

# 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

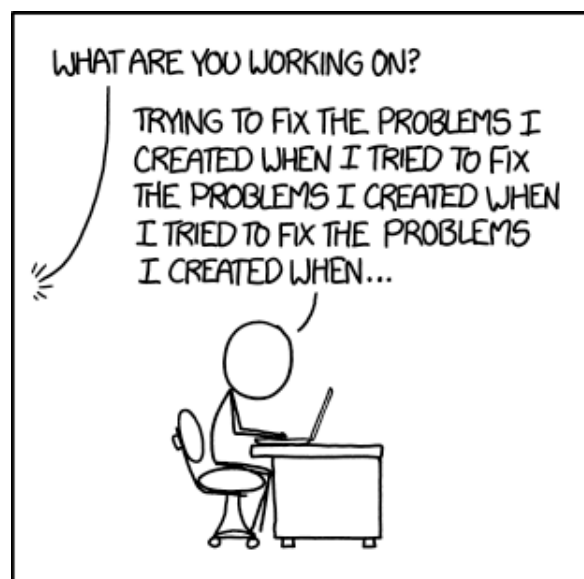
|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduzione</b>                                | <b>4</b>  |
| 1.1      | Notazioni . . . . .                                | 4         |
| 1.2      | Primo comandamento dell'informatica . . . . .      | 4         |
| 1.3      | Secondo comandamento dell'informatica . . . . .    | 4         |
| 1.4      | Terzo comandamento dell'informatica . . . . .      | 4         |
| 1.5      | Quarto comandamento dell'informatica . . . . .     | 4         |
| <b>2</b> | <b>Lezione Zero: Installazione</b>                 | <b>5</b>  |
| 2.1      | Installazione dell'ambiente: Pyzo . . . . .        | 5         |
| 2.2      | Installazione dell'interprete: Anaconda . . . . .  | 5         |
| 2.3      | Installazione dei pacchetti . . . . .              | 5         |
| <b>3</b> | <b>Prima lezione</b>                               | <b>6</b>  |
| 3.1      | Funzione print . . . . .                           | 6         |
| 3.2      | Commenti . . . . .                                 | 6         |
| 3.3      | Variabili . . . . .                                | 6         |
| 3.4      | Librerie . . . . .                                 | 8         |
| 3.5      | Gestione errori . . . . .                          | 9         |
| 3.5.1    | Try e except . . . . .                             | 9         |
| 3.5.2    | Raise Exception . . . . .                          | 10        |
| <b>4</b> | <b>Seconda lezione</b>                             | <b>11</b> |
| 4.1      | Gli array . . . . .                                | 11        |
| 4.2      | Tipi di array . . . . .                            | 11        |
| 4.3      | Array predefiniti . . . . .                        | 12        |
| 4.4      | Operazioni con gli array . . . . .                 | 12        |
| 4.5      | Matrici . . . . .                                  | 14        |
| <b>5</b> | <b>Terza lezione</b>                               | <b>16</b> |
| 5.1      | Le funzioni . . . . .                              | 16        |
| 5.2      | Istruzioni di controllo . . . . .                  | 16        |
| 5.2.1    | Espressioni condizionali: if, else, elif . . . . . | 16        |
| 5.2.2    | Cicli: while, for . . . . .                        | 17        |
| 5.3      | Ancora funzioni . . . . .                          | 19        |
| 5.4      | Grafici . . . . .                                  | 19        |
| 5.5      | Esercizio riassuntivo . . . . .                    | 22        |
| 5.6      | Prestazioni . . . . .                              | 23        |
| <b>6</b> | <b>Quarta lezione</b>                              | <b>24</b> |
| 6.1      | Importare file Python . . . . .                    | 24        |
| 6.2      | Fit . . . . .                                      | 24        |
| 6.3      | Risolvere numericamente le ODE . . . . .           | 28        |
| 6.3.1    | Esponenziale . . . . .                             | 28        |
| 6.3.2    | Pendolo . . . . .                                  | 30        |
| 6.3.3    | Animazione . . . . .                               | 31        |
| <b>A</b> | <b>Zeri di una funzione</b>                        | <b>33</b> |
| A.1      | Bisezione . . . . .                                | 33        |
| A.2      | Metodo di Newton . . . . .                         | 34        |
| A.3      | Zeri in più dimensioni . . . . .                   | 36        |
| <b>B</b> | <b>Risolvere numericamente le PDE</b>              | <b>39</b> |
| B.1      | Equazione del trasporto . . . . .                  | 39        |
| B.2      | Equazione del calore . . . . .                     | 40        |
| <b>C</b> | <b>Presa dati da foto</b>                          | <b>42</b> |

|          |   |           |
|----------|---|-----------|
| <b>D</b> | <b>Fit</b>  | <b>43</b> |
| D.1      | Fit con scipy . . . . .                               | 43        |
| D.2      | Fit circolare, metodo di Coope . . . . .              | 45        |
| D.3      | Fit di un’elissi, metodo di Halir e Flusser . . . . . | 48        |
| <b>E</b> | <b>Metodi Montecarlo</b>                              | <b>51</b> |
| E.1      | Generatori numeri pseudo-casuali . . . . .            | 51        |
| E.2      | Calcolo di Pi greco . . . . .                         | 52        |
| <b>F</b> | <b>Propagazione errori</b>                            | <b>54</b> |
| F.1      | Propagarli a mano . . . . .                           | 54        |
| F.2      | Uncertainties . . . . .                               | 54        |
| <b>G</b> | <b>Interpolazione</b>                                 | <b>56</b> |
| G.1      | Interpolazione lineare . . . . .                      | 56        |
| G.2      | Interpolazione Polinomiale . . . . .                  | 57        |
| G.3      | Scipy.interpolate . . . . .                           | 58        |
| <b>H</b> | <b>Programmazione a oggetti</b>                       | <b>60</b> |
| <b>I</b> | <b>Risolve numericamente le SDE</b>                   | <b>64</b> |
| I.1      | Processo di Ornstein–Uhlenbeck . . . . .              | 64        |
| I.2      | Moto geometrico Browniano . . . . .                   | 65        |
| <b>J</b> | <b>Machine Learning</b>                               | <b>67</b> |
| J.1      | Classificatore . . . . .                              | 67        |
| J.2      | Regressori . . . . .                                  | 68        |

# 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 tertium 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<sup>1</sup>. 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:

```
1 $ sudo apt -get install python3 -pip python3 - pyqt5
2 $ sudo python3 -m pip install pyzo -- upgrade
3 $ 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 munirsi 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.

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

oppure

```
1 /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:

```
1 install <nome_del_pacchetto >
```

oppure

```
1 pip install <nome_del_pacchetto >
```

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

```
1 import <nome_del_pacchetto >
```

se non succede nulla siete apposto

---

<sup>1</sup>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, quindi non un'intelligenza artificiale cresciuta su twitter, forse come prima cosa salterebbe tutti e il modo per comunicare è la funzione print, vediamo quindi il più classico degli esempi:

```
1 print('Hello world!')
2
3 [Output]
4 Hello world!
```

Questa funzione può stampare sia valori che espressioni:

```
1 print('Hello world!')
2 print('42')
3
4 print('Hello world!', 42)
5 print('Adesso vado \n a capo')
6
7 [Output]
8 Hello world!
9 42
10 Hello world! 42
11 Adesso vado
12 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.

```
1 #per i commenti che occupano una singola linea di codice si usa il cancelletto
2 #stampo hello world
3 print('Hello world!')
4
5 """
6 per un commento di maggiori linee di codice
7
8 vanno usate tre virgole per racchiuderlo
9 """
10 '''
11 ma van bene
12
13 anche tre apici
14 '''
15 [Output]
16 Hello world!
```

### 3.3 Variabili

Una variabile è un nome, un simbolo, che si dà 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:

```
1 numerointero= 13
2 numeroavirgolamobile= 13.
3
4 print('Numero intero:', numerointero, 'Numero in virgola mobile:', numeroavirgolamobile)
5 #oppure possiamo stampare in questo modo:
6 print(f'Numero intero: {numerointero}, Numero in virgola mobile: {numeroavirgolamobile}')
7
8 [Output]
9 Numero intero: 13 Numero in virgola mobile: 13.0
10 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()':

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

```

3 x = 7.
4 stringa = 'kebab'
5 lista = [1, 2., 'cane']
6 tupla = (42, 'balena')
7 dizionario = {'calza': 0, 'stampante': 0.5}
8
9 #stampiamole e stampiamone il tipo
10 print(n, type(n))
11 print(x, type(x))
12 print(stringa, type(stringa))
13 print(lista, type(lista))
14 print(tupla, type(tupla))
15 print(dizionario, type(dizionario))
16
17 [Output]
18 7 <class 'int'>
19 7.0 <class 'float'>
20 kebab <class 'str'>
21 [1, 2.0, 'cane'] <class 'list'>
22 (42, 'balena') <class 'tuple'>
23 {'calza': 0, 'stampante': 0.5} <class 'dict'>

```

Vediamo ora come fare le classiche operazioni matematiche:

```

1 x1 = 3
2 y1 = 4
3
4 somma = x1 + y1
5 prodotto = x1*y1
6 differenza = x1 - y1
7 rapporto = x1/y1
8 potenza = x1**y1
9
10 print(somma, prodotto, differenza, rapporto, potenza)
11 [Output]
12 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:

```

1 x = 0.1
2 y = 0.2
3 z = 0.3
4
5 print(x + y)
6 print(z)
7
8 [Output]
9 0.30000000000000004
10 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):

```

1 #variabili
2 x = 0.1
3 y = 0.2
4 z = 0.3
5 #sommo le prime due
6 t = x + y
7
8 """
9 applico una funzione che mi fornisce
10 una tupla contenente due numeri interi
11 il cui rapporto restituisce il numero iniziale.
12 output del tipo: (numeratore, denominatore)
13 """
14 print(t.as_integer_ratio())
15 print(z.as_integer_ratio())
16
17 [Output]
18 (1351079888211149, 4503599627370496)
19 (5404319552844595, 18014398509481984)

```

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

```
1 #definiamo una variabile
2 c = 3.141592653589793
3
4 #stampa come intero
5 print('%d' %c)
6
7 #stampa come reale
8 print('%f' %c) #di default stampa solo prime 6 cifre
9 print(f'{c}') #di default stampa tutte le cifre
10
11 #per scegliere il numero di cifre, ad esempio sette cifre
12 print('%7f' %c)
13 print(f'{c:.7f}')
14
15 [Output]
16 3
17 3.141593
18 3.141592653589793
19 3.1415927
20 3.1415927
```

Notare che il computer esegue un arrotondamento.

Una variabile può essere ridefinita e cambiare valore, il computer userà l'assegnazione più recente:

```
1 #definiamo una variabile
2 x = 30
3
4 """
5
6 operazioni varie
7
8
9 """
10
11 #ridefiniamo la variabile
12 x = 18
13
14 print('x=', x)
15
16 """
17 E' possibile anche sovrascrivere una variabile
18 con un numero che dipende dal suo valore precedente:
19 """
20 x = x + 1 #incrementiamo di uno
21 #Oppure:
22 x += 1
23 print('x=', x)
24
25 x = x * 2 #moltiplichiamo per due
26 #Oppure:
27 x *= 2
28 print('x=', x)
29
30 [Output]
31 x= 18
32 x= 20
33 x= 80
```

Come si vede i vari comandi  $x = x \text{ operazione numero}$  possono essere abbreviati con  $x \text{ operazione} = \text{numero}$ .

### 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:

```
1 import numpy
2
3 #per usare un contenuto di questa libreria basta scrivere numpy.contenuto
4 pigreco = numpy.pi
5 print(pigreco)
```



```

6
7 #Possiamo anche ribattezzare le librerie in questo modo
8 import numpy as np
9 #da ora all'interno del codice numpy si chiama np
10
11 eulerio = np.e
12 print(eulerio)
13
14 [Output]
15 3.141592653589793
16 2.718281828459045

```

```

1 import math
2
3 coseno=math.cos(0)
4 seno = math.sin(np.pi/2) #python usa di default gli angoli in radianti!!!
5 senosbagliato = math.sin(90)
6
7 print('Coseno di 0=', coseno, "\nSeno di pi/2=", seno, "\nSeno di 90=", senosbagliato)
8
9 #bisogna quindi stare attenti ad avere tutti gli angoli in radianti
10 angoloingradi = 45
11 #questa funzione converte gli angoli da gradi a radianti
12 angoloinradianti = math.radians(angoloingradi)
13
14 print("Angolo in gradi:", angoloingradi, "Angolo in radianti:", angoloinradianti)
15
16 [Output]
17 Coseno di 0= 1.0
18 Seno di pi/2= 1.0
19 Seno di 90= 0.8939966636005579
20 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 magari qualcosa è stato implementato male. Esiste un costrutto che ci permette di gestire gli errori in maniera tranquilla diciamo. Facciamo un semplice esempio:

```

1 a = 0
2 b = 1/a
3 print(b)
4
5 [Output]
6 Traceback (most recent call last):
7   File "<tmp 1>", line 5, in <module>
8     b = 1/a
9 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.

```

1 a = 0
2 try :
3     b = 1/a
4 except ZeroDivisionError:
5     b = 1
6
7 print(b)
8
9 [Output]
10 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.

```
1 """
2 leggo un valore da shell
3 uso del comando try per evitare che venga letto
4 qualcosa che non sia un numero: e.g. una stringa
5 """
6 try:
7     b = int(input('scegliere un valore:'))
8 except ValueError:
9     print('hai digitato qualcosa diverso da un numero, per favore ridigitare')
10    b = int(input('scegliere un valore:'))
11
12 #se si sbaglia a digitare di nuovo il codice si arresta per ValueError
13
14 #controllo se e' possibile proseguire
15 if b > 7 :
16     #se vero si blocca il codice sollevando l'eccezione
17     messaggio_di_errore = 'il valore scelto risulta insensato in quanto nulla supera 7, misura
18     massima di ogni cosa'
19     raise Exception(messaggio_di_errore)
20
21 [Output]
22 8
23 Traceback (most recent call last):
24   File "<tmp 1>", line 18, in <module>
25     raise Exception(messaggio_di_errore)
26 Exception: il valore scelto risulta insensato in quanto nulla supera 7, misura massima di ogni
27     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, 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:

```
1 import numpy as np
2
3 #Creiamo un array di 5 elementi
4 array1 = np.array([1.0, 2.0, 4.0, 8.0, 16.0]) #scrivere 2.0 equivale a scrivere 2.
5
6 print(array1)
7
8 #per accedere a un singolo elemento dell'array basta fare come segue:
9 elem = array1[1]
10
11 #ATTENZIONE! Gli indici, per Python, partono da 0, non da 1!
12 print(elem)
13
14 [Output]
15 [ 1.  2.  4.  8. 16.]
16 2.0
```

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

```
1 import numpy as np
2
3 array1=np.array([1.0, 2.0, 4.0, 8.0, 16.0])
4
5 #Aggiungiamo ora un numero in una certa posizione dell'array:
6 array1 = np.insert(array1, 4, 18)
7 '''
8 abbiamo aggiunto il numero 18 in quarta posizione, la sintassi e' :
9 np.insert(array a cui vogliamo aggiungere un numero, posizione dove aggiungerlo, numero)
10 '''
11 print(array1)
12
13 #Per aggiungere elementi in fondo ad un array esiste anche il comando append della libreria
  numpy:
14 array2 = np.append(array1, -4.)
15 print(array2)
16 #Mentre per togliere un elemento basta indicare il suo indice alla funzione remove di numpy:
17 array2 = np.delete(array2, 0)
18 print(array2)
19
20 [Output]
21 [ 1.  2.  4.  8. 18. 16.]
22 [ 1.  2.  4.  8. 18. 16. -4.]
23 [ 2.  4.  8. 18. 16. -4.]
```

### 4.2 Tipi di array

Come le variabili 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.

```
1 import numpy as np
2
3 array1 = np.array([1.0, 2.0, 4.0, 8.0, 16.0])
4
5 tipoarray1 = array1.dtype
6 print(tipoarray1)
7
8 a = np.array([0, 1, 2])
9 #abbiamo scritto solo numeri interi => array di interi
10
11 b = np.array([0., 1., 2.])
12 #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:

```

1 import numpy as np
2
3 array1 = np.array([1.0, 2.0, 4.0, 8.0, 16.0])
4
5 primi_tre = array1[0:3]
6 print('primi_tre = ', primi_tre)
7 """
8 Questa sintassi seleziona gli elementi di array1
9 dall'indice 0 incluso all'indice 3 escluso.
10 Il risultato e' ancora un array.
11 """
12
13 esempio = array1[1:-1]
14 print(esempio)
15 esempio = array1[-2:5]
16 print(esempio)
17 #Questo metodo accetta anche valori negativi, con effetti curiosi
18
19
20 elementi_pari = array1[0::2]
21 print('elementi_pari = ', elementi_pari)
22 """
23 In questo esempio invece, usando invece due volte il simbolo :
24 intendiamo prendere solo gli elementi dall'indice 0 saltando di 2 in 2.
25 Il risultato e' un array dei soli elementi di indice pari
26 """
27
28 rewind = array1[::-1]
29 print('rewind = ', rewind)
30 """
31 Anche qui possiamo usare valori negativi.
32 In particolare questo ci permette di saltare "all'indietro"
33 e, ad esempio, di invertire l'ordine di un'array con un solo comando
34 """
35
36 [Output]
37 primi_tre = [1. 2. 4.]
38 [2. 4. 8.]
39 [ 8. 16.]
40 elementi_pari = [ 1.  4. 16.]
41 rewind = [16.  8.  4.  2.  1.]

```

Grande comodità sono le operazioni 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 spaio.

```

1 import math
2 import numpy as np
3
4 v = np.array([4, 5, 6])
5 w = np.array([1.2, 3.4, 5.8])
6
7 #classiche operazioni
8 somma = v + w
9 sottr = v - w
10 molt = v * w
11 div = v / w
12
13 print(v, w)
14 print()
15 print(somma, sottr, molt, div)
16 print()
17 #altri esempi
18 print(v**2)
19 print(np.log10(w))
20
21 """
22 come dicevamo prima qui' otterremmo errore poiche'
23 math lavora solo con numeri o, volendo,
24 array unidimensionali lunghi uno
25 """
26 print(math.log10(w))
27
28 [Output]
29 [4 5 6] [1.2 3.4 5.8]
30
31 [ 5.2  8.4 11.8] [2.8 1.6 0.2] [ 4.8 17.  34.8] [3.33333333 1.47058824 1.03448276]

```

```

32 [16 25 36]
33 [0.07918125 0.53147892 0.76342799]
34 Traceback (most recent call last):
35   File "<tmp 1>", line 26, in <module>
36     print(math.log10(w))
37   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  $n$  e un vettore ad  $n$  componenti allora un array bidimensionale sarà una matrice.

```

1 import numpy as np
2
3 #esiste la funzione apposita di numpy per scrivere matrici.
4 matrice1 = np.matrix('1 2; 3 4; 5 6')
5 #Si scrivono essenzialmente i vettori riga della matrice separati da ;
6
7 #equivalente a:
8 matrice2 = np.matrix([[1, 2], [3, 4], [5,6]])
9
10 print(matrice1)
11 print(matrice2)
12
13
14 matricedizeri = np.zeros((3, 2)) #tre righe, due colonne: matrice 3x2
15 print('Matrice di zeri:\n', matricedizeri, '\n')
16 matricediuni = np.ones((3,2))
17 print('Matrice di uni:\n', matricediuni, '\n')
18
19 [Output]
20 [[1 2]
21  [3 4]
22  [5 6]]
23 [[1 2]
24  [3 4]
25  [5 6]]
26 Matrice di zeri:
27 [[0. 0.]
28  [0. 0.]
29  [0. 0.]]
30
31 Matrice di uni:
32 [[1. 1.]
33  [1. 1.]
34  [1. 1.]]

```

E ovviamente anche qui possiamo fare le varie operazioni matematiche:

```

1 import numpy as np
2
3 matrice1 = np.matrix('1 2; 3 4; 5 6')
4 matricediuni = np.ones((3,2))
5
6 sommadimatrici = matrice1 + matricediuni
7 print('Somma di matrici:\n', sommadimatrici)
8
9 matrice3 = np.matrix('3 4 5; 6 7 8') #matrice 2x3
10 prodottodimatrici = matrice1 * matrice3 #matrice 3x(2x2)x3
11 #alternativamente si potrebbe scrivere: prodottodimatrici = matrice1 @ matrice3
12
13 print('\nProdotto di matrici:\n', prodottodimatrici)
14
15 [Output]
16 Somma di matrici:
17 [[2. 3.]
18  [4. 5.]
19  [6. 7.]]
20
21 Prodotto di matrici:
22 [[15 18 21]

```

```
23 [33 40 47]
24 [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:

```
1 def area(a, b):
2     """
3     restituisce l'area del rettangolo
4     di lati a e b
5     """
6     A = a*b #calcolo dell'area
7     return A
8
9 #chiamiamo la funzione e stampiamo subito il risultato
10 print(area(3, 4))
11 print(area(2, 5))
12
13 """
14 Se la funzione non restituisce nulla
15 ma esegue solo un pezzo di codice,
16 si parla propriamente di procedura
17 e il valore restituito e' None.
18 """
19 def procedura(a):
20     a = a+1
21
22 print(procedura(2))
23
24 """
25 Volendo si possono creare anche funzioni
26 che non hanno valori in ingresso:
27 """
28 def pigreco():
29     return 3.14
30 print(pigreco())
31
32 [Output]
33 12
34 10
35 None
36 3.14
```

Portiamo all'attenzione due fatti importanti:

- 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 una 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:



```

1 def assoluto(x):
2     """
3     restituisce il valore assoluto di un numero
4     """
5     # se vero restituisci x
6     if x >= 0:
7         return x
8     #altrimenti restituisci -x
9     else:
10        return -x
11
12 print(assoluto(3))
13 print(assoluto(-3))
14
15 [Output]
16 3
17 3

```

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

```

1 def segno(x):
2     """
3     funzione per capire il segno di un numero
4     """
5     #se vero ....
6     if x > 0:
7         return 'Positivo'
8     #se invece ....
9     elif x == 0:
10        return 'Nullo'
11    #altrimenti ....
12    else:
13        return 'Negativo'
14
15 print(segno(5))
16 print(segno(0))
17 print(segno(-4))
18
19 [Output]
20 Positivo
21 Nullo
22 Negativo

```

### 5.2.2 Cicli: while, for

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

```

1 def fattoriale(n):
2     """
3     Restituisce il fattoriale di un numero
4     """
5     R = 1
6     #finche' e' vero fai ...
7     while n > 1:
8         R *= n
9         n -= 1
10    return R
11
12 print(fattoriale(5))
13
14 [Output]
15 120

```

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:

```

1 def fattoriale(n):
2     """
3     restituisce il fattoriale di un numero
4     """

```

```

5     R = 1
6     #finche' i non arriva ad n fai ...
7     for i in range(1, n+1):
8         R = R*i
9     return R
10
11 print(fattoriale(5))
12
13 [Output]
14 120

```

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  $\text{range}(1, n+1) = [1, 2, \dots, n]$ . 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:

```

1 import numpy as np
2
3 def trova_pari(array):
4     """
5     restituisce un array contenente solo
6     i numeri pari dell'array di partenza
7     """
8     R = np.array([]) #array da riempire
9     #per ogni elemento in array fai ...
10    for elem in array:
11        if elem%2 == 0:
12            R = np.append(R, elem)
13    return R
14
15 a = np.array([i for i in range(0, 11)])
16 """
17 il precedente e' un modo piu' conciso di scrivere:
18 a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
19 """
20 print(a)
21 print(trova_pari(a))
22
23 [Output]
24 [ 0  1  2  3  4  5  6  7  8  9 10]
25 [ 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:

```

1 import numpy as np
2
3 #creiamo un array
4 array = np.linspace(0, 1, 5)
5
6 """
7 in questo modo posso iterare contemporaneamente
8 sia sugli indici sia sugli elementi dell'array
9 """
10 for index, elem in enumerate(array):
11     print(index, elem)
12
13 [Output]
14 0 0.0
15 1 0.25
16 2 0.5
17 3 0.75
18 4 1.0

```

zip:

```

1 import numpy as np
2
3 #creiamo tre un array
4 array1 = np.linspace(0, 1, 5)
5 array2 = np.linspace(1, 2, 5)
6 array3 = np.linspace(2, 3, 5)

```

```

7  """
8  in questo modo posso iterare contemporaneamente
9  sugli elementi di tutti gli array
10 """
11 for a1, a2, a3 in zip(array1, array2, array3):
12     print(a1, a2, a3)
13
14 [Output]
15 0.0 1.0 2.0
16 0.25 1.25 2.25
17 0.5 1.5 2.5
18 0.75 1.75 2.75
19 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 single 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`):

```

1 def molt(*numeri):
2     """
3     restituisce il prodotto di n numeri
4     """
5     R = 1
6     for numero in numeri:
7         R *= numero
8     return R
9
10 print(molt(2, 7, 10, 11, 42))
11 print(molt(5, 5))
12 print(molt(10, 10, 2))
13
14 [Output]
15 64680
16 25
17 200

```

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:

```

1 #t[s] x[m]
2 1      1
3 2      4
4 3      9
5 4     16
6 5     25
7 6     36

```

Per leggerli:

```

1 import numpy as np
2
3 #Leggiamo da un file di testo classico
4 path = 'dati.txt'
5 dati1, dati2 = np.loadtxt(path, unpack=True)
6 """
7 unpack=True serve proprio a dire che vogliamo che
8 dati1 contenga la prima colonna e dati2 la seconda
9 La prima riga avendo il cancelletto verra' saltata
10 """
11

```

```

12 #se vogliamo invece che venga letto tutto come una matrice scriviamo:
13 path = 'dati.txt'
14 dati = np.loadtxt(path, unpack=True)
15 #dati sara' nella fattispecie una matrice con due colonne e 6 righe
16
17
18 #leggere da file.csv
19 path = 'dati.csv'
20 dati1, dati2 = np.loadtxt(path, usecols=[0,1], skiprows=1, delimiter=',', unpack=True)
21 """
22 a differenza di quanto sopra dobbiamo specificare le colonne da usare (contiamo da zero)
23 a causa poi dell'organizzazione dei .csv dobbiamo dire anche quante righe saltare
24 """

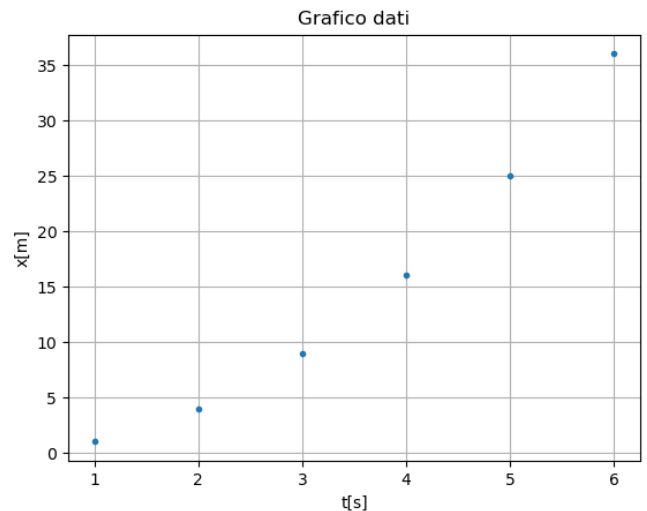
```

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

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 #Leggiamo da un file di testo classico
5 path = 'dati.txt'
6 dati1, dati2 = np.loadtxt(path, unpack=True)
7
8 plt.figure(1) #creiamo la figura
9
10 #titolo
11 plt.title('Grafico dati')
12 #nomi degli assi
13 plt.xlabel('t[s]')
14 plt.ylabel('x[m]')
15 #plot dei dati
16 plt.plot(dati1, dati2, marker='.', linestyle='')
17 #aggiungiamo una griglia
18 plt.grid()
19 #comando per mostrare a schermo il grafico
20 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:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def f(x):
5     """
6     restituisce il cubo di un numero
7     """
8     return x**3
9
10 def g(x):
11     """
12     restituisce il quadrato di un numero
13     """
14     return x**2
15
16 #array di numeri equispaziati nel range [-1,1] usiamo:
17 x = np.linspace(-1, 1, 40)
18
19 plt.figure(1) #creiamo la figura
20
21 #titolo
22 plt.title('Grafico funzioni')
23 #nomi degli assi
24 plt.xlabel('x')
25 plt.ylabel('f(x), g(x)')
26 #plot dei dati
27 plt.plot(x, f(x), marker='.', linestyle='--', color='blue', label='parabola')
28 plt.plot(x, g(x), marker='.', linestyle='--', color='red', label='cubica')

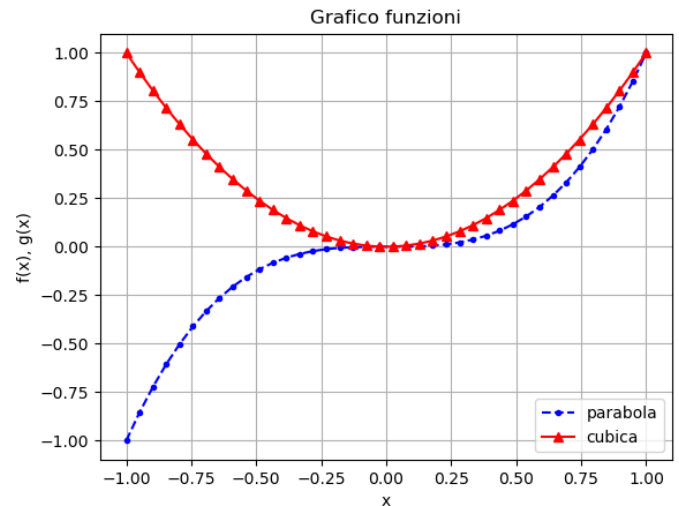
```

```

29 #aggiungiamo una leggenda
30 plt.legend(loc='best')
31 #aggiungiamo una griglia
32 plt.grid()
33 #comando per mostrare a schermo il grafico
34 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 conoscere tutti gli altri comandi possibili per migliorare/abbellire il grafico da adde 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 logaritmica o altro sull'asse x (analogo sarà sulle y mutatis mutandis).



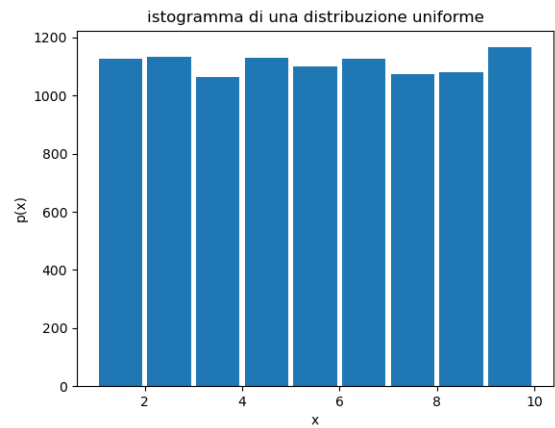
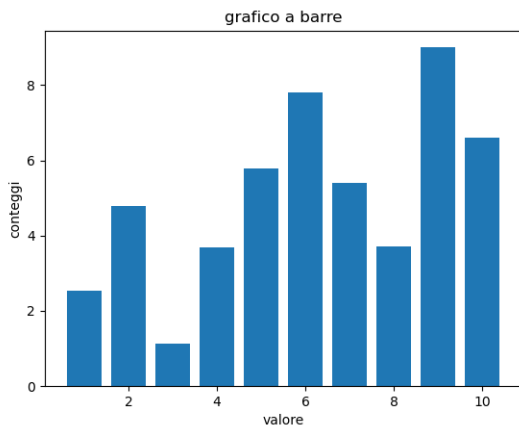
Ultima menzione da fare sono gli istogrammi (il fetish dei pallettari):

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 plt.figure(1)
5 plt.title('grafico a barre')
6 plt.xlabel('valore')
7 plt.ylabel('conteggi')
8 # Sull'asse x utilizziamo un array di 10 punti equispaziati.
9 x = np.linspace(1,10,10)
10 # Sull'asse y abbiamo, ad esempio, il seguente set di dati:
11 y = np.array([2.54, 4.78, 1.13, 3.68, 5.79, 7.80, 5.4, 3.7, 9.0, 6.6])
12
13 # Il comando per la creazione dell'istogramma corrispondente e':
14 plt.bar(x, y, align = 'center')
15
16 plt.figure(2)
17 plt.title('istogramma di una distribuzione uniforme')
18 plt.xlabel('x')
19 plt.ylabel('p(x)')
20
21 """
22 lista di numeri distribuiti uniformemente fra 0 e 10
23 si usa l'underscore nel for poiche' non serve usare
24 un'altra variabile. Avremmo potuto scrivere for i ...
25 ma la i non sarebbe comparsa da nessun'altra parte
26 sarebbe stato uno spreco
27 """
28 z = [np.random.uniform(10) for _ in range(10000)]
29 plt.hist(z, bins=10, rwidth=0.9)
30
31
32 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 riassume 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.

```

1 import numpy as np
2
3 def Area(l, n):
4     """
5     calcolo area di un poligono
6     regolare di lato l e numero lati n
7     """
8     a = 1/2 * l/np.tan(np.pi/n) #apotema
9     p = n*l                      #perimetro
10    A = p*a/2                    #area
11    return A
12
13 l = [3, 4, 5] #dimensioni dei lati, deve essere una terna pitagorica
14 n = np.arange(4, 12) #numero di lati dei poligoni
15
16 A3 = np.array([]) #array in cui ci saranno le aree dei poligoni di lato 3
17 A4 = np.array([]) #array in cui ci saranno le aree dei poligoni di lato 4
18 A5 = np.array([]) #array in cui ci saranno le aree dei poligoni di lato 5
19
20 for nn in n: #loop sul numero di lati
21     for ll in l: #lup sulla dimensione, quindi sul triangolo
22         A0 = Area(ll, nn) #calcolo dell'area
23
24         if ll == 3:
25             A3 = np.append(A3, A0)
26         elif ll == 4:
27             A4 = np.append(A4, A0)
28         elif ll == 5:
29             A5 = np.append(A5, A0)
30
31
32 for i in range(len(n)):
33     print(f'A3 + A4 = {A3[i]+A4[i]:.3f}')
34     print(f'A5 = {A5[i]:.3f}')
35
36 [Output]
37 A3 + A4 = 25.000
38 A5 = 25.000
39 A3 + A4 = 43.012
40 A5 = 43.012
41 A3 + A4 = 64.952
42 A5 = 64.952
43 A3 + A4 = 90.848
44 A5 = 90.848
45 A3 + A4 = 120.711
46 A5 = 120.711
47 A3 + A4 = 154.546
48 A5 = 154.546
49 A3 + A4 = 192.355
50 A5 = 192.355

```

```
51 A3 + A4 = 234.141
52 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:

```
1 import time
2 import numpy as np
3
4 #inizio a misurare il tempo
5 start = time.time()
6
7
8 a1 = 0          #variabile che conterra' il risultato
9 N = int(5e6)    # numero di iterazioni da fare = 5 x 10**6
10
11 #faccio il conto a 'mano'
12 for i in range(N):
13     a1 += np.sqrt(i)
14
15 #finisco di misurare il tempo
16 end = time.time()-start
17
18 print(end)
19
20 #inizio a misurare il tempo
21 start = time.time()
22
23 #stesso conto ma fatto tramite le librerie di python
24 a2 = sum(np.sqrt(np.arange(N)))
25
26 #finisco di misurare il tempo
27 end = time.time()-start
28
29 #sperabilmente sara' minore del tempo impiegato prima
30 print(end)
31
32 [Output]
33 11.588378429412842
34 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 stesso 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:

```
1 def f(x, n):
2     """
3     restituisce la potenza n-esima di un numero x
4     Parametri
5     -----
6     x, n : float
7
8     Return
9     -----
10    v : float
11        x**n
12    """
13
14    v = x**n
15
16    return v
17
18 if __name__ == '__main__':
19     #test
20     print(f(5, 2))
21
22 [Output]
23 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`:

```
1 import elevamento
2
3 print(elevamento.f(3, 3))
4
5 [Output]
6 27
```

Notiamo nel codice iniziale la presenza dell' `if`, esso serve per far sì 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

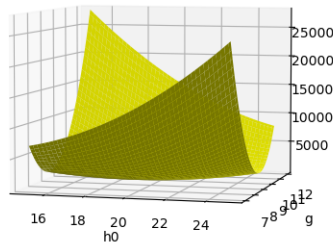
Nell'ambito della statistica un fit, cioè una regressione lineare o non che sia (dove la linearità è riferita ai parametri della funzione), è un metodo per trovare la funzione che meglio descrive l'andamento di alcuni dati. Nel caso di regressione lineare la procedura da eseguire non è troppo complicata, 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 legge 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.optimize.curve_fit()`". In ogni caso comunque l'idea di ciò che va fatto è trovare il minimo della seguente funzione:

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

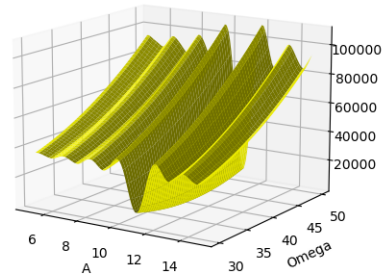
che nel caso in cui il termine dentro la somma sia distribuito in modo gaussiano allora la quantità  $S^2$  è distribuita come un chiquadro, e da qui si potrebbe fare tutta una discussione sulla significatività statistica di quello che andiamo a fare, che ovviamente noi non facciamo. Il problema della non linearità fondamentalmente si può esprimere nell'esistenza di minimi locali che potrebbero bloccare il fit dando valori per i parametri  $\theta_i$  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:



Chiquadro regressione lineare



Chiquadro regressione non-lineare



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

modello:  $y(t) = A \cos(\omega t)$

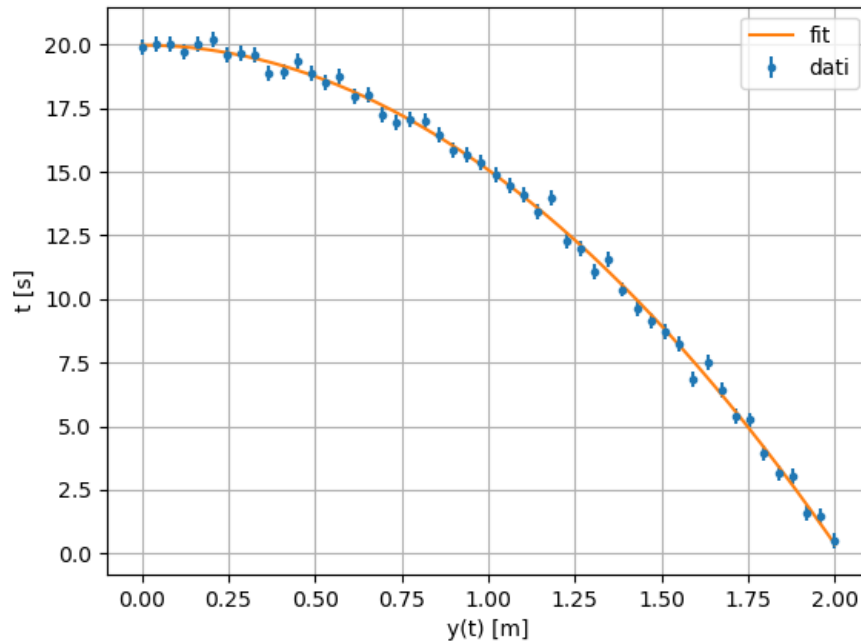
Vediamo come effettivamente siano presenti nel caso non lineare una serie di minimi locali che sarebbe meglio evitare (i codici per la realizzazione di grafici sono riportati a fine sezione). Vediamo ora un semplice esempio di codice:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.optimize import curve_fit
4
5 def Legge_oraria(t, h0, g):
6     """
7     Restituisce la legge oraria di caduta
8     di un corpo che parte da altezza h0 e
9     con una velocita' iniziale nulla
10    """
11    return h0 - 0.5*g*t**2
12
13 """
14 dati misurati:
15 xdata : fisicamemnte i tempi a cui osservo
16         la caduta del corpo non affetti da
17         errore
18 ydata : fisicamente la posizione del corpo
19         misurata a dati tempi xdata afetta
20         da errore
21 """
22
23 #misuro 50 tempi tra 0 e 2 secondi
24 xdata = np.linspace(0, 2, 50)
25
26 #legge di caduta del corpo
27 y = Legge_oraria(xdata, 20, 9.81)
28 rng = np.random.default_rng()
29 y_noise = 0.3 * rng.normal(size=xdata.size)
30 #dati misurati afferri da errore
31 ydata = y + y_noise
32 dydata = np.array(len(ydata)*[0.3])
33
34 #funzione che mi permette di vedere anche le barre d'errore
35 plt.errorbar(xdata, ydata, dydata, fmt='.', label='dati')
36
37 #array dei valori che mi aspetto, circa, di ottenere
38 init = np.array([15, 10])
39 #eseguo il fit
40 popt, pcov = curve_fit(Legge_oraria, xdata, ydata, init, sigma=dydata, absolute_sigma=False)
41
42 h0, g = popt
43 dh0, dg = np.sqrt(pcov.diagonal())
44 print(f'Altezza iniziale h0 = {h0:.3f} +- {dh0:.3f}')
45 print(f"Accelerazione di gravita' g = {g:.3f} +- {dg:.3f}")
46
47 #garfico del fit
48 t = np.linspace(np.min(xdata), np.max(xdata), 1000)
49 plt.plot(t, Legge_oraria(t, *popt), label='fit')
50
51 plt.grid()
52 plt.xlabel('y(t) [m]')
```

```

53 plt.ylabel('t [s]')
54 plt.legend(loc='best')
55 plt.show()
56
57 [Output]
58 Altezza iniziale h0 = 19.975 +- 0.064
59 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

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def Legge_oraria(t, h0, g):
5     """
6     Restituisce la legge oraria di caduta
7     di un corpo che parte da altezza h0 e
8     con una velocita' iniziale nulla
9     """
10    return h0 - 0.5*g*t**2
11
12 """
13 dati misurati:
14 xdata : fisicamemnte i tempi a cui osservo
15         la caduta del corpo non affetti da
16         errore
17 ydata : fisicamente la posizione del corpo
18         misurata a dati tempi xdata afetta
19         da errore
20 """
21
22 #misuro 50 tempi tra 0 e 2 secondi
23 xdata = np.linspace(0, 2, 50)
24
25 #legge di caduta del corpo
26 y = Legge_oraria(xdata, 20, 9.81)
27 rng = np.random.default_rng()
28 y_noise = 0.3 * rng.normal(size=xdata.size)
29 #dati misurati afferri da errore

```

```

30 ydata = y + y_noise
31 dydata = np.array(ydata.size*[0.3])
32
33 N = 100
34 S2 = np.zeros((N, N))
35 h0 = np.linspace(15, 25, N)
36 g = np.linspace(7, 12, N)
37 for i in range(N):
38     for j in range(N):
39         S2[i, j] = (((ydata - Legge_oraria(xdata, h0[i], g[j]))/dydata)**2).sum()
40
41 #grafico del chi quadro
42 fig = plt.figure(1)
43 gridx, gridy = np.meshgrid(h0, g)
44 ax = fig.add_subplot(projection='3d')
45 ax.plot_surface(gridx, gridy, S2, color='yellow')
46 ax.set_title('Chiquadro regressione lineare')
47 ax.set_xlabel('h0')
48 ax.set_ylabel('g')
49 plt.show()

```

Caso non lineare

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.optimize import curve_fit
4
5 def Legge_oraria(t, A, omega):
6     """
7     Restituisce la legge oraria di un corpo che
8     oscilla con ampiezza A e frequenza omega
9     """
10    return A*np.cos(omega*t)
11
12    """
13    dati misurati:
14    xdata : fisicamemnte i tempi a cui osservo
15            l'ossilazione del corpo non
16            affetti da errore
17    ydata : fisicamente la posizione del corpo
18            misurata a dati tempi xdata afetta
19            da errore
20    """
21
22    #misuro 50 tempi tra 0 e 2 secondi
23    xdata = np.linspace(0, 2, 50)
24
25    #legge di oscillazione del corpo
26    y = Legge_oraria(xdata, 10, 42)
27    rng = np.random.default_rng()
28    y_noise = 0.3 * rng.normal(size=xdata.size)
29    #dati misurati afferri da errore
30    ydata = y + y_noise
31    dydata = np.array(ydata.size*[0.3])
32
33    N = 100
34    S2 = np.zeros((N, N))
35    A = np.linspace(5, 15, N)
36    O = np.linspace(30, 50, N)
37    for i in range(N):
38        for j in range(N):
39            S2[i, j] = (((ydata - Legge_oraria(xdata, A[i], O[j]))/dydata)**2).sum()
40
41    #grafico chiquadro
42    fig = plt.figure(1)
43    gridx, gridy = np.meshgrid(A, O)
44    ax = fig.add_subplot(projection='3d')
45    ax.plot_surface(gridx, gridy, S2, color='yellow')
46    ax.set_title('Chiquadro regressione non-lineare')
47    ax.set_xlabel('A')
48    ax.set_ylabel('Oomega')
49    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 equazioni 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:

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

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  $dt \rightarrow 0$ . 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:

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

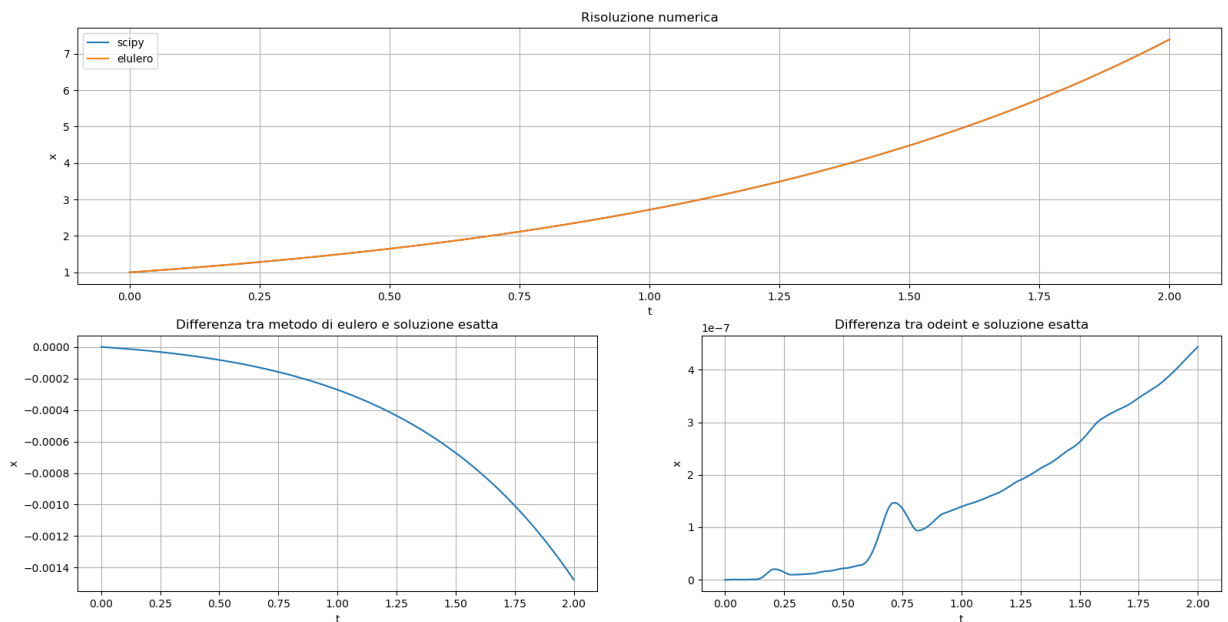
Abbiamo Una funzione incognita  $x(t)$  di cui sappiamo che la derivata è uguale a se stessa e che calcolata in zero restituisce uno. Nella fattispecie la soluzione è semplice, si tratta di un esponenziale crescente, tuttavia vediamo come risolvere numericamente tale equazione.

```
1 import numpy as np
2 import scipy.integrate
3 import matplotlib.pyplot as plt
4
5 #parametri
6 x0 = 1      #condizione iniziale
7 tf = 2      #fino a dove integrare
8 N = 10000   #numero di punti
9
10 #odeint
11 def ODE_1(y, t):
12     """
13     equazione da risolvere per odeint
14     """
15     x = y
16     dydt = x
17     return dydt
18
19
20 y0 = [x0] #x(0)
21 t = np.linspace(0, tf, N+1)
22 sol = scipy.integrate.odeint(ODE_1, y0, t)
23
24 x_scipy = sol[:,0]
25
26 #metodo di eulero
27 def ODE_2(x):
28     """
29     equazione da risolvere per eulero
30     """
31     x_dot = x
32     return x_dot
33
34 def eulero(N, tf, x0):
35     """
36     si usa che dx/dt = (x[i+1]-x[i])/dt
37     che e' praticamente la definizione di rapporto incrementale
38     discretizzata la derivata sappiamo a cosa eguagliarla
39     perche dx/dt = g(x(t)) nella fattispecie g(x) = x
40     quindi discretizzando tutto:
41     (x[i+1]-x[i])/dt = x[i]
```

```

42     da cui si isola x[i+1]
43     """
44     dt = tf/N #passo di integrazione
45     x = np.zeros(N+1)
46     x[0] = x0
47
48     for i in range(N):
49         x[i+1] = x[i] + dt*ODE_2(x[i])
50
51     return x
52
53 x_eulero = eulero(N, tf, x0)
54
55 plt.figure(1)
56
57 ax1 = plt.subplot(211)
58 ax1.set_title('Risoluzione numerica')
59 ax1.set_xlabel('t')
60 ax1.set_ylabel('x')
61 ax1.plot(t, x_scipy, label='scipy')
62 ax1.plot(t, x_eulero, label='eulero')
63 ax1.legend(loc='best')
64 ax1.grid()
65
66 ax2 = plt.subplot(223)
67 ax2.set_title('Differenza tra metodo di eulero e soluzione esatta')
68 ax2.set_xlabel('t')
69 ax2.set_ylabel('x')
70 ax2.plot(t, x_eulero-np.exp(t))
71 ax2.grid()
72
73
74 ax3 = plt.subplot(224)
75 ax3.set_title('Differenza tra odeint e soluzione esatta')
76 ax3.set_xlabel('t')
77 ax3.set_ylabel('x')
78 ax3.plot(t, x_scipy-np.exp(t))
79 ax3.grid()
80
81 plt.show()

```



Vediamo che entrambi i metodi sembrano funzionare bene, scipy usa un integratore migliore rispetto ad euler infatti vediamo che la differenza fra le due soluzioni è 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:

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

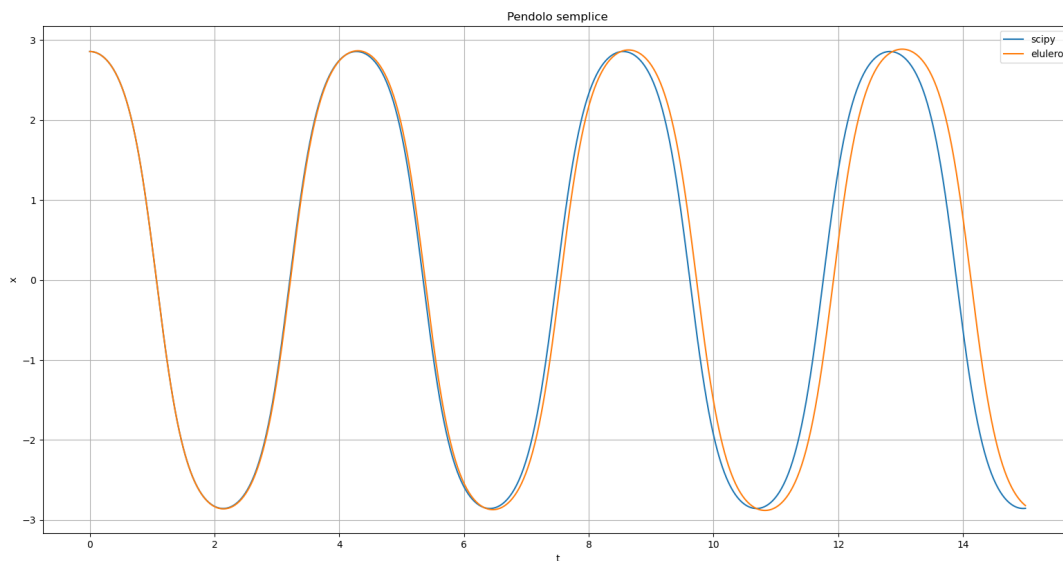
È la famosa equazione del pendolo semplice che approssimata dà luogo all'oscillatore armonico ovvero a tutta la fisica. Vediamo come si modifica il codice di sopra ora:

```
1 import numpy as np
2 import scipy.integrate
3 import matplotlib.pyplot as plt
4
5 #parametri
6 N = 100000          #numero di punti
7 l = 1               #lunghezza pendolo
8 g = 9.81            #accelerazione di gravita'
9 o0 = g/l            #frequenza piccole oscillazioni
10 v0 = 0              #condizioni iniziali velocita'
11 x0 = np.pi/1.1     #condizioni iniziali posizione
12 tf = 15             #fin dove integrare
13
14 #odeint
15 def ODE_1(y, t):
16     """
17     equazione da risolvere per odeint
18     """
19     theta, omega = y
20     dydt = [omega, - o0*np.sin(theta)]
21     return dydt
22
23
24 y0 = [x0, v0] #x(0), x'(0)
25 t = np.linspace(0, tf, N+1)
26 sol = scipy.integrate.odeint(ODE_1, y0, t)
27
28 x_scipy = sol[:,0]
29
30 #metodo di eulero
31 def ODE_2(x, v):
32     """
33     equazione da risolvere per eulero
34     """
35     x_dot = v
36     v_dot = -o0*np.sin(x)
37     return x_dot, v_dot
38
39 def eulero(N, tf, x0, v0):
40     """
41     si usa che dx/dt = (x[i+1]-x[i])/dt
42     che e' praticamente la definizione di rapporto incrementale
43     discretizzata la derivata sappiamo a cosa eguagliarla
44     perche dx/dt = g(x(t)) nella fattispecie g(x) = x
45     quindi discretizzando tutto:
46     (x[i+1]-x[i])/dt = x[i]
47     da cui si isola x[i+1]
48     """
49     dt = tf/N #passo di integrazione
50     x = np.zeros(N+1)
51     v = np.zeros(N+1)
52     x[0], v[0] = x0, v0
53
54     for i in range(N):
55         dx, dv = ODE_2(x[i], v[i])
56         x[i+1] = x[i] + dt*dx
57         v[i+1] = v[i] + dt*dv
58
59     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='eulero')
71 plt.legend(loc='best')
72 plt.grid()
73
74 plt.show()

```



Dal grafico vediamo che le due soluzioni si distaccano, questo è dovuto al fatto che l'integrazione con il metodo di euler 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).

### 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()".

```

1 import numpy as np
2 import scipy.integrate
3 from matplotlib import animation
4 import matplotlib.pyplot as plt
5
6 #parametri
7 N = 10000          #numero di punti
8 l = 1              #lunghezza pendolo
9 g = 9.81           #accelerazione di gravita'
10 o0 = g/l           #frequenza piccole oscillazioni
11 v0 = 0             #condizioni iniziali velocita'
12 x0 = np.pi/1.1    #condizioni iniziali posizione
13 tf = 15            #fin dove integrare
14
15 #odeint
16 def ODE_1(y, t):

```

```

17     """
18     equazione da risolvere per odeint
19     """
20     theta, omega = y
21     dydt = [omega, - o0*np.sin(theta)]
22     return dydt
23
24
25 y0 = [x0, v0] #x(0), x'(0)
26 t = np.linspace(0, tf, N+1)
27 sol = scipy.integrate.odeint(ODE_1, y0, t)
28
29 #passaggio in cartesiane
30 theta = sol[:,0]
31 x = l*np.sin(theta)
32 y = -l*np.cos(theta)
33
34 #grafico e bellurie
35 fig = plt.figure(1, figsize=(10, 6))
36 plt.suptitle('Pendolo semplice')
37 ax = fig.add_subplot(121)
38 time_template = 'time = %.1fs'
39 time_text = ax.text(0.05, 0.9, '', transform=ax.transAxes)
40 plt.xlim(-2, 2)
41 plt.ylim(-2, 2)
42 plt.gca().set_aspect('equal', adjustable='box')
43
44 #coordinate del perno e della pallina
45 xf, yf = [0,x[0]],[0,y[0]]
46
47 line1, = plt.plot(xf, yf, linestyle='-', marker='o',color='k')
48
49 plt.grid()
50
51 def animate(i):
52     """
53     funzione che a ogni i aggiorna le corrdinate della pallina
54     """
55     xf[1] = x[i]
56     yf[1] = y[i]
57     line1.set_data(xf, yf)
58     time_text.set_text(time_template % (i*t[1]))
59
60     return line1, time_text
61
62 #funzione che fa l'animazione vera e propria
63 anim = animation.FuncAnimation(fig, animate, frames=range(0, len(t), 5), interval=1, blit=True
64     , repeat=True)
65
66 plt.subplot(122)
67 plt.ylabel(r'$\theta$(t) [rad]')
68 plt.xlabel('t [s]')
69 plt.plot(t, theta)
70 plt.grid()
71 plt.show()

```

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



## A Zeri di una funzione

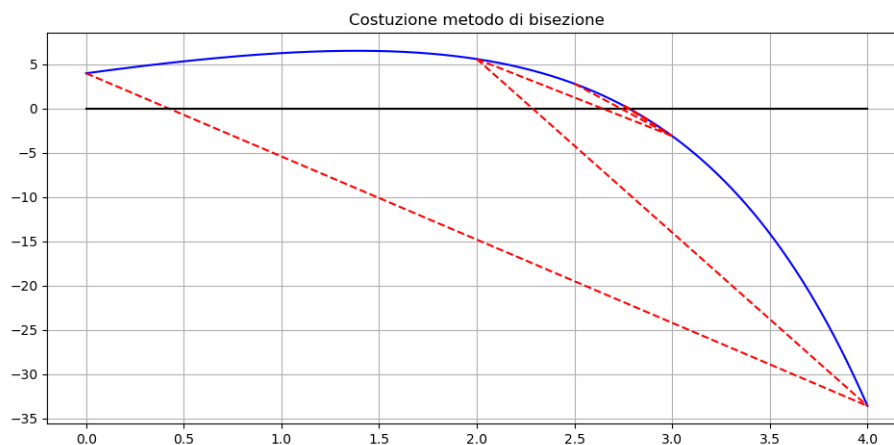
Capita spesso la necessità di trovare gli zeri di una funzione, o più, per risolvere un'equazione o un sistema di equazioni. Brevemente vedremo due metodi per la risoluzione di un'equazione, quindi per trovare lo zero, o gli zeri, di una funzione: il metodo di bisezione e il metodo di Newton, o delle 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):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 def f(x) :
6     """
7     funzione di cui trovare lo zero
8     """
9     return 5.0+4.0*x-np.exp(x)
10
11 a = 0.0 #estremo sinistro dell'intervallo
12 b = 4.0 #estremo destro dell'intervallo
13 t = 1.0e-15 #tolleranza
14
15 x=np.linspace(a, b, 1000)
16 #plot per vedere come scegliere gli estremi
17 plt.figure(1)
18 plt.plot(x, f(x))
19 plt.grid()
20 plt.show()
21
22 ##metodo bisezione
23 fa = f(a)
24 fb = f(b)
25 if fa*fb>0:
26     print("potrebbero esserci piu' soluzioni" , fa , fb)
27     """
28     Potrebbero esserci piu' zeri anche se la condizione non fosse verificata
29     Ma se la condizione e' verificata allora di certo ci sono piu' soluzioni
30     non e' un se e solo se
31     """
32
33 iter = 1
34 #fai finche' l'intervallo e' piu' grande della tolleranza
35 while (b-a) > t:
36     c = (a+b)/2.0 #punto medio
37     fc = f(c)
38     #se hanno lo stesso segno allora c e' piu' vicino allo zero che a
39     if fc*fa > 0:
40         a = c
41     #altrimenti e' b ad essere piu' lontano
42     else:
43         b = c
44     iter += 1
45
46 print(iter , " iterazioni necessarie:")
47 print("x0 = " ,c)
48 print("accuracy = " , '{:.2e}'.format(b-a))
49 print("f (x0)=" ,f(c))
50
51 [Output]
52 53 iterazioni necessarie:
53 x0 = 2.780080782051699
54 accuracy = 8.88e-16
55 f (x0)= 7.105427357601002e-15
```

Vediamo graficamente cosa succede:



Per generare il grafico precedente si può fare così:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 a = 0.0 #estremo sinistro dell'intervallo
5 b = 4.0 #estremo destro dell'intervallo
6 t = 1.0e-15 #tolleranza
7
8 plt.figure(2)
9 plt.title('Costuzione metodo di bisezione')
10 plt.plot(x, f(x), 'b')
11 plt.plot([a, b], [f(a), f(b)], linestyle='--', c='r')
12 plt.plot(x, x*0, 'k')
13
14 iter = 1
15 #fai finche' l'intervallo e' piu' grande della tolleranza
16 while (b-a) > t:
17     c = (a+b)/2.0 #punto medio
18     fc = f(c)
19     #se hanno lo stesso segno allora c e' piu' vicino allo zero che a
20     if fc*fa > 0:
21         a = c
22     #altrimenti e' b che e' piu' lontano
23     else:
24         b = c
25     iter += 1
26     plt.plot([a, b], [f(a), f(b)], linestyle='--', c='r')
27
28 plt.grid()
29 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 quindi 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.

```

1 import sympy as sp
2
3 x = sp.Symbol('x')
4 f = sp.tan(x)-x #funzione di cui trovare gli zeri
5 df = sp.diff(f, x) #derivata della funzione f
6 t = 1e-13 #tolleranza

```

```

7
8 def tangenti(x0, t):
9     iter = 1
10    while abs(f.subs(x, x0))>=t:
11        x0 = x0 - ( f.subs(x,x0) / df.subs(x,x0) )
12        iter += 1
13        if iter > 10000 or abs(f.subs(x, x0))>500:
14            if iter > 10000:
15                raise Exception('troppe iterazioni')
16
17            if abs(f.subs(x, x0))>500:
18                raise Exception('la soluzione sta divergendo\nscegliere meglio il punto di
19                partenza')
20
21    return x0, iter
22
23 #valore iniziale da cui partire
24 init = 4.4
25
26
27 xs, iter = tangenti(init, t)
28
29 print(iter , " iterazioni necessarie")
30
31
32 print("xs= %.15f" %xs)
33
34 print("|f(xs)|= %e" %abs(f.subs(x,xs)))
35
36 [Output]
37 7 iterazioni necessarie
38 xs= 4.493409457909064
39 |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:

```

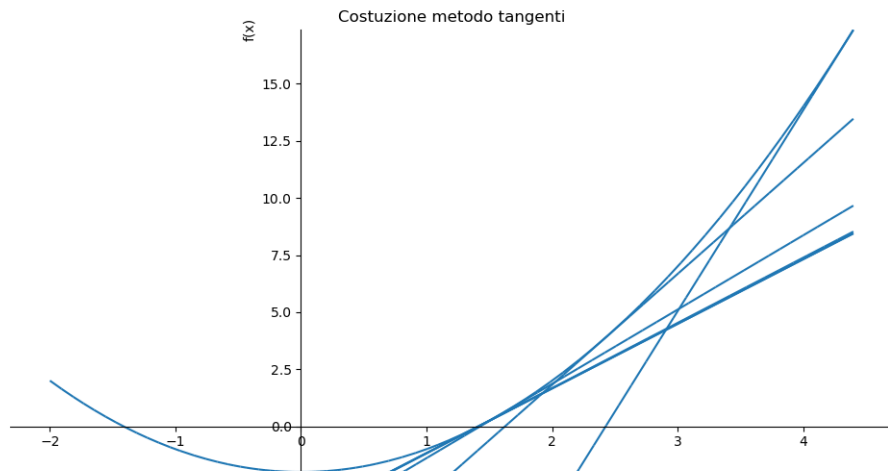
1 import sympy as sp
2 from sympy.plotting import plot
3
4 x = sp.Symbol('x')
5 f = x**2 -2 #funzione di cui trovare gli zeri
6 df = sp.diff(f, x) #derivata della funzione f
7 t = 1e-13 #tolleranza
8 init = 4.4
9 P1 = plot(f, (x, -2, init), ylim=(-2.2, f.subs(x,init)), show=False, title='Costuzione metodo
10 tangenti')
11
12 def tangenti(x0, t):
13     iter = 1
14     while abs(f.subs(x, x0))>=t:
15         P2 = plot(f.subs(x,x0)+(x-x0)*df.subs(x,x0), (x, -2, init), ylim=(-2.2, f.subs(x,init)
16         ), show=False)
17         P1.extend(P2)
18         x0 = x0 - ( f.subs(x,x0) / df.subs(x,x0) )
19         iter += 1
20         if iter > 10000 or abs(f.subs(x, x0))>500:
21             if iter > 10000:
22                 raise Exception('troppe iterazioni')
23
24             if abs(f.subs(x, x0))>500:
25                 raise Exception('la soluzione sta divergendo\nscegliere meglio il punto di
26                 partenza')
27
28     return x0, iter
29
30 #valore iniziale da cui partire
31 xs, iter = tangenti(init, t)
32
33 print(iter , " iterazioni necessarie")
34 print("xs= %.15f" %xs)
35 print("|f(xs)|= %e" %abs(f.subs(x,xs)))
36
37 P1.show()
38
39

```

```

37 [Output]
38 7 iterazioni necessarie
39 xs= 1.414213562373095
40 |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:

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

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:

```

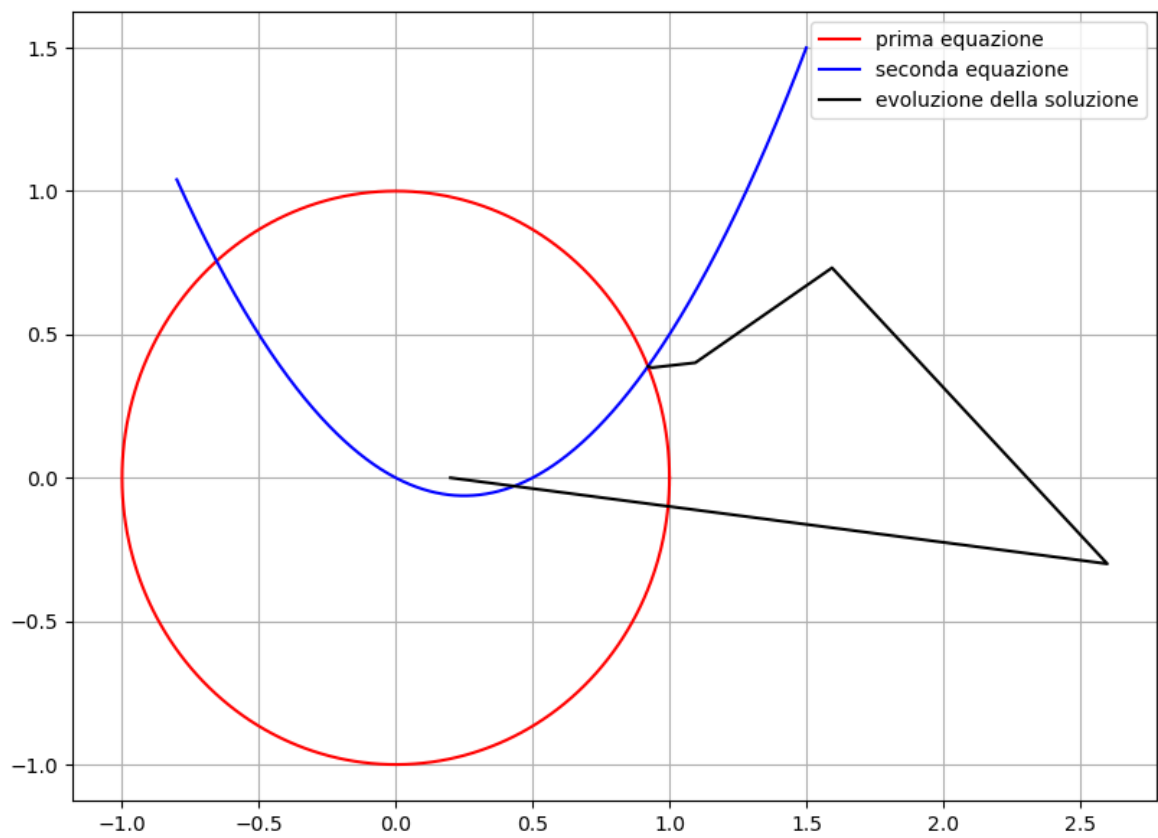
1
2 import sympy as sp
3 import numpy as np
4 import matplotlib.pyplot as plt
5 from scipy.optimize import fsolve, root
6
7 ## risoluzione "manuale"
8 x = sp.Symbol('x')
9 y = sp.Symbol('y')
10
11 f1 = x**2 + y**2 - 1
12 f2 = y - x**2 + x/2
13
14 f1x = sp.diff(f1, x)
15 f1y = sp.diff(f1, y)
16 f2x = sp.diff(f2, x)
17 f2y = sp.diff(f2, y)
18
19 #valori iniziali da cui partire che vengono

```

```

20 #ongi volta aggiornati fino ad arrivare alla soluzione
21 x0 = 0.2
22 y0 = 0.0
23
24 tau = 1e-10
25 iter = 0
26 xs = np.array([x0])
27 ys = np.array([y0])
28
29 while (abs(f1.subs(x, x0).subs(y, y0))>tau and abs(f2.subs(x, x0).subs(y, y0)) > tau):
30     #calcolo jacobiano
31     a11 = f1x.subs(x, x0).subs(y, y0)
32     a12 = f2x.subs(x, x0).subs(y, y0)
33     a21 = f1y.subs(x, x0).subs(y, y0)
34     a22 = f2y.subs(x, x0).subs(y, y0)
35
36     det_a = a11*a22 - a12*a21
37     #calcolo inverso
38     b11 = a22/det_a
39     b12 = -a21/det_a
40     b21 = -a12/det_a
41     b22 = a11/det_a
42
43     f1_0 = f1.subs(x, x0).subs(y, y0)
44     f2_0 = f2.subs(x, x0).subs(y, y0)
45     #risolvo
46     d1 = -(b11*f1_0 + b12*f2_0)
47     d2 = -(b21*f1_0 + b22*f2_0)
48     #aggiorno le coordinate
49     x0 = x0 + d1
50     y0 = y0 + d2
51     #conservo per fare il plot
52     xs = np.insert(xs, len(xs), x0)
53     ys = np.insert(ys, len(ys), y0)
54
55     iter += 1
56
57 print(f"x_0: {xs[-1]} e y_0: {ys[-1]} raggiunti in {iter} iterazioni")
58
59 t = np.linspace(0, 2*np.pi, 1000)
60 z = np.linspace(-0.8, 1.5, 1000)
61
62 plt.figure(1)
63 plt.grid()
64 plt.plot(np.cos(t), np.sin(t), 'r', label='prima equazione')
65 plt.plot(z, z**2- z/2, 'b', label='seconda equazione')
66 plt.plot(xs, ys, 'k', label='evoluzione della soluzione')
67 plt.legend(loc='best')
68 plt.show()
69
70 ##risoluzione con fsolve di scipy
71
72 def sistema(V):
73     x1, x2 = V
74     r1 = x1**2 + x2**2 - 1
75     r2 = x2 - x1**2 + x1/2
76     return[r1 , r2]
77
78 start = (0.2, 0.0)
79 sol = fsolve(sistema , start, xtol=1e-10)
80 print("soluzione con fsolve:", sol)
81
82 ##risoluzione con root di scipy
83
84 def sistema(V):
85     x1, x2 = V
86     r1 = x1**2 + x2**2 - 1
87     r2 = x2 - x1**2 + x1/2
88     return[r1 , r2]
89
90 start = (0.2, 0.0)
91 sol = root(sistema, start, method='hybr')
92 print("soluzione con root:", sol.x)

```



## 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 vedremo, 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  $u_j^n = \xi^n \exp i k j \Delta x$  e si vede che l'ampiezza  $\xi$  diverge per ogni scelta di  $\Delta t$  e  $\Delta x$  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  $\frac{v\Delta t}{\Delta x} < 1$

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import matplotlib.animation as animation
4
5 N = 100      #numero punti sulle x
6 T = 400      #numero di punti nel tempo
7 v = 1        #velocita' di propagazione
8 dt = 0.001   #passo temporale
9 dx = 0.01    #passo spaziale
10
11 alpha = v*dt/dx #<1
12 print(alpha)
13
14 Sol = np.zeros((N+1, T))
15 sol_v = np.zeros(N+1)
16 sol_n = np.zeros(N+1)
17
18 #condizione iniziale
19 q = 2*np.pi
20 x = np.linspace(0, (N+1)*dx, N+1)
21 sol_v = np.sin(q*1*x)
22 Sol[:, 0] = sol_v
23
24 #evoluzione temporale con lax
25 for time in range(1, T):
26     for j in range(1, N):
27         sol_n[j] = 0.5*(sol_v[j+1]*(1 - alpha)) + 0.5*(sol_v[j-1]*(1 + alpha))
28
29     #condizione periodiche al bordo
30     sol_n[0] = sol_n[N-1]
31     sol_n[N] = sol_n[1]
32
33     #aggiorno la soluzione
34     sol_v = sol_n
35
36     #conservo la soluzione per l'animazione
37     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     line.set_data(x, Sol[:, i])
63     return line,
64
65 anim = animation.FuncAnimation(fig, animate, frames=np.arange(0, T, 1) ,interval=10, blit=True
66     , 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 timestep = 5000 #punti sul tempo
9 T = np.zeros((N,timestep))
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,timestep):
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])

```



```

24 #     T[0,time]=T[1,time] #per avere bordi non fissi
25 #     T[N-1,time]=T[N-2,time]
26
27 fig = plt.figure(1)
28 ax = fig.gca(projection='3d')
29 gridx, gridy = np.meshgrid(range(tstep), range(N))
30 ax.plot_surface(gridx,gridy,T, cmap=mp.cm.coolwarm,vmax=250,linewidth=0,rstride=2, cstride
    =100)
31 ax.set_title('Diffusione del calore')
32 ax.set_xlabel('Tempo')
33 ax.set_ylabel('Lunghezza')
34 ax.set_zlabel('Temperatura')
35
36 fig = plt.figure(2)
37 plt.xlim(np.min(x), np.max(x))
38 plt.ylim(np.min(T), np.max(T))
39
40 line, = plt.plot([], [], 'b')
41 def animate(i):
42     line.set_data(x, T[:,i])
43     return line,
44
45
46 anim = animation.FuncAnimation(fig, animate, frames=tstep, interval=10, blit=True, repeat=True
    )
47
48 plt.grid()
49 plt.title('Diffusione del calore')
50 plt.xlabel('Distanza')
51 plt.ylabel('Temperatura')
52
53 #anim.save('calore.mp4', fps=30, extra_args=['-vcodec', 'libx264'])
54
55 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).

```
1 import matplotlib as mp
2 import matplotlib.pyplot as plt
3
4
5 #il file txt su cui scrivere se non esiste viene creato automaticamente
6
7 path_dati = "C:\\Users\\franc\\Desktop\\dati0.txt"
8 path_img = "C:\\Users\\franc\\Documents\\DatiL\\datiL3\\FIS2\\e0verm\\DSC_0005.jpg"
9
10 fig, ax = plt.subplots()
11
12 img = mp.image.imread(path_img)
13
14 ax.imshow(img)
15
16
17 def onclick(event):
18     #apre file, il permesso e' a altrimenti sovrascriverebbe i dati
19     file= open(path_dati, "a")
20
21     x=event.xdata
22     y=event.ydata
23     print('x=%f, y=%f' %(x, y)) #stampa i dati sulla shell
24
25     #scrive i dati sul file belli pronti per essere letti da codice del fit
26     file.write(str(x))
27     file.write('\t')
28     file.write(str(y))
29     file.write('\n')
30     file.close() #chiude il file
31
32
33 fig.canvas.mpl_connect('button_press_event', onclick)
34
35
36 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

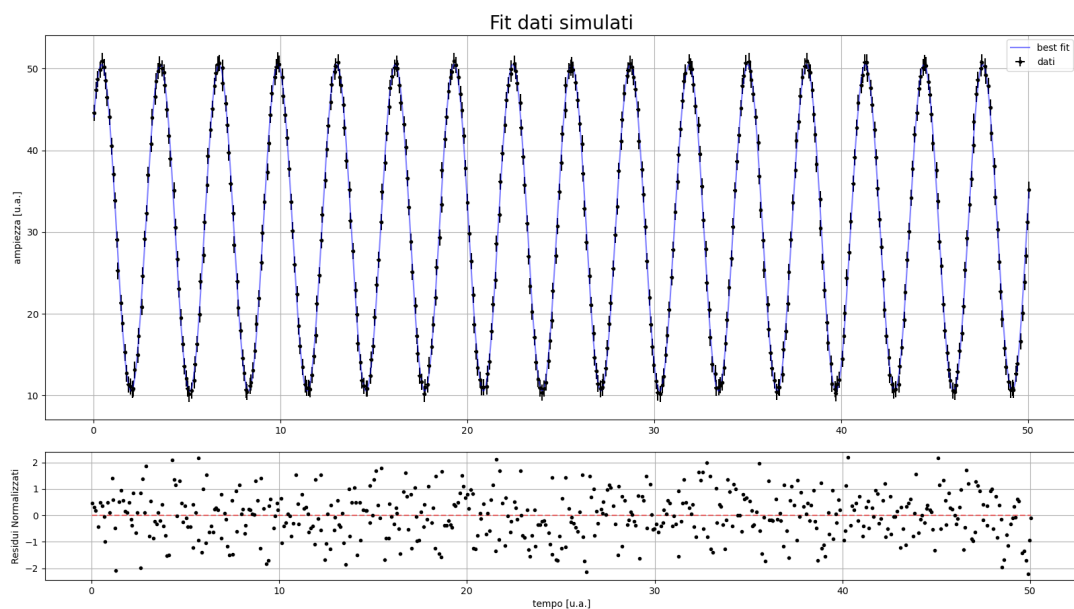
La libreria scipy grazie alla funzione "curve\_fit()" ci permette di eseguire un gran numero di fit; riportiamo sotto un esempio di codice e relativi risultati e grafico:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.optimize import curve_fit
4
5 #Importiamo i dati (va inserito il path assoluto per permettere di trovare) e definiamo la
  funzione di fit:
6 #x, y= np.loadtxt(r'C:\Users\franc\Desktop\datiL\Datil2\onda.txt', unpack = True)
7 N = 500
8 ex, ey = 0.1, 1
9 dy = np.array(N*[ey])
10 dx = np.array(N*[ex])
11 x = np.linspace(0, 50, N)
12
13 A1 = 20
14 o1 = 2
15 v1 = 30
16 phi = np.pi/4
17
18 y = A1*np.sin(o1*x + phi) + v1
19 k = np.random.uniform(0, ey, N)
20 l = np.random.uniform(0, ex, N)
21 y = y + k #aggiungo errore
22 x = x + l
23
24 def f(x, A, o, f, v):
25     '''funzione modello
26     '''
27     return A*np.sin(o*x + f) + v
28
29 """
30 definiamo un array di parametri iniziali contenente
31 i valori numerici che ci si aspetta il fit restituisca,
32 per aiutare la convergenza dello stesso:
33 init = np.array([A, o, f, v])
34 """
35 init = np.array([25, 2.1, 3, 29])
36
37
38 #Eseguiamo il fit e stampiamo i risultati:
39 pars, covm = curve_fit(f, x, y, init, sigma=dy, absolute_sigma=False)
40 print('A = %.5f +- %.5f ' % (pars[0], np.sqrt(covm.diagonal()[0])))
41 print('o = %.5f +- %.5f ' % (pars[1], np.sqrt(covm.diagonal()[1])))
42 print('f = %.5f +- %.5f ' % (pars[2], np.sqrt(covm.diagonal()[2])))
43 print('v = %.5f +- %.5f ' % (pars[3], np.sqrt(covm.diagonal()[3])))
44
45 #Calcoliamo il chi quadro, indice ,per quanto possibile, della bonta' del fit:
46 chisq = sum(((y - f(x, *pars))/dy)**2.)
47 ndof = len(y) - len(pars)
48 print(f'chi quadro = {chisq:.3f} ({ndof:d} dof)')
49
50
51 #Definiamo un matrice di zeri che divvera' la matrice di correlazione:
52 c=np.zeros((len(pars),len(pars)))
53 #Calcoliamo le correlazioni e le inseriamo nella matrice:
54 for i in range(0, len(pars)):
55     for j in range(0, len(pars)):
56         c[i][j] = (covm[i][j])/(np.sqrt(covm.diagonal()[i])*np.sqrt(covm.diagonal()[j]))
57 print(c) #matrice di correlazione
58
59
60 #Grafichiamo il risultato
61 fig1 = plt.figure(1)
62 #Parte superiore contenente il fit:
63 frame1=fig1.add_axes((.1,.35,.8,.6))
64 #frame1=fig1.add_axes((trasla lateralmente, trasla verticalmente, larghezza, altezza))
```

```

65 frame1.set_title('Fit dati simulati',fontsize=20)
66 plt.ylabel('ampiezza [u.a.]',fontsize=10)
67 #plt.ticklabel_format(axis = 'both', style = 'sci', scilimits = (0,0))#notazione scientifica
   sugliassi
68 plt.grid()
69
70 #grafichiamo i punti e relative barre d'errore
71 plt.errorbar(x, y, dy, dx, fmt='.', color='black', label='dati')
72 t = np.linspace(np.min(x),np.max(x), 10000)
73 s = f(t, *pars)
74 plt.plot(t,s, color='blue', alpha=0.5, label='best fit') #grafico del best fit
75 plt.legend(loc='best')#inserisce la legenda nel posto migliore
76
77
78 #Parte inferiore contenente i residui
79 frame2=fig1.add_axes((.1,.1,.8,.2))
80
81 #Calcolo i residui normalizzati
82 ff = (y-f(x, *pars))/dy
83 frame2.set_ylabel('Residui Normalizzati')
84 plt.xlabel('tempo [u.a.]',fontsize=10)
85 #plt.ticklabel_format(axis = 'both', style = 'sci', scilimits = (0,0))
86
87
88 plt.plot(t, 0*t, color='red', linestyle='--', alpha=0.5) #grafico la retta costantemente zero
89 plt.plot(x, ff, '.', color='black') #grafico i residui normalizzati
90 plt.grid()
91
92 plt.show()
93
94 [Output]
95 A = 19.95561 +- 0.05547
96 o = 1.99991 +- 0.00019
97 f = 6.96973 +- 0.00565
98 v = 30.54562 +- 0.03930
99 chi quadro = 382.795 (496 dof)
100 [[ 1. 0.00576834 -0.00302643 0.00468354]
101 [ 0.00576834 1. -0.86984711 -0.02110156]
102 [-0.00302643 -0.86984711 1. 0.02174393]
103 [ 0.00468354 -0.02110156 0.02174393 1. ]]

```



## D.2 Fit circolare, metodo di Coope

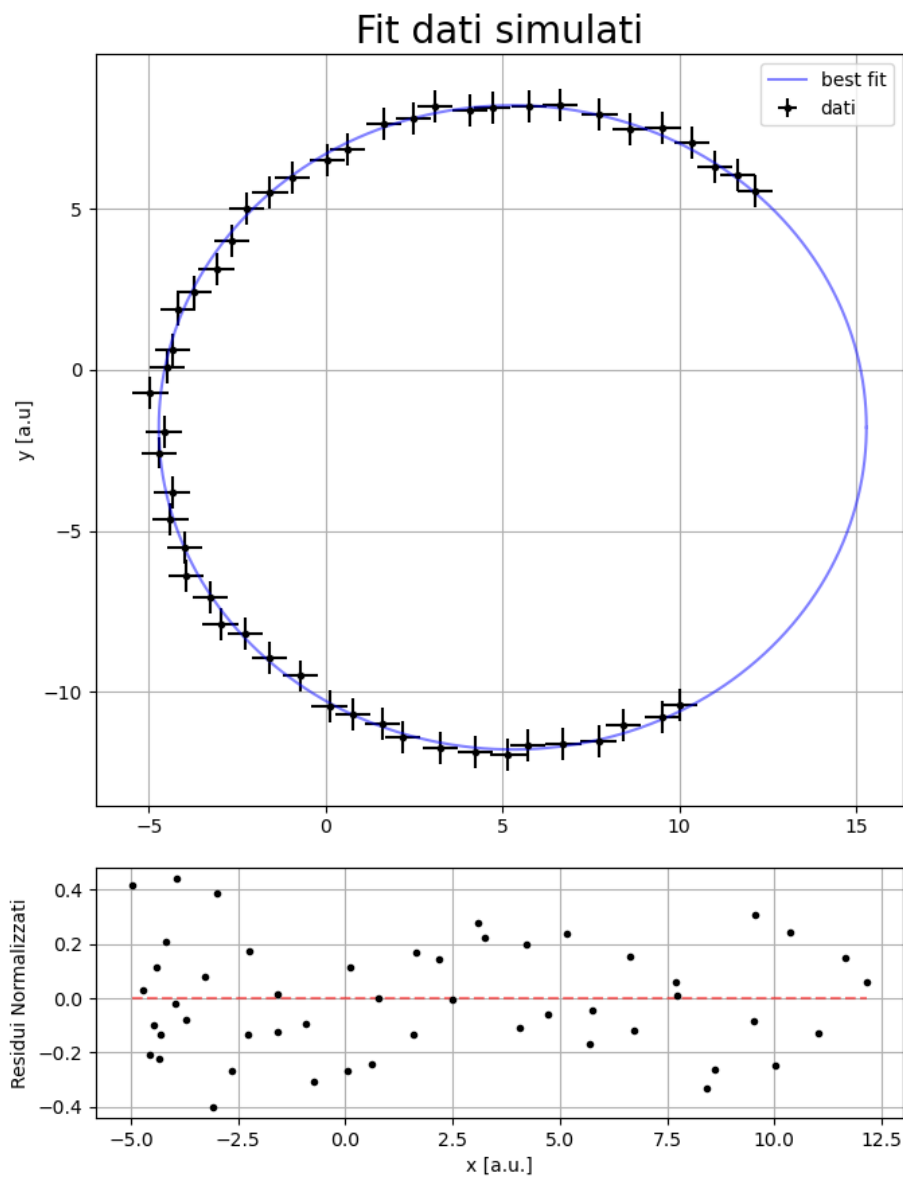
Ci sono casi in cui, come per un circonferenza o un'ellisse, curve fit non è comodo da usare, in quanto non si tratta di vere e proprie funzioni. Mostriamo un esempio di fit circolare seguito con il metodo di Coope e riportiamo qui il link all'articolo originale: <https://core.ac.uk/download/pdf/35472611.pdf>

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 def cerchio(xc, yc, r, N, phi_min=0, phi_max=2*np.pi):
6     """
7     Restituisce un cerchio di centro (xc, yc) e di raggio r
8     phi e' il parametro di "percorrenza" del cerchio
9     """
10
11     phi = np.linspace(phi_min, phi_max, N)
12
13     x = xc + r*np.cos(phi)
14     y = yc + r*np.sin(phi)
15
16     return x, y
17
18
19 def fitcerchio(pt):
20     """
21     fit di un cerchio con metodo di coope
22     Parameters
23     -----
24     pt : 2Darray
25         contiene le coordinate del cerchio
26
27     Returns
28     -----
29     c : 1Darray
30         array con le coordinate del centro del cerchio
31     r : float
32         raggio del cerchio
33     d : 1Darray
34         array con gli errori associati a c ed r
35     A1 : 2Darray
36         matrice di covarianza
37     """
38
39     npt = len(pt[0])
40     S = np.column_stack((pt.T, np.ones(npt)))
41     y = (pt**2).sum(axis=0)
42
43     A = S.T @ S #@ e' il prodotto matriciale
44     b = S.T @ y
45     sol = np.linalg.solve(A, b)
46
47     c = 0.5*sol[:-1]
48     r = np.sqrt(sol[-1] + c.T @ c)
49
50     d = np.zeros(3)
51     A1 = np.linalg.inv(A)
52
53     for i in range(3):
54         d[i] = np.sqrt(A1[i,i])
55     return c, r, d, A1
56
57
58 if __name__ == "__main__":
59     #numero di punti
60     N = 50
61     #parametri cerchio
62     xc, yc, r1 = 5, -2, 10
63     #errori
64     ex, ey= 0.5, 0.5
65     dy = np.array(N*[ey])
66     dx = np.array(N*[ex])
67     dr = np.sqrt(dx**2 + dy**2)
68     k = np.random.uniform(0, ex, N)
69     l = np.random.uniform(0, ey, N)
70     #creiamo il cerchio
71     x, y = cerchio(xc, yc, r1, N, np.pi/4, 5/3*np.pi)
```

```

72 x = x + k #aggiungo errore
73 y = y + l
74
75 a = np.array([x, y])
76 c, r, d, A = fitcerchio(a) #fit
77
78 print(f'x_c = {c[0]:.5f} +- {d[0]:.5f}; valore esatto={xc:.5f}')
79 print(f'y_c = {c[1]:.5f} +- {d[1]:.5f}; valore esatto={yc:.5f}')
80 print(f'r = {r:.5f} +- {d[2]:.5f}; valore esatto={r1:.5f}')
81
82
83 chisq = sum(((np.sqrt((x-c[0])**2 + (y-c[1])**2) - r)/dr)**2.)
84 ndof = N - 3
85 print(f'chi quadro = {chisq:.3f} ({ndof:d} dof)')
86
87 corr=np.zeros((3,3))
88 for i in range(0, 3):
89     for j in range(0, 3):
90         corr[i][j]=(A[i][j])/(np.sqrt(A.diagonal()[i])*np.sqrt(A.diagonal()[j]))
91 print(corr)
92
93 #plot
94 fig1 = plt.figure(1, figsize=(7.5,9.3))
95 frame1=fig1.add_axes((.1,.35,.8,.6))
96 #frame1=fig1.add_axes((trasla lateralmente, trasla verticalmente, larghezza, altezza))
97 frame1.set_title('Fit dati simulati',fontsize=20)
98 plt.ylabel('y [a.u.],fontsize=10)
99 plt.grid()
100
101 plt.errorbar(x, y, dy, dx, fmt='.', color='black', label='dati')
102 xx, yy = cerchio(c[0], c[1], r, 10000)
103 plt.plot(xx, yy, color='blue', alpha=0.5, label='best fit')
104 plt.legend(loc='best')
105
106
107 frame2=fig1.add_axes((.1,.1,.8,.2))
108 frame2.set_ylabel('Residui Normalizzati')
109 plt.xlabel('x [a.u.],fontsize=10)
110
111 ff=(np.sqrt((x-c[0])**2 + (y-c[1])**2) - r)/dr
112 x1=np.linspace(np.min(x),np.max(x), 1000)
113 plt.plot(x1, 0*x1, color='red', linestyle='--', alpha=0.5)
114 plt.plot(x, ff, '.', color='black')
115 plt.grid()
116
117 plt.show()
118
119 [Output]
120 x_c = 5.28657 +- 0.02684; valore esatto=5.00000
121 y_c = -1.78606 +- 0.01830; valore esatto=-2.00000
122 r = 10.00627 +- 0.15355; valore esatto=10.00000
123 chi quadro = 2.075 (47 dof)
124 [[ 1. -0.11448697 -0.35569543]
125 [-0.11448697 1. 0.19860027]
126 [-0.35569543 0.19860027 1. ]]

```



### D.3 Fit di un'ellisse, 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.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 def ellisse(parametri, n, tmin=0, tmax=2*np.pi):
6     """
7     Resistuisce un'ellisse di centro (x0, y0),
8     di semiassi maggiore e minore (semi_M, semi_m)
9     inclinata di un anglo (phi) rispetto all'asse x
10    t e' il parametro di "percorrenza" dell'ellisse
11    """
12
13    x0, y0, semi_M, semi_m, phi = parametri
14    t = np.linspace(tmin, tmax, n)
15
16    x = x0 + semi_M*np.cos(t)*np.cos(phi) - semi_m*np.sin(t)*np.sin(phi)
17    y = y0 + semi_M*np.cos(t)*np.sin(phi) + semi_m*np.sin(t)*np.cos(phi)
18
19    return x, y
20
21
22 def cartesiano_a_polari(coef):
23     """
24     Converte i coefficienti di:  $ax^2 + bxy + cy^2 + dx + fy + g = 0$ 
25     nei coefficienti polari: centro, semiassi, inclinazione ed eccentricita'
26     Per dubbi sulla geometria: https://mathworld.wolfram.com/Ellipse.html
27     """
28     #i termini misti presentano un 2 nella forma piu' generale
29     a = coef[0]
30     b = coef[1]/2
31     c = coef[2]
32     d = coef[3]/2
33     f = coef[4]/2
34     g = coef[5]
35
36     #Controlliamo sia un ellisse (i.e. il fit sia venuto bene, forse)
37     den = b**2 - a*c
38     if den > 0:
39         Error = 'I coefficienti passati non sono un ellisse: b^2 - 4ac deve essere negativo'
40         raise ValueError(Error)
41
42     #Troviamo il centro dell'ellisse
43     x0, y0 = (c*d - b*f)/den, (a*f - b*d)/den
44
45     num = 2*(a*f**2 + c*d**2 + g*b**2 - 2*b*d*f - a*c*g)
46     fac = np.sqrt((a - c)**2 + 4*b**2)
47     #Troviamo i semiassi maggiori e minori
48     semi_M = np.sqrt(num/den/(fac - a - c))
49     semi_m = np.sqrt(num/den/(-fac - a - c))
50
51     #Controlliamo che il semiasse maggiore sia maggiore
52     M_gt_m = True
53     if semi_M < semi_m:
54         M_gt_m = False
55         semi_M, semi_m = semi_m, semi_M
56
57     #Troviamo l'eccentricita'
58     r = (semi_m/semi_M)**2
59     if r > 1:
60         r = 1/r
61     e = np.sqrt(1 - r)
62
63     #Troviamo l'angolo di inclinazione del semiasse maggiore dall'asse x
64     #l'angolo come solito e misurato in senso antiorario
65     if b == 0:
66         if a < c:
67             phi = 0
68         else:
69             phi = np.pi/2
70     else:
71
```



```

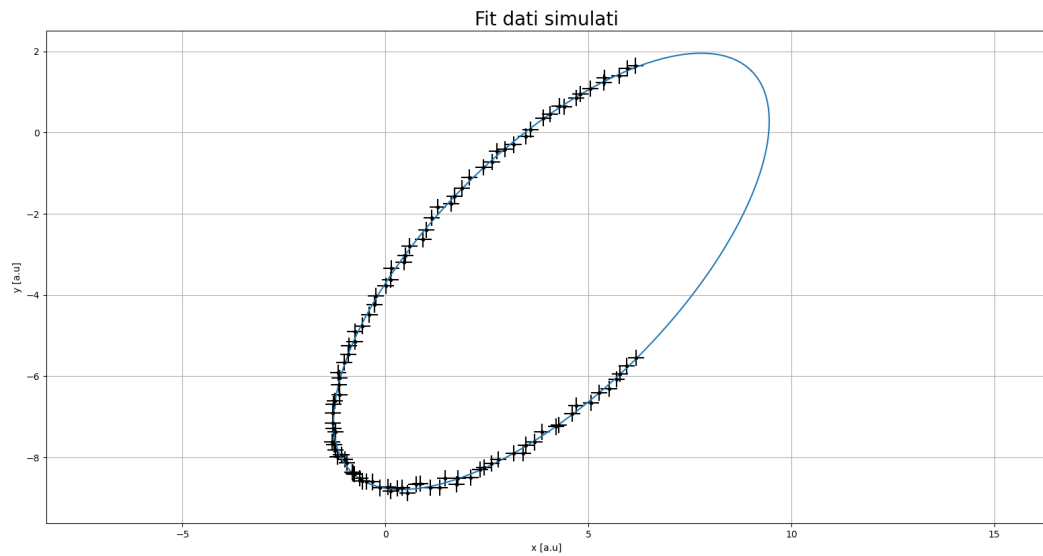
72     phi = np.arctan((2*b)/(a - c))/2
73     if a > c:
74         phi += np.pi/2
75
76     if not M_gt_m :
77         phi += np.pi/2
78
79     #periodicita' della rotazione
80     phi = phi % np.pi
81
82     return x0, y0, semi_M, semi_m, e, phi
83
84
85 def fit_ellisse(x, y):
86     """
87     Basato sull'articolo di Halir and Flusser,
88     "Numerically stable direct
89     least squares fitting of ellipses".
90     """
91
92     D1 = np.vstack([x**2, x*y, y**2]).T
93     D2 = np.vstack([x, y, np.ones(len(x))]).T
94
95     S1 = D1.T @ D1
96     S2 = D1.T @ D2
97     S3 = D2.T @ D2
98
99     T = -np.linalg.inv(S3) @ S2.T
100    M = S1 + S2 @ T
101    C = np.array(((0, 0, 2), (0, -1, 0), (2, 0, 0)), dtype=float)
102    M = np.linalg.inv(C) @ M
103
104    eigval, eigvec = np.linalg.eig(M)
105    cond = 4*eigvec[0]*eigvec[2] - eigvec[1]**2
106    ak = eigvec[:, cond > 0]
107
108    return np.concatenate((ak, T @ ak)).ravel()
109
110
111 if __name__ == "__main__":
112
113     #numero di punti
114     N = 100
115     #parametri dell'ellissi
116     x0, y0 = 4, -3.5
117     semi_M, semi_m = 7, 3
118     phi = np.pi/4
119     #eccentricita' non fondamentale per la creazione
120     r = (semi_m/semi_M)**2
121     if r > 1:
122         r = 1/r
123     e = np.sqrt(1 - r)
124
125     #errori
126     ex, ey = 0.2, 0.2
127     dy = np.array(N*[ey])
128     dx = np.array(N*[ex])
129     #creiamo l'ellisse
130     x, y = ellisse((x0, y0, semi_M, semi_m, phi), N, np.pi/4, 3/2*np.pi)
131     k = np.random.uniform(0, ex, N)
132     l = np.random.uniform(0, ey, N)
133     x = x + k #aggiungo errore
134     y = y + l
135
136     coef_cart = fit_ellisse(x, y) #fit
137
138     print('valori esatti:')
139     print(f'x0:{x0:.4f}, y0:{y0:.4f}, semi_M:{semi_M:.4f}, semi_m:{semi_m:.4f}, phi:{phi:.4f},
140           e:{e:.4f}')
141     x0, y0, semi_M, semi_m, e, phi = cartesiano_a_polari(coef_cart)
142     print('valori fittati')
143     print(f'x0:{x0:.4f}, y0:{y0:.4f}, semi_M:{semi_M:.4f}, semi_m:{semi_m:.4f}, phi:{phi:.4f},
144           e:{e:.4f}')
145
146     #plot
147     plt.figure(1)

```

```

146 plt.title('Fit dati simulati',fontsize=20)
147 plt.ylabel('y [a.u]',fontsize=10)
148 plt.xlabel('x [a.u]',fontsize=10)
149 plt.axis('equal')
150 plt.errorbar(x, y, dy, dx, fmt='.', color='black', label='dati')
151 x, y = ellisse((x0, y0, semi_M, semi_m, phi), 1000)
152 plt.plot(x, y)
153 plt.grid()
154 plt.show()
155
156 [Output]
157 valori esatti:
158 x0:4.0000, y0:-3.5000, semi_M:7.0000, semi_m:3.0000, phi:0.7854, e:0.9035
159 valori fittati
160 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:

```
1 import numpy as np
2
3 def GEN(r0, n=1, M=2**64, a=6364136223846793005, c=1442695040888963407, norm=True):
4     """
5     generatore congruenziale lineare
6     Parametri
7     -----
8     r0 : int
9         seed della generazione
10    n : int, opzionale
11        dimensione lista da generare, di default e' 1
12    M : int, opzionale
13        periodo del generatore di default e' 2**64
14    a : int, opzionale
15        moltiplicatore del generatore, di default e' 6364136223846793005
16    c : int, opzionale
17        incremento del generatore, di default e' 1442695040888963407
18    norm : bool, opzionale
19        se True il numero restituito e' fra zero ed 1
20
21    Returns
22    -----
23    r : list
24        lista con numeri distribuiti casualmente
25    """
26    if n==1:
27        r = (a*r0 + c)%M
28    else:
29        r = []
30        x = r0
31        for i in range(1, n):
32            x = (a*x + c)%M
33            r.append(x)
34
35    if norm :
36        if n==1:
37            return float(r)/(M-1)
38        else :
39            return [float(e1)/(M-1) for e1 in r]
40    else :
41        return r
42
43 if __name__ == '__main__':
44     seed = 42
45
46     print(GEN(seed, n=5))
47     momenti1 = [np.mean(np.array(GEN(seed, n=int(5e5))))**i for i in range(1, 10)]
48     momenti2 = [1/(1+p) for p in range(1, 10)]
49
50     for M1, M2 in zip(momenti1, momenti2):
51         print(f'{M1:.3f}, {M2:.3f}')
52
53 [Output]
```

```

54 [0.5682303266439077, 0.22546342894775137, 0.41283831882951183, 0.6303980498395979]
55 0.500, 0.500
56 0.333, 0.333
57 0.250, 0.250
58 0.200, 0.200
59 0.167, 0.167
60 0.143, 0.143
61 0.125, 0.125
62 0.111, 0.111
63 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  $\pi$ . 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  $\pi$  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  $\pi/4$ .

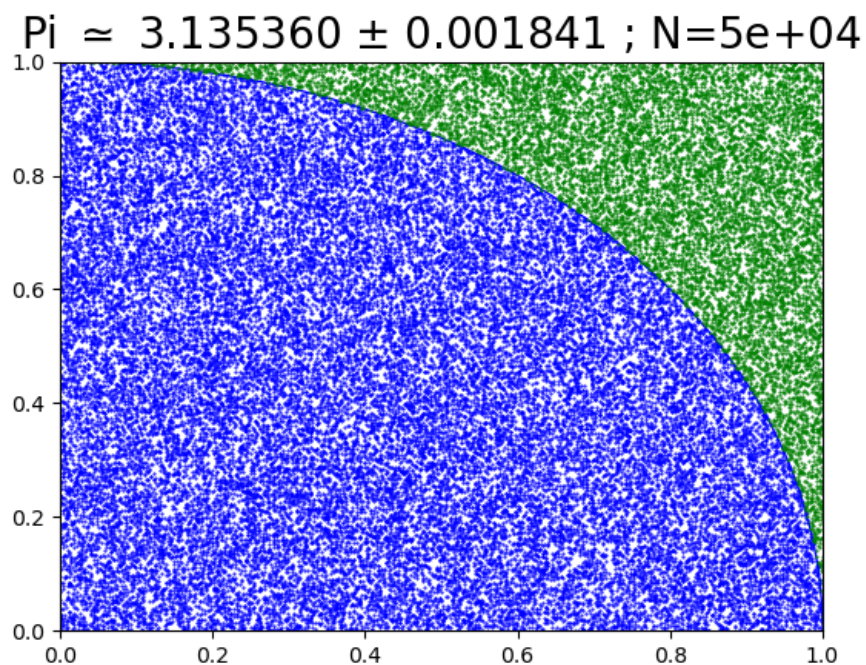
```

1 import time
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 N = int(5e4)
6 start_time=time.time()
7
8 x = np.linspace(0,1, 10000)
9
10 def f(x):
11     """
12     semi circonferenza superiore
13     """
14     return np.sqrt(1-x**2)
15
16 c = 0
17
18 for i in range(1,N):
19     #genero due variabili casuali uniformi fra 0 e 1
20     a = np.random.rand()
21     b = np.random.rand()
22     r = a**2 + b**2
23     #se vero aggiorno c di 1
24     if r < 1:
25         plt.errorbar(a, b, fmt='.', markersize=1, color='blue')
26         c += 1
27     else:
28         plt.errorbar(a, b, fmt='.', markersize=1, color='green')
29
30 #moltiplico per quattro essendo su un solo quadrante
31 Pi = 4*c/N
32 #propagazione errore, viene dalla binomiale
33 dPi = np.sqrt(c/N * (1-c/N))/np.sqrt(N)
34 print('%f +- %f' %(Pi, dPi))
35 print(np.pi)
36 print(abs((Pi-np.pi)/np.pi))
37
38 plt.figure(1)
39 plt.title('$\pi$ simeq $f$ $\pm$ $d$ ; N=%.0e' %(Pi, dPi, N), fontsize=20)
40 plt.xlim(0,1)
41 plt.ylim(0,1)
42 plt.plot(x, f(x),color='blue', lw=1)
43 plt.show()
44
45 print("--- %s seconds ---" % (time.time() - start_time))
46
47 [Output]
48 3.135360 +- 0.001841
49 3.141592653589793

```

```
50 0.001983915254790065
51 --- 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 sei 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 modificare all'occorrenza si potrebbe provare così:

```
1 import numpy as np
2 import sympy as sp
3
4 x = sp.Symbol('x')
5 y = sp.Symbol('y')
6 z = sp.Symbol('z')
7 t = sp.Symbol('t')
8
9 def Errore(x1, dx1, y1, dy1, z1, dz1, t1, dt1):
10     """
11     Prende in input certe quantità con un errore
12     e propaga l'errore su una certa funzione di queste
13     """
14     #funzione su cui propagare l'errore da modifica all'occorrenza
15     f1 = ((x-y)/(z+t))
16
17     #valor medio
18     f = float(f1.subs(x,x1).subs(y,y1).subs(z,z1).subs(t,t1))
19
20     #derivate parziali calcolate nel punto
21     a = sp.diff(f1, x).subs(x,x1).subs(y,y1).subs(z,z1).subs(t,t1)
22     b = sp.diff(f1, y).subs(x,x1).subs(y,y1).subs(z,z1).subs(t,t1)
23     c = sp.diff(f1, z).subs(x,x1).subs(y,y1).subs(z,z1).subs(t,t1)
24     d = sp.diff(f1, t).subs(x,x1).subs(y,y1).subs(z,z1).subs(t,t1)
25
26     #somma dei vari contributi
27     df1 = ((a*dx1)**2 + (b*dy1)**2 + (c*dz1)**2 + (d*dt1)**2 )
28     df = np.sqrt(float(df1))
29
30     return f, df
31
32
33 print(Erore(1, 0.1, 2, 0.1, 3, 0.1, 2, 0.1))
34
35 [Output]
36 (-0.2, 0.02884441020371192)
```

### F.2 Uncertainties

Oppure volendo si potrebbe usare questa comoda libreria:

```
1 from uncertainties import ufloat
2 import uncertainties.umath as um
3
4 #il primo argomento e' il valore centrale, il secondo l'errore
5 x = ufloat(7.1, 0.2)
6 y = ufloat(12.3, 0.7)
7
8
9 print(x)
10 print(2*x-y)
11 print(um.log(x**y))
12
13 [Output]
14 7.10+/-0.20
15 1.9+/-0.8
16 24.1+/-1.4
```

Vediamo ora un riassunto della propagazione degli errori:

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

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

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

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

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

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

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

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

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

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

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

## 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 certi 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:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def f(x, xx, yy):
5     """
6     restituisce l'interpolazione dei punti xx yy
7     x puo' essere un singolo valore in cui calcolare
8     la funzione interpolante o un intero array
9     """
10    #proviamo se x e' un array
11    try :
12        n = len(x)
13        x_in = np.min(xx) <= np.min(x) and np.max(xx) >= np.max(x)
14    except TypeError:
15        n = 1
16        x_in = np.min(xx) <= x <= np.max(xx)
17
18    #se il valore non e' nel range corretto e' impossibile fare il conto
19    if not x_in :
20        a = 'uno o diversi valori in cui calcolare la funzione'
21        b = ' interpolante sono fuori dal range di interpolazione'
22        errore = a+b
23        raise Exception(errore)
24
25    #array che conterra' l'interpolazione
26    F = np.zeros(n)
27
28    if n == 1 :
29        #controlla dove e' la x e trovo l'indice dell'array
30        #per sapere in che range bisogna interpolare
31        for j in range(len(xx)-1):
32            if xx[j] <= x <= xx[j+1]:
33                i = j
34
35        A = yy[i] * (xx[i+1] - x)/(xx[i+1] - xx[i])
36        B = yy[i+1] * (x - xx[i])/(xx[i+1] - xx[i])
37        F[0] = A + B
38
39    else:
40        #per ogni valore dell'array in cui voglio calcolare l'interpolazione
41        for k, x in enumerate(x):
42            #controlla dove e' la x e trovo l'indice dell'array
43            #per sapere in che range bisogna interpolare
44            for j in range(len(xx)-1):
45                if xx[j] <= x <= xx[j+1]:
46                    i = j
47
48            A = yy[i] * (xx[i+1] - x)/(xx[i+1] - xx[i])
49            B = yy[i+1] * (x - xx[i])/(xx[i+1] - xx[i])
50            F[k] = A + B
51
52    return F
53
54 if __name__ == '__main__':
55     x = np.linspace(0, 1, 10)
```

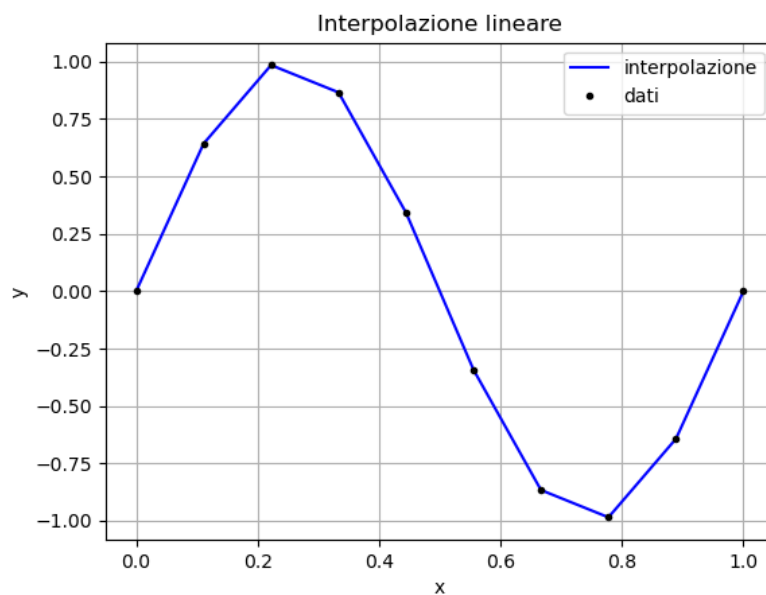


```

56 y = np.sin(2*np.pi*x)
57 z = np.linspace(0, 1, 100)
58
59 plt.figure(1)
60 plt.title('Interpolazione lineare')
61 plt.xlabel('x')
62 plt.ylabel('y')
63 plt.plot(z, f(z, x, y), 'b', label='interpolazione')
64 plt.plot(x, y, marker='.', linestyle='', c='k', label='dati')
65 plt.legend(loc='best')
66 plt.grid()
67 plt.show()

```

La funzione scritta prende due array "xx" e "yy" che sono i dati da interpolare, e una variabile "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 dà 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:

```

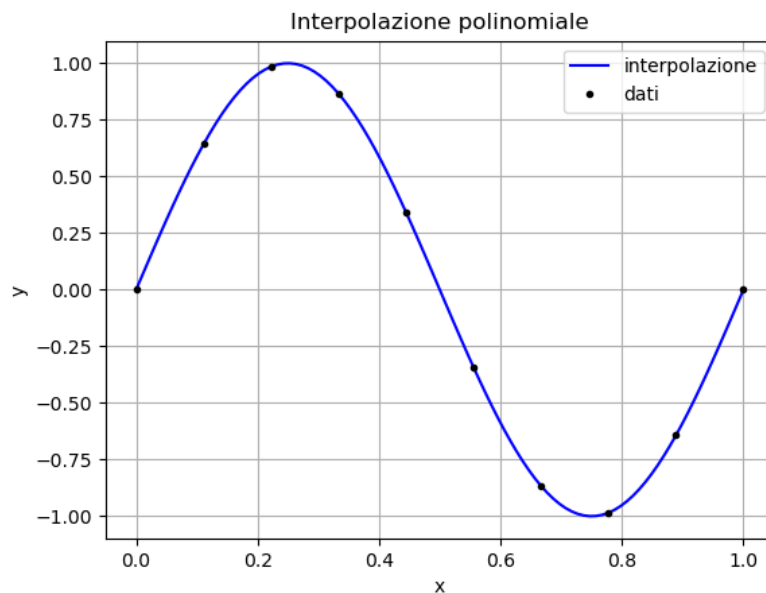
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 N = 10
5 x = np.linspace(0, 1, N)
6 y = np.sin(2*np.pi*x)
7
8 #Matrice di Vandermonde
9 A = np.zeros((N, N))
10 A[:,0] = 1
11 for i in range(1, N):
12     A[:,i] = x**i

```

```

13
14 #risolvo il sistema, la soluzione sono i coefficienti 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).

```

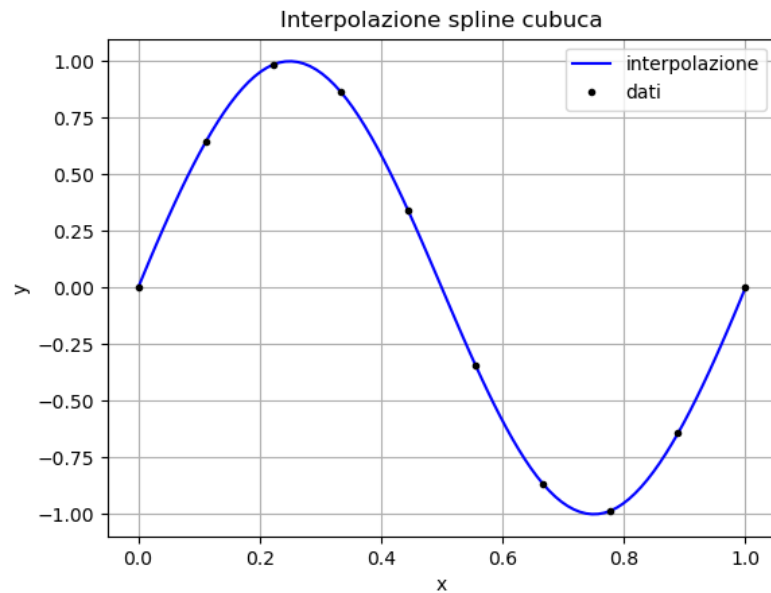
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.interpolate import InterpolatedUnivariateSpline
4
5 N = 10
6 x = np.linspace(0, 1, N)
7 y = np.sin(2*np.pi*x)
8
9 #interpolazione con una, spline cubica (k=3)
10 s3 = InterpolatedUnivariateSpline(x, y, k=3)
11
12 z = np.linspace(0, 1, 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 orientato 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:

```
1 import numpy as np
2
3 class Pallina:
4     """
5     Classe che rappresenta una pallina
6     intesa come oggetto puntiforme
7     """
8
9     def __init__(self, x, y, vx, vy, m):
10         """
11         costruttore della classe, verra' chiamato
12         quando creeremo l'istanza della classe
13         in input prende la posizione, la velocita' e massa
14         che sono le quantita' che identificano la pallina
15         che saranno gli attributi della classe;
16         il costruttore e' un particolare metodi della
17         classe per questo si utilizzano gli underscore.
18         il primo parametro che passiamo (self) rappresenta
19         l'istanza della classe (self e' un nome di default)
20         questo perche' la classe e' un modello generico che
21         deve valere per ogni 'pallina'
22         """
23         #posizione
24         self.x = x
25         self.y = y
26         #velocita'
27         self.vx = vx
28         self.vy = vy
29         self.massa = m
30
31     def energia_cinetica(self):
32         """
33         ad ogni metodo della classe viene passato
34         come primo argomento self, quindi l'istanza
35         calcoliamo l'evergia cinetica
36         """
37         #una volta creati gli attributi non e necessario passarli ai vari metodi
38         ene_k = 0.5*self.massa*(self.vx **2 + self.vy**2)
39         return ene_k
40
41     def energia_potenziale_gravitazionale(self, g):
42         """
43         calcoliamo l'energia potenziale; all'interno
44         di un campo gravitazionale di intensita' g
45         la nostra pallina ha energia per il semplice
46         fatto di essere in un qualche punto del campo
47         """
48
49         #supponiamo g sia diretta verso il basso
50         ene_u = self.massa*g*self.y
51         return ene_u
52
53 g = 9.81 #acc di gravita' che vorrei tanto mettere uguale a 1
54
55 #creo l'istanza della classe
56 p1 = Pallina(1, 1, 2, 2, 1)
57 #chiamo i metodi sull'istanza
58 ene_tot_p1 = p1.energia_cinetica() + p1.energia_potenziale_gravitazionale(g)
59 print('energia pallina 1:', ene_tot_p1)
60
61 #creo l'istanza della classe
62 p2 = Pallina(2, 2, 2, 2, 1)
63 #chiamo i metodi sull'istanza
64 ene_tot_p2 = p2.energia_cinetica() + p2.energia_potenziale_gravitazionale(g)
65 print('energia pallina 2:', ene_tot_p2)
66
67 [Output]
68 energia pallina 1: 13.81
```

69 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:

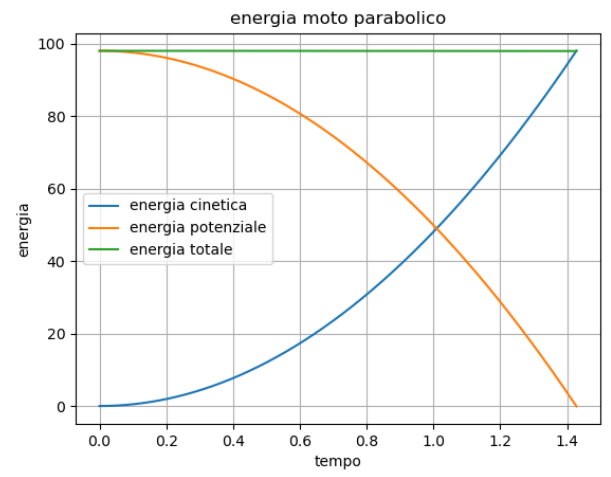
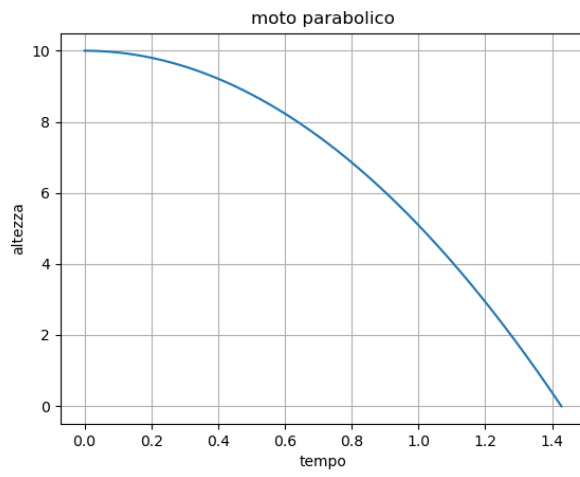
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 class Pallina:
5     """
6     Classe che rappresenta una pallina
7     intesa come oggetto puntiforme
8     """
9
10    def __init__(self, x, y, vx, vy, m):
11        """
12        costruttore della classe, verra' chiamato
13        quando creeremo l'istanza della classe
14        in input prende la posizione, la velocita' e massa
15        che sono le quantita' che identificano la pallina
16        che saranno gli attributi della classe;
17        il costruttore e' un particolare metodi della
18        classe per questo si utilizzano gli underscore.
19        il primo parametro che passiamo (self) rappresenta
20        l'istanza della classe (self e' un nome di default)
21        questo perche' la classe e' un modello generico che
22        deve valere per ogni 'pallina'
23        """
24        #posizione
25        self.x = x
26        self.y = y
27        #velocita'
28        self.vx = vx
29        self.vy = vy
30        self.massa = m
31
32    def energia_cinetica(self):
33        """
34        ad ogni metodo della classe viene passato
35        come primo argomento self, quindi l'istanza
36        calcoliamo l'energia cinetica
37        """
38        #una volta creati gli attributi non e necessario passarli ai vari metodi
39        ene_k = 0.5*self.massa*(self.vx **2 + self.vy**2)
40        return ene_k
41
42    def energia_potenziale_gravitazionale(self, g):
43        """
44        calcoliamo l'energia potenziale; all'interno
45        di un campo gravitazionale di intensita' g
46        la nostra pallina ha energia per il semplice
47        fatto di essere in un qualche punto del campo
48        """
49        #supponiamo g sia diretta verso il basso
50        ene_u = self.massa*g*self.y
51        return ene_u
52
53    #aggiornamento posizione e velocita' con eulero
54    def n_vel(self, fx, fy, dt):
55        """
56        date le componenti della forza e il passo temporale
57        aggiorni le componenti della velocita'
58        """
59        self.vx += fx*dt
60        self.vy += fy*dt
61
62    def n_pos(self, dt):
63        """
64        dato il passo temporale aggiorni le posizioni
65        """
66        self.x += self.vx*dt
67        self.y += self.vy*dt
68
69
70
71 if __name__ == "__main__":
72
```

```

73 g = 9.81 #acc di gravita' che vorrei tanto mettere uguale a 1
74 p = Pallina(0, 10, 0, 0, 1)
75
76 # simulazione moto
77 N = 1500
78 dt = 0.001
79
80 #salvo le posizioni iniziali
81 x = np.array([])
82 y = np.array([])
83 x = np.insert(x, len(x), p.x)
84 y = np.insert(y, len(y), p.y)
85 #array del tempo
86 t = np.array([])
87 t = np.insert(t, len(t), 0.0)
88
89 #energia iniziale
90 ene_cin = np.array([])
91 ene_pot = np.array([])
92 ene_cin = np.insert(ene_cin, len(ene_cin), p.energia_cinetica())
93 ene_pot = np.insert(ene_pot, len(ene_pot), p.energia_potenziale_gravitazionale(g))
94
95
96 for i in range(1, N):
97
98     #se arrivi a terra fermati
99     if y[-1]<0: break
100
101     #moto di caduta libera
102     fx = 0
103     fy = -g
104
105     #aggiorno le posizioni
106     p.n_vel(fx, fy, dt)
107     p.n_pos(dt)
108
109     #salvo le posizioni
110     x = np.insert(x, len(x), p.x)
111     y = np.insert(y, len(y), p.y)
112
113     #calcolo energia
114     ene_cin = np.insert(ene_cin, len(ene_cin), p.energia_cinetica())
115     ene_pot = np.insert(ene_pot, len(ene_pot), p.energia_potenziale_gravitazionale(g))
116
117     t = np.insert(t, len(t), i*dt)
118
119
120 #plot
121 plt.figure(1)
122 plt.title('moto parabolico')
123 plt.xlabel('tempo')
124 plt.ylabel('altezza')
125 plt.plot(t, y)
126 plt.grid()
127
128 plt.figure(2)
129 plt.title('energia moto parabolico')
130 plt.xlabel('tempo')
131 plt.ylabel('energia')
132 plt.plot(t, ene_cin, label='energia cinetica')
133 plt.plot(t, ene_pot, label='energia potenziale')
134 plt.plot(t, ene_cin+ene_pot, label='energia totale')
135 plt.legend(loc='best')
136 plt.grid()
137
138 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.



# 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 piccolo 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  $\theta, \sigma$  costanti positive e  $\mu$  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:

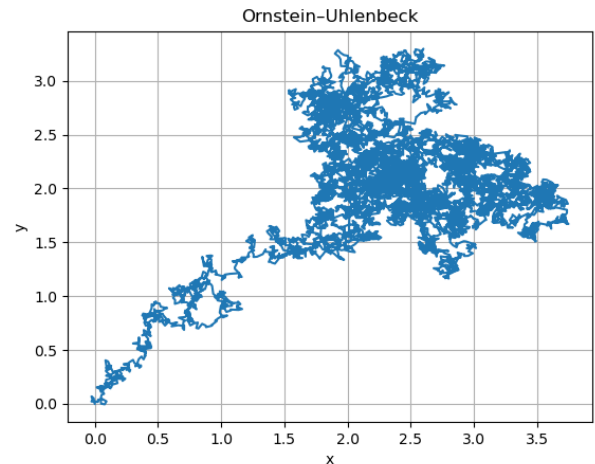
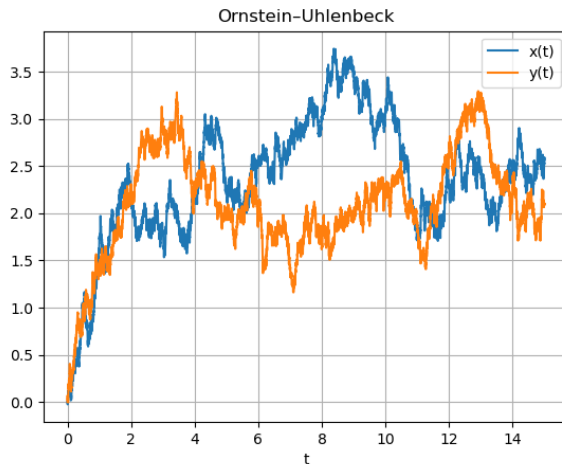
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def f(z):
5     """
6     funzione che moltiplica il dt
7     """
8     theta = 0.7
9     mu = 2.5
10    return theta * (mu - z)
11
12 def g():
13     """
14     funzione che moltiplica il processo di wiener
15     """
16     sigma = 0.6
17     return sigma
18
19 def dW(delta_t):
20     """
21     processo di wiener trattato come variabile gaussiana
22     """
23     return np.random.normal(loc=0.0, scale=np.sqrt(delta_t))
24
25 #parametri simulazione
26 N = 10000
27 tf = 15
28 dt = tf/N
29
30 ts = np.zeros(N + 1)
31 ys = np.zeros(N + 1)
32 xs = np.zeros(N + 1)
33
34 ys[0], xs[0] = 0, 0 #condizioni iniziali
35
36 for i in range(N):
37     ys[i+1] = ys[i] + f(ys[i]) * dt + g() * dW(dt)
38     xs[i+1] = xs[i] + f(xs[i]) * dt + g() * dW(dt)
39     ts[i+1] = ts[i] + dt
40
41 plt.figure(1)
42 plt.plot(xs, ys)
43 plt.title('Ornstein Uhlenbeck')
44 plt.xlabel("x")
45 plt.ylabel("y")
46 plt.grid()
47
48 plt.figure(2)
```



```

49 plt.plot(ts, xs, label='x(t)')
50 plt.plot(ts, ys, label='y(t)')
51 plt.title('Ornstein Uhlenbeck')
52 plt.xlabel("t")
53 plt.legend()
54 plt.grid()
55
56 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 ( $dx = f(x)dt + g(x)dW$ ):

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

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

```

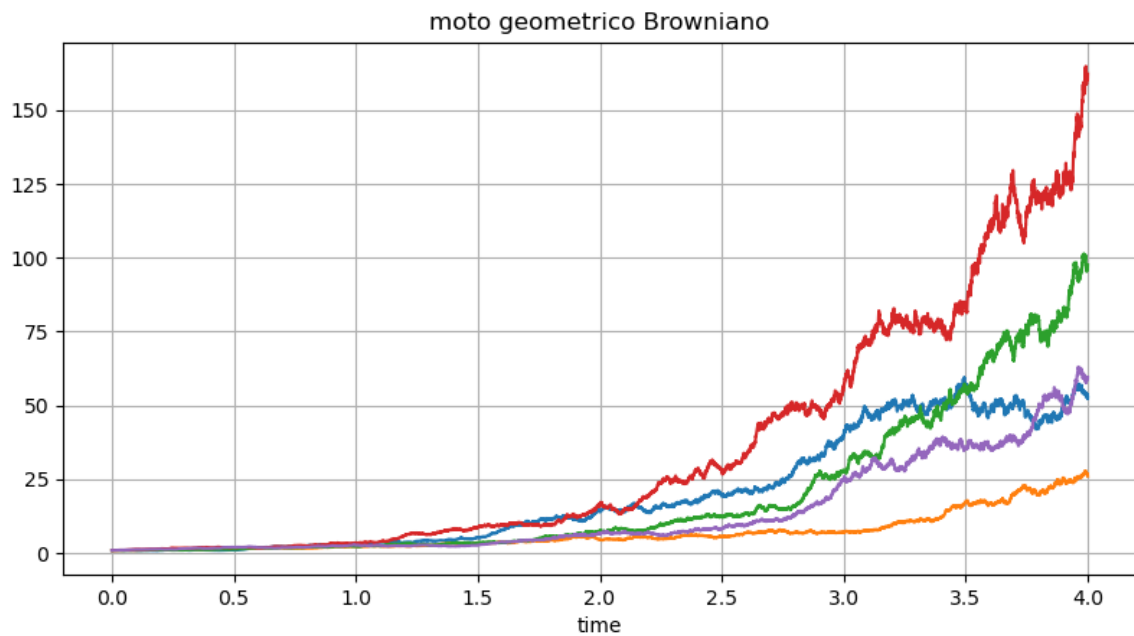
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # geometric Brownian motion
5 # Heun method
6
7 def f(z):
8     """
9     funzione che moltiplica il dt
10    """
11    mu = 1
12    return mu*z
13
14 def g(z):
15     """
16     funzione che moltiplica il processo di wiener
17    """
18    sigma = 0.5
19    return sigma*z
20
21 def dg():
22     """
23     derivata di g
24    """
25    sigma = 0.5

```

```

26     return sigma
27
28 def dW(delta_t):
29     """
30     processo di wiener trattato come variabile gaussiana
31     """
32     return np.random.normal(loc=0.0, scale=np.sqrt(delta_t))
33
34
35 #parametri simulazioni
36 N = 10000
37 tf = 4
38 dt = tf/N
39 #faccio 5 simulazioni diverse
40 for _ in range(5):
41     #array dove conservare la soluzione, ogni volta inizializzati
42     ts = np.zeros(N + 1)
43     ys = np.zeros(N + 1)
44
45     ys[0] = 1#condizioni iniziali
46
47     for i in range(N):
48         ts[i+1] = ts[i] + dt
49         y0 = ys[i] + f(ys[i])*dt + g(ys[i])*dW(dt) + 0.5*g(ys[i])*dg()*(dW(dt)**2)
50         y1 = ys[i] + f(y0)*dt + g(y0)*dW(dt) + 0.5*g(y0)*dg()*(dW(dt)**2)
51         ys[i+1] = 0.5*(y0 + y1)
52
53     plt.plot(ts, ys)
54
55 plt.figure(1)
56 plt.title('moto geometrico Browniano')
57 plt.xlabel("time")
58 plt.grid()
59
60 plt.show()

```



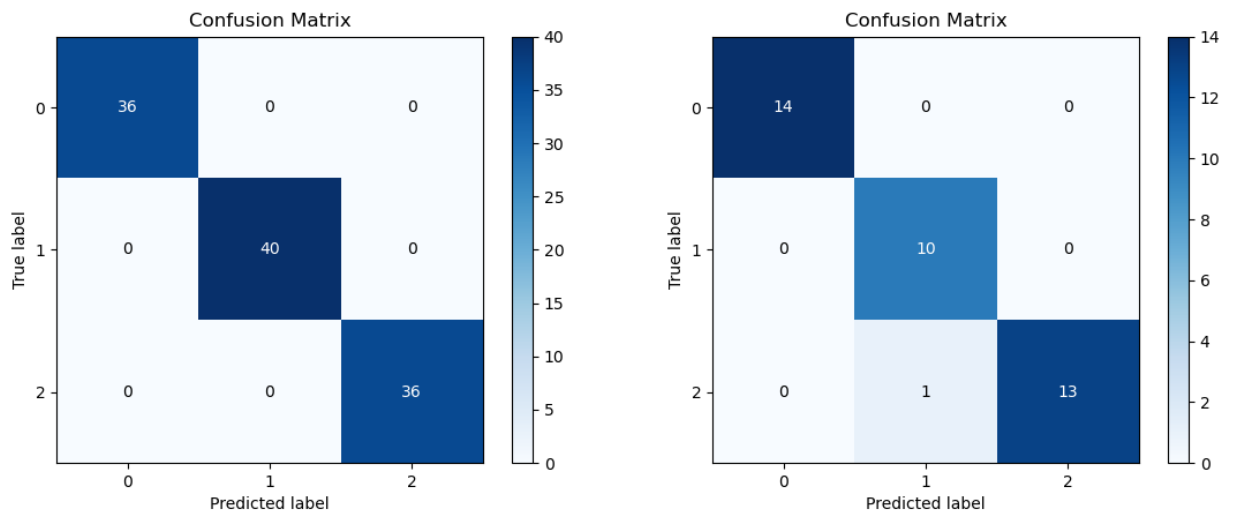
## J Machine Learning

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:

```
1 import scikitplot as skplt
2 import matplotlib.pyplot as plt
3 from sklearn import datasets
4 from sklearn.model_selection import train_test_split
5 from sklearn.tree import DecisionTreeClassifier
6 from sklearn.metrics import accuracy_score
7
8 #dati che verranno utilizzati
9 iris_dataset = datasets.load_iris()
10 #print(iris_dataset["DESCR"])
11
12 #caratteristiche, dati in input
13 x = iris_dataset.data
14
15 #output, cioe' quello che il modello dovrebbe predire
16 y = iris_dataset.target
17
18 #divido i dati, un parte li uso per addestrare, l'altra per
19 #testare se il modello ha imparato bene
20 x_train, x_test, y_train, y_test = train_test_split(x, y)
21
22 #modello non addestrato, classificatore
23 #un classificatore prende i dati e restituisce un categoria
24 modello = DecisionTreeClassifier()
25
26 #addestro il modello
27 modello.fit(x_train, y_train)
28
29 #predizioni sui dati su cui ha imparato
30 predizione_train = modello.predict(x_train)
31
32 #predizioni su nuovi dati
33 predizione_test = modello.predict(x_test)
34
35 #misuro l'accuratezza sia dell'addestramento che del test
36 #questo puo' dare informazioni su over fitting o meno
37 #sinceramente non so come
38 print("accuratezza train")
39 print(accuracy_score(y_train, predizione_train))
40
41 print("accuratezza test")
42 print(accuracy_score(y_test, predizione_test))
43
44 #Rappresentazione grafica dei quanto e' stato bravo il modello
45 #sulle y c'e' la risposta che il modello doveva dare e
46 #sulle x ci sta la predizione che il modello ha dato, quindi gli
47 #elementi fuori diagonali sono le risposte sbagliate
48 skplt.metrics.plot_confusion_matrix(y_train, predizione_train)
49 skplt.metrics.plot_confusion_matrix(y_test, predizione_test)
50
51 plt.show()
52
53 [Output]
54 accuratezza train
55 1.0
56 accuratezza test
57 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...

```

1 import numpy as np
2 from sklearn.datasets import load_boston
3 from sklearn.linear_model import LinearRegression
4 from sklearn.metrics import mean_absolute_error
5 from sklearn.model_selection import train_test_split
6
7 """
8 utilizzo di un regressore lineare per predire
9 Il prezzo di una casa dati certe informazioni
10 """
11 dataset = load_boston()
12
13 #print(dataset["DESCR"])
14 #primi 13 elementi della prima riga
15 #print(dataset["data"][0])
16 #ultimo elemento della prima riga
17 #print(dataset["target"][0])
18
19 #caratteristiche
20 X = dataset["data"]
21
22 #output, cioè quello che il modello dovrebbe predire
23 y = dataset["target"]
24
25 #divido i dati, una parte li uso per addestrare, l'altra per
26 #testare se il modello ha imparato bene
27 X_train, X_test, y_train, y_test = train_test_split(X, y)
28
29 #modello non addestrato è un regressore
30 #un regressore prende i dati e restituisce un numero, una stima di qualcosa
31 modello = LinearRegression()
32
33 #addestrare il modello
34 modello.fit(X_train, y_train)
35
36 #predizioni
37 p_train = modello.predict(X_train)
38 p_test = modello.predict(X_test)
39
40 #errori sulla predizione
41 dp_train = mean_absolute_error(y_train, p_train)
42 dp_test = mean_absolute_error(y_test, p_test)
43
44 print("train", np.mean(y_train), "+-", dp_train)

```

```
45 print("test ", np.mean(y_test), "+-", dp_test)
46
47 [Output]
48 train 22.59815303430079 +- 3.1207001914064647
49 test 22.337795275590548 +- 3.806645940024761
```

See you Space Cowboy ...