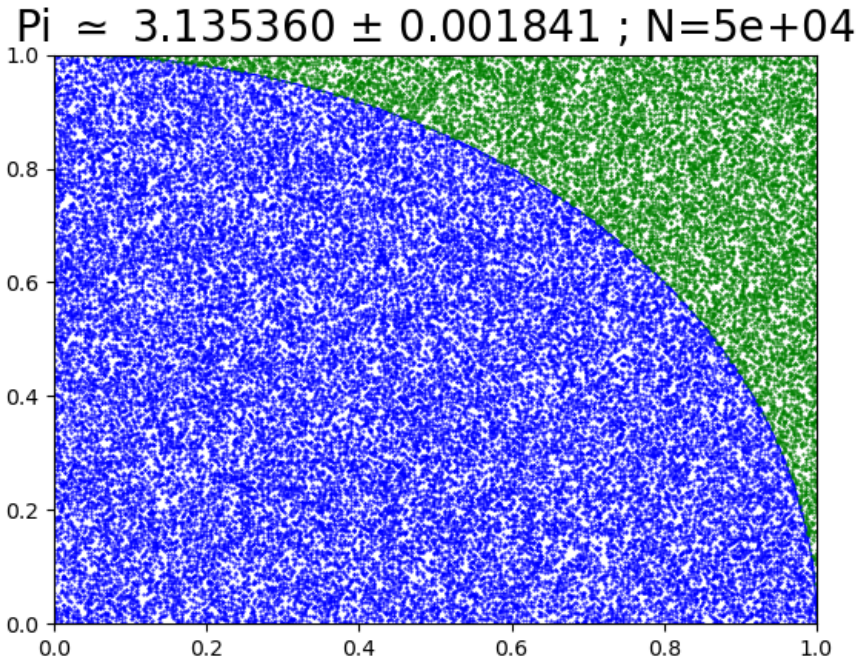
E.2 Calcolo di Pi greco

Prendete dei coriandoli e buttateli a caso su una mattonella con un cerchio disegnato sopra e contando quanti sono dentro al cerchio rispetto al totale avete calcolato π. Fondamentalmente quel che si fa è il calcolo di un'area (i.e. un'integrale) e benché ci siano modi più efficienti il calcolo di π è un classico esempio e quindi non mancheremo di esporlo. La particolarità di usare un metodo Monte-Carlo infatti si vede in alte dimensioni poiché a differenza dei possibili metodi di integrazione che uno può inventarsi l'errore dato da Monte-Carlo non dipende dalla dimensione ma va sempre come $1/\sqrt{N}$. Per comodità l'esempio è fatto su un quarto di circonferenza quindi la probabilità che il coriandolo sia dentro è $\pi/4$.

```
import time
import numpy as np
import matplotlib.pyplot as plt

N = int(5e4)
start_time=time.time()

x = np.linspace(0,1, 10000)

def f(x):
    """
    semi circonferenza superiore
    """
    return np.sqrt(1-x**2)

c = 0

for i in range(1,N):
    #genero due variabili casuali uniformi fra 0 e 1
    a = np.random.rand()
    b = np.random.rand()
    r = a**2 + b**2
    #se vero aggiorno c di 1
    if r < 1:
        plt.errorbar(a, b, fmt='.', markersize=1, color='blue')
        c += 1
    else:
        plt.errorbar(a, b, fmt='.', markersize=1,  color='green')

#moltiplico per quattro essendo su un solo quadrante
Pi = 4*c/N
#propagazione errore, viene dalla binomiale
dPi = np.sqrt(c/N * (1-c/N))/np.sqrt(N)
print('%f +- %f' %(Pi, dPi))
print(np.pi)
print(abs((Pi-np.pi)/np.pi))

plt.figure(1)
plt.title('Pi $\simeq$ %f $\pm$ %f ; N=%.0e' %(Pi, dPi, N), fontsize=20)
plt.xlim(0,1)
plt.ylim(0,1)
plt.plot(x, f(x),color='blue', lw=1)
plt.show()

print("--- %s seconds ---" % (time.time() - start_time))

[Output]
3.135360 +- 0.001841
3.1415926535897793
```

```
0.001983915254790065
--- 29.91716456413269 seconds ---
```

Cambiando N si può controllare quanto velocemente questo metodo converga, e si vedrà che non è velocissimo ma va beh, come dicevamo prima siamo solo in due dimensioni. Abbiamo usato la libreria time per misurare il tempo impiegato ma in realtà molto del tempo va nella costruzione del grafico, senza di esso e con lo stesso numero di punti otteniamo lo stesso risultato in 0.08 secondi.



F Propagazione errori

Può capitare spesso che vadano propagati degli errori, purtroppo. A Laboratorio 1 si vede che il modo di propagarli è fare le derivate, e in casi più semplici ci sono dei trucchetti. Noi per andare sul sicuro faremo sempre le derivate, dove il guaio è che è facile sbagliare i calcoli, ma per fortuna noi li facciamo fare al computer.

F.1 Propagarli a mano

Volendo scrivere un breve codice facile da modifcare all'occorenza si potrebbe provare così:

```
import numpy as np
import sympy as sp

x = sp.Symbol('x')
y = sp.Symbol('y')
z = sp.Symbol('z')
t = sp.Symbol('t')

def Errore(x1, dx1, y1, dy1, z1, dz1, t1, dt1):
    """
    Prende in input certe quantita' con un errore
    e propaga l'errore su una certa funzione di queste
    """
    #funzione su cui propagare l'errore da modifica all'occorenza
    f1 = ((x-y)/(z+t))

    #valor medio
    f = float(f1.subs(x,x1).subs(y,y1).subs(z,z1).subs(t,t1))

    #derivate parziali calcolate nel punto
    a = sp.diff(f1, x).subs(x,x1).subs(y,y1).subs(z,z1).subs(t,t1)
    b = sp.diff(f1, y).subs(x,x1).subs(y,y1).subs(z,z1).subs(t,t1)
    c = sp.diff(f1, z).subs(x,x1).subs(y,y1).subs(z,z1).subs(t,t1)
    d = sp.diff(f1, t).subs(x,x1).subs(y,y1).subs(z,z1).subs(t,t1)

    #somma dei vari contributi
    df1 = ((a*dx1)**2 + (b*dy1)**2+ (c*dz1)**2 + (d*dt1)**2 )
    df = np.sqrt(float(df1))

    return f, df


print(Errore(1, 0.1, 2, 0.1, 3, 0.1, 2, 0.1))

[Output]
(-0.2, 0.02884441020371192)
```

F.2 Uncertainties

Oppure volendo si potrebbe usare questa comoda libreria:

```
from uncertainties import ufloat
import uncertainties.umath as um

#il pimo argomento e' il valore centrale, il secondo l'errore
x = ufloat(7.1, 0.2)
y = ufloat(12.3, 0.7)


print(x)
print(2*x-y)
print(um.log(x**y))

[Output]
7.10+/-0.20
1.9+/-0.8
24.1+/-1.4
```

Vediamo ora un riassunto della propagazione degli errori:

$$\text{PRECISE NUMBER} + \text{PRECISE NUMBER} = \text{SLIGHTLY LESS PRECISE NUMBER}$$

$$\text{PRECISE NUMBER} \times \text{PRECISE NUMBER} = \text{SLIGHTLY LESS PRECISE NUMBER}$$

$$\text{PRECISE NUMBER} + \text{GARBAGE} = \text{GARBAGE}$$

$$\text{PRECISE NUMBER} \times \text{GARBAGE} = \text{GARBAGE}$$

$$\sqrt{\text{GARBAGE}} = \text{LESS BAD GARBAGE}$$

$$(\text{GARBAGE})^2 = \text{WORSE GARBAGE}$$

$$\frac{1}{N}\sum\left(\text{N PIECES OF STATISTICALLY INDEPENDENT GARBAGE}\right) = \text{BETTER GARBAGE}$$

$$\left(\text{PRECISE NUMBER}\right)^{\text{GARBAGE}} = \text{MUCH WORSE GARBAGE}$$

$$\text{GARBAGE} - \text{GARBAGE} = \text{MUCH WORSE GARBAGE}$$

$$\frac{\text{PRECISE NUMBER}}{\text{GARBAGE} - \text{GARBAGE}} = \text{MUCH WORSE GARBAGE, POSSIBLE DIVISION BY ZERO}$$

$$\text{GARBAGE} \times \bigcirc = \text{PRECISE NUMBER}$$

G Interpolazione

Nel mondo della fisica computazionale, ad esempio nel mondo dell'astrofisica computazionale, capita spesso che alcune cose siano tabulate. Ovvero per fare una qualche simulazione si prendono delle certe quantità a loro volta frutto in genere di simulazioni e che quindi sono date per passi; se però fossimo interessati ad analizzare determinati valori magari in un range con un passo più piccolo della tabella, dobbiamo necessariamente interpolare, per cercare di capire cosa succede tra i due punti nella tabella.

G.1 Interpolazione lineare

Il modo più semplice è unire i punti con una retta, ovvero eseguire un'interpolazione lineare. Dati due punti consecutivi nella tabella indicati come (x_i, y_i) e (x_{i+1}, y_{i+1}) l'interpolazione lineare non fa altro che assegnare ad ogni valore di x compreso nell'intervallo $[x_i, x_{i+1}]$ la media ponderata tra y_i e y_{i+1}.

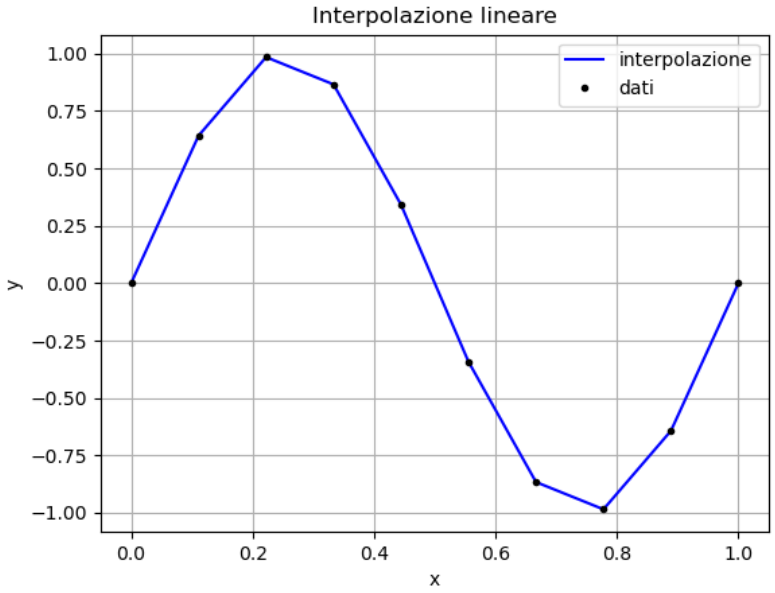
$$f(x) = \frac{x_{i+1} - x}{x_{i+1} - x_i} y_i + \frac{x - x_i}{x_{i+1} - x_i} y_{i+1}$$

L'espressione precedente non è altro quindi che la retta che unisce i due punti. Vediamo il codice:

```
import numpy as np
import matplotlib.pyplot as plt

def f(x, xx, yy):
    """
    restituisce l'interpolazione dei punti xx yy
    x puo' essere un singolo valore in cui calcolare
    la funzione interpolante o un intero array
    """
    #proviamo se x e' un array
    try :
        n = len(x)
        x_in = np.min(xx) <= np.min(x) and np.max(xx) >= np.max(x)
    except TypeError:
        n = 1
        x_in = np.min(xx) <= x <= np.max(xx)

    #se il valore non e' nel range corretto e' impossibile fare il conto
    if not x_in :
        a = 'uno o diversi valori in cui calcolare la funzione'
        b = ' interpolante sono fuori dal range di interpolazione'
        errore = a+b
        raise Exception(errore)

    #array che conterra' l'interpolazione
    F = np.zeros(n)

    if n == 1 :
        #controllo dove e' la x e trovo l'indice dell'array
        #per sapere in che range bisogna interpolare
        for j in range(len(xx)-1):
            if xx[j] <= x <= xx[j+1]:
                i = j

        A = yy[i] * (xx[i+1] - x)/(xx[i+1] - xx[i])
        B = yy[i+1] * (x - xx[i])/(xx[i+1] - xx[i])
        F[0] = A + B

    else:
        #per ogni valore dell'array in cui voglio calcolare l'interpolazione
        for k, x in enumerate(x):
            #controllo dove e' la x e trovo l'indice dell'array
            #per sapere in che range bisogna interpolare
            for j in range(len(xx)-1):
                if xx[j] <= x <= xx[j+1]:
                    i = j

            A = yy[i] * (xx[i+1] - x)/(xx[i+1] - xx[i])
            B = yy[i+1] * (x - xx[i])/(xx[i+1] - xx[i])
            F[k] = A + B

    return F

if __name__ == '__main__':
    x = np.linspace(0, 1, 10)
```

```
    y = np.sin(2*np.pi*x)
    z = np.linspace(0, 1, 100)

    plt.figure(1)
    plt.title('Interpolazione lineare')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.plot(z, f(z, x, y), 'b', label='interpolazione')
    plt.plot(x, y, marker='.', linestyle='', c='k', label='dati')
    plt.legend(loc='best')
    plt.grid()
    plt.show()
```

La funzione scritta prende due array "xx" e "yy" che sono i dati da interpolare, e una veriabile "x" che può essere un singolo punto dell'intervallo o un intero array per ottenere una curva. Si è usato un try except perché chiaramente Python da errore se si calcola la lunghezza di un numero. Vi è poi il sollevamento di un'eccezione in caso i valori di interesse siano fuori dagli estremi della tabella dove non si può dire nulla quindi il codice si interrompe. Vediamo il risultato:



G.2 Interpolazione Polinomiale

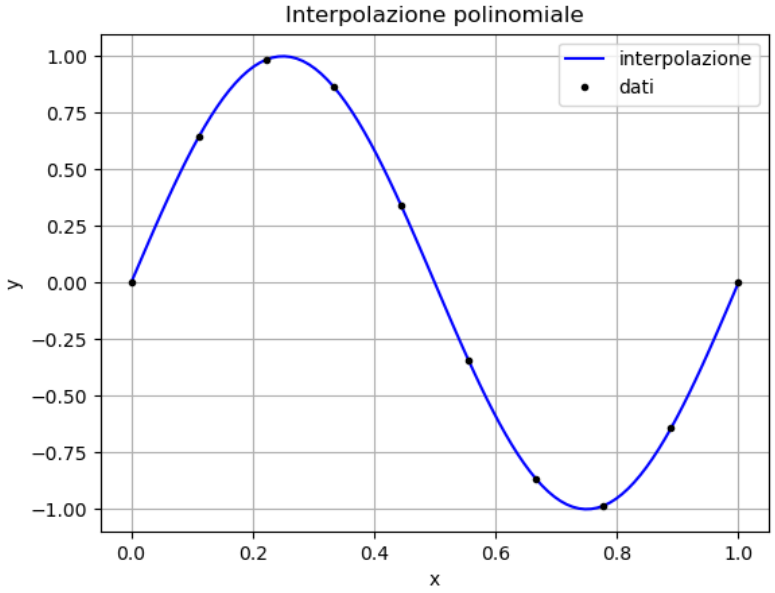
Un altro modo per interpolare è usare un polinomio di grado $n-1$ per n punti e si può fare facilmente con la matrice di Vandermonde, il problema è che tale matrice è mal condizionata, quindi non funziona sempre benissimo e il costo computazionale è alto dato che bisogna invertire una matrice.

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \dots & x_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \dots & x_m^{n-1} \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ \vdots \\ s_{n-1} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix}$$

Dove le x e le y sono i nostri dati tabulati e gli s_i sono i coefficienti del polinomio. Vediamo un semplice esempio di codice:

```
import numpy as np
import matplotlib.pyplot as plt

N = 10
x = np.linspace(0, 1, N)
y = np.sin(2*np.pi*x)

#Matrice di Vandermonde
A = np.zeros((N, N))
A[:,0] = 1
for i in range(1, N):
    A[:,i] = x**i
```

```
13
14 #risolvo il sistema, la soluzione sono i coefficenti del polinomio
15 s = np.linalg.solve(A, y)
16
17
18 def f(s, zz):
19     '''
20     funzione per fare il grafico
21     '''
22     n = len(zz)
23     y = np.zeros(n)
24     for i , z in enumerate(zz):
25         y[i] = sum([s[j]*z**j for j in range(len(s))])
26     return y
27
28 z = np.linspace(0, 1, 100)
29
30 plt.figure(1)
31 plt.title('Interpolazione polinomiale')
32 plt.xlabel('x')
33 plt.ylabel('y')
34 plt.plot(z, f(s, z), 'b', label='interpolazione')
35 plt.plot(x, y, marker='.', linestyle='', c='k', label='dati')
36 plt.legend(loc='best')
37 plt.grid()
38 plt.show()
```

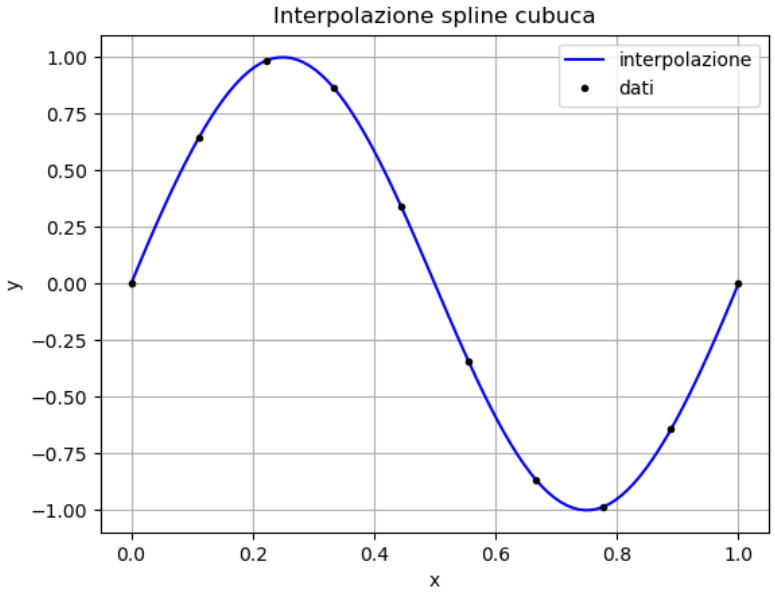


G.3 Scipy.interpolate

Ovviamente esiste una libreria di Python che ci permette facilmente di eseguire le interpolazioni, riportiamo un semplice esempio (ricordando sempre che il modo migliore per capire a pieno è leggere la documentazione).

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.interpolate import InterpolatedUnivariateSpline
4
5 N = 10
6 x = np.linspace(0, 1, N)
7 y = np.sin(2*np.pi*x)
8
9 #interpolazione con una, spline cucbica (k=3)
10 s3 = InterpolatedUnivariateSpline(x, y, k=3)
11
12 z = np.linspace(0, 1, 100)
13
14 plt.figure(1)
15 plt.title('Interpolazione spline cubica')
16 plt.xlabel('x')
```

```
17 plt.ylabel('y')
18 plt.plot(z, s3(z), 'b', label='interpolazione')
19 plt.plot(x, y, marker='.', linestyle='', c='k', label='dati')
20 plt.legend(loc='best')
21 plt.grid()
22 plt.show()
```



H Programmazione a oggetti

Python è un linguaggio che permette la programmazione ad oggetti. Molto di Python stesso è scritto con programmazione orientata ad oggetti. Al di là dello spiegare cosa è effettivamente la programmazione ad oggetti, ci limiteremo ad illustrare semplici esempi. Quello che faremo è utilizzare le classi, fondamentalmente creare una classe vuol dire definire un oggetto. All'interno della classe è possibile definire delle funzioni che verranno chiamati metodi. Vediamo un semplice esempio:

```
import numpy as np

class Pallina:
    """
    Classe che rappresenta una pallina
    intesa come oggetto puntiforme
    """

    def __init__(self, x, y, vx, vy, m):
        """
        costruttore della classe, verra' chiamato
        quando creeremo l'istanza della classe
        in input prende la posizione, la velocita' e massa
        che sono le quantita' che identificano la pallina
        che saranno gli attributi della classe;
        il costruttore e' un particolare metodi della
        classe per questo si utilizzano gli underscore.
        il primo parametro che passiamo (self) rappresenta
        l'istanza della classe (self e' un nome di default)
        questo perche' la classe e' un modello generico che
        deve valere per ogni 'pallina'
        """
        #posizione
        self.x = x
        self.y = y
        #velocita'
        self.vx = vx
        self.vy = vy
        self.massa = m

    def energia_cinetica(self):
        """
        ad ogni metodo della classe viene passato
        come primo argomento self, quindi l'istanza
        calcoliamo l'evergia cinetica
        """
        #una volta creati gli attributi non e necessario passarli ai vari metodi
        ene_k = 0.5*self.massa*(self.vx **2 + self.vy**2)
        return ene_k

    def energia_potenziale_gravitazionale(self, g):
        """
        calcolimao l'energia potenziale; all'interno
        di un campo gravitazionale di intensita' g
        la nostra pallina ha energia per il semplice
        fatto di essere in un qualche punto del campo
        """

        #supponiamo g sia diretta verso il basso
        ene_u = self.massa*g*self.y
        return ene_u

g = 9.81 #acc di gravita' che vorrei tanto mettere uguale a 1

#creo l'istanza della classe
p1 = Pallina(1, 1, 2, 2, 1)
#chiamo i metodi sull'istanza
ene_tot_p1 = p1.energia_cinetica() + p1.energia_potenziale_gravitazionale(g)
print('energia pallina 1:', ene_tot_p1)

#creo l'istanza della classe
p2 = Pallina(2, 2, 2, 2, 1)
#chiamo i metodi sull'istanza
ene_tot_p2 = p2.energia_cinetica() + p2.energia_potenziale_gravitazionale(g)
print('energia pallina 2:',ene_tot_p2)

[Output]
energia pallina 1: 13.81
```

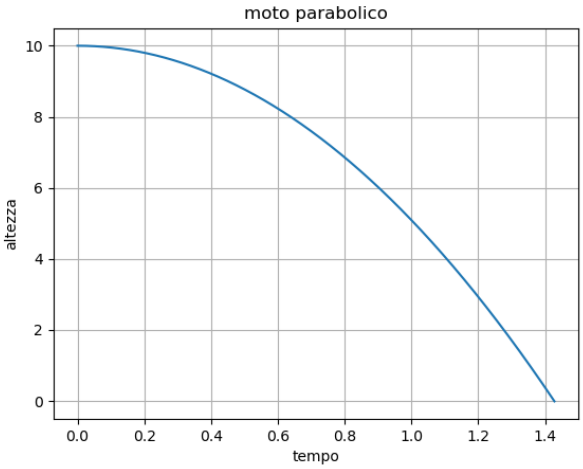
```
energia pallina 2: 23.62
```

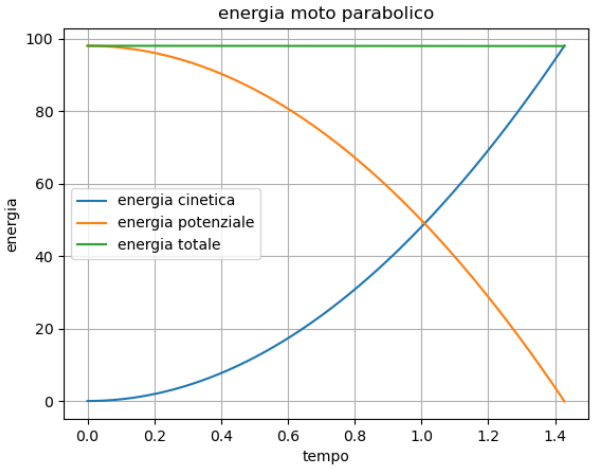
Volendo potremmo creare dei nuovi metodi per aggiornare velocità e posizioni, in modo da magari costruire una dinamica della nostra pallina e ricostruirne il moto. Vediamo il caso semplice di moto di caduta libera:

```python
import numpy as np
import matplotlib.pyplot as plt

class Pallina:
    """
    Classe che rappresenta una pallina
    intesa come ogetto puntiforme
    """

    def __init__(self, x, y, vx, vy, m):
        """
        costruttore della classe, verra' chiamato
        quando creeremo l'istanza della classe
        in input prende la posizione, la velocita' e massa
        che sono le quantita' che identificano la pallina
        che saranno gli attributi della classe;
        il costruttore e' un particolare metodi della
        classe per questo si utilizzano gli underscore.
        il primo parametro che passiamo (self) rappresenta
        l'istanza della classe (self e' un nome di default)
        questo perche' la classe e' un modello generico che
        deve valere per ogni 'pallina'
        """
        #posizione
        self.x = x
        self.y = y
        #velocita'
        self.vx = vx
        self.vy = vy
        self.massa = m

    def energia_cinetica(self):
        """
        ad ogni metodo della classe viene passato
        come primo argomento self, quindi l'istanza
        calcoliamo l'energia cinetica
        """
        #una volta creati gli attributi non e necessario passarli ai vari metodi
        ene_k = 0.5*self.massa*(self.vx **2 + self.vy**2)
        return ene_k

    def energia_potenziale_gravitazionale(self, g):
        """
        calcolimao l'energia potenziale; all'interno
        di un campo gravitazionale di intensita' g
        la nostra pallina ha energia per il semplice
        fatto di essere in un qualche punto del campo
        """
        #supponiamo g sia diretta verso il basso
        ene_u = self.massa*g*self.y
        return ene_u

    #aggiornamento posizione e velocita' con eulero
    def n_vel(self, fx, fy, dt):
        """
        date le componenti della forza e il passo temporale
        aggiorno le componenti della velocita'
        """
        self.vx += fx*dt
        self.vy += fy*dt

    def n_pos(self, dt):
        """
        dato il passo temporale aggiorno le posizioni
        """
        self.x += self.vx*dt
        self.y += self.vy*dt



if __name__ == "__main__":

```

```
    g = 9.81 #acc di gravita' che vorrei tanto mettere uguale a 1
    p = Pallina(0, 10, 0, 0, 1)

    # simulazione moto
    N = 1500
    dt = 0.001

    #salvo le posizioni iniziali
    x = np.array([])
    y = np.array([])
    x = np.insert(x, len(x), p.x)
    y = np.insert(y, len(y), p.y)
    #array del tempo
    t = np.array([])
    t = np.insert(t, len(t), 0.0)

    #energia iniziale
    ene_cin = np.array([])
    ene_pot = np.array([])
    ene_cin = np.insert(ene_cin, len(ene_cin), p.energia_cinetica())
    ene_pot = np.insert(ene_pot, len(ene_pot), p.energia_potenziale_gravitazionale(g))


    for i in range(1, N):

        #se arrivi a terra fermati
        if y[-1]<0: break

        #moto di cadula libera
        fx = 0
        fy = -g

        #aggiorno le posizioni
        p.n_vel(fx, fy, dt)
        p.n_pos(dt)

        #salvo le posizioni
        x = np.insert(x, len(x), p.x)
        y = np.insert(y, len(y), p.y)

        #calcolo energia
        ene_cin = np.insert(ene_cin, len(ene_cin), p.energia_cinetica())
        ene_pot = np.insert(ene_pot, len(ene_pot), p.energia_potenziale_gravitazionale(g))

        t = np.insert(t, len(t), i*dt)


    #plot
    plt.figure(1)
    plt.title('moto parabolico')
    plt.xlabel('tempo')
    plt.ylabel('altezza')
    plt.plot(t, y)
    plt.grid()

    plt.figure(2)
    plt.title('energia moto parabolico')
    plt.xlabel('tempo')
    plt.ylabel('energia')
    plt.plot(t, ene_cin, label='energia cinetica')
    plt.plot(t, ene_pot, label='energia potenziale')
    plt.plot(t, ene_cin+ene_pot, label='energia totale')
    plt.legend(loc='best')
    plt.grid()

    plt.show()
```

Non è propriamente necessario usare i metodi per cambiare gli attributi di una classe, si possono utilizzare cose quali le property e i setter, ma non ce ne cureremo. Volendo ora grazie alla nostra classe potremmo creare n palline ognuna con delle condizioni iniziali diverse, inserirle in una lista e ciclarci sopra per vedere come evolvono, provate se vi va.


moto parabolico
altezza
tempo
0
2
4
6
8
10
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4


energia moto parabolico
energia
tempo
energia cinetica
energia potenziale
energia totale
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

I Risolve numericamente le SDE

Nelle sezioni precedenti abbiamo parlato delle equazioni differenziali alle derivate ordinarie (ODE) e alle derivate parziali (PDE); veniamo ora a fare un picolo accenno alle equazioni differenziali stocastiche (SDE). Le SDE sono equazioni in cui un termine è un processo stocastico e quindi anche la soluzione sarà un processo stocastico; sono utilizzate per modellare gli andamenti dei mercati o un qualche fenomeno soggetto a fluttuazioni termiche. Vedremo due semplici esempi, il moto geometrico Browniano e un processo di Ornstein–Uhlenbeck.

I.1 Processo di Ornstein–Uhlenbeck

Un processo di Ornstein–Uhlenbeck è descritto dalla seguente equazione stocastica:

$$dx = \theta(\mu - x)\,dt + \sigma\,dW$$

dove θ, σ constanti positive e μ costante, mentre dW è un processo di Wiener. In genere data una certa SDE della forma:

$$dx = f(x)dt + g(x)dW,$$

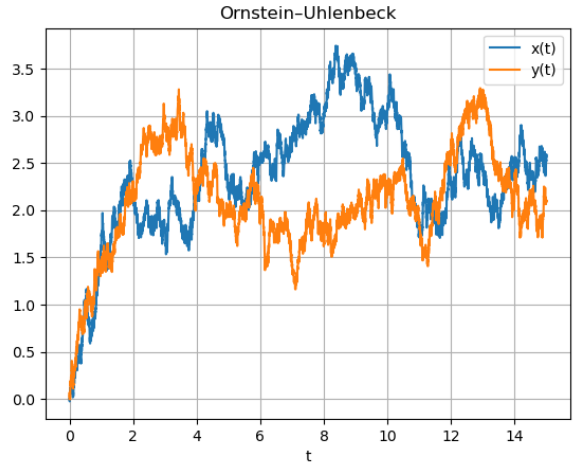
possiamo risolverla nel seguente modo (metodo di Euler–Maruyama):

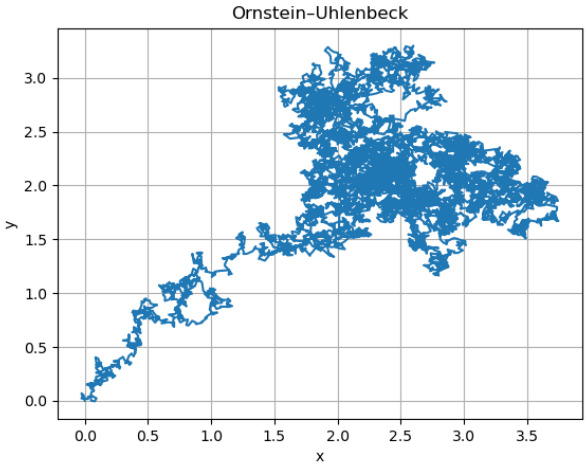
$$x_{n+1} = x_n + f(x_n)dt + g(x_n)dW$$

dove dt ora è il passo di integrazione e dW, che sarebbe un integrale stocastico, lo trattiamo una variabile gaussiana di media zero e varianza uguale alla radice del passo di integrazione. Vediamo un semplice esempio:

```
import numpy as np
import matplotlib.pyplot as plt

def f(z):
    """
    funzione che moltiplica il dt
    """
    theta = 0.7
    mu = 2.5
    return theta * (mu - z)

def g():
    """
    funzione che moltimplica il processo di wiener
    """
    sigma = 0.6
    return sigma

def dW(delta_t):
    """
    processo di wiener trattato come variabile gaussiana
    """
    return np.random.normal(loc=0.0, scale=np.sqrt(delta_t))

#parametri simulazione
N = 10000
tf = 15
dt = tf/N

ts = np.zeros(N + 1)
ys = np.zeros(N + 1)
xs = np.zeros(N + 1)

ys[0], xs[0] = 0, 0 #condizioni inizali

for i in range(N):
    ys[i+1] = ys[i] + f(ys[i]) * dt + g() * dW(dt)
    xs[i+1] = xs[i] + f(xs[i]) * dt + g() * dW(dt)
    ts[i+1] = ts[i] + dt

plt.figure(1)
plt.plot(xs, ys)
plt.title('Ornstein Uhlenbeck')
plt.xlabel("x")
plt.ylabel("y")
plt.grid()

plt.figure(2)
```

```
plt.plot(ts, xs, label='x(t)')
plt.plot(ts, ys, label='y(t)')
plt.title('Ornstein Uhlenbeck')
plt.xlabel("t")
plt.legend()
plt.grid()

plt.show()
```





I.2 Moto geometrico Browniano

Il moto geometrico browniano è un moto browniano esponenziale, che ha applicazioni nella descrizione dei mercati finanziari ad esempio; l'equazione associata è:

$$dx = \mu x dt + \sigma x dW$$

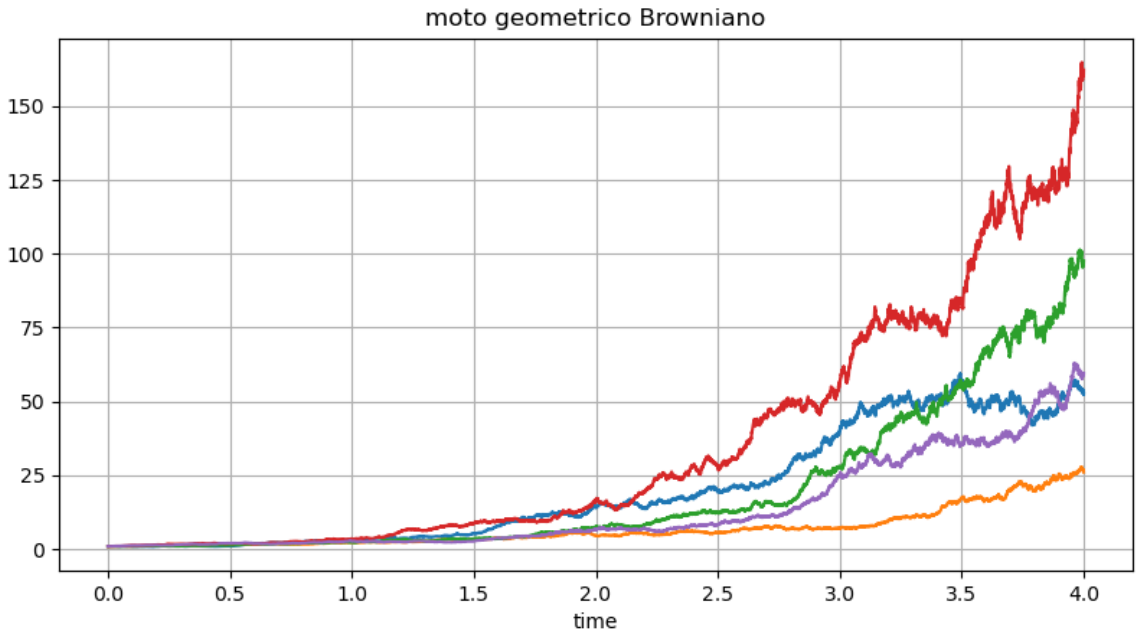
Vedremo per risolverla il metodo di heun che si può scrivere così, facendo riferimento all'equazione generica di sopra($dx = f(x)dt + g(x)dW$):

$$\begin{cases} \overline{x} = x_n + z_1 g(x_n) + f(x_n)dt + \frac{1}{2}g(x_n)g^{'}(x_n)z_1^2 \\ \hat{x} = x_n + z_1 g(\overline{x}) + f(\overline{x})dt + \frac{1}{2}g(\overline{x})g^{'}(\overline{x})z_1^2 \\ x_{n+1} = \frac{1}{2}(\hat{x} + \overline{x}) \end{cases}$$

dove z_1 rappresenta il processo di wiener ed è sempre una variabile gaussiana a media zero e varianza dt. Vediamone l'implementazione:

```
import numpy as np
import matplotlib.pyplot as plt

# geometric Brownian motion
# Heun method

def f(z):
    """
    funzione che moltiplica il dt
    """
    mu = 1
    return mu*z

def g(z):
    """
    funzione che moltimplica il processo di wiener
    """
    sigma = 0.5
    return sigma*z

def dg():
    """
    derivata di g
    """
    sigma = 0.5
```

```
    return sigma

def dW(delta_t):
    """
    processo di wiener trattato come variabile gaussiana
    """
    return np.random.normal(loc=0.0, scale=np.sqrt(delta_t))


#parametri simulazioni
N = 10000
tf = 4
dt = tf/N
#faccio 5 simulazioni diverse
for _ in range(5):
    #array dove conservare la soluzione, ogni volta inizializzati
    ts = np.zeros(N + 1)
    ys = np.zeros(N + 1)

    ys[0] = 1#condizioni iniziali

    for i in range(N):
        ts[i+1] = ts[i] + dt
        y0 = ys[i] + f(ys[i])*dt + g(ys[i])*dW(dt) + 0.5*g(ys[i])*dg()*(dW(dt)**2)
        y1 = ys[i] + f(y0)*dt    + g(y0)*dW(dt)    + 0.5*g(y0)*dg()*(dW(dt)**2)
        ys[i+1] = 0.5*(y0 + y1)

    plt.plot(ts, ys)

plt.figure(1)
plt.title('moto geometrico Browniano')
plt.xlabel("time")
plt.grid()

plt.show()
```



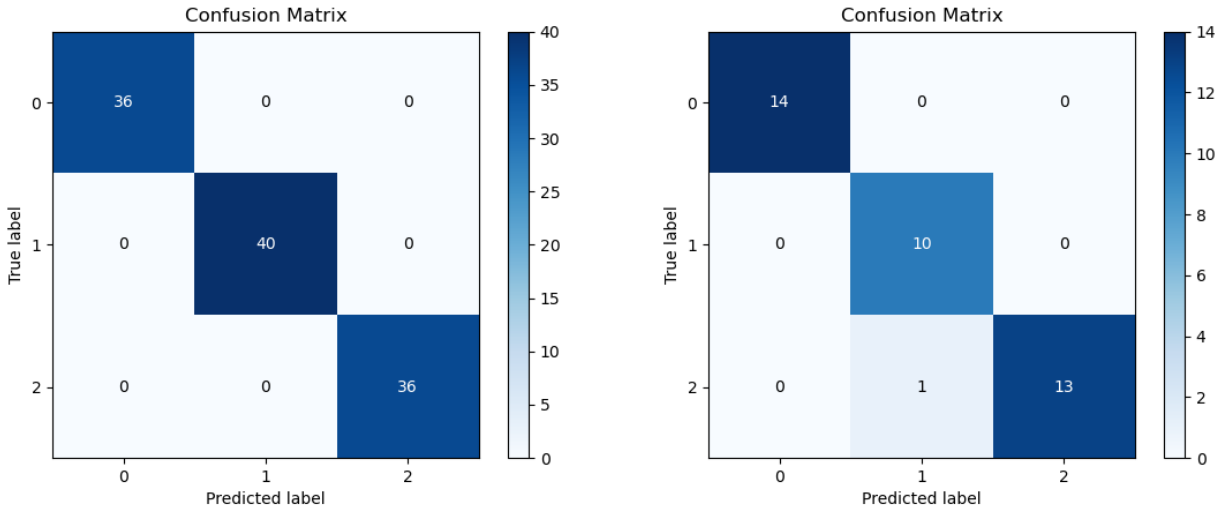
J Machine Learning

Essendo il computer molto stupido a qualcuno è venuta la brillante idea di cercare di educarlo, tanto per divertirsi un po'. Cominciamo quindi questa *descensio ad inferos* di cui al più faremo i primi gradini, utilizzando la libreria 'sklearn'. Vediamo quindi un esempio che possa essere il corrispettivo di un "Hello World". Ci limiteremo al machine learning supervisionato ovvero sappiamo sia input che output. Mostreremo un esempio di classificatore e uno di regressore.

J.1 Classificatore

Un algoritmo classificatore fondamentalmente prende dei dati e restituisce una categoria, quindi classifica i dati in input. Vediamo un esempio:

```
import scikitplot as skplt
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

#dati che verrano utilizzati
iris_dataset = datasets.load_iris()
#print(iris_dataset["DESCR"])

#caratteristiche, dati in input
x = iris_dataset.data

#output, cioe' quello che il modello dovrebbe predire
y = iris_dataset.target

#divido i dati, un parte li uso per addestrare, l'altra per
#testare se il modello ha imparato bene
x_train, x_test, y_train, y_test = train_test_split(x, y)

#modello non addestrato, classificatore
#un classificatore prende i dati e restituisce un categoria
modello = DecisionTreeClassifier()

#addestro il modello
modello.fit(x_train, y_train)

#predizioni sui dati su cui ha imparato
predizione_train = modello.predict(x_train)

#predizioni su nuovoi dati
predizione_test = modello.predict(x_test)

#misuro l'accuratezza sia dell'addestramento che del test
#questo puo' dare informazioni su over fitting o meno
#sinceramente non so come
print("accuratezza train")
print(accuracy_score(y_train, predizione_train))

print("accuratezza test")
print(accuracy_score(y_test, predizione_test))

#Rappresentazione grafica dei quanto e' stato bravo il modello
#sulle y c'e' la risposta che il modello doveva dare e
#sulle x ci sta la predizione che il modello ha dato, quindi gli
#elementi fuori diagonali sono le risposte sbagliate
skplt.metrics.plot_confusion_matrix(y_train,predizione_train)
skplt.metrics.plot_confusion_matrix(y_test,predizione_test)

plt.show()

[Output]
accuratezza train
1.0
accuratezza test
0.9736842105263158
```



La matrice di sinistra ci fa vedere come la predizione sui dati che abbiamo usato per addestrarla sia andata bene, infatti avevamo accuratezza uno; a sinistra vediamo che il modello ha dato una sola risposta sbagliata sui dati di test.

J.2 Regressori

Un regressore prende dei dati e restituisce un numero; è un po' come quando si fa un fit, circa...

```
import numpy as np
from sklearn.datasets import load_boston
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import train_test_split

"""
utilizzo di un regressore lineare per predirre
Il prrezzo di una casa dati certe informazioni
"""
dataset = load_boston()

#print(dataset["DESCR"])
#primi 13 elemnti della prima riga
#print(dataset["data"][0])
#ultimo elemnto della prima riga
#print(dataset["target"][0])

#caratteristiche
X = dataset["data"]

#output, cioe' quello che il modello dovrebbe predire
y = dataset["target"]

#divido i dati, un parte li uso per addestrare, l'altra per
#testare se il modello ha imparato bene
X_train, X_test, y_train, y_test = train_test_split(X, y)

#modello non addrestato e' un regressore
#un regressore prende i dati e restituisce un numero, una stima di qualcosa
modello = LinearRegression()

#addestro il modello
modello.fit(X_train, y_train)

#predizioni
p_train = modello.predict(X_train)
p_test = modello.predict(X_test)

#errori sulla predizione
dp_train = mean_absolute_error(y_train, p_train)
dp_test = mean_absolute_error(y_test, p_test)

print("train", np.mean(y_train),"+-", dp_train)
```

```
print("test ", np.mean(y_test), "+-", dp_test)

[Output]
train 22.59815303430079 +- 3.1207001914064647
test  22.337795275590548 +- 3.806645940024761
```

See you Space Cowboy ...