

4 Brevi lezioni di python

Francesco Zeno Costanzo *

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

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

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

Indice

1	Scopo delle note e Caveat	4
2	Introduzione	5
2.1	Notazioni	5
2.2	Primo comandamento dell'informatica	5
2.3	Secondo comandamento dell'informatica	5
2.4	Terzo comandamento dell'informatica	5
2.5	Quarto comandamento dell'informatica	5
3	Lezione Zero: Installazione	6
3.1	Installazione dell'ambiente: Pyzo	6
3.2	Installazione dell'interprete: Anaconda	6
3.3	Installazione dei pacchetti	6
4	Prima lezione	7
4.1	Funzione print	7
4.2	Commenti	7
4.3	Variabili	7
4.4	Librerie	10
4.5	Gestione errori	10
4.5.1	Try e except	11
4.5.2	Raise Exception	11
4.6	Come leggere il Traceback	11
5	Seconda lezione	13
5.1	Gli array	13
5.2	Tipi di array	14
5.3	Array predefiniti	14
5.4	Operazioni con gli array	15
5.5	Matrici	16
6	Terza lezione	18
6.1	Le funzioni	18
6.2	Istruzioni di controllo	18
6.2.1	Espressioni condizionali: if, else, elif	18
6.2.2	Cicli: while, for	19
6.3	Ancora funzioni	21
6.4	Grafici	21
6.5	Esercizio riassuntivo	24
6.6	Prestazioni	25
7	Quarta lezione	26
7.1	Importare file Python	26
7.2	Fit	26
7.3	Dietro curve fit: Levenberg-Marquardt	30
A	Calcolo degli integrali	37
B	Risolvere numericamente le ODE	40
B.1	Esponenziale	40
B.2	Pendolo	42
B.3	Animazione	46
C	Sistemi lineari	48
C.1	Metodo Gauss-Seidel	48
C.2	Successive over-relaxation	48
C.3	Metodo del gradiente coniugato	50

D Zeri di una funzione	53
D.1 Bisezione	53
D.2 Metodo di Newton	54
D.3 Zeri in più dimensioni	56
E Risolvere numericamente le PDE	60
E.1 Equazione del trasporto	60
E.2 Equazione del calore	61
F Presa dati da foto	63
G Trasformate di Fourier	64
G.1 DFT	64
G.2 FFT	65
G.3 RFFT	66
H Fit	69
H.1 Fit con scipy	69
H.2 Fit circolare, metodo di Coope	71
H.3 Fit di un'elissi, metodo di Halir e Flusser	74
I Autovalori e autovettori	77
J Metodi Montecarlo	81
J.1 Generatori numeri pseudo-casuali	81
J.2 Calcolo di Pi greco	82
K Propagazione errori	84
K.1 Propagarli a mano	84
K.2 Uncertainties	84
L Interpolazione	86
L.1 Interpolazione lineare	86
L.2 Interpolazione Polinomiale	87
L.3 Scipy.interpolate	88
M Programmazione a oggetti	90
M.1 Più che una funzione	93
N Risolve numericamente le SDE	96
N.1 Processo di Ornstein–Uhlenbeck	96
N.2 Moto geometrico Browniano	97
O Ottimizzazione	99
O.1 Discesa del gradiente	99
O.2 Principio di inerzia	100
P Machine Learning	102
P.1 Classificatore	102
P.2 Regressori	103
P.3 Salvare il modello	104
Q Creare un eseguibile	107
Q.1 Eseguitibile windows	107
Q.2 Eseguitibile linux	108
R Bibliografia e Conclusioni	109

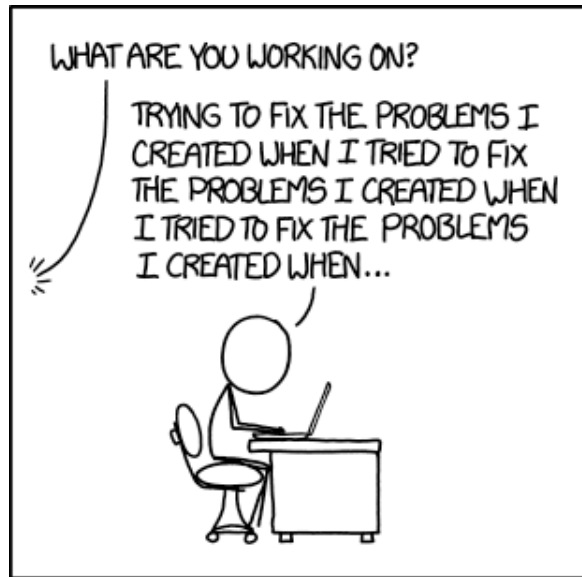
1 Scopo delle note e Caveat

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. Sono in realtà presenti più elementi del necessario nella volontà o di approfondire alcuni aspetti o di dare un assaggio di quello che nella vita di un fisico capita spesso di utilizzare (gli argomenti introdotti sono molti e una trattazione esaustiva richiederebbe troppo lavoro).

Nelle note sono presenti i codici in modo che possano essere consistenti da sole. Tuttavia copiare e incollare i codici potrebbe creare fastidio quando saranno eseguiti quindi conviene piuttosto copiarli a mano o prenderli dalla cartella dove sono presenti anche queste note: <https://github.com/Francesco-Zeno-Costanzo/4BLP>.

2 Introduzione

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.



2.1 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).

2.2 Primo comandamento dell'informatica

Se funziona quanto basta
non toccare che si guasta.

2.3 Secondo comandamento dell'informatica

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

2.4 Terzo comandamento dell'informatica

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

2.5 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.

3 Lezione Zero: Installazione

3.1 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 bit¹. 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.

3.2 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
```

3.3 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

¹Al 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>

4 Prima lezione

4.1 Funzione print

Se un macchina fosse senziente e gentile, quindi non un'intelligenza artificiale cresciuta su twitter, forse come prima cosa saluterebbe tutti e il modo per comunicare è la funzione print, che ci permette di stampare a schermo (sulla shell) delle informazioni contenute nel codice. 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
```

4.2 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!
```

4.3 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 variabili 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
```

Nel codice di sopra il nome delle variabili è alquanto esplicativo del loro significato, ed è in genere buona norma dare nomi che siano intuitivi. Ora non vi dico che dovete chiamare una variabile: "momentoagolaresullasseZ", che ci vogliono tre anni solo a scriverla ma di certo chiamarla "Lz" piuttosto che "a" o "pippo" è una strada che andrebbe perseguita. 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'>

```

Dizionari, liste, tuple, possono essere elementi molto utili. Tutti essi sono indicizzati, i primi due sono modificabili le tuple invece sono immutabili, come abbiamo visto non ci sono particolari problemi di casting, essi possono contenere elementi di vario genere. I dizionari inoltre, utilizzando un sistema chiave valore possono essere utili per gestire alcuni output di codici lunghi o complessi; per esempio sono molto utili nella programmazione in parallelo, dove l'ordine si perde quindi un dizionario è un ottimo modo per tenere traccia di tutta l'esecuzione. Più avanti parleremo un po' di di questo altro tipo di variabili. Vediamo intanto come fare le classiche operazioni matematiche tra variabili:

```

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. A seconda del tipo della variabile ci si ferma ad tot di cifre decimali, 8 per i float32, 16 per i float64; dove il numero che segue la parola float indica il numero di bit che il computer usa per scrivere il numero. Vediamo un classico esempio:

```

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)

```

Vediamo quindi che un numero reale è scritto in realtà, e altrimenti non si potrebbe fare, come numero razionale. 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 (attenzione mi raccomando che ci vuole poco che una cosa del genere fornisca errori):

```

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 $x \text{ operazione} = \text{numero}$.

4.4 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 (ogni tanto però è interessante andare a vedere cosa ci sta dietro, ma ne parleremo, non molto, più avanti). 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 eulero = np.e
12 print(eulero)
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). Altra cosa interessante è che essendo la documentazione scritta sul codice, esattamente come qui noi scriviamo i commenti, potete consultarla anche da shell. Su Pyzo vi basta scrivere sulla shell: `nomefunzione?`, mentre se usate una shell normale, stile quella di ubuntu dovete prima digitare `python` oppure `python3` sulla shell e vi si aprirà l'ambiente python dove potete importare i pacchetti come se scriveste codice e per vedere la documentazione vi basta fare `help(nomefunzione)`.

4.5 Gestione errori

È molto facile scrivere codice che produca errore, magari perché 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 (al più possiamo appellarci alla censura cosmica e mettere un'orizzonte a vestire la divisione per zero) e quindi il computer ci da errore. Possiamo aggirare il problema, evitando così il second impact, in due modi diciamo:

4.5.1 Try e except

Possiamo utilizzare il costrutto try except dicendo al computer: prova a fare la divisione e, sia mai funziona, se questa però 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
```

Il fatto che alle riga 3 e 5 abbiamo lasciato uno spazio è di fondamentale importanza in quanto Python è case sensitive e richiede che per certi comandi quello spazio ci sia in modo da avere chiaro il comportamento del codice. Lo vedremo meglio quando faremo le funzioni.

4.5.2 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
```

Non abbiamo ancora spiegato cosa è un if ma dal codice direi che non risulta di difficile intuizione.

4.6 Come leggere il Traceback

Quelli che abbiamo sopra sono esempi di errori che danno come risultato in genere l'interruzione del codice. Quindi quando la shell di Pyzo si tinge di rosso cremisi che nemmeno fosse appena finita una battaglia campale di Vlad figlio del drago voivoda di Valacchia è buona norma leggere attentamente il messaggio di errore, il traceback (che come suggerisce il nome va letto al contrario; da sotto verso sopra), e capire cosa si è sbagliato. Il traceback infatti è la catena di eventi che hanno portato all'errore. Analizziamo il caso di sopra.

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

La prima riga ci fa capire che c'è stato un errore.

La seconda ci dice che nel file chiamato " < tmp1 > " (perché mi ero dimenticato di salvarlo ancora, sennò ci sarebbe il path del file) alla linea 5, nel codice eseguito(se fosse dentro una funzione ci sarebbe, oltre a questa, un' altra riga uguale con il nome della funzione al posto di < module >) è successo qualcosa.

La terza linea è la linea di codice a cui è avvenuto il misfatto.

La quarta ci da due informazioni, separate dai due punti: il tipo di errore che è avvenuto e le informazioni riguardo ad esso.

Ora capisco che prima vi dico di leggerlo al contrario e poi qui ve lo descrivo nel normale ordine di lettura, ma è solo per farvi capire il significato delle varie linee di errore. Facciamo adesso un esempio un po' più complicato, leggendolo correttamente, in cui useremo funzioni quindi magari potete tornare a rileggerlo dopo se volete.

```
1 def err(c):
2     b = 1/c
3     return b
4
5 def run(c):
6     print(err(c))
7
8 if __name__ == "__main__":
9     run(0)
10
11 [Output]
12 Traceback (most recent call last):
13   File "C:\Users\franc\Documents\GitHub\4BLP\Prima Lezione\codiciL1\err_trace.py", line 9, in
14     <module>
15     run(0)
16   File "C:\Users\franc\Documents\GitHub\4BLP\Prima Lezione\codiciL1\err_trace.py", line 6, in
17     run
18     print(err(c))
19   File "C:\Users\franc\Documents\GitHub\4BLP\Prima Lezione\codiciL1\err_trace.py", line 2, in
20     err
21     b = 1/c
22 ZeroDivisionError: division by zero
```

Leggiamolo insieme, partendo dal basso abbiamo:

C'è stata una divisione per zero che ha causato l'errore di tipo ZeroDivision error alla linea 2 nella funzione "err" contenuta nel codice avente quel path (sta volta il codice era salvato).

L'errore è stato causato dalla chiamata della funzione "err" da parte della funzione "run" a linea 6, contenuta nello stesso file, hanno infatti stesso path.

Ma ancora prima l'errore è causato dalla chiamata della funzione "run" (che ha chiamato err, che ha prodotto l'errore) all'interno dello stesso codice, alla linea 9. Insomma fiera dell'est di Branduardi spostati proprio. Se proprio non capite che vi sta dicendo, buona norma è copiare la riga più bassa del traceback e incollarla su Google. Qualcuno avrà già avuto il vostro problema.

5 Seconda lezione

Ripetiamo tutti insieme: Python conta da zero.

5.1 Gli array

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

```
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.]
```

Analogamente ciò può essere fatto per le liste con le funzioni native, quindi non di numpy, append e pop. Più corretto sarebbe dire che esse sono dei metodi della classe che gestisce le liste, infatti come vediamo nel prossimo esempio la sintassi è leggermente diversa, ma non vale la pena complicarci troppo la vita.

```
1 # lista iniziale
2 lista = [1, 2, 3, 4]
3 print(lista)
4
5 #aggiungo un elemento in coda, quindi alla fine della lista
6 lista.append(42)
7 print(lista)
8
9 #rimuovo l'ultimo elemento
10 lista.pop()
11 print(lista)
12
13 [Output]
14 [1, 2, 3, 4]
15 [1, 2, 3, 4, 42]
16 [1, 2, 3, 4]
```

5.2 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
```

5.3 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.          ]

```

5.4 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
33 [16 25 36]
34 [0.07918125 0.53147892 0.76342799]
35 Traceback (most recent call last):
36   File "<tmp 1>", line 26, in <module>
37     print(math.log10(w))
38 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. Ai fisici piace dire che un vettore è un qualcosa che trasforma come un vettore, e con questi esempi potrete capire che una lista non "trasforma" come un vettore mentre un array di numpy sì. Per questo nella programmazione scientifica se ne fa largo uso.

5.5 Matrici

Se un array unidimensionale lungo n è 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]]
```

Ci siamo fermati alle matrici, oggetti a due indici, ma volendo avremmo potuto creare oggetti a più indici (i famosi tensori, tanto sempre numeri sono) ad esempio "np.ones((3,3,3))" creerebbe un oggetto a tre indici di uni, che possiamo vedere ad esempio come un vettore a tre componenti, ciascuna delle quali è una matrice 3×3 . Se questo vi sembra strano aspettate di fare meccanica quantistica e ne riparlamo.

Ora è importante far notare una cosa: l'operazione di assegnazione con gli array è delicata, anche con le liste:

```
1 import numpy as np
2
3 a = np.array([1, 2, 3, 4])
4 print(f"array iniziale: {a}, id: {id(a)}")
5
6 b = a
7 b[0] = 7
8
9 print(f"array iniziale: {a}, id: {id(a)}")
10 print(f"array finale : {b}, id: {id(b)}")
11
12 #usiamo ora copy invece che l'assegnazione
13
14 a = np.array([1, 2, 3, 4])
15 print(f"array iniziale: {a}, id: {id(a)}")
16
17 b = np.copy(a)
18 b[0] = 7
19
20 print(f"array iniziale: {a}, id: {id(a)}")
21 print(f"array finale : {b}, id: {id(b)}")
22
23 [Output]
24 array iniziale: [1 2 3 4], id: 2226551695088
25 array iniziale: [7 2 3 4], id: 2226551695088
26 array finale : [7 2 3 4], id: 2226551695088
27 array iniziale: [1 2 3 4], id: 2226573280912
28 array iniziale: [1 2 3 4], id: 2226573280912
29 array finale : [7 2 3 4], id: 2226578310224
```

Come vedete se usiamo l'operato di assegnazione anche l'array iniziale cambia poiché sia a che b sono riferiti allo stesso indirizzo di memoria, mentre usando la funzione "copy" ora il secondo array ha un diverso indirizzo e quindi il problema non si pone più.

6 Terza lezione

6.1 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 importati:

- È fondamentale in Python che il corpo della funzione sia indentato, per seguire un raggruppamento logico del codice. esattamente come avevamo visto per il comando "try" e come vedremo per i cicli. Insomma le parti indentate del codice devono essere logicamente connesse.
- 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 "A", ma posso tranquillamente definire una nuova variabile "A" al di fuori della funzione e non avrei problemi di sovrascrittura.

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.

6.2 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.

6.2.1 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
```

6.2.2 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, della serie che fa prima a decadere il protone.

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.

6.3 Ancora funzioni

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

```

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.

6.4 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 """

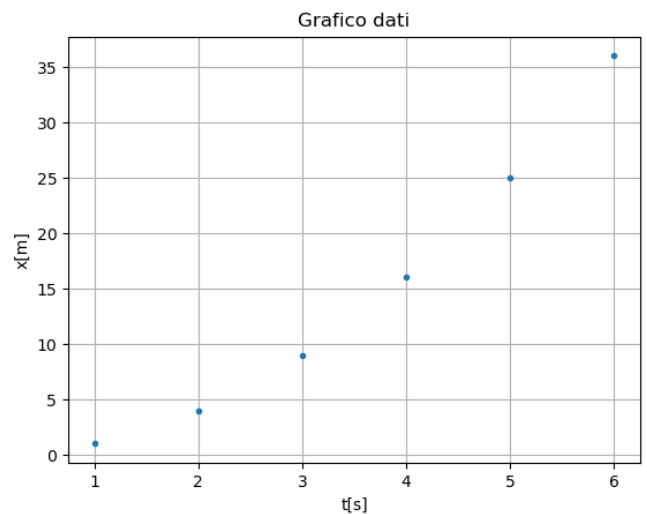
```

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')

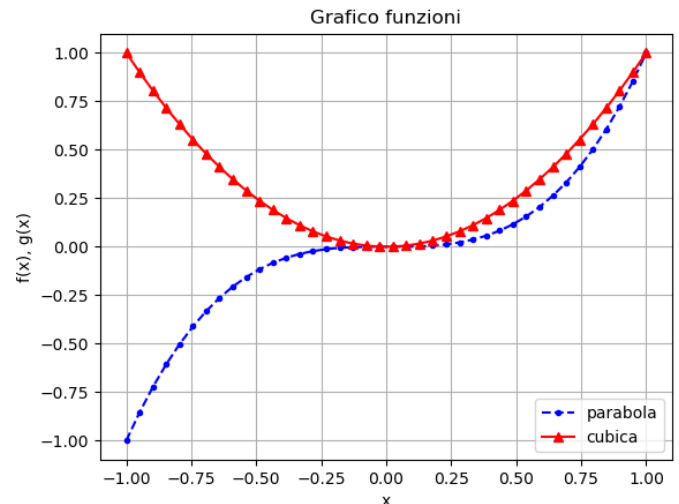
```

```

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 consona, 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).



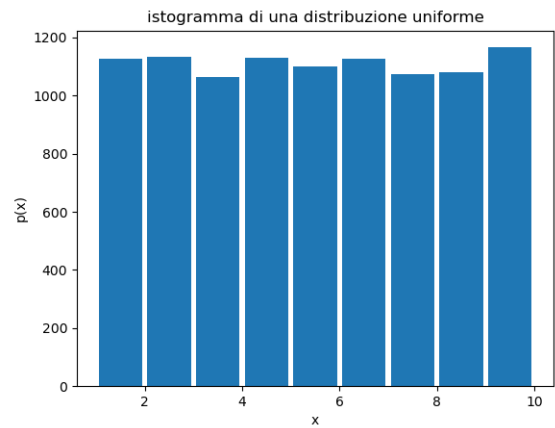
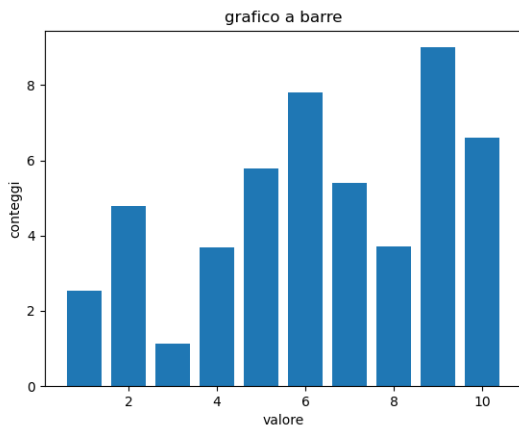
Ultima menzione da fare sono gli istogrammi (il fetish dei pallettari):

```

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).



6.5 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.

6.6 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
```

Abbiamo usato la libreria time per misurare il tempo, "time.time()" restituisce i numeri di secondi trascorsi dall'inizio dell'epoca unix (1 gennaio 1970). Vediamo che quindi usando le funzioni di numpy, (np.arange) e le funzioni della libreria standard di Python (sum), è possibile fare lo steso conto in un tempo molto minore che tramite un ciclo for. Questo perché le librerie non sono totalmente in Python ma in molta parte in C e/o fortran.

7 Quarta lezione

7.1 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.

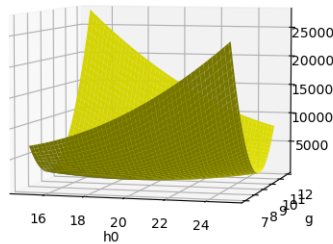
7.2 Fit

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:

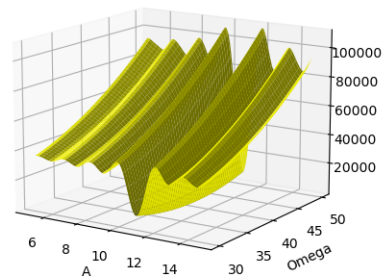
$$S^2(\{\theta_i\}) = \sum_i \frac{(y_i - f(x_i; \{\theta_j\}))^2}{\sigma_{y_i}^2} \quad (1)$$

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 θ_i non realistici; mentre per una regressione lineare il minimo è solo uno e assoluto. Prima di vedere il codice vediamo brevemente due grafici della quantità S^2 , che con un po' di abuso di notazione chiamiamo chiquadro, nel caso di regressione lineare e non:

Chiquadro regressione lineare



Chiquadro regressione non-lineare



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

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

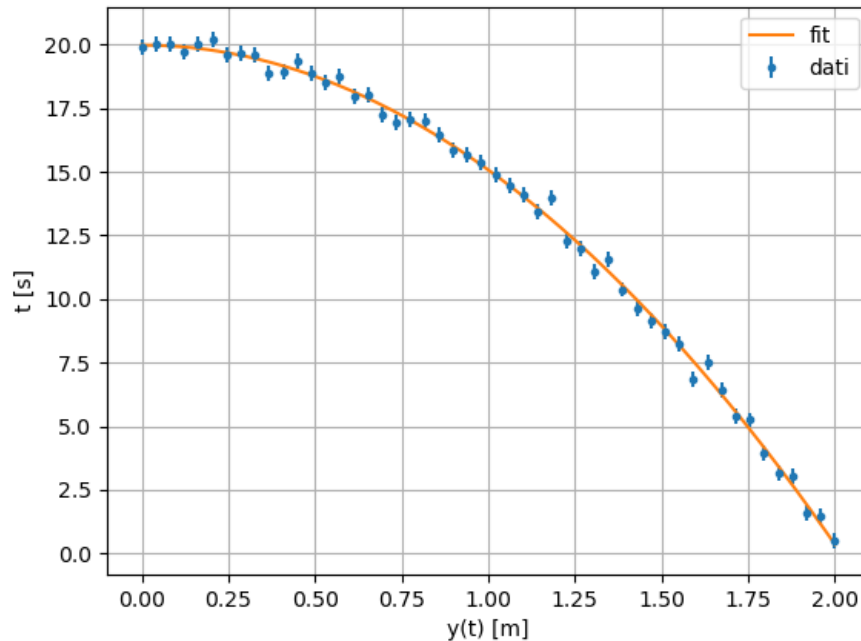
Vediamo come effettivamente siano presenti nel caso non lineare una serie di minimi locali che sarebbe meglio evitare (i codici per la realizzazione di grafici sono riportati a fine sezione). Vediamo ora un semplice esempio di codice:

```
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 : fisicamente i tempi a cui osservo
15            l'oscillazione del corpo non
16            affetti da errore
17    ydata : fisicamente la posizione del corpo
18            misurata a dati tempi xdata affetta
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 affetti 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()

```

7.3 Dietro curve fit: Levenberg-Marquardt

Vogliamo ora provare ad andare dietro la libreria e vedere cosa fa effettivamente curve fit. Chiaramente i metodi di fit implementati sono molti e diversi, a seconda delle esigenze; per semplicità perciò andiamo a vedere quello che viene usato di default: Levenberg-Marquardt. Questo è un metodo iterativo, il che spiega la sensibilità ai valori iniziali, caratteristica di ogni metodo iterativo. Consideriamo la nostra funzione di fit f la quale dipende da una variabile indipendente e da un insieme di parametri θ , il quale fondamentalmente è un vettore di \mathbb{R}^m . Possiamo espandere f in serie di Taylor intorno ad un valore dei nostri parametri:

$$f(x_i, \theta_j + \delta_j) \simeq f(x_i, \theta_j) + J_{ij}\delta_j \quad (2)$$

dove δ_j è lo spostamento che viene fatto ad ogni passo dell'iterazione e J_{ij} è il gradiente di f , o jacobiano se volete:

$$J_{ij} = \frac{\partial f(x_i, \theta_j)}{\partial \theta_j} = \begin{bmatrix} \frac{\partial f(x_1, \theta_1)}{\partial \theta_1} & \dots & \frac{\partial f(x_1, \theta_m)}{\partial \theta_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(x_n, \theta_1)}{\partial \theta_1} & \dots & \frac{\partial f(x_n, \theta_m)}{\partial \theta_m} \end{bmatrix} \quad (3)$$

Che è una matrice $m \times n$ con $m < n$ altrimenti il metodo non funziona e dobbiamo adottare altre strategie. Per trovare il valore di δ espandiamo la (1):

$$\begin{aligned} S^2(\theta + \delta) &\simeq \sum_{i=1}^n \frac{(y_i - f(x_i, \theta) - J_{ij}\delta_j)^2}{\sigma_{y_i}^2} \\ &= (y - f(x, \theta) - J\delta)^T W (y - f(x, \theta) - J\delta) \\ &= (y - f(x, \theta))^T W (y - f(x, \theta)) - (y - f(x, \theta))^T W J\delta - (J\delta)^T W (y - f(x, \theta)) + (J\delta)^T W (J\delta) \\ &= (y - f(x, \theta))^T W (y - f(x, \theta)) - 2(y - f(x, \theta))^T W J\delta + \delta^T J^T W (J\delta) \end{aligned} \quad (4)$$

Dove W è tale che $W_{ii} = 1/\sigma_{y_i}^2$ e derivando rispetto a δ otteniamo il metodo di Gauss-Newton:

$$\frac{\partial S^2(\theta + \delta)}{\partial \delta} = -2(y - f(x, \theta))^T W J + 2\delta^T J^T W J = 0 \quad (5)$$

per cui facendo il trasposto a tutto otteniamo:

$$(J^T W J)\delta = J^T W (y - f(x, \theta)) \quad (6)$$

La quale si risolve per δ . Per migliorare la convergenza del metodo si introduce un parametro di damping λ e l'equazione diventa:

$$(J^T W J - \lambda \text{diag}(J^T W J))\delta = J^T W (y - f(x, \theta)) \quad (7)$$

Il valore di λ viene cambiato a seconda se ci avviciniamo o meno alla soluzione giusta. Se ci stiamo avvicinando ne riduciamo il valore, andando verso il metodo di Gauss-Newton; mentre se ci allontaniamo ne aumentiamo il valore in modo che l'algoritmo si comporti più come un gradiente discendente (di cui in appendice ci sarà un esempio). La domanda è: come capiamo se ci stiamo avvicinando alla soluzione? Calcoliamo:

$$\begin{aligned} \rho(\delta) &= \frac{S^2(x, \theta) - S^2(x, \theta + \delta)}{|(y - f(x, \theta) - J\delta)^T W (y - f(x, \theta) - J\delta)|} \\ &= \frac{S^2(x, \theta) - S^2(x, \theta + \delta)}{|\delta^T (\lambda \text{diag}(J^T W J)\delta + J^T W (y - f(x, \theta)))|} \end{aligned} \quad (8)$$

se $\rho(\delta) > \varepsilon_1$ la mossa è accettata e riduciamo λ senno rimaniamo nella vecchia posizione. Altra domanda a cui rispondere è: quando siamo arrivati a convergenza? definiamo:

$$R1 = \max(|J^T W (y - f(x, \theta))|) \quad (9)$$

$$R2 = \max(|\delta/\theta|) \quad (10)$$

$$R3 = |S^2(x, \theta)/(n - m) - 1| \quad (11)$$

Se una di queste quantità è minore di una certa tolleranza allora l'algoritmo termina. Rimane ora un'ultima domanda a cui rispondere e possiamo passare al codice. Dato che ci servono gli errori sui parametri di fit: come calcoliamo la matrice di covarianza? Basta calcolare:

$$\text{Cov} = (J^T W J)^{-1} \quad (12)$$

quindi gli errori saranno semplicemente la radice degli elementi sulla diagonale, e le altre entrate le correlazioni fra parametri.

Passiamo ora al codice:

```

1  """
2  the code performs a linear and non linear regression
3  Levenberg-Marquardt algorithm. You have to choose
4  some parameters delicately to make the result make sense
5  """
6
7  import numpy as np
8  import matplotlib.pyplot as plt
9
10
11 def lm_fit(func, x, y, x0, sigma=None, tol=1e-6, dense_output=False, absolute_sigma=False):
12     """
13     Implementation of Levenberg-Marquardt algorithm
14     for non-linear least squares. This algorithm interpolates
15     between the Gauss-Newton algorithm and the method of
16     gradient descent. It is iterative optimization algorithms
17     so finds only a local minimum. So you have to be careful
18     about the values you pass in x0
19
20     Parameters
21     -----
22     f : callable
23         fit function
24     x : 1darray
25         the independent variable where the data is measured.
26     y : 1darray
27         the dependent data, y <= f(x, {\theta})
28     x0 : 1darray
29         initial guess
30     sigma : None or 1darray
31         the uncertainty on y, if None sigma=np.ones(len(y))
32     tol : float
33         required tollerance, the algorithm stop if one of this quantities
34         R1 = np.max(abs(J.T @ W @ (y - func(x, *x0))))
35         R2 = np.max(abs(d/x0))
36         R3 = sum(((y - func(x, *x0))/dy)**2)/(N - M) - 1
37         is smaller than tol
38
39     dense_output : bool, optional dafult False
40         if true all iteration are returned
41     absolute_sigma : bool, optional dafult False
42         If True, sigma is used in an absolute sense and
43         the estimated parameter covariance pcov reflects
44         these absolute values.
45         pcov(absolute_sigma=False) = pcov(absolute_sigma=True) * chisq(popt)/(M-N)
46
47     Returns
48     -----
49     x0 : 1d array or ndarray
50         array solution
51     pcov : 2darray
52         The estimated covariance of popt
53     iter : int
54         number of iteration
55     """
56
57     iter = 0           #initialize iteration counter
58     h = 1e-7           #increment for derivatives
59     l = 1e-3           #damping factor
60     f = 10             #factor for update damping factor
61     M = len(x0)         #number of variable
62     N = len(x)          #number of data
63     s = np.zeros(M)     #auxiliary array for derivatives
64     J = np.zeros((N, M)) #gradient
65     #some trashold
66     eps_1 = 1e-1
67     eps_2 = tol
68     eps_3 = tol
69     eps_4 = tol
70
71     if sigma is None :   #error on data
72         W = np.diag(1/np.ones(N))
73         dy = np.ones(N)
74     else :
75         W = np.diag(1/sigma**2)
76         dy = sigma

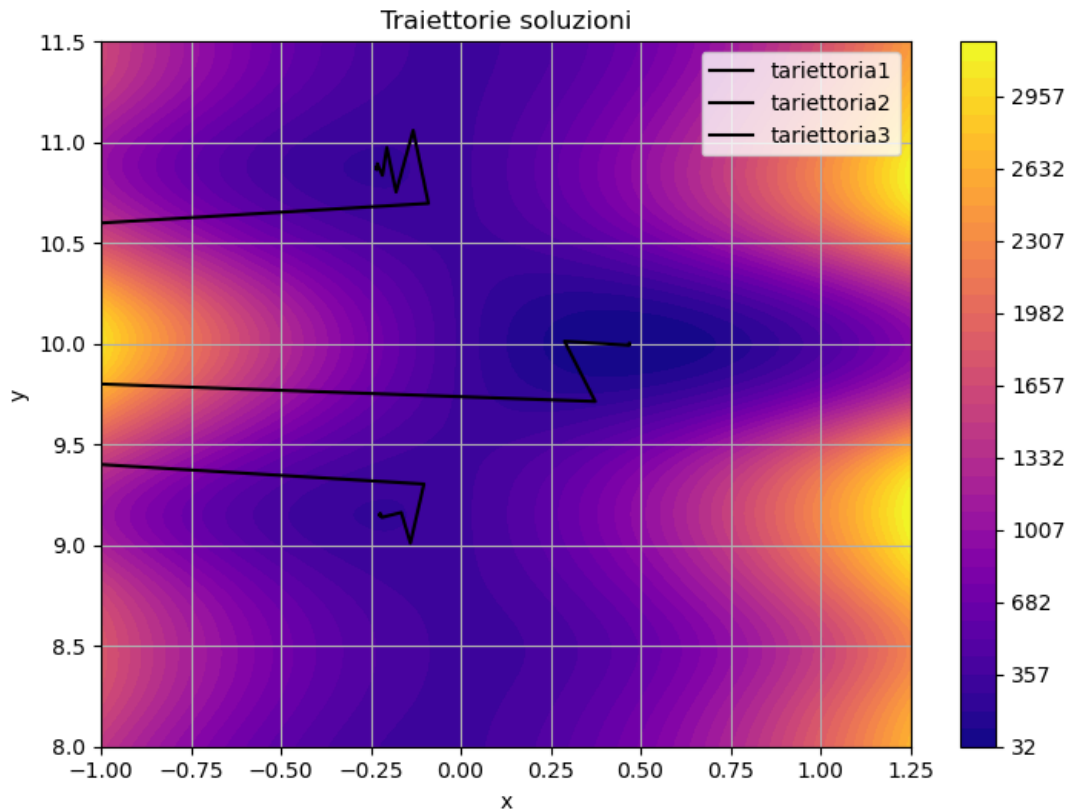
```

```

77
78
79 if dense_output:          #to store solution
80     X = []
81     X.append(x0)
82
83 while True:
84     #jacobian computation
85     for i in range(M):          #loop over variables
86         s[i] = 1                #we select one variable at a time
87         dz1 = x0 + s*h          #step forward
88         dz2 = x0 - s*h          #step backward
89         J[:,i] = (func(x, *dz1) - func(x, *dz2))/(2*h) #derivative along z's direction
90         s[:] = 0                #reset to select the other
91     variables
92
93     JtJ = J.T @ W @ J           #matrix multiplication, JtJ is an MxM matrix
94     dia = np.eye(M)*np.diag(JtJ) #dia_ii = JtJ_ii ; dia_ij = 0
95     res = (y - func(x, *x0))     #residuals
96     b = J.T @ W @ res           #ordinate or dependent variable values of
97     system
98     d = np.linalg.solve(JtJ + l*dia, b) #system solution
99     x_n = x0 + d                #solution at new time
100
101     # compute the metric
102     chisq_v = sum((res/dy)**2)
103     chisq_n = sum(((y - func(x, *x_n))/dy)**2)
104
105     rho = chisq_v - chisq_n
106     den = abs( d.T @ (l*np.diag(JtJ)*d + J.T @ W @ res))
107     rho = rho/den
108     # acceptance
109     if rho > eps_1 :             #if i'm closer to the solution
110         x0 = x_n                #update solution
111         l /= f                   #reduce damping factor
112     else:
113         l *= f                   #else magnify
114
115     # Convergence criteria
116     R1 = np.max(abs(J.T @ W @ (y - func(x, *x0))))
117     R2 = np.max(abs(d/x0))
118     R3 = abs(sum(((y - func(x, *x0))/dy)**2)/(N - M) - 1)
119
120     if R1 < eps_2 or R2 < eps_3 or R3 < eps_4: #break condition
121         break
122
123     iter += 1
124
125     if dense_output:
126         X.append(x0)
127
128 #compute covariance matrix
129 pcov = np.linalg.inv(JtJ)
130
131 if not absolute_sigma:
132     s_sq = sum(((y - func(x, *x0))/dy)**2)/(N - M)
133     pcov = pcov * s_sq
134
135 if not dense_output:
136     return x0, pcov, iter
137 else :
138     X = np.array(X)
139     return X, pcov, iter

```

Il parametro `dense_output` è stato inserito per fare un plot interessante per far vedere la dipendenza dalle condizioni iniziali. Non riportiamo l'intero codice per non appesantire, la restante parte trattava solo di fare il plot delle curve di livello. In ogni caso è disponibile nell'apposita cartella il codice intero. Questo è il primo vero codice che fa qualcosa di molto complicato esso usa tutto quanto spiegato fin'ora e adesso è evidente l'importanza di mettere commenti, dare nomi sensati e rendere leggibile il codice. Bisogna ricordarsi che i codici in genere vengono scritti una volta ma letti tante volte quindi la chiarezza non va dosata con parsimonia.



Questo grafico rappresenta le curve di livello del modello non lineare $y(t) = A \cos(\omega t)$ ed è facile vedere come partendo da condizioni diverse il fit si incastrerà in minimi locali. L'asse y corrisponde a ω mentre l'asse x corrisponde ad A . Vediamo ora di testare i risultati del codice fittando qualcosa di un po' più bruttino.

```

1  """
2  Test
3  """
4  import numpy as np
5  import matplotlib.pyplot as plt
6  from scipy.optimize import curve_fit
7  from Lev_Maqq import lm_fit
8
9
10 def f(t, A, o1, o2, f1, f2, v, tau):
11     """fit function
12     """
13     return A*np.cos(t*o1 + f1)*np.cos(t*o2 + f2)*np.exp(-t/tau) + v
14
15 ##data
16 x = np.linspace(0, 20, 1000)
17 y = f(x, 200, 10.5, 0.5, np.pi/2, np.pi/4, 42, 25)
18 rng = np.random.default_rng(seed=69420)
19 y_noise = 1 * rng.normal(size=x.size)
20 y = y + y_noise
21 dy = np.array(y.size*[1])
22
23 ##confronto
24
25 init = np.array([101, 10.5, 0.475, 1.5, 0.6, 35, 20])
26
27 pars1, covm1, iter = lm_fit(f, x, y, init, sigma=dy, tol=1e-8)
28 dpar1 = np.sqrt(covm1.diagonal())
29 pars2, covm2 = curve_fit(f, x, y, init, sigma=dy)
30 dpar2 = np.sqrt(covm2.diagonal())
31 print("-----codice-----|-----scipy-----")
32 for i, p1, dp1, p2, dp2 in zip(range(len(init)), pars1, dpar1, pars2, dpar2):
33     print(f"pars{i} = {p1:.5f} +- {dp1:.5f} ; {p2:.5f} +- {dp2:.5f}")
34
35 print(f"numero di iterazioni = {iter}")
36

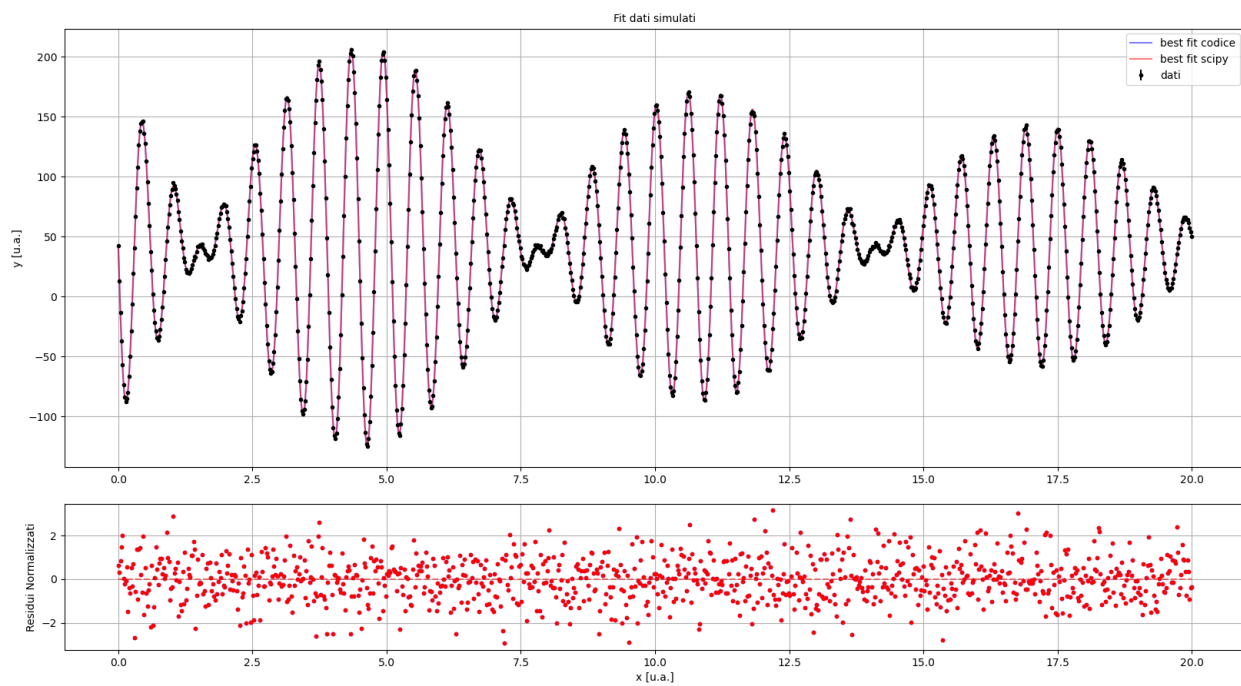
```

```

37 chisq1 = sum(((y - f(x, *pars1))/dy)**2.)
38 chisq2 = sum(((y - f(x, *pars2))/dy)**2.)
39 ndof = len(y) - len(pars1)
40 print(f'chi quadro codice = {chisq1:.3f} ({ndof:d} dof)')
41 print(f'chi quadro numpy = {chisq2:.3f} ({ndof:d} dof)')
42
43
44 c1 = np.zeros((len(pars1),len(pars1)))
45 c2 = np.zeros((len(pars1),len(pars1)))
46 #Calcoliamo le correlazioni e le inseriamo nella matrice:
47 for i in range(0, len(pars1)):
48     for j in range(0, len(pars1)):
49         c1[i][j] = (covm1[i][j])/(np.sqrt(covm1.diagonal()[i])*np.sqrt(covm1.diagonal()[j]))
50         c2[i][j] = (covm2[i][j])/(np.sqrt(covm2.diagonal()[i])*np.sqrt(covm2.diagonal()[j]))
51 #print(c1) #matrice di correlazione
52 #print(c2)
53
54 ##Plot
55 #Grafichiamo il risultato
56 fig1 = plt.figure(1)
57 #Parte superiore contenente il fit:
58 frame1=fig1.add_axes((.1,.35,.8,.6))
59 #frame1=fig1.add_axes((trasla lateralmente, trasla verticalmente, larghezza, altezza))
60 frame1.set_title('Fit dati simulati',fontsize=10)
61 plt.ylabel('y [u.a.]',fontsize=10)
62 plt.grid()
63
64
65 plt.errorbar(x, y, dy, fmt='.', color='black', label='dati') #grafico i punti
66 t = np.linspace(np.min(x), np.max(x), 10000)
67 plt.plot(t, f(t, *pars1), color='blue', alpha=0.5, label='best fit codice') #grafico del best
68     fit
69 plt.plot(t, f(t, *pars2), color='red', alpha=0.5, label='best fit scipy') #grafico del best
70     fit scipy
71 plt.legend(loc='best')#inserisce la legenda nel posto migliore
72
73 #Parte inferiore contenente i residui
74 frame2=fig1.add_axes((.1,.1,.8,.2))
75
76 #Calcolo i residui normalizzati
77 ff1 = (y - f(x, *pars1))/dy
78 ff2 = (y - f(x, *pars2))/dy
79 frame2.set_ylabel('Residui Normalizzati')
80 plt.xlabel('x [u.a.]',fontsize=10)
81
82 plt.plot(t, 0*t, color='red', linestyle='--', alpha=0.5) #grafico la retta costantemente zero
83 plt.plot(x, ff1, '.', color='blue') #grafico i residui normalizzati
84 plt.plot(x, ff2, '.', color='red') #grafico i residui normalizzati scipy
85 plt.grid()
86 plt.show()
87
88 [Output]
89
90 -----codice-----|-----scipy-----
91 pars0 = 199.85504 +- 0.17712 ; 199.85504 +- 0.17712
92 pars1 = 10.50005 +- 0.00009 ; 10.50005 +- 0.00009
93 pars2 = 0.49990 +- 0.00008 ; 0.49990 +- 0.00008
94 pars3 = 1.57040 +- 0.00087 ; 1.57040 +- 0.00087
95 pars4 = 0.78579 +- 0.00067 ; 0.78579 +- 0.00067
96 pars5 = 41.92350 +- 0.03125 ; 41.92350 +- 0.03125
97 pars6 = 24.99194 +- 0.05652 ; 24.99194 +- 0.05652
98 numero di iterazioni = 6
99 chi quadro codice = 969.017 (993 dof)
100 chi quadro numpy = 969.017 (993 dof)

```

Non abbiamo stampato la matrice di covarianza per avere un po' più di ordine. Vediamo che tra i due non ci sono differenze, siamo felici. Vediamo anche il grafico:



Qui iniziano le appendici
Hic sunt dracones



A Calcolo degli integrali

Avere a che fare con integrali di cui non si conosce l'espressione analitica è una cosa estremamente comune in fisica, anche troppo. Bisogna quindi trovare un modo per poter calcolare i vari integrali. Molti sono i metodi usati, qui voglia illustrare un metodo di quadratura gaussiana, il metodo di Gauss-Legendre. Questo metodo ci permette di approssimare l'integrale con una somma pesata:

$$\int_{-1}^1 f(x) \simeq \sum_{i=1}^n w_i f(x_i) \quad , \quad (13)$$

dove n è il numero di punti, x_i sono le radici del polinomio di Legendre di grado n $P_n(x)$, e w_i sono definiti come:

$$w_i = \frac{2}{(1 - x_i^2)(P'_n(x_i))^2} \quad . \quad (14)$$

Questo però ci dà solo integrali nell'intervallo $[-1, 1]$, problema facilmente risolvibile considerando un cambio di variabili:

$$I_{ab} = \int_a^b f(x) \simeq \frac{b-a}{2} \sum_{i=1}^n w_i f\left(x_i \frac{b-a}{2} + \frac{b+a}{2}\right) \quad . \quad (15)$$

Ora in linea di principio basterebbe questo, calcoliamo quella somma una volta ed abbiamo finito. Però vogliamo fare qualcosa di più, vogliamo fare un algoritmo adattivo. Quello che facciamo è: scelti a e b estremi in integrazione calcoliamo l'integrale poi detto $m = (a+b)/2$ il punto medio calcoliamo l'integrale nei due sotto intervalli; se la differenza fra l'integrale intero e la somma degli integrali sui due sotto intervalli è piccola, abbiamo finito, altrimenti ripartiamo ricorsivamente dividendo ciascun intervallo in altri due intervalli et cetera. Possiamo così attribuire facilmente un errore al nostro integrale, useremo proprio la differenza che usiamo come criterio di convergenza. Vediamo dal punto di vista del codice:

```
1 """
2 code for integration with Gauss-Legendre quadrature
3 """
4
5 import math
6 from scipy.special import roots_legendre
7
8 def adaptive_gaussleg_quadrature(f, a, b, args=(), tol=1e-8, n=5):
9     """
10     Splits an integral into the right and left half
11     and compares it to the integral on the whole interval
12     if tolerance is not satisfied, left and right are
13     splitted into smaller intervals and so on until the
14     tolerance is satisfied they are added to the sum
15
16     Parameters
17     -----
18     f : callable
19         function to integrate
20     a : float
21         beginning of the interval
22     b : float
23         end of the interval
24     args : tuple, optional
25         extra arguments to pass to f
26     tol : float, optional
27         tollerance, default 1e-8
28     n : int, optional
29         number of nodes, default n=5
30
31     Return
32     -----
33     Int : float
34         \int_a^b f
35     diff : float
36         error on Int
37     """
38
39     #interval divison
40     m = a + (b - a)/2
41     #compute integral
42     Int = gauss_leg(f, a, b, args, n)
43     I_r = gauss_leg(f, m, b, args, n)
44     I_l = gauss_leg(f, a, m, args, n)
```

```

45 #check tollerance
46 diff = abs( Int - (I_l + I_r) )
47 if diff < tol:
48     return Int, diff
49 #recursive call
50 div1, err1 = adaptive_gaussleg_quadrature(f, m, b, args, tol, n)
51 div2, err2 = adaptive_gaussleg_quadrature(f, a, m, args, tol, n)
52 #sum of all integrals in the several intervals
53 x = div1 + div2
54 r = err1 + err2
55
56 return x, r
57
58
59 def gauss_leg(f, a, b, args=(), n=5):
60     """
61     Calculation of the integral of f
62     with the Gaussian-Legendre quadrature method
63
64     Parameters
65     -----
66     f : callable
67         function to integrate
68     a : float
69         beginning of the interval
70     b : float
71         end of the interval
72     args : tuple, optional
73         extra arguments to pass to f
74     n : int
75         number of nodes, default n=5
76
77     Return
78     -----
79     Inte : float
80         \int_a^b f
81     """
82
83     #b must be grather tha a
84     if b < a:
85         a, b = b, a
86     else : pass
87
88     Inte = 0
89     dxdxi = (b - a)/2
90     roots, weights = roots_legendre(n)
91
92     for x_i, w_i in zip(roots, weights):
93         Inte += w_i * f(x_i * dxdxi + (b + a)/2, *args)
94
95     Inte *= dxdxi
96
97     return Inte
98
99
100 def test():
101     """ little test
102     """
103     def h(x):
104         """sine
105         """
106         return math.sin(x)
107     def f(x, a1):
108         """gaussian
109         """
110         return math.exp(-x**2/(2*a1))/(math.sqrt(2*math.pi)*a1)
111     def g(x, a1, a2):
112         """fermi dirac
113         """
114         return math.exp(-(x - a1)/a2)/(1 + math.exp(-(x - a1)/a2))
115
116     I, dI = adaptive_gaussleg_quadrature(h, 0, math.pi)
117     print('Integral value is', I, '+-', dI)
118     I, dI = adaptive_gaussleg_quadrature(f, -100, 100, args=(1,))
119     print('Integral value is', I, '+-', dI)
120     I, dI = adaptive_gaussleg_quadrature(g, 0, 20, args=(10, 0.02))

```

```

121     print('Integral value is', I, '+-', dI)
122
123
124 if __name__ == '__main__':
125     test()
126
127
128 [Output]
129 Integral value is 2.0000000000791305 +- 7.905831544974262e-11
130 Integral value is 1.00000000051542988 +- 5.442503145328187e-09
131 Integral value is 10.0 +- 0.0

```

Volendo fare un confronto con una funzione Python quella adatta è "scipy.integrate.quad" la quale adopera sempre una quadratura gaussiana ma con pesi e nodi diversi, infatti usa la tecnica: 21-point Gauss–Kronrod quadrature, con una leggera modifica. Non se ne riporta l'implementazione in quanto sarebbe analogo tolta la parte dei nodi che però son tabulati (più di 400 linee di codice). Nella cartella sarà tuttavia presente per chi fosse interessato. Giusto per completezza facciamo notare che in Gauss–Kronrod l'errore è calcolato in maniera diversa e più efficiente. Si considerano infatti due integrali che denominiamo GN, KM, rispettivamente per Gauss e Kronrod, uno calcolato in N punti e l'altro in M e come errore si considera la differenza del valore assoluto. L'implementazione di scipy è basata su (G10, K21). Nella cartella è presente il codice in cui sono raccolte le seguenti possibili composizioni ('gkdata.py'): (G7, K15), (G10, K21), (G15, K31), (G20, K41), (G25, K51), (G30, K61).

B 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.

B.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.

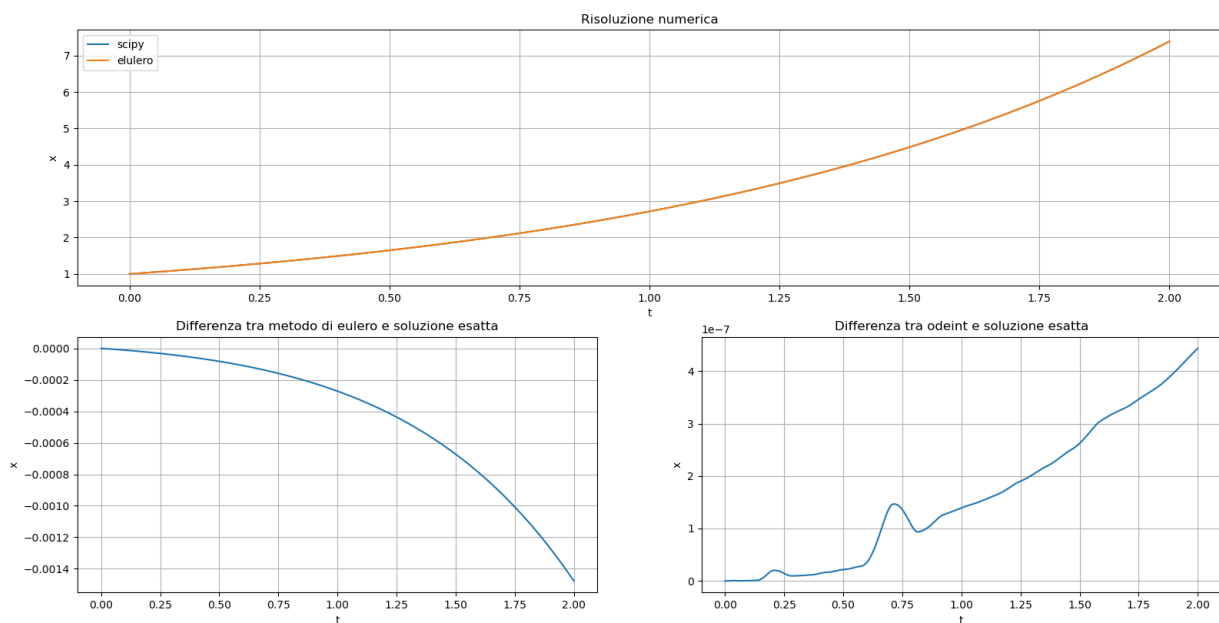
```
1 import numpy as np
2 import scipy.integrate
3 import matplotlib.pyplot as plt
4
5 #parametri
6 x0 = 1      #condizione iniziale
7 tf = 2      #fino a dove integrare
8 N = 10000   #numero di punti
9
10 #odeint
11 def ODE_1(y, t):
12     """
13     equazione da risolvere per odeint
14     """
15     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 eulero 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;

B.2 Pendolo

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

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

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:

```

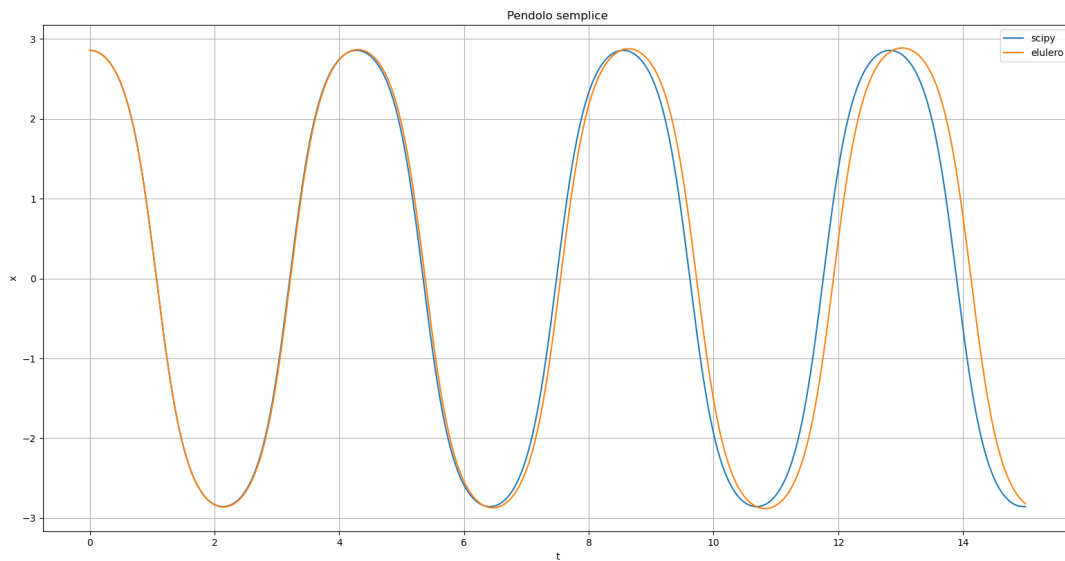
1 import numpy as np
2 import scipy.integrate
3 import matplotlib.pyplot as plt
4
5 #parametri
6 N = 100000          #numero di punti
7 l = 1               #lunghezza pendolo
8 g = 9.81            #accelerazione di gravita'
9 o0 = g/l            #frequenza piccole oscillazioni
10 v0 = 0              #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 le due soluzioni distaccarsi, 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 uno troppo piccolo lo è (esistono poi algoritmi adattivi in cui il valore del passo può cambiare durante l'integrazione). Vediamo un esempio di come varia l'errore nel calcolo di una derivata numerica al variare dell'incremento:

```

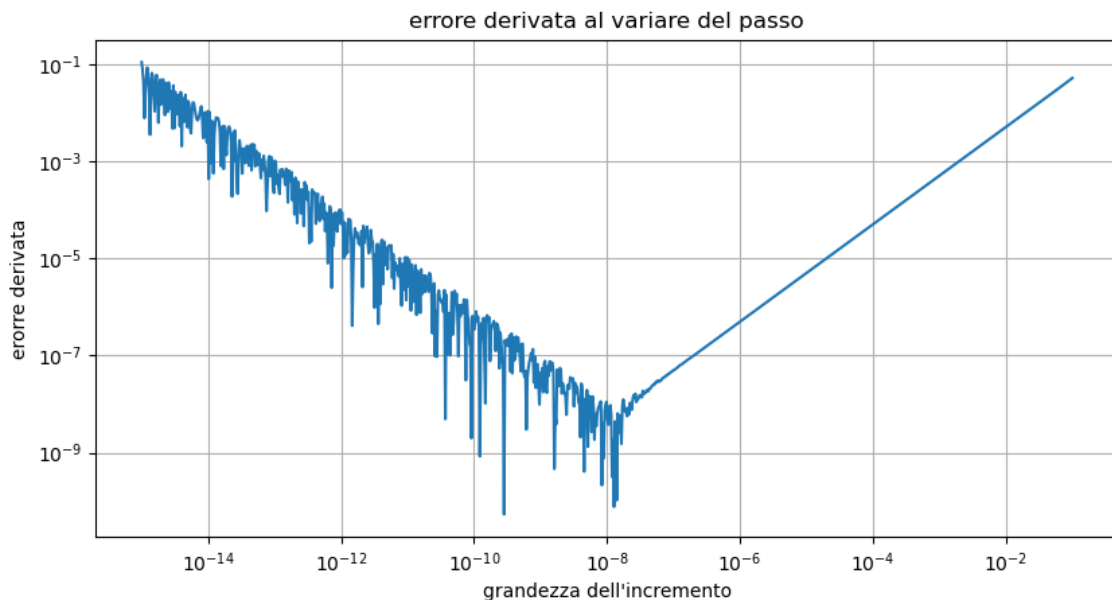
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def f(x):
5     '''
6     funzione di cui calcolare la derivata
7     '''
8     return np.exp(x)
9
10 def df(f, x, h):
11     """
12     derivata di f
13     """
14     dy = (f(x+h) - f(x))/h
15     return dy
16
17 #array del passo di discretizzazione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

```

```

25 plt.plot(h, abs(df(f, 0, h)-f(0)))
26
27 plt.xscale('log')
28 plt.yscale('log')
29 plt.grid()
30 plt.show()

```



vediamo quindi come un passo di 10^{-14} che intuitivamente potremmo credere migliore da lo stesso errore di un passo di 10^{-2} . Inoltre tutta la trattazione è immancabilmente affetta dal metodo di approssimazione che usiamo. Un algoritmo di alto ordine darà risultati migliori con un passo grande piuttosto che uno piccolo. Vediamo diversi modi di approssimare una derivata prima:

$$f' = \frac{f(x+h) - f(x)}{h} \quad (16)$$

$$f' = \frac{f(x) - f(x-h)}{h} \quad (17)$$

$$f' = \frac{f(x+h) - f(x-h)}{2h} \quad (18)$$

$$f' = \frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h} \quad (19)$$

$$f' = \frac{3f(x) - 4f(x-h) + f(x-2h)}{2h} \quad (20)$$

$$f' = \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h} \quad (21)$$

Rispettivamente i primi due primo ordine, i seguenti tre del secondo ordine e il terzo del quarto ordine. Vediamo un piccolo codice nel quale vi potete divertire a cambiare la grandezza passo per rendervi conto di quanto dicevamo:

```

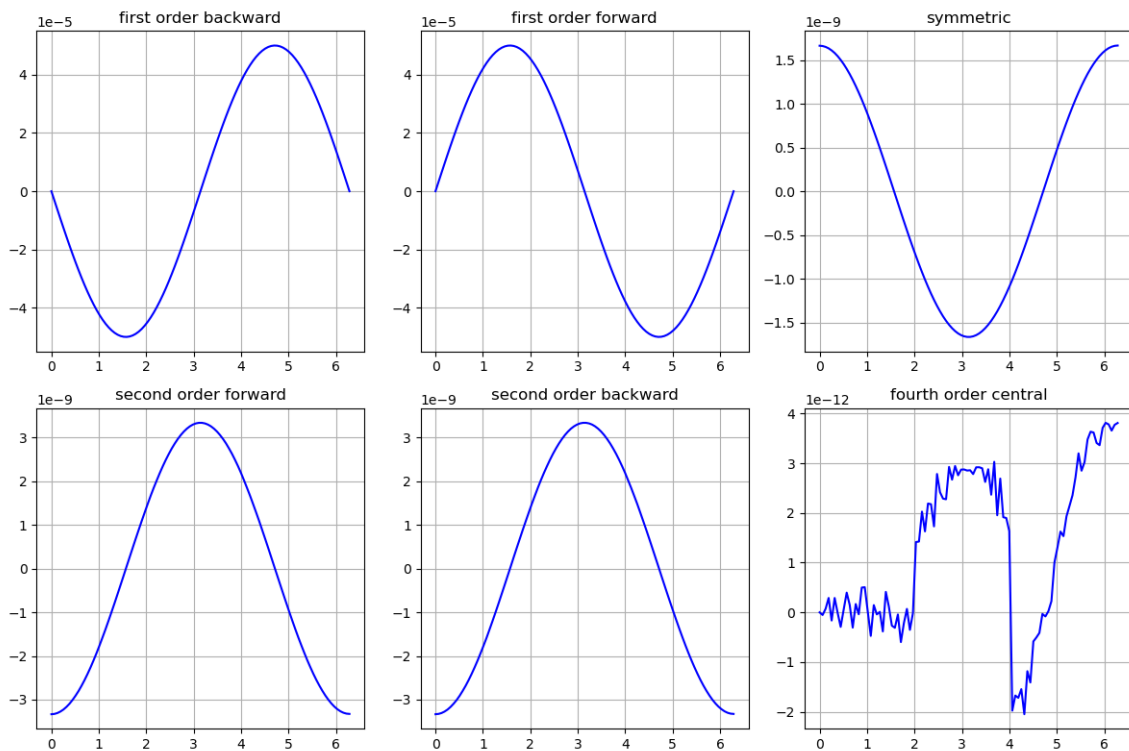
1 """
2 Code that compiles some method of calculating a derivative to various orders of accuracy
3 """
4 import numpy as np
5 import matplotlib.pyplot as plt
6
7
8 def d1b(f, x0, dx):
9     '''first order, backward derivative
10     '''
11     dfdx = (f(x0) - f(x0-dx))/dx
12     return dfdx
13
14 def d1f(f, x0, dx):

```

```

15     '''first order, forward derivative
16     '''
17     dfdx = (f(x0+dx) - f(x0))/dx
18     return dfdx
19
20 def d2c(f, x0, dx):
21     '''second order, symmetric derivative
22     '''
23     dfdx = (f(x0+dx) - f(x0-dx))/(2.0*dx)
24     return dfdx
25
26 def d2f(f, x0, dx):
27     ''' second order forward derivative
28     '''
29     dfdx = ( - 3.0*f(x0) + 4.0*f(x0+dx) - f(x0+2.0*dx) )/(2.0*dx)
30     return dfdx
31
32 def d2b(f, x0, dx):
33     ''' second order backward derivative
34     '''
35     dfdx = ( 3.0*f(x0) - 4.0*f(x0-dx) + f(x0-2.0*dx) )/(2.0*dx)
36     return dfdx
37
38 def d4c(f, x0, dx):
39     ''' fourth order centered derivative
40     '''
41     dfdx = ( -f(x0+2*dx) + 8.0*f(x0+dx) - 8.0*f(x0-dx) + f(x0-2*dx) )/(12.0*dx)
42     return dfdx
43
44 #=====
45 # Plot
46 #=====
47
48 plt.figure(1, figsize=(12, 8))
49 x = np.linspace(0, 2*np.pi, 100)
50 f = np.sin
51 g = np.cos
52 h = 1e-4
53 Df = [d1b, d1f, d2c, d2f, d2b, d4c]
54 l = ['first order backward', 'first order forward', 'symmetric',
55      'second order forward', 'second order backward', 'fourth order central']
56
57 for i, df in enumerate(Df):
58     plt.subplot(2, 3, i+1)
59     plt.title(l[i])
60     plt.plot(x, g(x)-np.array([df(f, t, h) for t in x]), 'blue')
61     plt.grid()
62
63 plt.tight_layout()
64 plt.show()

```



Insomma avrete capito che è un argomento delicato.

B.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 coordinate 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.

C Sistemi lineari

Spesso capita, come abbiamo visto sopra con il caso dei fit, di dover risolvere dei sistemi di equazioni o di invertire una matrice. Abbiamo un sistema del tipo:

$$Ax = b \quad (22)$$

dove

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

C.1 Metodo Gauss-Seidel

Detta L una matrice triangolare inferiore e U matrice triangolare superiore, con diagonale nulla tale che $A = L + U$

$$L = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \quad (23)$$

Allora abbiamo:

$$Ax = b \quad (24)$$

$$(L + U)x = b \quad (25)$$

$$Lx + Ux = b \quad (26)$$

$$Lx = b - Ux \quad (27)$$

e quindi abbiamo in maniera iterativa:

$$x^{k+1} = L^{-1}(b - Ux^k) \quad (28)$$

e per la riga i -esima di x^{k+1} possiamo scrivere:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^n a_{ij} x_j^k \right) \quad (29)$$

Questo metodo converge solo per alcune caratteristiche della matrice A : A deve essere simmetrica definita positiva, oppure deve essere dominante diagonale ($|a_{ii}| \geq \sum_{j \neq i} |a_{ij}| \quad \forall i$). Esiste una versione modificata di questo metodo chiamata: successive over-relaxation (SOR)

C.2 Successive over-relaxation

Se ora scomponiamo A come $D + L + U$:

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \quad L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix} \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \quad (30)$$

Introducendo il parametro ω di overrelaxation l'algoritmo diventa:

$$x_i^{k+1} = (1 - \omega)x_i^k + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^n a_{ij} x_j^k \right) \quad (31)$$

Vediamo quindi che per $\omega = 1$ recuperiamo il metodo precedente. Vediamo ora un esempio di codice e vedremo che due esecuzioni con ω diversi cambiano molto. In particolare $0 < \omega < 2$ e valori più vicini a due ne migliorano la convergenza.


```

1 import time
2 import numpy as np
3
4 def SOR(A, b, omg, tol=1e-6):
5     """
6     Implementation of successive over-relaxation method for solve  $A @ x = b$ ;
7     A must be:
8     1) symmetric (i.e.  $A.T = A$ )
9     2) positive-definite (i.e.  $x.T @ A @ x > 0$  for all non-zero  $x$ )
10    3) real.
11
12    Parameters
13    -----
14    A : 2darray
15        matrix of system.
16    b : 1darray
17        Ordinate or dependent variable values.
18    tol : float, optional
19        required tollerance default 1e-6
20
21    Return
22    -----
23    x : 1darray
24        solution of system
25    iter : int
26        number of iteration
27    """
28    x = np.zeros(len(b))
29    iter = 0
30    while True:
31
32        x_new = np.zeros(len(x))
33
34        for i in range(A.shape[0]):
35            s1 = np.dot(A[i, :i], x_new[:i])
36            s2 = np.dot(A[i, i + 1:], x[i + 1:])
37            x_new[i] = (1 - omg)*x[i] + omg*(b[i] - s1 - s2) / A[i, i]
38
39        res = np.sqrt(np.sum((A @ x_new - b)**2))
40        if res < tol:
41            break
42
43        x = x_new
44        iter += 1
45
46    return x, iter
47
48 if __name__ == "__main__":
49     np.random.seed(69420)
50     N = 20
51     A = np.random.normal(size=[N, N])
52     A = A.T @ A #per garantire la convergenza del metodo
53     b = np.random.normal(size=[N])
54
55     start = time.time()
56     x1, iter = SOR(A, b, 1.9, 1e-8)
57     print(f"number of iteration = {iter}")
58     print(f"Elapsed time = {time.time()-start}")
59
60     start = time.time()
61     x2 = np.linalg.solve(A, b)
62     print(f"Elapsed time = {time.time()-start}")
63
64     d = np.sqrt(np.sum((x1 - x2)**2))
65     print(f'difference with numpy = {d}')
66
67 [Output] (omg=1)
68 number of iteration = 10794
69 Elapsed time = 1.4843645095825195
70 Elapsed time = 0.0
71 difference with numpy = 4.5979072526656344e-07
72
73 [Output] (omg=1.9)
74 number of iteration = 2590
75 Elapsed time = 0.35565781593322754
76 Elapsed time = 0.0

```

```
77 difference with numpy = 2.1632731207202056e-08
```

C.3 Metodo del gradiente coniugato

Un altro modo per risolvere sistemi lineari, e che ci permettere di far crescere la dimensione dalla matrice, è il metodo del gradiente coniugato. La matrice A come sopra deve essere definita positiva. L'algoritmo è iterativo e l'aggiornamento della posizione è fatto secondo la seguente regola:

$$x_{k+1} = x_k + \alpha_k p_k \quad (32)$$

dove p_k è la direzione di discesa ed è scelto in modo che sia ortogonale rispetto al prodotto scalare indotto da A : $\langle p_j, p_k \rangle_A = 0, \forall j = 0, \dots, k-1$. Mentre α_k è la grandezza del passo, $\frac{p_k^T r_k}{p_k^T A p_k}$. Vediamone l'implementazione:

```
1  """
2  Implementation and test for
3  conjugate gradient method
4  """
5
6  import time
7  import numpy as np
8  import matplotlib.pyplot as plt
9  from scipy.sparse.linalg import cg
10
11
12  def conj_grad(A, b, tol=1e-6, dense_output=False):
13      """
14      Implementation of conjugate gradient method for solve A @ x = b
15      A must be:
16      1) symmetric (i.e. A.T = A)
17      2) positive-definite (i.e. x.T @ A @ x > 0 for all non-zero x)
18      3) real.
19
20      Parameters
21      -----
22      A : 2darray
23          matrix of system.
24      b : 1darray
25          Ordinate or dependent variable values.
26      tol : float, optional
27          required tollerance default 1e-6
28      dense_output : bool, optional
29          if True all iteration of th solution, error and number
30          of iteration are stored and returned, default is False
31
32      Return
33      -----
34      x : 1darray
35          solution of system
36      err : float
37          error of solution
38
39      if dense_output :
40      s : 2darray
41          all iteration of solution
42      e : 1darray
43          error of all iteration,
44      iter : int
45          number of iteration
46
47      """
48
49      N = len(b)
50      x = np.zeros(N) #initia guess
51
52      r = b - A @ x #residuals
53      p = r #descent direction
54      r2 = sum(r*r) #norm^2 residuals
55
56      if dense_output:
57          s = []
58          e = []
59          s.append(x)
60          e.append(np.sqrt(r2))
61
```

```

62     iter = 0
63
64     while True:
65
66         Ap = A @ p           #computation of
67         alpha = r2 / (p @ Ap) #descent's step
68
69         x = x + alpha * p     #update position
70         r = r - alpha * Ap    #update residuals
71
72         r2_new = sum(r*r)     #norm^2 new residuals
73         beta = r2_new/r2      #compute step for p
74
75         r2 = r2_new           #update norm
76
77         if dense_output:
78             s.append(x)
79             e.append(np.sqrt(r2))
80
81         if np.sqrt(r2_new) < tol : #break condition
82             break
83
84         p = r + beta * p      #update p
85         iter += 1
86
87     if not dense_output:
88         err = np.sqrt(r2_new)
89         return x, err
90     else:
91         return np.array(s), np.array(e), iter
92
93
94 if __name__ == '__main__':
95
96     np.random.seed(69420)
97
98     N = 1000
99     P = np.random.normal(size=[N, N])
100    A = np.dot(P.T, P) #deve essere simmetrica e semidef >0
101    b = np.random.normal(size=[N])
102
103    t1 = time.time()
104    sol, err, iter = conj_grad(A, b, 1e-8, dense_output=True)
105    x1 = sol[-1]
106    t2 = time.time()
107    print(f'numero di iterazioni: {iter}')
108    print(f'Elapsed time          = {t2 - t1}')
109
110    t1 = time.time()
111    x2 = np.linalg.solve(A, b)
112    t2 = time.time()
113    print(f'Elapsed time numpy = {t2 - t1}')
114
115    t1 = time.time()
116    x3, exit_code = cg(A, b)
117    t2 = time.time()
118    print(f'Elapsed time scipy = {t2 - t1}')
119
120    print('confronto soluzioni')
121    print(f"distanza delle due soluzioni(cg-n) = {np.sqrt(np.sum((x1-x2)**2))}")
122    print(f"distanza delle due soluzioni(cg-s) = {np.sqrt(np.sum((x1-x3)**2))}")
123
124
125    plt.figure(1)
126    plt.grid()
127    plt.plot(abs(err))
128    plt.xlabel('iteration')
129    plt.ylabel('error')
130    #plt.xscale('log')
131    plt.yscale('log')
132    plt.show()
133
134 [Output]
135 numero di iterazioni: 1916
136 Elapsed time          = 1.2752277851104736
137 Elapsed time numpy = 0.03131294250488281

```

```
138 Elapsed time scipy = 1.1133522987365723
139 confronto soluzioni
140 distanza delle due soluzioni(cg-n) = 3.295292002340236e-08
141 distanza delle due soluzioni(cg-s) = 5.375356794116895e-07
```

Come è possibile vedere ne abbiamo guadagnato molto con questo metodo, risulta essere più veloce e non fatica non matrici molto grosse.

D Zeri di una funzione

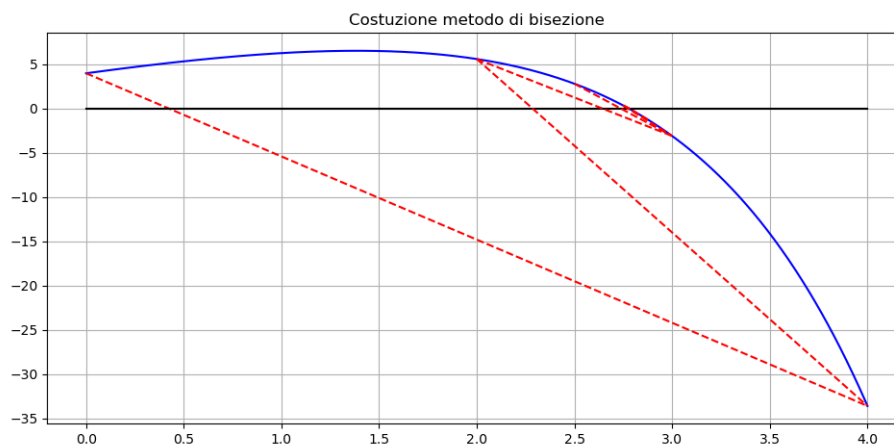
Capita spesso la necessità di trovare gli zeri di una funzione, o più, per risolvere un'equazione o un sistema di equazioni. Brevemente vedremo due metodi per la risoluzione di un'equazione, quindi per trovare lo zero, o gli zeri, di una funzione: il metodo di bisezione e il metodo di Newton, o delle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

D.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...

D.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    return x0, iter
21
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:

```

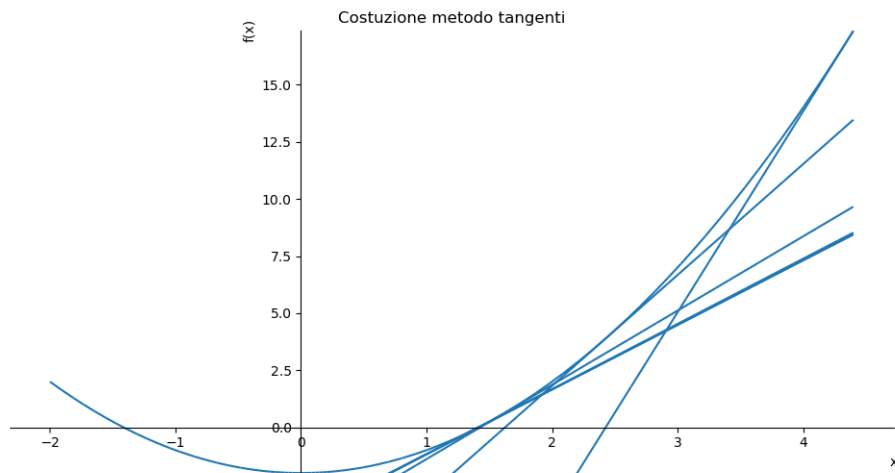
1 import sympy as sp
2 from sympy.plotting import plot
3
4 x = sp.Symbol('x')
5 f = x**2 -2 #funzione di cui trovare gli zeri
6 df = sp.diff(f, x) #derivata della funzione f
7 t = 1e-13 #tolleranza
8 init = 4.4
9 P1 = plot(f, (x, -2, init), ylim=(-2.2, f.subs(x,init)), show=False, title='Costuzione metodo
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     return x0, iter
28
29 #valore iniziale da cui partire
30 xs, iter = tangenti(init, t)
31
32 print(iter , " iterazioni necessarie")
33 print("xs= %.15f" %xs)
34 print("|f(xs)|= %e" %abs(f.subs(x,xs)))
35
36 P1.show()

```

```

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...

D.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. Passiamo quindi da un sistema non lineare a risolvere diverse volte, per ogni iterazione, un sistema lineare:

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

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 possiamo anche visualizzare l'evoluzione della soluzione, 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 newton method for nonlinear system of equations
3 """
4 import numpy as np
5 import matplotlib.pyplot as plt
6 from scipy.optimize import fsolve, root
7
8 #=====
9 # Newron method implementation
10 #=====
11
12 def newton(f, start, tol, args=(), dense_output=False, max_it=1000):
13     '''
14     Generalizzazion of newton method for n equations
15
16     Parameters
17     -----
18     f : callable

```



```

19     A vector function to find a root of.
20     start : ndarray
21     Initial guess. The method is very sensitive to this
22     must be chosen carefully
23     tol :float, optional, default 1e-8
24     required tollerance
25     args : tuple, optional
26     Extra arguments passed to f
27     dense_output : bool, optional
28     true for full and number of iteration
29     max_it : int, optional, default 1000
30     after max_it iteration the code stop raising an exception
31
32     Return
33     -----
34     x0 : ndarray
35     solution of the system
36     if dense_output=True all iteration are returned
37     in a matrix called X and also the number of iteration
38     ,,,
39
40     # initial guess
41     x0 = start
42     f0 = f(x0, *args)
43     # for the computation of jacobian
44     nd = len(x0)
45     df = np.zeros((nd, nd))
46     h = 1e-8
47     s = np.zeros(nd)
48     #for full output
49     X = []
50     if dense_output : X.append(x0)
51     # count
52     n_iter = 0
53
54     while True:
55
56         # compute jacobian using symmetric derivative
57         for i in range(nd):      # loop over functions
58             for j in range(nd):  # loop over variables
59                 s[j] = 1
60                 xr, xl = x0 + h*s, x0 - h*s
61                 df[i, j] = (f(xr, *args) - f(xl, *args) )/h
62                 s[:] = 0
63
64         # update solution
65         delta = np.linalg.solve(df, f0)
66         x0 = x0 - delta
67         f0 = f(x0, *args)
68
69         if dense_output:
70             n_iter += 1
71             X.append(x0)
72
73         # stop condition
74         if all(abs(f0) < tol):
75             break
76         # check iterations
77         if n_iter > max_it :
78             err_msg = 'too many iteration, failure to converge, change initial guess'
79             raise Exception(err_msg)
80
81         if dense_output:
82             return np.array(X), n_iter
83         else:
84             return x0
85
86     #=====
87     # System to solve
88     #=====
89
90     def system(V):
91         x1, x2 = V
92
93         r1 = x1**2 + x2**2 - 1 #x3
94         r2 = x2 - x1**2 + x1/2 #+ x3/5

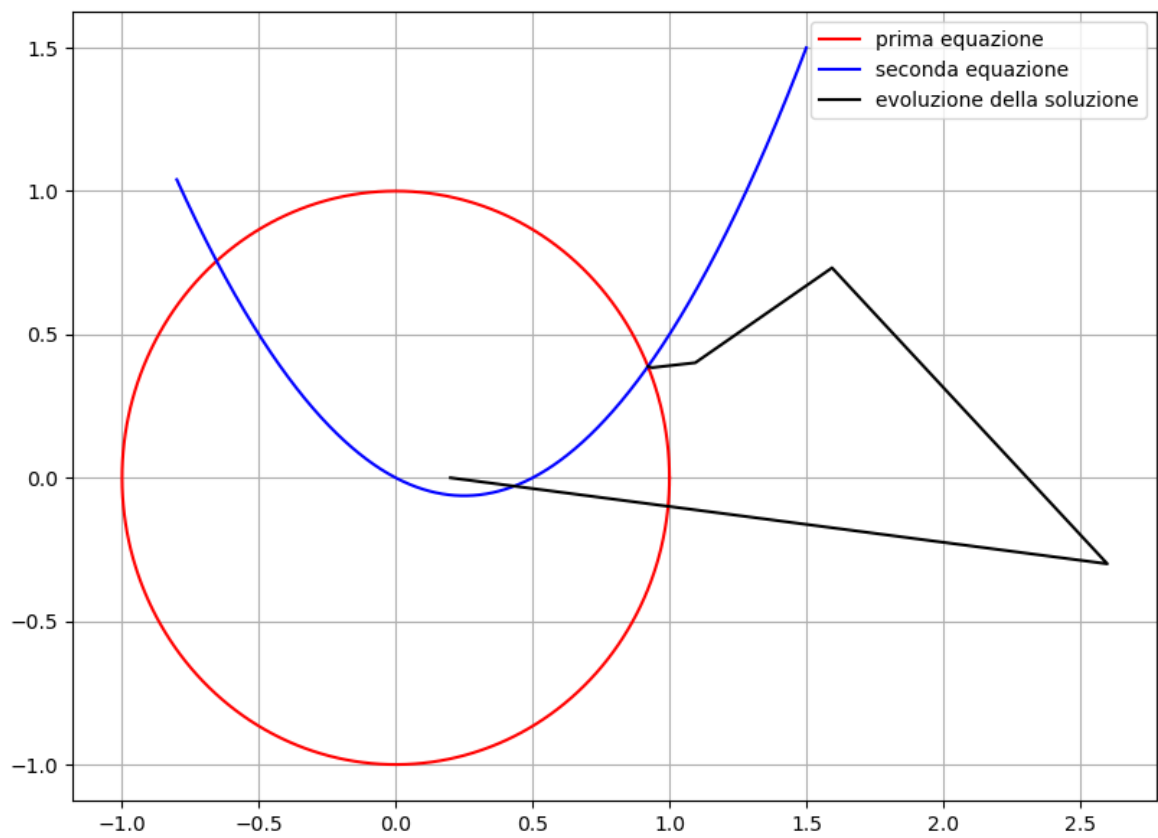
```

```

95     #r3 = x3**2 + 5*x2 - 7
96
97     R = np.array([r1, r2])#, r3])
98     return R
99
100 #=====
101 # Solution and compare with scipy
102 #=====
103
104 init = np.array([0.2, 0.0])
105 tol = 1e-12
106 sol, n_iter = newton(system, init, tol=tol, dense_output=True)
107 xs, ys = sol.T
108 print("Solution with newton: ", *sol[-1], "in", n_iter, "iterations")
109
110 sol = root(system, init, method='hybr', tol=tol)
111 print("Solution with root: ", *sol.x)
112
113 sol = fsolve(system, init, xtol=tol)
114 print("Solution with fsolve: ", *sol)
115
116 #=====
117 # Plot
118 #=====
119
120 t = np.linspace(0, 2*np.pi, 1000)
121 z = np.linspace(-0.8, 1.5, 1000)
122 plt.figure(1)
123 plt.grid()
124 plt.plot(np.cos(t), np.sin(t), 'r', label='first equation')
125 plt.plot(z, z**2 - z/2, 'b', label='second equation')
126 plt.plot(xs, ys, 'k', label='evolution of solution')
127 plt.legend(loc='best')
128 plt.show()
129
130 [Output]
131 Solution with newton:  0.9214908788160613  0.38840000033316596 in 7 iterations
132 Solution with root:    0.9214908788160611  0.388400000333166
133 Solution with fsolve:  0.9214908788160611  0.388400000333166

```

Vediamo che il nostro codice funziona ci sono però un paio di problemi. Esso presenta problemi simili a quello unidimensionale. Ci sono casi in cui lo jacobiano può non esistere, o essere singolare. Inoltre se già gli algoritmi sofisticati sono sensibili alle condizioni iniziali, questo metodo, essendo molto base, lo è infinitamente di più. Provate a cambiare un po' il valore della guess iniziale e vedrete il macello che si combina. Una strategia spesso usata, che è ciò che fa scipy (non a caso le funzioni se avete notato sono nella libreria "scipy.optimize", la stessa di "curve_fit"), è quella di promuovere il problema dalla ricerca di uno zero alla ricerca di un minimo. Ovvero si passa da cercare lo zero di f_i alla ricerca del minimo della somma su i delle f_i al quadrato $S^2 = \sum_i f_i^2$. Con qualche modifica si può anche usare metodo visto per fare i fit ma la cosa migliore e in genere più usata è risolvere questi problemi con degli algoritmi chiamati "trust-region".



E 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:

E.1 Equazione del trasporto

L'equazione di nostro interesse è:

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

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

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

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

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

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

```
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.

E.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()

```

F 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()
```

G Trasformate di Fourier

G.1 DFT

La trasformata di Fourier è una trasformata integrale che ci permette di cambiare dominio della nostra funzione: ad da tempo a frequenze o da spazio a numero d'onda.

$$f(t) = \mathcal{F}(f(t))(\omega) = \int_0^\infty \tilde{f}(\omega) e^{-i\omega t} dt \quad (33)$$

$$\tilde{f}(\omega) = \mathcal{F}^{-1}(\tilde{f}(\omega))(t) = \frac{1}{2\pi} \int_0^\infty f(t) e^{-i\omega t} d\omega \quad (34)$$

e gode di varie proprietà:

$$\mathcal{F}(D^k f) = (-i\omega)^k \mathcal{F}(f) \quad \mathcal{F}(f(t-a)) = e^{i\omega a} \mathcal{F}(f(t)) \quad \mathcal{F}(e^{i\omega a} f(t)) = \tilde{f}(\omega-a) \quad \mathcal{F}((it)^k f) = D^k \mathcal{F}(f) \quad (35)$$

Andando nel discreto abbiamo:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn}, \quad k = 0, \dots, N-1 \quad (36)$$

e non è difficile vedere che fondamentalmente la trasformata di Fourier discreta (DFT) è un prodotto matrice per vettore:

$$X_k = W_{kn} x_n \quad W_{kn} = e^{-\frac{2\pi i}{N} kn} \quad (37)$$

e l'anti trasformata non sarà altro che:

$$X_k = \frac{1}{N} W_{kn}^{-1} x_n \quad W_{kn}^{-1} = e^{\frac{2\pi i}{N} kn} \quad (38)$$

Dunque è facile notare che la complessità dell'algoritmo è $\mathcal{O}(N^2)$; vediamone una semplice implementazione:

```
1 """
2 Implementation of DFT
3 """
4 import time
5 import numpy as np
6 import matplotlib.pyplot as plt
7
8
9 def DFT(x, anti=-1):
10     '''
11     Compute the discrete Fourier Transform of the 1D array x
12
13     Parameters
14     -----
15     x : 1darray
16         data to transform
17     anti : int, optional
18         -1 trasform
19         1 anti trasform
20
21     Return
22     -----
23     dft : 1d array
24         dft or anti dft of x
25     '''
26
27     N = len(x)          # length of array
28     n = np.arange(N)    # array from 0 to N
29     k = n[:, None]      # transposed of n written as a Nx1 matrix
30     # is equivalent to k = np.reshape(n, (N, 1))
31     # so k * n will be a N x N matrix
32
33     M = np.exp(anti * 2j * np.pi * k * n / N)
34     dft = M @ x
35
36     if anti == 1:
37         return dft/N
38     else:
39         return dft
```


G.2 FFT

I signori Cooley e Tukey si inventarono un modo per accelerare un po' il calcolo della DFT e crearono la FFT (trasformata di Fourier veloce) la quale ha ordine $\mathcal{O}(N \ln_2(N))$. Qui vedremo il caso più semplice, quello in cui l'array da trasformare deve essere lungo necessariamente una potenza di 2 (FFT radix 2). Esistono anche altri algoritmi, chiamati a radice mista, in cui possiamo rilassare questo vincolo, ad esempio le funzioni di numpy non hanno questo vincolo. vediamo brevemente l'algoritmo:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \quad (39)$$

$$= \sum_{n=0}^{N/2-1} x_{2n} e^{-\frac{2\pi i}{N} k(2n)} + \sum_{n=0}^{N/2-1} x_{2n+1} e^{-\frac{2\pi i}{N} k(2n+1)} \quad (40)$$

$$= \sum_{n=0}^{N/2-1} x_{2n} e^{-\frac{2\pi i}{N} k(2n)} + e^{-\frac{2\pi i}{N} k} \sum_{n=0}^{N/2-1} x_{2n+1} e^{-\frac{2\pi i}{N} kn} \quad (41)$$

$$= DFT(x_{2n}) + e^{-\frac{2\pi i}{N} k} DFT(x_{2n+1}) \quad (42)$$

e quindi ripetiamo ricorsivamente questa divisione, fino ad arrivare ad un N_{min} che blocca la ricorsione e ed esegue una DFT come vista sopra; quindi N/N_{min} DFT:

```

1 """
2 Implementetion of FFT
3 """
4 import time
5 import numpy as np
6 import matplotlib.pyplot as plt
7
8
9 def FFT(x, anti=1):
10     '''
11     A recursive implementation of the Cooley-Tukey FFT
12
13     Parameters
14     -----
15     x : 1darray
16         data to transform
17     anti : int, optional
18         -1 trasform
19         1 anti trasform
20
21     Return
22     -----
23     fft : 1d array
24         fft or anti fft of x
25     '''
26
27     N = x.shape[0]
28
29     if N % 2 > 0:
30         raise ValueError("size of x must be a power of 2")
31     elif N <= 32:
32         return DFT(x, anti)
33     else:
34         X_even = FFT(x[0::2])
35         X_odd = FFT(x[1::2])
36         factor = np.exp(-anti*2j * np.pi * np.arange(N) / N)
37         return np.concatenate([X_even + factor[:N / 2] * X_odd,
38                                X_even + factor[N / 2:] * X_odd])

```

cerchiamo di capire perché questa divisione accelera il calcolo.

- $\frac{N}{2} \longrightarrow 2 \frac{N^2}{2^2} + N = \frac{N^2}{2} + N$
- $\frac{N}{4} \longrightarrow 2 \left(2 \left(\frac{N}{4} \right)^2 + \frac{N}{2} \right) + N = \frac{N^2}{4} + 2N$
- \vdots
- $\frac{N}{2^p} \longrightarrow \frac{N^2}{2^p} + pN = \frac{N^2}{N} + \ln_2(N)N \longrightarrow \mathcal{O}(N \ln_2(N))$

Dove abbiamo usato che $N = 2^m$ allora al massimo deve essere $p = m = \ln_2(N)$.

Possiamo però costruire un algoritmo iterativo che ottimizzi il codice sopra scritto e la vettorizzazione di Python ci aiuta:

```

1  """
2  Implementetion of FFT
3  """
4  import time
5  import numpy as np
6  import matplotlib.pyplot as plt
7
8  def FFT(x, anti=-1):
9      '''
10     Compute the Fast Fourier Transform of the 1D array x.
11     Using non recursive Cooley-Tukey FFT.
12     In recursive FFT implementation, at the lowest
13     recursion level we must perform N/N_min DFT.
14     The efficiency of the algorithm would benefit by
15     computing these matrix-vector products all at once
16     as a single matrix-matrix product.
17     At each level of recursion, we also perform
18     duplicate operations which can be vectorized.
19
20     Parameters
21     -----
22     x : 1darray
23         data to transform
24     anti : int, optional
25         -1 trasform
26         1 anti trasform
27
28     Return
29     -----
30     fft : 1d array
31         fft or anti fft of x
32
33     '''
34     N = len(x)
35
36     if np.log2(N) % 1 > 0:
37         msg_err = "The size of x must be a apower of 2"
38         raise ValueError(msg_err)
39
40     # stop criterion
41     N_min = min(N, 2**2)
42
43     # DFT on all length-N_min sub-problems
44     n = np.arange(N_min)
45     k = n[:, None]
46     M = np.exp(anti * 2j * np.pi * n * k / N_min)
47     X = np.dot(M, x.reshape((N_min, -1)))
48
49     while X.shape[0] < N:
50         # first part of the matrix, the one on the left
51         X_even = X[:, X.shape[1] // 2 :] # all rows, first X.shape[1]//2 columns
52         # second part of the matrix, the one on the right
53         X_odd = X[:, X.shape[1] // 2:] # all rows, second X.shape[1]//2 columns
54
55         f = np.exp(anti * 1j * np.pi * np.arange(X.shape[0]) / X.shape[0])[:, None]
56         X = np.vstack([X_even + f*X_odd, X_even - f*X_odd]) # re-merge the matrix
57
58     fft = X.ravel() # flattens the array
59     # from matrix Nx1 to array with length N
60
61     if anti == 1:
62         return fft/N
63     else :
64         return fft

```

G.3 RFFT

Ultima interessante implementazione è il caso in cui l'input sia reale, per cui è possibile definire una variabile complessa fare una FFT lunga la meta e ricostruire lo spettro con le proprietà di simmetria della FFT. Se

siamo interessati alle frequenze positive, che è il caso usuale basta fondamentalmente prendere i primi $N/2 + 1$ elementi della nostra FFT.

```

1 def RFFT(x, anti=-1):
2     '''
3     Compute the fft for real value using FFT
4     only values corresponding to positive
5     frequencies are returned.
6
7     The transform is implemented by passing to
8     a complex variable z = x[2n] + j x[2n+1]
9     then an fft of length N/2 is calculated
10    For the inverse we adopt an other method
11    (the previous method didn't work,
12    if you know how to fix it ... I would be grateful)
13
14    Parameters
15    -----
16    x : 1darray
17        data to transform
18    anti : int, optional
19        -1 trasform
20        1 anti trasform
21
22    Return
23    -----
24    rfft : 1d array
25        rfft or anti rfft of x
26    '''
27    if anti == -1 :
28        z = x[0::2] + 1j * x[1::2] # Splitting odd and even
29        Zf = FFT(z)
30        Zc = np.array([Zf[-k] for k in range(len(z))]).conj()
31        Zx = 0.5 * (Zf + Zc)
32        Zy = -0.5j * (Zf - Zc)
33
34        N = len(x)
35        W = np.exp(- 2j * np.pi * np.arange(N//2) / N)
36        Z = np.concatenate([Zx + W*Zy, Zx - W*Zy])
37
38        return Z[:N//2+1]
39
40    if anti == 1 :
41        # we use the fft symmetries to reconstruct the whole spectrum
42        N = 2*(len(x)-1) # length of final array
43        x1 = x[:-1] # cut last value
44        S = len(x1) # length of new array
45        xn = np.zeros(N, dtype=complex)
46        xn[0:S] = x1
47        xx = x[1:] # cut first element, zero frequency mode
48        xx = xx[::-1] # rewind array
49        xn[S:N] = xx.conj()
50        z = FFT(xn, anti=1)
51
52    return np.real(z)

```

Ultima cosa da vedere è come creare l'array delle frequenze, quello che sarebbe np.fft.fftfreq:

```

1 def fft_freq(n, d, real):
2     '''
3     Return the Discrete Fourier Transform sample frequencies.
4     if real = False then:
5     f = [0, 1, ..., n/2-1, -n/2, ..., -1] / (d*n) if n is even
6     f = [0, 1, ..., (n-1)/2, -(n-1)/2, ..., -1] / (d*n) if n is odd
7     else :
8     f = [0, 1, ..., n/2-1, n/2] / (d*n) if n is even
9     f = [0, 1, ..., (n-1)/2-1, (n-1)/2] / (d*n) if n is odd
10
11    Parameters
12    -----
13    n : int
14        length of array that you transform
15
16    d : float
17        Sample spacing (inverse of the sampling rate).
18        If the data array is in seconds
19        the frequencies will be in hertz

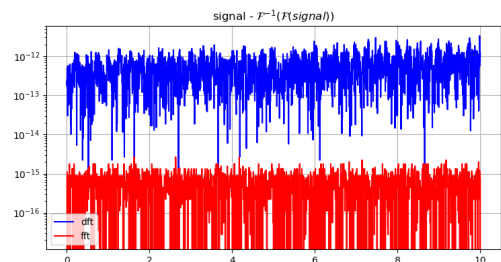
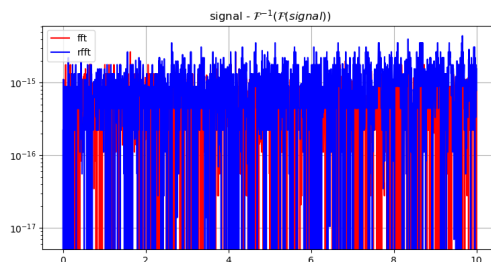
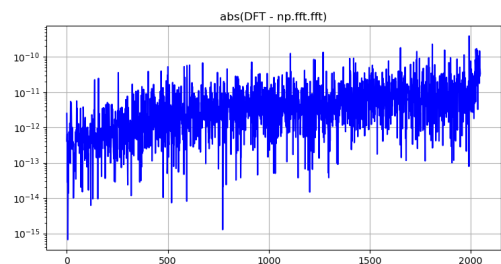
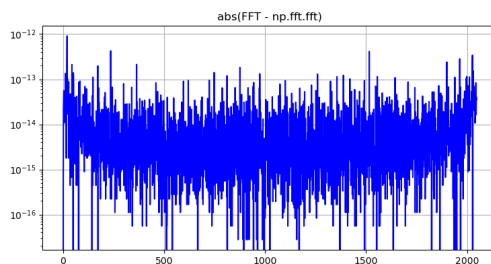
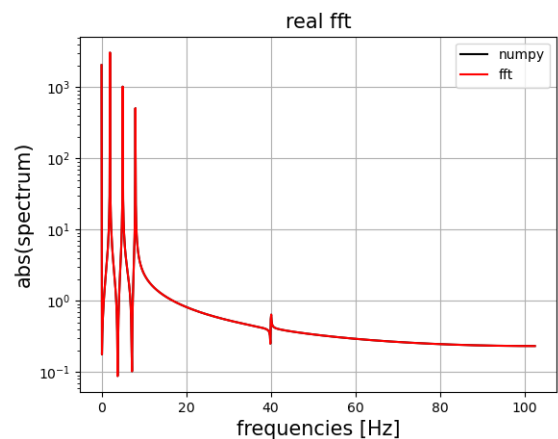
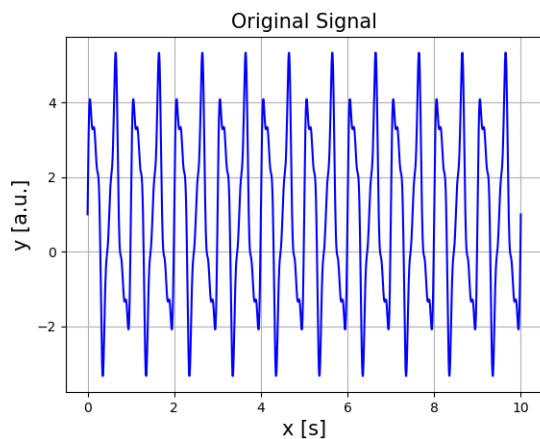
```

```

20     real : bool
21         false for fft
22         true for rfft
23
24     Returns
25     -----
26     f: 1d array
27         Array of length n containing the sample frequencies.
28     '''
29     if not real:
30         if n%2 == 0:
31             f1 = np.array([i for i in range(0, n//2)])
32             f2 = np.array([i for i in range(-n//2, 0)])
33             return np.concatenate((f1, f2))/(d*n)
34         else :
35             f1 = np.array([i for i in range((n-1)//2 + 1)])
36             f2 = np.array([i for i in range(-(n-1)//2, 0)])
37             return np.concatenate((f1, f2))/(d*n)
38     if real:
39         if n%2 == 0:
40             f1 = np.array([i for i in range(0, n//2 + 1)])
41             return f1 / (d*n)
42         else :
43             f1 = np.array([i for i in range((n-1)//2 + 1)])
44             return f1 / (d*n)

```

Fatto ciò possiamo chiamare le nostre funzioni e vedere i risultati. La restante parte del codice non verrà mostrata perché si tratta solo di plot, il codice intero è comunque disponibile. Tutti i codici sopra sono pezzi di un unico grande codice.



H 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.

H.1 Fit con scipy

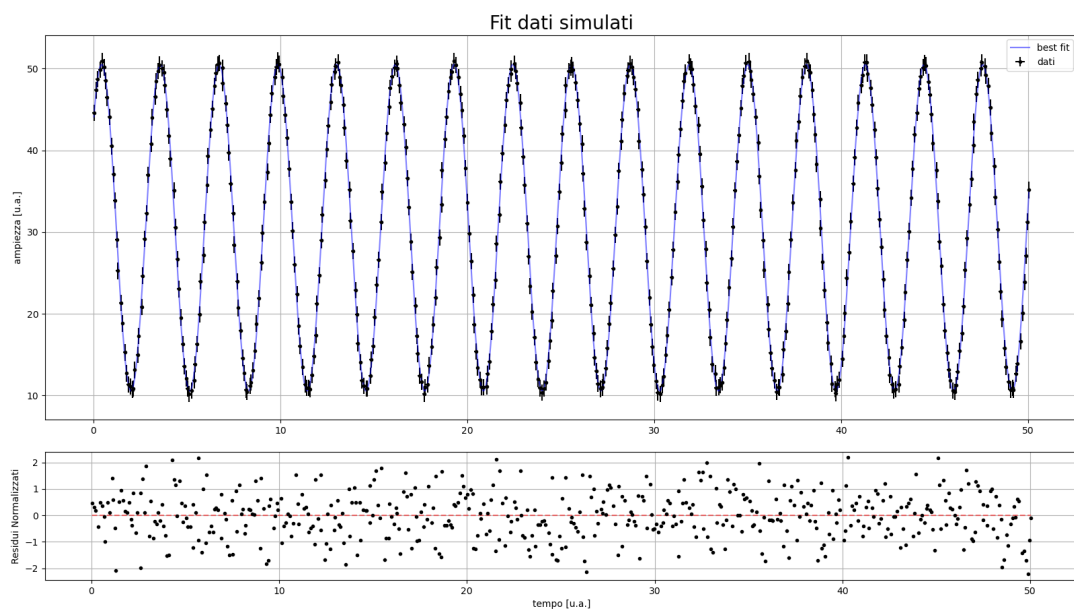
La libreria scipy grazie alla funzione "curve_fit()" ci permette di eseguire un gran numero di fit; riportiamo sotto un esempio di codice e relativi risultati e grafico:

```
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. ]]

```



H.2 Fit circolare, metodo di Coope

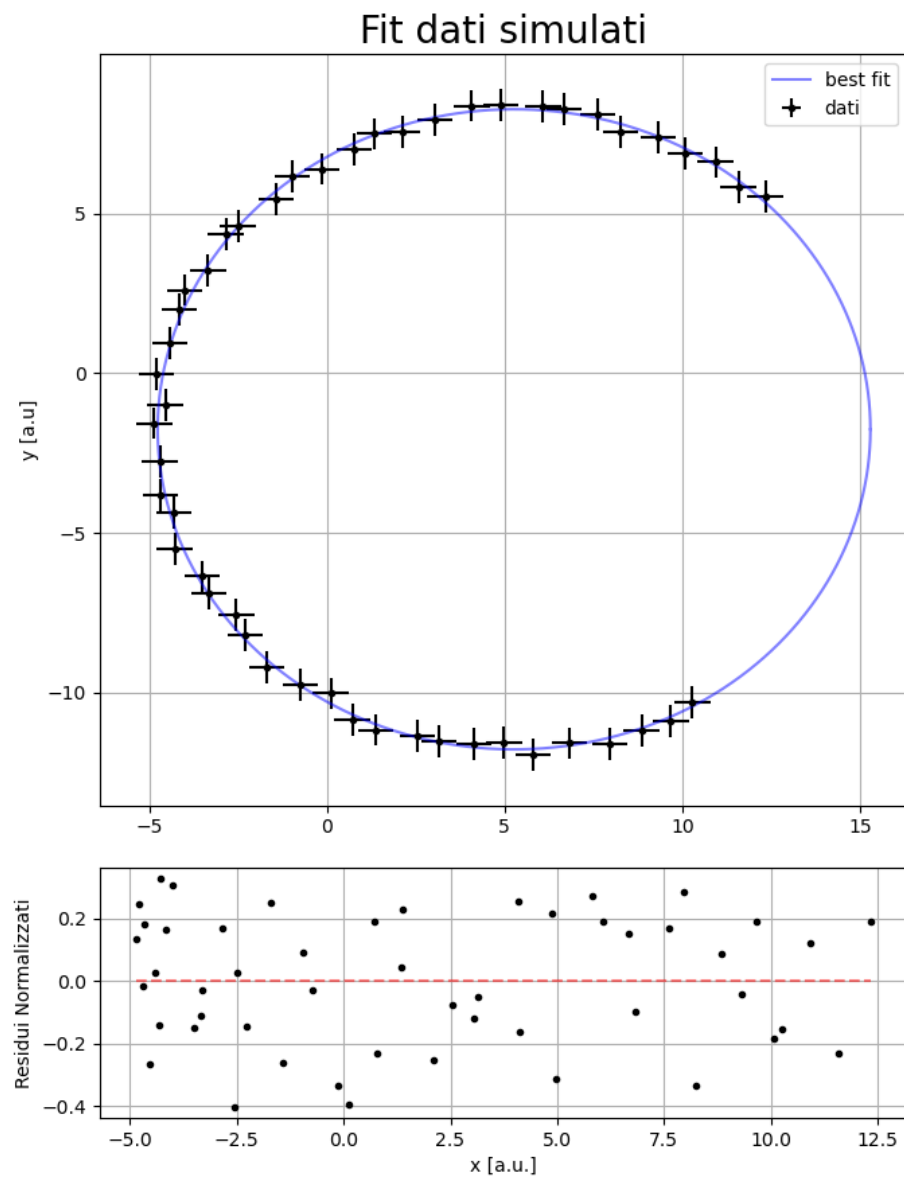
Ci sono casi in cui, come per un circonferenza o un'ellisse, curve fit non è comodo da usare, in quanto non si tratta di vere e proprie funzioni. Mostriamo un esempio di fit circolare seguito con il metodo di Coope e riportiamo qui il link all'articolo originale: <https://core.ac.uk/download/pdf/35472611.pdf>

```
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, w=None):
20     """
21     fit di un cerchio con metodo di coope
22     Parameters
23     -----
24     pt : 2Darray
25         contiene le coordinate del cerchio
26     w : None or 1Darray
27         w = np.sqrt(dx**2 + dy**2)
28         if None => w = np.ones(len(pt[0]))
29
30
31     Returns
32     -----
33     c : 1Darray
34         array con le coordinate del centro del cerchio
35     r : float
36         raggio del cerchio
37     d : 1Darray
38         array con gli errori associati a c ed r
39     A1 : 2Darray
40         matrice di covarianza
41     """
42     npt = len(pt[0])
43
44     S = np.column_stack((pt.T, np.ones(npt)))
45     y = (pt**2).sum(axis=0)
46
47     if w is None:
48         w = np.ones(npt)
49
50     w = np.diag(1/w)
51
52     A = S.T @ w @ S #@ -> prodotto matriciale
53     b = S.T @ w @ y
54     sol = np.linalg.solve(A, b)
55
56     c = 0.5*sol[:-1]
57     r = np.sqrt(sol[-1] + c.T @ c)
58
59     d = np.zeros(3)
60     A1 = np.linalg.inv(A)
61
62     for i in range(3):
63         d[i] = np.sqrt(A1[i,i])
64     return c, r, d, A1
65
66
67 if __name__ == "__main__":
68     np.random.seed(69420)
69     #numero di punti
70     N = 50
71     #parametri cerchio
```

```

72 xc, yc, r1 = 5, -2, 10
73 #errori
74 ex, ey = 0.5, 0.5
75 dy = np.array(N*[ey])
76 dx = np.array(N*[ex])
77 dr = np.sqrt(dx**2 + dy**2)
78 k = np.random.uniform(0, ex, N)
79 l = np.random.uniform(0, ey, N)
80 #creiamo il cerchio
81 x, y = cerchio(xc, yc, r1, N, np.pi/4, 5/3*np.pi)
82 x = x + k #aggiungo errore
83 y = y + l
84
85 a = np.array([x, y])
86 c, r, d, A = fitcerchio(a, dr) #fit
87
88 print(f'x_c = {c[0]:.5f} +- {d[0]:.5f}; valore esatto = {xc:.5f}')
89 print(f'y_c = {c[1]:.5f} +- {d[1]:.5f}; valore esatto = {yc:.5f}')
90 print(f'r = {r:.5f} +- {d[2]:.5f}; valore esatto = {r1:.5f}')
91
92
93 chisq = sum(((np.sqrt((x-c[0])**2 + (y-c[1])**2) - r)/dr)**2.)
94 ndof = N - 3
95 print(f'chi quadro = {chisq:.3f} ({ndof:d} dof)')
96
97 corr=np.zeros((3,3))
98 for i in range(0, 3):
99     for j in range(0, 3):
100         corr[i][j]=(A[i][j])/(np.sqrt(A.diagonal()[i])*np.sqrt(A.diagonal()[j]))
101 print(corr)
102
103 #plot
104 fig1 = plt.figure(1, figsize=(7.5,9.3))
105 frame1=fig1.add_axes((.1,.35,.8,.6))
106 #frame1=fig1.add_axes((trasla lateralmente, trasla verticalmente, larghezza, altezza))
107 frame1.set_title('Fit dati simulati',fontsize=20)
108 plt.ylabel('y [a.u]',fontsize=10)
109 plt.grid()
110
111 plt.errorbar(x, y, dy, dx, fmt='.', color='black', label='dati')
112 xx, yy = cerchio(c[0], c[1], r, 10000)
113 plt.plot(xx, yy, color='blue', alpha=0.5, label='best fit')
114 plt.legend(loc='best')
115
116
117 frame2=fig1.add_axes((.1,.1,.8,.2))
118 frame2.set_ylabel('Residui Normalizzati')
119 plt.xlabel('x [a.u.]',fontsize=10)
120
121 ff=(np.sqrt((x-c[0])**2 + (y-c[1])**2) - r)/dr
122 x1=np.linspace(np.min(x),np.max(x), 1000)
123 plt.plot(x1, 0*x1, color='red', linestyle='--', alpha=0.5)
124 plt.plot(x, ff, '.', color='black')
125 plt.grid()
126
127 plt.show()
128
129 [Output]
130 x_c = 5.26652 +- 0.02239; valore esatto = 5.00000
131 y_c = -1.75547 +- 0.01536; valore esatto = -2.00000
132 r = 10.02356 +- 0.12864; valore esatto = 10.00000
133 chi quadro = 2.125 (47 dof)
134 [[ 1. -0.09873297 -0.3480512 ]
135 [-0.09873297 1. 0.18918522]
136 [-0.3480512 0.18918522 1. ]]

```

H.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://autotracer.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     Converta i coefficienti di:  $ax^2 + bxy + cy^2 + dx + ey + g = 0$ 
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     e = 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         phi = np.arctan2(2*b, a - c)
```

```

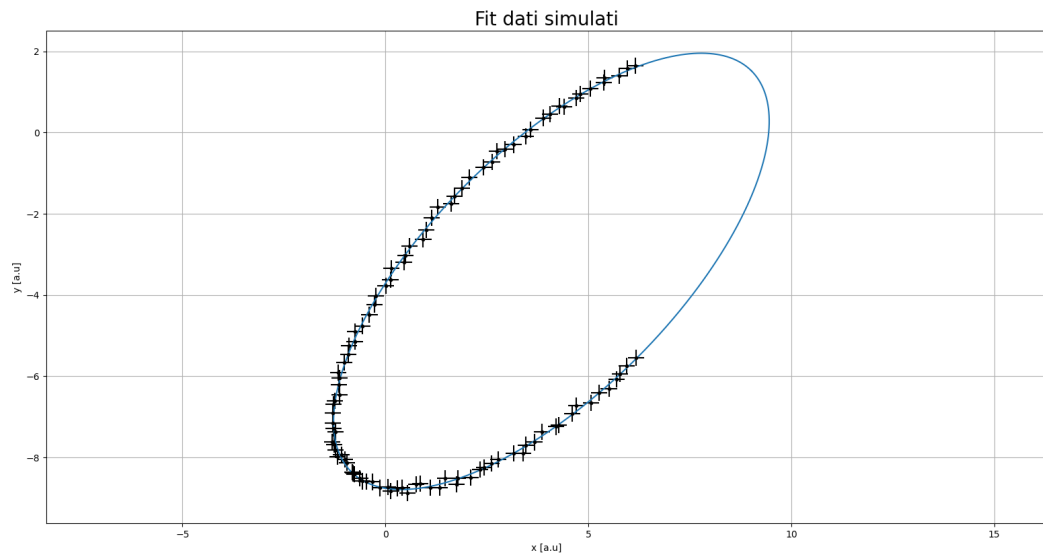
72     phi = np.arctan((2*b)/(a - c))/2
73     if a > c:
74         phi += np.pi/2
75
76     if not M_gt_m :
77         phi += np.pi/2
78
79     #periodicita' della rotazione
80     phi = phi % np.pi
81
82     return x0, y0, semi_M, semi_m, e, phi
83
84
85 def fit_ellisse(x, y):
86     """
87     Basato 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

```



I Autovalori e autovettori

Uno dei problemi che spesso capita di dover affrontare in fisica è di dover diagonalizzare una matrice cioè trovare λ e \mathbf{v} tale che:

$$A\mathbf{v} = \lambda\mathbf{v} \quad . \quad (43)$$

Il più semplice metodo da poter implementare è il metodo delle potenze, il quale è un algoritmo iterativo la cui regola di iterazione è:

$$\mathbf{v}_{k+1} = \frac{A\mathbf{v}_k}{\|A\mathbf{v}_k\|} = \frac{A^k\mathbf{v}_0}{\|A^k\mathbf{v}_0\|} \quad . \quad (44)$$

L'idea è abbastanza semplice, se A è diagonalizzabile allora possiamo decomporre un vettore generico nella base degli autovettori di A :

$$\mathbf{v}_0 = \alpha_1 v_1 + \dots + \alpha_n v_n \quad , \quad (45)$$

Quindi applicando la potenza ennesima di A a \mathbf{v}_0 si ottiene:

$$A^k\mathbf{v}_0 = \alpha_1 A^k v_1 + \dots + \alpha_n A^k v_n \quad (46)$$

$$= \alpha_1 \lambda_1^k v_1 + \dots + \alpha_n \lambda_n^k v_n \quad (47)$$

$$= \alpha_1 \lambda_1^k \left(v_1 + \dots + \frac{c_n}{c_1} \left(\frac{\lambda_n}{\lambda_1} \right)^k v_n \right) \quad . \quad (48)$$

Quindi se gli autovalori sono tutti diversi e sono ordinati in ordine decrescente $\lambda_1 > \dots > \lambda_n$ allora tutti i termini dentro la parentesi tendono a zero per k che va all'infinito in quanto minori di uno. Questo metodo converge all'autovalore maggiore della matrice. Se volessimo trovarli tutti possiamo sempre usare questo metodo ma integrarlo con un metodo di ortogonalizzazione ad esempio Gram-Schmidt che chiamiamo ad ogni passo. Come criteri di convergenza possiamo mettere o la distanza tra iterazione successive dell'autovettore oppure la radice del valore assoluto della differenza tra iterazione successive dell'autovalore. Usiamo la radice perché la convergenza è quadratica negli autovalori:

$$\begin{aligned} R_{1,2} &= \|\mathbf{v}_{k+1} \pm \mathbf{v}_k\| \\ R_3 &= \sqrt{|\lambda_{k+1} - \lambda_k|} \quad , \end{aligned} \quad (49)$$

dove il \pm deriva dal fatto che sia $+\mathbf{v}$ che $-\mathbf{v}$ sono autovettori.

Capita sovente però che magari non si sia interessati a tutti gli autovalori, magari solo ai più grandi o a i più piccoli. Per i più grandi possiamo applicare l'algoritmo così come lo abbiamo descritto. Ma se fossimo interessati ai più piccoli? Per questo secondo basta sostituire A con la sua inversa in quanto gli autovalori di A^{-1} sono il reciproco degli autovalori di A quindi il più grande autovalore di A^{-1} sarà il più piccolo di A , proprio come volevamo; questo si chiama metodo delle potenze inverso. Vediamo ora il codice:

```
1 import numpy as np
2
3 def eig(M, k=None, tol=1e-10, magnitude='small'):
4     '''
5     Compute the eigenvalue decomposition
6     of the symmetric matrix A using power iteration
7     or inverse iteration.
8     Inverse iteration is the same of power iteration
9     but we use M^-1 instead of M so the eigenvalues
10    are the reciprocal.
11
12    Parameters
13    -----
14    M : 2darray
15        N x N matrix, symmetric
16    k : None or int, if None k=N
17        number of eigenvalues and eigenvectors to find,
18        if k<N then k eigenvectors corresponding to the k
19        largest eigenvalues will be found
20    tol : float, optional default 1e-10
21        required tolerance
22    magnitude : string, optional, default small
23        if magnitude == 'small' the smallest eigenvalues
24        and their relative eigenvectors will be computed
25        if magnitude == 'big' the biggest eigenvalues
26        and their relative eigenvectors will be computed
27
28    Return
29    -----
```

```

30 eigval : 1darray
31     array of eigenvalues
32 eigvec : 2darray
33     kxk matrix, the column eigvec[:, i] is the
34     ormalized eigenvector corresponding to the
35     eigenvalue eigval[i]
36 counts : 1darray
37     how many iteration are made for each eigenvector
38 '''
39
40 if magnitude == 'small':
41     A = np.copy(np.linalg.inv(M))
42 if magnitude == 'big':
43     A = np.copy(M)
44
45 N = A.shape[0]
46 if k is None:
47     k = N
48
49 eigvec = [] # will contain the eignvectors
50 eigval = [] # will contain the eignvalues
51 counts = [] # will contain the numero of iteration of each eigenvalue
52
53 for _ in range(k):
54
55     v_p = np.random.randn(N) #initial vector
56     v_p = v_p / np.sqrt(sum(v_p**2))
57     l_v = np.random.random()
58     Iter= 0
59
60     while True:
61         l_o = l_v
62         v_o = v_p # update vector
63         v_p = np.dot(A, v_p) # compute new vector
64         v_p /= np.sqrt(sum(v_p**2)) # normalization
65         v_p = gs(v_p, eigvec) # orthogonalization respect
66         # all eigenvectors find previously
67         #eigenvalue of v_p, A @ v_p = l_v * v_p
68         #multiplying by the transposed => (A @ v_p) @ v_p.T = l_v
69         #using v_p @ v_p.T = 1
70         l_v = np.dot(np.dot(A, v_p), v_p)
71
72         R1 = np.sqrt(sum((v_p - v_o)**2))
73         R2 = np.sqrt(sum((v_o + v_p)**2))
74         R3 = np.sqrt(abs(l_v - l_o)) # In eigenvalues the convergence is quadratic
75
76         Iter += 1
77         if R1 < tol or R2 < tol or R3 < tol:
78             break
79
80     eigvec.append(v_p)
81     eigval.append(l_v)
82     counts.append(Iter)
83
84 if magnitude == 'small':
85     eigval = 1/np.array(eigval)
86     eigvec = np.array(eigvec).T
87 if magnitude == 'big':
88     eigvec = np.array(eigvec).T
89     eigval = np.array(eigval)
90
91 return eigvec, eigval, counts
92
93 def gs(v, eigvec):
94     '''
95     Gram-Schmidt process for orthogonalization
96
97     Parameters
98     -----
99
100    v : 1darray
101        vector to be orthogonalized
102    eigvec : list
103        list of normalized eigenvectors, vectors
104        with respect to which to orthogonalize v
105

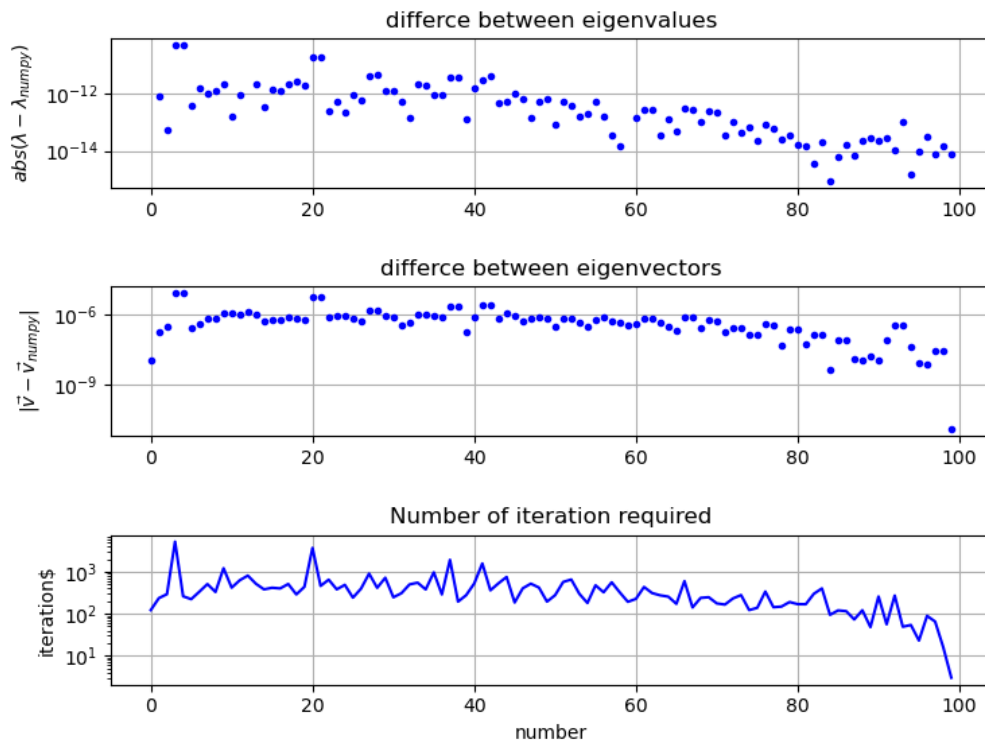
```

```

106 Return
107 -----
108 v : 1darray
109     vector orthogonal to the eigenvector set
110     ,,,
111
112 for i in range(len(eigvec)):
113     v = v - np.dot(eigvec[i], v) * eigvec[i]
114 return v

```

Vediamo ora i risultati due test del nostro algoritmo. Cominciamo generando una matrice random grazie a numpy e per averla simmetrica consideriamone il prodotto per se stessa trasposta: "P = np.random.normal(size=[n, n]) H = np.dot(P.T, P)"; quindi diagonalizziamo, prendiamo $n = 100$, H settando "magnitude='big'". Mostriamo solo il risultato, il codice lo troverete scritto nella cartella, confrontando il risultato con la diagonalizzazione fatta da numpy:



Dal punto di vista del tempo chiaramente numpy nemmeno fa fatica: 9.13726 secondi noi contro 0.0090 di numpy. Comunque il risultato è soddisfacente.

Vediamo ora un test un po' più fisico dove siamo interessati agli autovalori più piccoli. Consideriamo una matrice 'a bischero' fatta del tipo:

$$H = -\frac{1}{2h^2} \begin{pmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & 1 & -2 & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{pmatrix} + \begin{pmatrix} V(x_1) & 0 & 0 & \cdots & 0 \\ 0 & V(x_2) & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & 0 & V(x_{n-1}) & 0 \\ 0 & \cdots & 0 & 0 & V(x_n) \end{pmatrix} \quad (50)$$

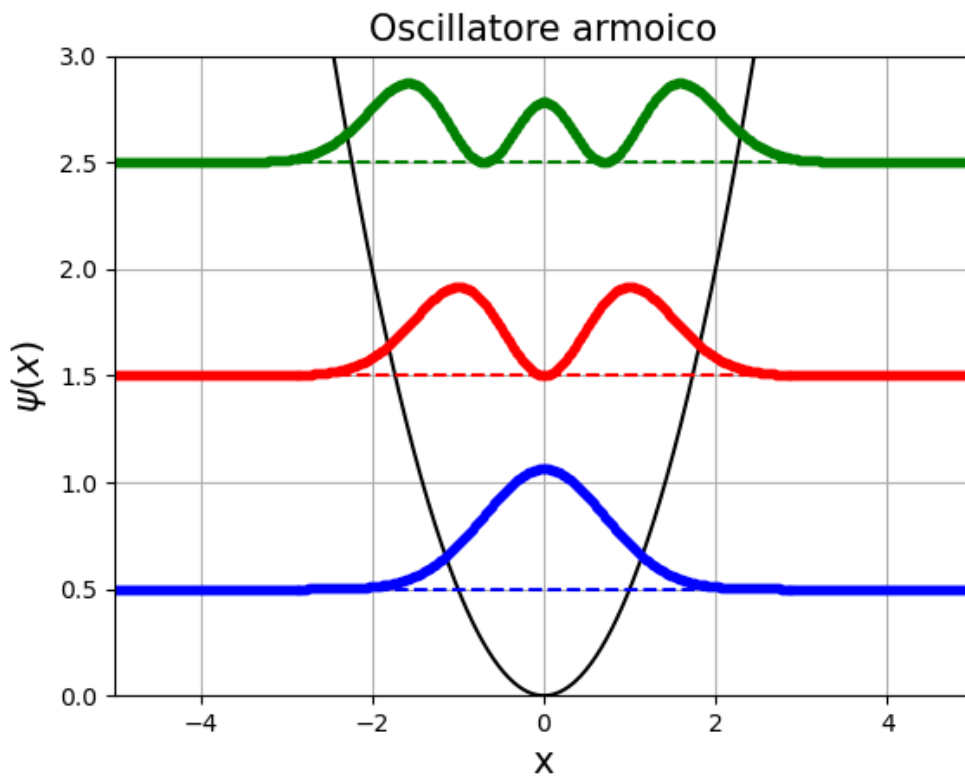
dove h è la spaziatura tra x_i e x_{i+1} il quale è un array in un certo range in un certo numero di punti. Probabilmente avrete notato che la matrice scritta sopra non è altro che la discretizzazione dell'equazione di Schrödinger :

$$H\psi = \left(\frac{1}{2} \frac{\partial^2}{\partial x^2} + V(x) \right) \psi = E\psi \quad (51)$$

Qui chiaramente siamo interessati solo ai livelli energetici minori in quanto l'approssimazione del laplaciano che abbiamo fatto peggiora man mano che gli stati sono sempre più estesi, discretizzare così infatti significa mettersi dentro una scatola di lato fissato ma chiaramente noi vorremmo La scatola infinita e per i livelli più

bassi l'approssimazione è buona. Per semplicità consideriamo il caso dell'oscillatore armonico $V(x) = x^2/2$ e troviamo i dieci autovalori più bassi ($E = n + 1/2$). Grafichiamo anche poi per bellezza tre autovettori, o autofunzioni, con il caveat che ogni autovettore va diviso per \sqrt{h} . Come sempre il codice è già scritto e lo trovate tutto insieme, noi qui mostriamo solo i risultati:

teorico	calcolato	errore
0.5	0.50049	4.88e-04
1.5	1.50144	1.44e-03
2.5	2.50234	2.34e-03
3.5	3.50319	3.19e-03
4.5	4.50399	3.99e-03
5.5	5.50474	4.74e-03
6.5	6.50544	5.44e-03
7.5	7.50609	6.09e-03
8.5	8.50669	6.69e-03
9.5	9.50724	7.24e-03



Per un totale di tempo di esecuzione di 0.84998 secondi.

J 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).

J.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.

J.2 Calcolo di Pi greco

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

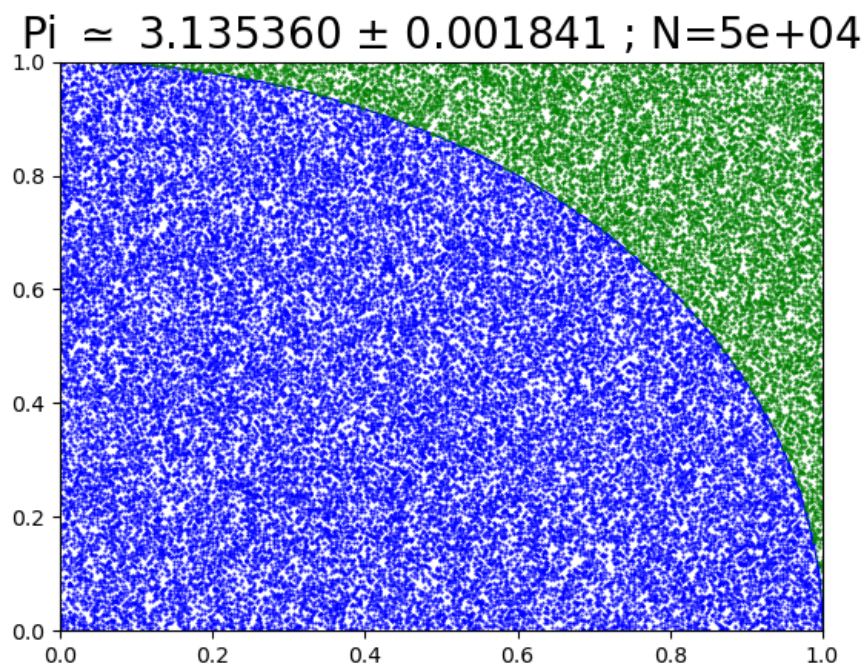
```

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 $\pm$ $ %f ; 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.



K Propagazione errori

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

K.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)
```

K.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:

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

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

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

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

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

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

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

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

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

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

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

L Interpolazione

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

L.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:

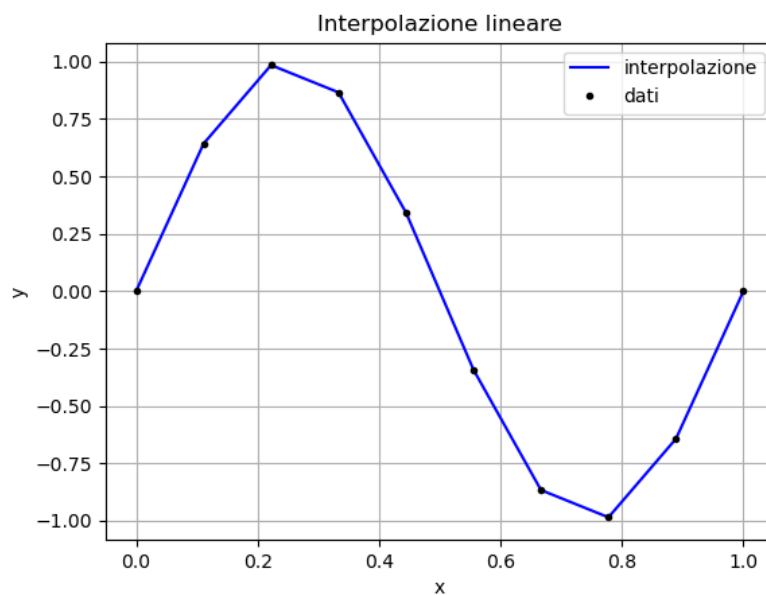
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def f(x, xx, yy):
5     """
6     restituisce l'interpolazione dei punti xx yy
7     x puo' essere un singolo valore in cui calcolare
8     la funzione interpolante o un intero array
9     """
10    #proviamo se x e' un array
11    try :
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:



L.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:

```

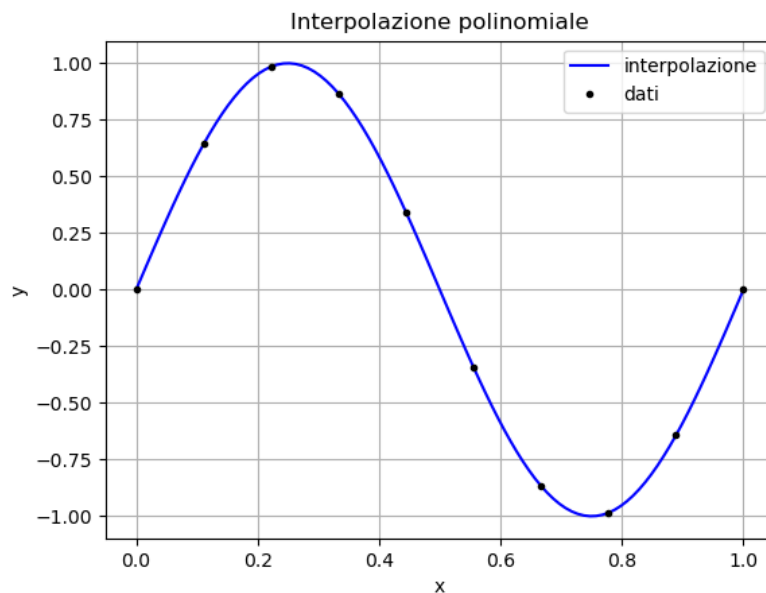
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 N = 10
5 x = np.linspace(0, 1, N)
6 y = np.sin(2*np.pi*x)
7
8 #Matrice di Vandermonde
9 A = np.zeros((N, N))
10 A[:,0] = 1
11 for i in range(1, N):
12     A[:,i] = x**i

```

```

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()

```



L.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')

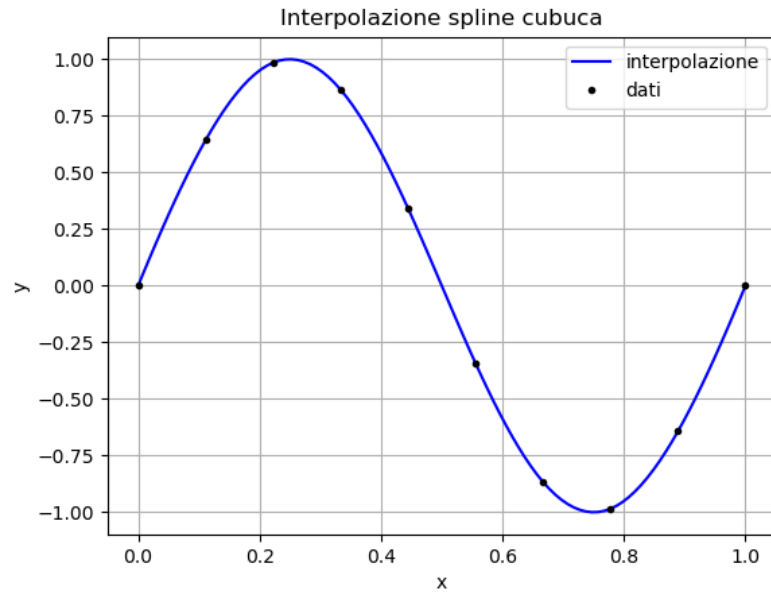
```



```

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()

```



M Programmazione a oggetti

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

```
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:

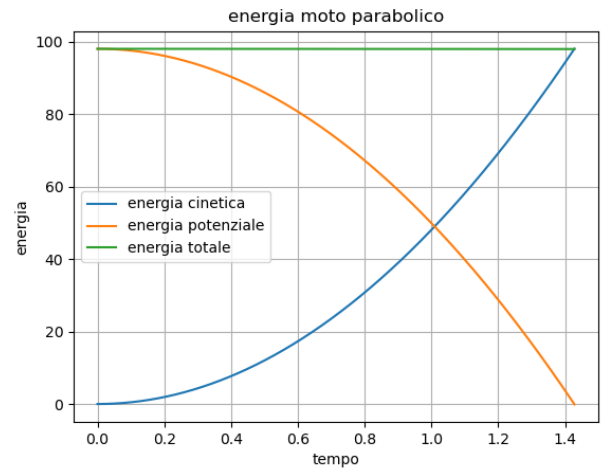
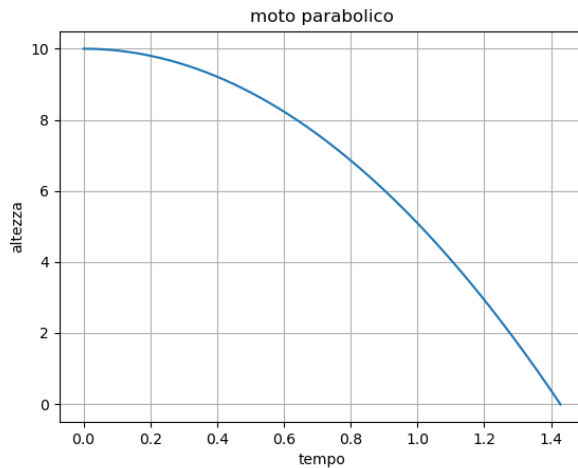
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 class Pallina:
5     """
6     Classe che rappresenta una pallina
7     intesa come oggetto puntiforme
8     """
9
10    def __init__(self, x, y, vx, vy, m):
11        """
12        costruttore della classe, verra' chiamato
13        quando 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.



M.1 Più che una funzione

Le classi sono molto utili perchè ci permettono di poter fare più cose che una funzione, ma possono anche essere chiamate come una funzione; ad esempio grazie al metodo "`__call__`" l'istanza che creiamo sarà una funzione. Abbiamo visto prima un codice che esegue un'interpolazione lineare; se ci fermiamo a pensare un attimo quella funzione ogni volta che viene chiamata rifà tutto il conto. Con l'utilizzo di una classe possiamo calcolarci nel costruttore i coefficienti del polinomio e poi chiamare la funzione creata, la quale sa già i coefficienti del polinomio e quindi sarà effettivamente più veloce. Vediamo il caso di una spline cubica:

```
1 class CubicSpline:
2     '''
3     1-D interpolating natural cubic spline
4
5     Parameters
6     -----
7     xx : 1darray
8         value on x must be strictly increasing
9     yy : 1darray
10        value on y
11
12     Example
13     -----
14     >>>import numpy as np
15     >>>import matplotlib.pyplot as plt
16     >>>x = np.linspace(0, 1, 10)
17     >>>y = np.sin(2*np.pi*x)
18     >>>F = CubicSpline(x, y)
19     >>>print(F(0.2))
20     0.9508316728694627
21
22     >>>z = np.linspace(0, 1, 100)
23     >>>plt.figure(1)
24     >>>plt.title('Spline interpolation')
25     >>>plt.xlabel('x')
26     >>>plt.ylabel('y')
27     >>>plt.plot(z, F(z), 'b', label='Cubic')
28     >>>plt.plot(x, y, marker='.', linestyle='', c='k', label='data')
29     >>>plt.legend(loc='best')
30     >>>plt.grid()
31     >>>plt.show()
32     '''
33
34     def __init__(self, xx, yy):
35
36         self.x = xx                # x data
37         self.y = yy                # y data will be the constant of polynomial
38         self.N = len(xx)           # len of data
39         alpha = np.zeros(self.N-1) # auxiliar array
40         self.b = np.zeros(self.N-1) # linear term
41         self.c = np.zeros(self.N)   # quadratic term
42         self.d = np.zeros(self.N-1) # cubic term
43         l = np.zeros(self.N)        # auxiliar array
44         z = np.zeros(self.N)        # auxiliar array
45         mu = np.zeros(self.N)       # auxiliar array
```

```

46
47     if not np.all(np.diff(xx) > 0.0):
48         raise ValueError('x must be strictly increasing')
49
50     dx = xx[1:] - xx[:-1]
51     a = yy
52
53     for i in range(1, self.N-1):
54         alpha[i] = 3*(a[i+1] - a[i])/dx[i] - 3*(a[i] - a[i-1])/dx[i-1]
55
56     l[0] = 1.0
57     z[0] = 0.0
58     mu[0] = 0.0
59
60     for i in range(1, self.N-1):
61         l[i] = 2.0*(xx[i+1] - xx[i-1]) - dx[i-1]*mu[i-1]
62         mu[i] = dx[i]/l[i]
63         z[i] = (alpha[i] - dx[i-1]*z[i-1])/l[i]
64
65     l[self.N-1] = 1.0
66     z[self.N-1] = 0.0
67     self.c[self.N-1] = 0.0
68
69     #Coefficient's computation
70     for i in range(self.N-2, -1, -1):
71
72         self.c[i] = z[i] - mu[i]*self.c[i+1]
73         self.b[i] = (a[i+1] - a[i])/dx[i] - dx[i]*(self.c[i+1] + 2.0*self.c[i])/3.0
74         self.d[i] = (self.c[i+1] - self.c[i])/(3.0*dx[i])
75
76
77     def __call__(self, x):
78         '''
79         x : float or 1darray
80             when we want compute the function
81         '''
82         n = self.check(x)
83
84         if n == 1 :
85
86             for j in range(self.N-1):
87                 if self.x[j] <= x <= self.x[j+1]:
88                     i = j
89                     break
90
91             q = (x - self.x[i])
92             return self.d[j]*q**3.0 + self.c[j]*q**2.0 + self.b[j]*q + self.y[j]
93
94         else:
95             F = np.zeros(n)
96             for k, x1 in enumerate(x):
97
98                 for j in range(len(self.x)-1):
99                     if self.x[j] <= x1 <= self.x[j+1]:
100                         i = j
101                         break
102
103                 q = (x1 - self.x[i])
104                 F[k] = self.d[j]*q**3.0 + self.c[j]*q**2.0 + self.b[j]*q + self.y[j]
105
106             return F
107
108
109     def check(self, x):
110         try :
111             n = len(x)
112             x_in = np.min(self.x) <= np.min(x) and np.max(self.x) >= np.max(x)
113         except TypeError:
114             n = 1
115             x_in = np.min(self.x) <= x <= np.max(self.x)
116
117         # if the value is not in the correct range it is impossible to count
118         if not x_in :
119             errore = 'Value out of range'
120             raise Exception(errore)
121

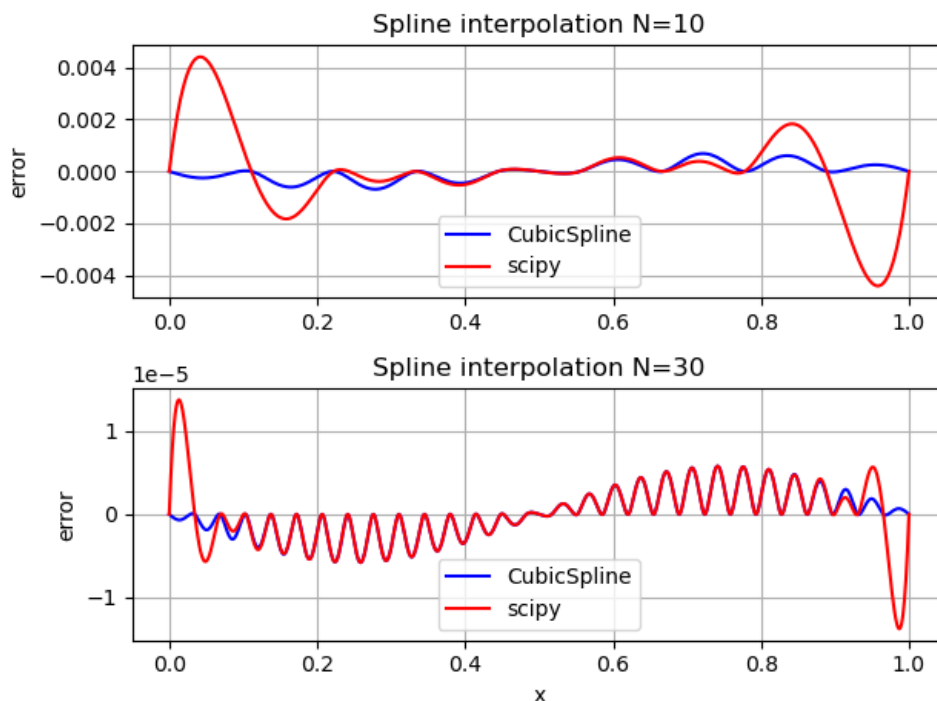
```

```

122         return n
123 if __name__ == '__main__':
124
125     x = np.linspace(0, 1, 10)
126     y = np.sin(2*np.pi*x)
127
128     z = np.linspace(0, 1, 1000)
129     G = CubicSpline(x, y)
130
131     print(G(0.2))
132
133     plt.figure(1)
134     plt.title('Spline interpolation')
135     plt.xlabel('x')
136     plt.ylabel('y')
137     plt.plot(z, G(z), 'r', label='Cubic')
138     plt.plot(x, y, marker='.', linestyle='', c='k', label='data')
139     plt.legend(loc='best')
140     plt.grid()
141     plt.show()
142
143 [Output]
144 0.9508316728694627

```

Fondamentalmente che succede: noi creiamo G, istanza della classe CubicSpline, nel farlo chiamiamo il costruttore il quale calcola i coefficienti dei polinomi interpolanti. Ora però grazie al metodo “__call__” G è un oggetto chiamabile. Avete presente quando in altri codici passando funzioni ad altre funzioni nella documentazione scrivevamo callable? Ecco è proprio questo, è come se la vostra funzione fosse il metodo “__call__” della classe. Quindi chiamando l’istanza viene eseguito il metodo “__call__” il quale, nel nostro caso, sa già i dati necessari e sa calcolarci la spline. Giusto per completezza precisiamo che questa spline cubica non è esattamente la stessa spline cubica di scipy che abbiamo visto sopra; questa si chiama spline naturale e ai bordi si comporta un pò diversamente rispetto a scipy (meglio anche). Per vederlo calcoliamo la differenza fra l’interpolazione calcolata in z (l’array nel codice) e i valori che restituisce “np.sin(2*np.pi*z)”; per brevità riportiamo solo il grafico, certo che possiate scrivervi da voi questo piccolo codice di confronto. Facciamo due casi, interpoliamo prima 10 e poi 30 punti:



Il codice sopra mostrato è riportato nella cartella interpolazioni, benché mostrato in questa sezione. Inoltre è riportato la stessa implementazione per il caso lineare.

N 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.

N.1 Processo di Ornstein–Uhlenbeck

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

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

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

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

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

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

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

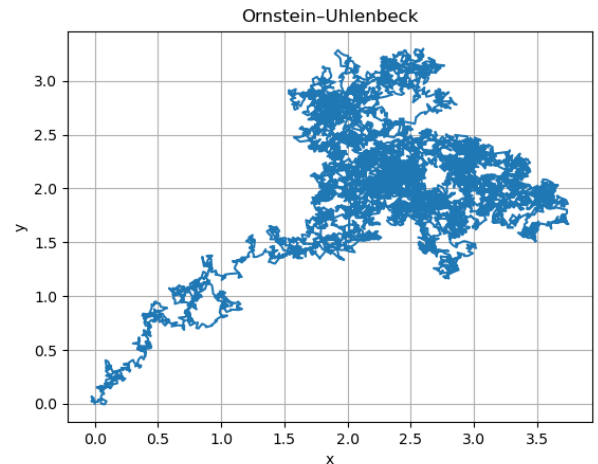
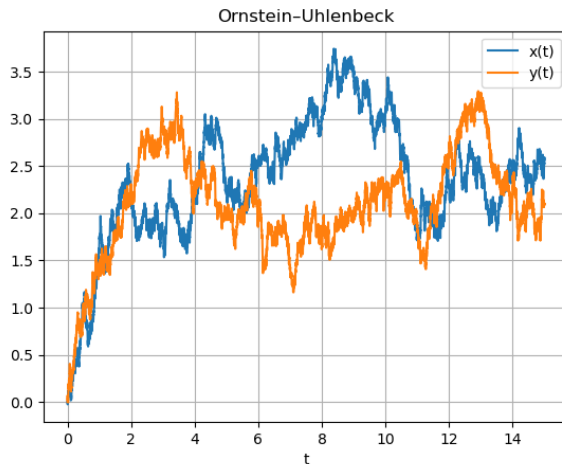
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def f(z):
5     """
6     funzione che moltiplica il dt
7     """
8     theta = 0.7
9     mu = 2.5
10    return theta * (mu - z)
11
12 def g():
13    """
14    funzione che moltiplica il processo di wiener
15    """
16    sigma = 0.6
17    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()

```



N.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:

```

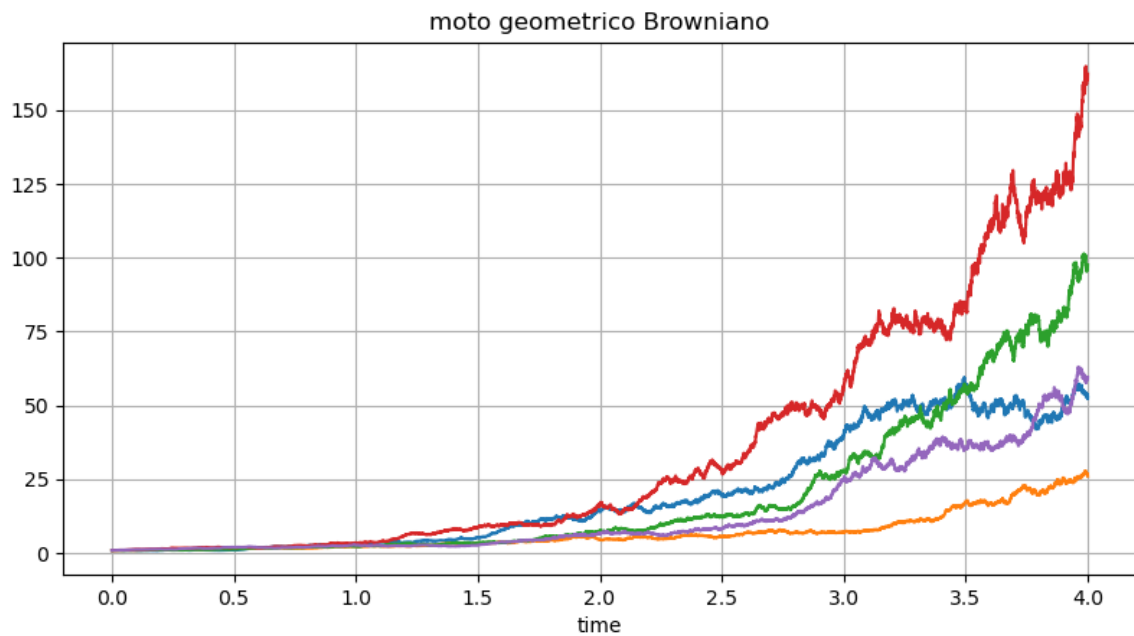
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # geometric Brownian motion
5 # Heun method
6
7 def f(z):
8     """
9     funzione che moltiplica il dt
10    """
11    mu = 1
12    return mu*z
13
14 def g(z):
15     """
16     funzione che moltiplica il processo di wiener
17    """
18    sigma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

```



O Ottimizzazione

Per quanto la discussione fatta nella quarta lezione con `curve_fit` sia appunto di ottimizzazione, parliamo brevemente degli algoritmi di ottimizzazione. Tratteremo del gradiente discendente e di una sua modifica spiegando al computer che esiste il principio di inerzia. Sia $F(x) : \mathbb{R}^n \rightarrow \mathbb{R}$ la funzione da minimizzare.

O.1 Discesa del gradiente

La regola del gradiente discendente ci fa aggiornare iterativamente il punto di minimo con:

$$x_{n+1} = x_n - \alpha_n \nabla F(x_n) \quad (52)$$

Ci sono vari modi per scegliere α_n noi lo supporremo costante, scelta non delle migliori, perché può allungare la convergenza del metodo. Ricordiamo inoltre che i metodi iterativi convergono solo a minimi locali quindi bisogna scegliere attentamente il punto iniziale.

```
1 def grad_disc(f, x0, tol, step):
2     """
3     implementation of gradient descent
4     you have to be careful about the values
5     you pass in x0 if the function has more minima
6     and also the value of steps is a delicate choice
7     to be made wisely
8
9     Parameters
10    -----
11    f : callable
12        function to find the minimum,
13        can be f(x), f(x,y) and so on
14    x0 : ndarray
15        initial guess, to choose carefully
16    tol : float
17        required tollerance
18        the function stops when all components
19        of the gradient have smaller than tol
20    step : float
21        size of step to do, to choose carefully
22
23    Returns
24    -----
25    X : ndarray
26        array with all steps of solution
27    iter : int
28        number of iteration
29    """
30    iter = 0                #initialize iteration counter
31    h = 1e-7               #increment for derivatives
32    X = []                 #to store solution
33    M = len(x0)            #number of variable
34    s = np.zeros(M)        #auxiliary array for derivatives
35    grad = np.zeros(M)     #gradient
36
37    while True:
38        #gradient computation
39        for i in range(M):
40            s[i] = 1        #loop over variables
41                            #we select one variable at a time
42            dz1 = x0 + s*h  #step forward
43            dz2 = x0 - s*h  #step backward
44            grad[i] = (f(*dz1) - f(*dz2))/(2*h) #derivative along z's direction
45            s[:] = 0        #reset to select the other variables
46
47        if all(abs(grad) < tol):
48            break
49
50        x0 = x0 - step*grad #move towards the minimum
51        X.append(x0)        #store iteration
52        iter += 1           #update counter
53
54    X = np.array(X)
55    return X, iter
```

O.2 Principio di inerzia

Fondamentalmente se vediamo l'evoluzione della soluzione è come se una pallina stesse scendendo lungo una verde vallata, cioè la nostra F è il potenziale a cui la pallina è soggetta, quindi in analogia con la fisica si introduce il : gradiente discendente con momento. Cioè sia ggiunge un termine di velocità e un termine di attrito dipendente dalla velocità in modo che l'algoritmo non oscilli.

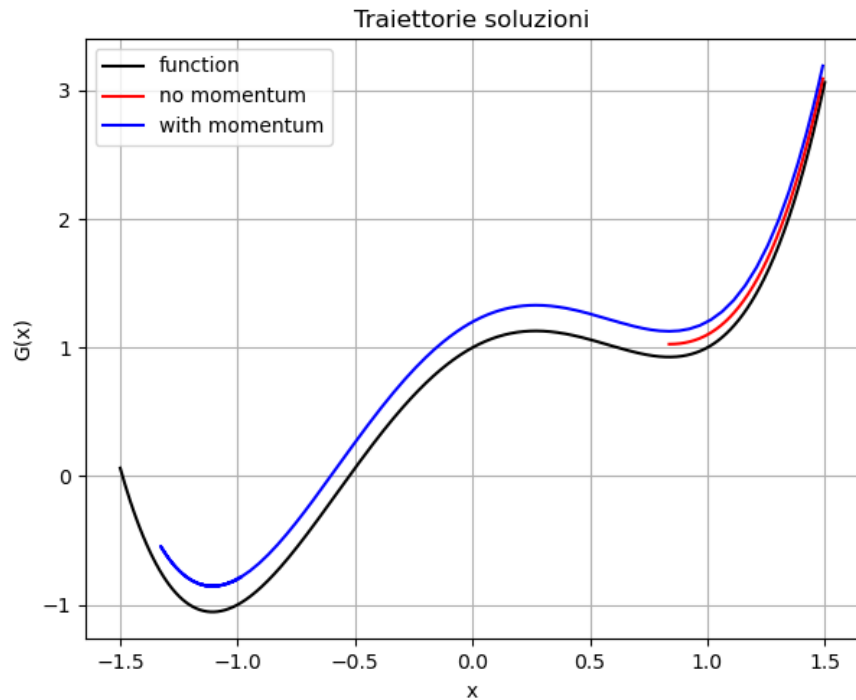
$$w_{n+1} = \beta w_n + \nabla F(x_n) \quad (53)$$

$$x_{n+1} = x_n - \alpha w_{n+1} \quad (54)$$

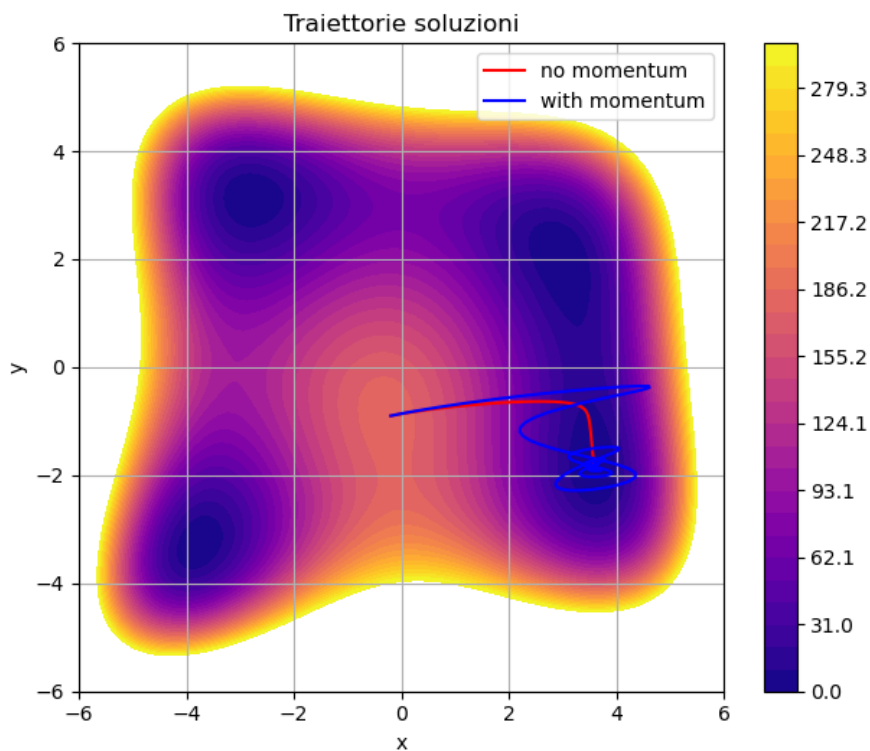
α è sempre lo step mentre β rappresenta pittoricamente il coefficiente di attrito. Se $\beta = 1$ l'algoritmo oscilla tra due punti con lo stesso valore di F (si conserva l'energia) quindi bisogna avere $\beta < 1$.

```
1 def grad_disc_m(f, x0, tol, alpha, beta):
2     """
3     implementation of gradient descent with momentum
4     you have to be careful about the values
5     you pass in x0 if the function has more minima
6     and also the value of alpha an beta is a
7     delicate choice to be made wisely
8
9     Parameters
10    -----
11    f : callable
12        function to find the minimum,
13        can be f(x), f(x,y) and so on
14    x0 : 1darray
15        initial guess, to choose carefully
16    tol : float
17        required tollerance
18        the function stops when all components
19        of the gradient have smaller than tol
20    alpha : float
21        size of step to do, to choose carefully
22    beta : float
23        size of step to do for velocity,
24        to choose carefully, if beta = 0
25        we get the method of gradient
26        descent without momentum
27
28    Returns
29    -----
30    X : ndarray
31        array with all steps of solution
32    iter : int
33        number of iteration
34    """
35    iter = 0                #initialize iteration counter
36    h = 1e-7               #increment for derivatives
37    X = []                 #to store solution
38    M = len(x0)            #number of variable
39    s = np.zeros(M)        #auxiliary array for derivatives
40    grad = np.zeros(M)     #gradient
41    w = np.zeros(M)       #velocity, momentum
42
43    while True:
44        #gradient computation
45        for i in range(M): #loop over variables
46            s[i] = 1       #we select one variable at a time
47            dz1 = x0 + s*h  #step forward
48            dz2 = x0 - s*h  #step backward
49            grad[i] = (f(*dz1) - f(*dz2))/(2*h) #derivative along z's direction
50            s[:] = 0       #reset to select the other variables
51
52        if all(abs(grad) < tol):
53            break
54
55        w = beta*w + grad  #update velocity
56        x0 = x0 - alpha*w  #update position move towards the minimum
57        X.append(x0)       #store iteration
58        iter += 1          #update counter
59
60    X = np.array(X)
61    return X, iter
```

Vediamo ora due grafici per capire il risultato, non mostriamo il codice in quanto si tratta semplicemente di chiamare le funzioni e fare i plot.



Le funzioni sono state chiamate con i parametri $\text{grad_disc}(G, x_0, 1e-8, 1e-3)$ e $\text{grad_disc_m}(G, x_0, 1e-8, 1e-3, 0.953)$ la funzione da minimizzare è $F(x) = (x^2 - 1)^2 + x$ vediamo che il primo data una infelice scelta del punto di partenza si incastra in un minimo locale. Il secondo invece arriva al minimo locale con un valore della velocità non nullo e quindi riesce a scavalcare il massimo locale se β fosse stato 0.9 anche lui si sarebbe bloccato lì. Le curve sono state traslate per maggiore leggibilità. Con gli stessi parametri vediamo un esempio bidimensionale (il codice è lo stesso la funzione è implementata in modo generico). $F(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$.



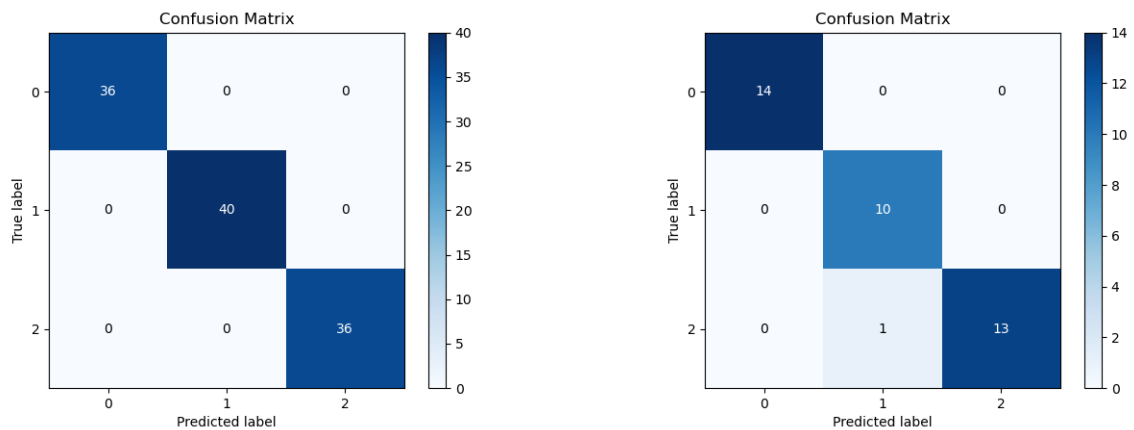
P Machine Learning

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.

P.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, cioè quello che il modello dovrebbe predire
16 y = iris_dataset.target
17
18 #divido i dati, una 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 una categoria
24 modello = DecisionTreeClassifier()
25
26 #addestramento del 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 può 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 di quanto è stato bravo il modello
45 #sulle y c'è 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.

P.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
```

P.3 Salvare il modello

Ora giustamente voi mi potreste obiettare: Sì tutto molto bello, ma se spengo il computer il modello muore e devo riaddestrarlo. Effettivamente avete ragione ma chiaramente ci sta una soluzione a tutto ciò, nessuno vuole perdere tempo a riaddestrare il modello ogni volta che accende il pc. Vedremo due comandi della libreria 'joblib' che ci permetteranno di salvare il modello e di caricarlo poi su un altro codice. Per fare un semplice esempio che possa scalare bene con i parametri, consideriamo un classificatore a cui diamo in input una matrice $N \times M$ contenete delle curve del tipo $y(x) = x^k$ dove $k \in [0, 1]$. Prendiamo per k ad esempio 5 valori, e calcoliamo M curve in totale ognuna lunga N . Per rendere le cose un po' più complicate per la macchina il range in cui calcolare le curve è scelto a caso. Vediamo ora il codice:

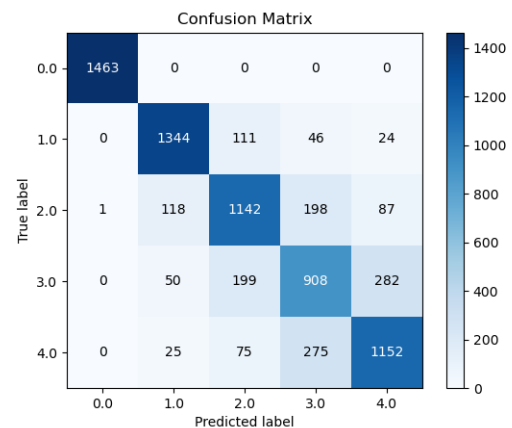
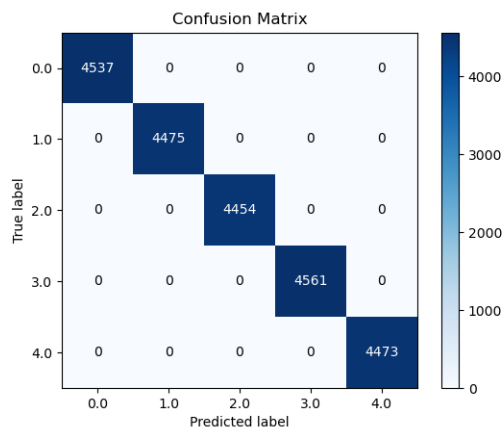
```
1 '''
2 code that trains a model and saves it
3 '''
4 import joblib
5 import numpy as np
6 import scikitplot as skplt
7 import matplotlib.pyplot as plt
8 from sklearn.metrics import accuracy_score
9 from sklearn.tree import DecisionTreeClassifier
10 from sklearn.model_selection import train_test_split
11
12 path = r'C:\Users\franc\desktop\mod.sav'
13
14 #=====
15 # Creation of data set
16 #=====
17
18 M = 30000 # numer data
19 N = 200   # len of each curve
20
21 X = np.zeros((N, M))           # matrix of featurers
22 d = np.linspace(0, 1, 5)       # parameter of curve
23 t = np.zeros(M)                # target index of d
24
25 for i in range(M):
26     # random interval
27     x1, x2 = np.random.random(2)*5
28     # each features is nothing but a curve y=x**k with k element of d
29     k = d[i%len(d)]
30     X[:, i] = np.linspace(x1, x2, N)**k
31     # the target must be integer so we use the corrispective indices
32     t[i] = i%len(d)
33
34
35 #=====
36 # Creation and training of model
37 #=====
38
39 x = X.T
40 y = t
41 # split fro train ad test
42 x_train, x_test, y_train, y_test = train_test_split(x, y)
43
44 # define the model
45 modello = DecisionTreeClassifier()
46 # I train the model
47 modello.fit(x_train, y_train)
48 # predictions about the data on which it has learned
49 prediction_train = modello.predict(x_train)
50 # prediction on new data
51 prediction_test  = modello.predict(x_test)
52
53 # accuracy
54 print("accuracy train")
55 print(accuracy_score(y_train, prediction_train))
56
57 print("accuracy test")
58 print(accuracy_score(y_test, prediction_test))
59
60 #=====
61 # Plot confusion matrix
```



```

62 #=====
63
64 skplt.metrics.plot_confusion_matrix(y_train, prediction_train)
65 skplt.metrics.plot_confusion_matrix(y_test, prediction_test)
66
67 plt.show()
68
69 #=====
70 # Save model
71 #=====
72
73 joblib.dump(modello, path)
74
75 [Output]
76 accuracy train
77 1.0
78 accuracy test
79 0.8012

```



Per rendere fattibile la cosa il target non è il valore dell'esponente ma l'indice dell'array in cui il valore è contenuto. Ora che abbiamo salvato il modello, possiamo spegnere il computer e se ci capita per strada un set di dati per cui la nostra macchina è stata addestrata possiamo riusarlo. Vediamo come si fa:

```

1 '''
2 code that takes a model saved on your computer and uses it to make predictions
3 '''
4 import joblib
5 import numpy as np
6 import scikitplot as skplt
7 import matplotlib.pyplot as plt
8 from sklearn.metrics import accuracy_score
9
10 modello = joblib.load(r'C:\users\franc\desktop\mod.sav')
11
12 #=====
13 # Creation of data set with same features
14 #=====
15
16 M = 1000 # numer data
17 N = 200 # len of each curve
18
19 X = np.zeros((N, M)) # matrix of featurrs
20 d = np.linspace(0, 1, 5) # parameter of curve
21 t = np.zeros(M) # target index of d
22
23 for i in range(M):
24     # random interval
25     x1, x2 = np.random.random(2)*5
26     # each features is nothing but a curve y=x**k with k element of d
27     k = np.random.choice(d)
28     X[:, i] = np.linspace(x1, x2, N)**k
29     # the target must be integer so we use the corrispective indices
30     t[i] = np.where(k==d)[0][0]
31
32
33 #=====

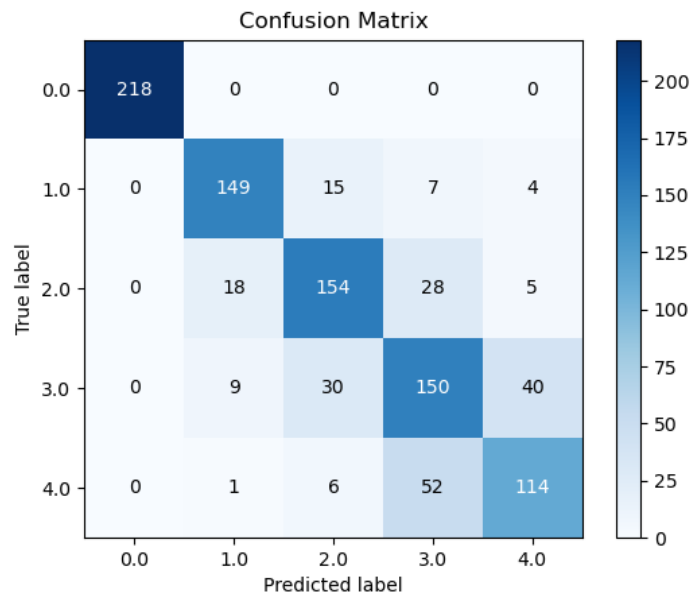
```

```

34 # Prediction on new data
35 #=====
36
37 prediction = modello.predict(X.T)
38
39 print(f'Accuracy score = {accuracy_score(t, prediction)}')
40
41 skplt.metrics.plot_confusion_matrix(t, prediction)
42
43 plt.show()
44
45 [Output]
46 Accuracy score = 0.785

```

Vediamo che l'accuratezza è abbastanza buona, parente di quella sui dati di test del precedente codice. Qui il parametro che possiamo settare in maniera diversa è M , se proviamo a cambiare N otterremo un errore, perché chiaramente la macchina sa fare solo quello per cui è stata addestrata.



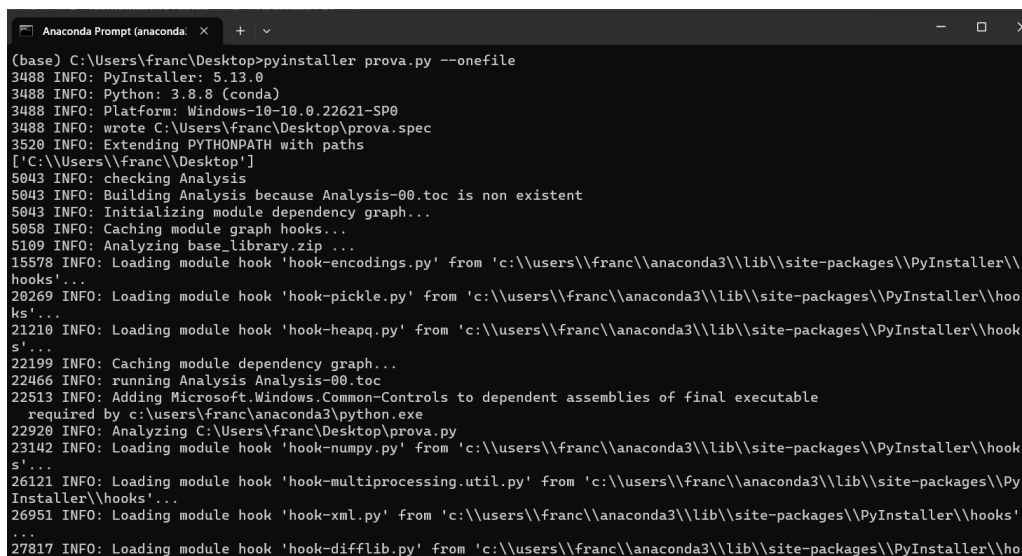
Q Creare un eseguibile

Abbiamo visto che Python è un linguaggio interpretato e non compilato quindi non viene creato un eseguibile. Supponiamo però vi venga in mente di creare un eseguibile, come facciamo? Ci serve un pacchetto: pyinstaller, che ci permette di creare un eseguibile, con il caveat che se l'eseguibile lo creo su un certo sistema, esso potrà essere eseguito solo sullo stesso sistema; detto semplice se create un eseguibile su windows non potete eseguirlo su linux e viceversa. Come prima cosa creiamo quindi un piccolo file python di prova, che chiameremo prova.py (poca fantasia).

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(0, 1, 100)
5
6 for i in np.logspace(-1, 1, 50):
7     plt.plot(x, x**i)
8
9 plt.grid()
10 plt.show()
```

Q.1 Eseguibile windows

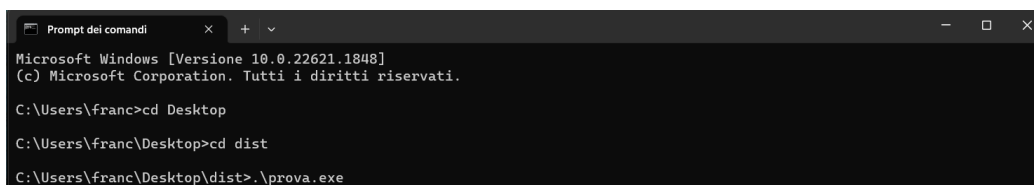
Dopo aver salvato il file dobbiamo installare pyinstaller e lo si può fare comodamente dalla shell di pyzo con pip install. Fatto ciò chiudiamo pyzo ed apriamo un prompt di comandi di anaconda/python, basta cercare sulla barra di ricerca windows: anaconda prompt ed eseguire (cioè scrivere e premere invio) la seguente linea: pyinstaller prova.py --onefile. Na cosa tipo questa:



```

Anaconda Prompt (anaconda) x + v
(base) C:\Users\franc\Desktop>pyinstaller prova.py --onefile
3488 INFO: PyInstaller: 5.13.0
3488 INFO: Python: 3.8.8 (conda)
3488 INFO: Platform: Windows-10-10.0.22621-SP0
3488 INFO: wrote C:\Users\franc\Desktop\prova.spec
3520 INFO: Extending PYTHONPATH with paths
['C:\\Users\\franc\\Desktop']
5043 INFO: checking Analysis
5043 INFO: Building Analysis because Analysis-00.toc is non existent
5043 INFO: Initializing module dependency graph...
5058 INFO: Caching module graph hooks...
5109 INFO: Analyzing base_library.zip ...
15578 INFO: Loading module hook 'hook-encodings.py' from 'c:\\users\\franc\\anaconda3\\lib\\site-packages\\PyInstaller\\hooks'...
28269 INFO: Loading module hook 'hook-pickle.py' from 'c:\\users\\franc\\anaconda3\\lib\\site-packages\\PyInstaller\\hooks'...
21210 INFO: Loading module hook 'hook-heapq.py' from 'c:\\users\\franc\\anaconda3\\lib\\site-packages\\PyInstaller\\hooks'...
22199 INFO: Caching module dependency graph...
22466 INFO: running Analysis Analysis-00.toc
22513 INFO: Adding Microsoft.Windows.Common-Controls to dependent assemblies of final executable
required by c:\users\franc\anaconda3\python.exe
22920 INFO: Analyzing C:\Users\franc\Desktop\prova.py
23142 INFO: Loading module hook 'hook-numpy.py' from 'c:\\users\\franc\\anaconda3\\lib\\site-packages\\PyInstaller\\hooks'...
26121 INFO: Loading module hook 'hook-multiprocessing.util.py' from 'c:\\users\\franc\\anaconda3\\lib\\site-packages\\PyInstaller\\hooks'...
26951 INFO: Loading module hook 'hook-xml.py' from 'c:\\users\\franc\\anaconda3\\lib\\site-packages\\PyInstaller\\hooks'...
27817 INFO: Loading module hook 'hook-difflib.py' from 'c:\\users\\franc\\anaconda3\\lib\\site-packages\\PyInstaller\\ho
```

Verranno stampate tante cose sulla shell e l'operazione potrebbe richiedere un po'. Quando avrà finito, cioè quando riappare la scritta che termina con > l'eseguibile sarà creato. Verrà creato nella cartella dove è salvato il file tre cose: un file. spec, una cartella nominata build e un'altra cartella nominata dist, che è quella che contiene l'eseguibile vero e proprio. Per eseguirlo bisogna aprire ora il prompt dei comandi di windows, andare nella cartella dove è l'eseguibile ed eseguirlo:

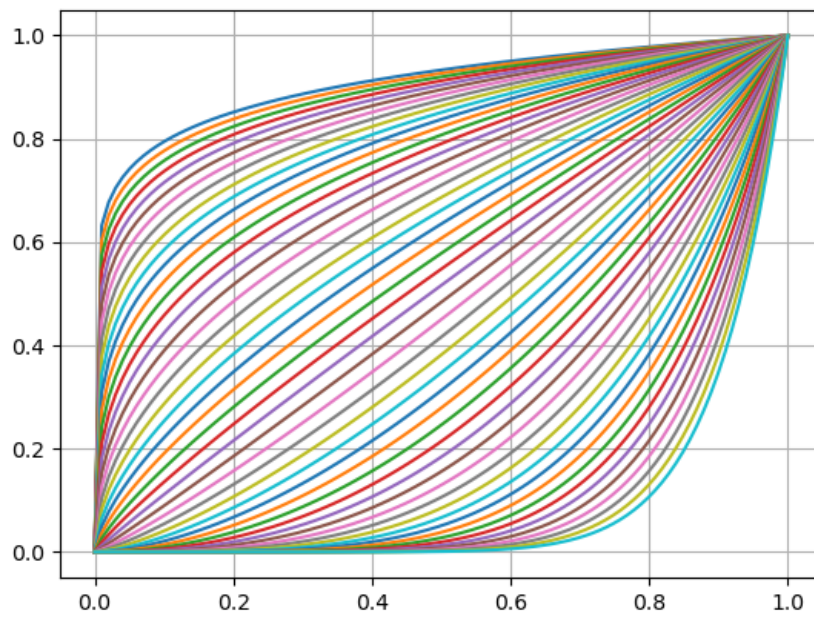


```

Prompt dei comandi x + v
Microsoft Windows [Versione 10.0.22621.1848]
(c) Microsoft Corporation. Tutti i diritti riservati.

C:\Users\franc>cd Desktop
C:\Users\franc\Desktop>cd dist
C:\Users\franc\Desktop\dist>.\prova.exe
```

e si avrà l'output desiderato, (per la serie grafici carini):



Q.2 Eseguibile linux

Per linux il discorso è analogo, i comandi da usare sono gli stessi, solo che ora la shell è sempre la stella. I nomi delle cartelle e dei file saranno uguali. Non so perché dovrebbe venire in mente di creare un eseguibile, da quali arcani motivi possiate essere spinti, ma comunque questo è un modo, sinceramente mai fatto roba del genere se non per scriverlo qui.

R Bibliografia e Conclusioni

Leggetevi il cazzo di manga.

Leggetevi la documentazione.

A parte gli scherzi, in questa sezione dovrebbe come giusto che sia esserci una bibliografia, delle referenze giustamente. La sua assenza è dovuta al fatto che, al di là degli autori della prima versione delle lezioni, che qui è stata un po' rivista ed ampliata, e che ho già citato a inizio di queste note e che ringrazio ancora, tutto il resto è una raccolta di argomenti che ho appreso nel corso di 4 anni, cercando su internet come farle e studiando da lì. Quindi andare a rintracciare tutto sarebbe un po' difficile. Di certo però wikipedia è stata utile per non parlare di stackoverflow. Inoltre molti esempi si possono trovare sulla documentazione dei pacchetti. Quindi ecco, diciamo che la bibliografia di queste note è un po' tutto internet.

Come conclusione vorrei solo dirvi che ok potrei essermi lasciato andare con le appendici, ma è giusto per dare, a chi voglia leggerle, un'idea, una breve introduzione, un impatto non brusco con la programmazione e il calcolo scientifico. Ho inserito in queste appendici delle introduzioni a tutto quello che penso che uno studente di fisica possa trovarsi ad affrontare nel corso degli anni. Per chi segue il corso di Python dell'aisf non mi aspetto minimamente che vengano lette tutte, spero che le lezioni vere e proprie risultino utili, e che magari, a distanza, essendosi impratichito con l'uso di python, qualcuno si ricordi che esistono queste appendici e che magari torni a vederle e leggerle.

See you Space Cowboy ...