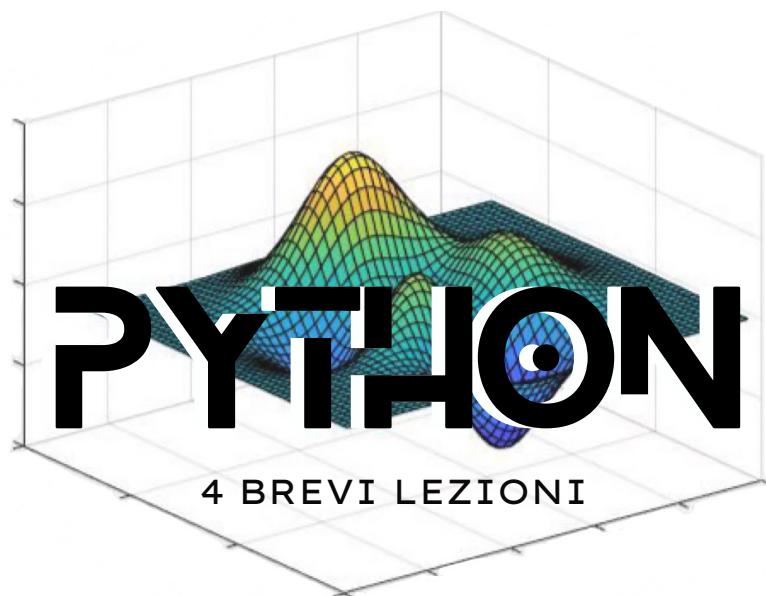


# (Le non) 4 (e per nulla) Brevi lezioni di Python

Francesco Zeno Costanzo

(Do you remember the) 21 (st night of) September 2025



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)

# Indice

<b>1 Scopo, storia delle note e caveat</b>	<b>5</b>
<b>2 Introduzione</b>	<b>6</b>
2.1 Notazioni . . . . .	6
2.2 I 4 (per ora) comandamenti dell'informatica . . . . .	6
<b>3 Lezione Zero: Installazione</b>	<b>7</b>
3.1 Installazione dell'ambiente: Pyzo . . . . .	7
3.2 Installazione dell'interprete: Anaconda . . . . .	7
3.3 Installazione dei pacchetti . . . . .	7
<b>4 The Zen of Python</b>	<b>8</b>
<b>5 Prima lezione</b>	<b>9</b>
5.1 Funzione print . . . . .	9
5.2 Commenti . . . . .	9
5.3 Variabili . . . . .	9
5.4 Operatore di Warlus . . . . .	12
5.5 Librerie . . . . .	13
5.6 Come leggere il Traceback . . . . .	14
5.7 Un paio di trick . . . . .	15
<b>6 Seconda lezione</b>	<b>16</b>
6.1 Gli array . . . . .	16
6.2 Tipi di array . . . . .	17
6.3 Array predefiniti . . . . .	17
6.4 Operazioni con gli array . . . . .	18
6.5 Maschere . . . . .	19
6.6 Matrici . . . . .	20
6.7 Esercizi . . . . .	21
<b>7 Terza lezione</b>	<b>27</b>
7.1 Le funzioni . . . . .	27
7.2 Istruzioni di controllo . . . . .	27
7.2.1 Espressioni condizionali: if, else, elif . . . . .	27
7.2.2 Cicli: while, for . . . . .	28
7.3 Ancora funzioni . . . . .	30
7.4 Funzioni lambda . . . . .	30
7.5 Sempre più funzioni: i decoratori . . . . .	31
7.6 Grafici . . . . .	37
7.6.1 Istogrammi . . . . .	39
7.6.2 I colori . . . . .	40
7.7 Standard input . . . . .	42
7.8 Prestazioni: <i>pure</i> Python vs librerie . . . . .	44
7.9 Prestazioni: globale vs locale . . . . .	44
7.10 Gestione errori . . . . .	45
7.10.1 Try e except . . . . .	46
7.10.2 Raise Exception . . . . .	46
7.11 Logging . . . . .	46
7.12 Generatori . . . . .	49
7.12.1 Problema delle 8 regine . . . . .	51
7.13 Esercizi . . . . .	52
<b>8 Quarta lezione</b>	<b>59</b>
8.1 Importare file Python . . . . .	59
8.2 Fit . . . . .	59
8.2.1 Init . . . . .	64
8.3 Dietro curve fit: Levenberg-Marquardt . . . . .	66
<b>9 Lezione Bonus: Version control</b>	<b>72</b>

<b>A Sistemi lineari</b>	<b>75</b>
A.1 Metodo Gauss-Seidel . . . . .	75
A.2 Successive over-relaxation . . . . .	75
A.3 Metodo del gradiente coniugato . . . . .	77
A.4 Matrici tridiagonali . . . . .	79
<b>B Zeri di una funzione</b>	<b>81</b>
B.1 Bisezione . . . . .	81
B.2 Metodo di Newton . . . . .	82
B.2.1 Calcolo simbolico . . . . .	83
B.3 Zeri in più dimensioni . . . . .	85
<b>C Risolvere numericamente le ODE: IVP</b>	<b>89</b>
C.1 Calcolo delle derivate . . . . .	89
C.2 ODE: caso esponenziale . . . . .	92
C.3 Sistema di ODE: pendolo semplice . . . . .	94
C.4 Animazione . . . . .	96
<b>D Risolvere numericamente le ODE: BVP</b>	<b>98</b>
D.1 Shooting . . . . .	98
D.2 Relaxation . . . . .	102
<b>E Calcolo degli integrali</b>	<b>106</b>
<b>F Diagonalizzazione</b>	<b>109</b>
F.1 Metodo delle potenze . . . . .	109
F.2 Equazione di Schrödinger . . . . .	111
F.3 Algoritmo QR . . . . .	113
F.4 Lanczos . . . . .	114
F.4.1 Matrix-less . . . . .	116
<b>G Trasformate di Fourier</b>	<b>120</b>
G.1 DFT . . . . .	120
G.2 FFT . . . . .	121
G.2.1 Standard FFT iterativa . . . . .	123
G.3 RFFT . . . . .	125
G.4 Applicazioni delle FFT . . . . .	128
G.4.1 Derivate con FFT . . . . .	128
G.4.2 Altre applicazioni . . . . .	130
<b>H Presa dati da foto</b>	<b>131</b>
<b>I Fit</b>	<b>132</b>
I.1 Fit circolare, metodo di Coope . . . . .	132
I.2 Fit di un'ellisse, metodo di Halir e Flusser . . . . .	135
<b>J Interpolazione</b>	<b>138</b>
J.1 Interpolazione lineare . . . . .	138
J.2 Interpolazione Polinomiale . . . . .	139
J.3 Scipy.interpolate . . . . .	140
<b>K Programmazione a oggetti</b>	<b>142</b>
K.1 Problema N-body . . . . .	142
K.2 Breve compendio di meccanica Hamiltoniana . . . . .	145
K.3 Più che una funzione . . . . .	148
<b>L Propagazione errori</b>	<b>153</b>
L.1 Propagarli a mano . . . . .	153
L.2 Uncertainties . . . . .	154
L.3 Unit test . . . . .	154

<b>M Metodi Montecarlo</b>	<b>157</b>
M.1 Generatori numeri pseudo-casuali . . . . .	157
M.2 Calcolo di Pi greco . . . . .	158
M.3 Radice di N . . . . .	159
M.4 Importance sampling . . . . .	162
M.4.1 Importance vs Simple . . . . .	164
M.5 Metropolis (Hastings) . . . . .	166
<b>N Risolvere numericamente le PDE</b>	<b>171</b>
N.1 Equazione del trasporto . . . . .	171
N.2 Equazione del calore . . . . .	174
N.3 Equazione di Burgers . . . . .	175
N.4 Equazione di laplace . . . . .	177
N.5 Equazione di Schrödinger . . . . .	179
<b>O Risolve numericamente le SDE</b>	<b>182</b>
O.1 Processo di Ornstein–Uhlenbeck . . . . .	182
O.2 Moto geometrico Browniano . . . . .	183
<b>P Ottimizzazione</b>	<b>185</b>
P.1 Discesa del gradiente . . . . .	185
P.2 Principio di inerzia . . . . .	186
<b>Q Machine Learning</b>	<b>188</b>
Q.1 Classificatore . . . . .	188
Q.2 Regressori . . . . .	189
Q.3 Salvare il modello . . . . .	190
Q.4 Clustering . . . . .	192
Q.4.1 K-Means . . . . .	192
Q.4.2 DBSCAN . . . . .	194
<b>R Reti neurali</b>	<b>198</b>
R.1 Reti Neurali da zero . . . . .	198
R.2 PINN . . . . .	205
<b>S Calcolo parallelo</b>	<b>213</b>
S.1 Numba . . . . .	213
S.1.1 Equazione al laplaciano in 2 dimensioni . . . . .	214
S.1.2 Equazione delle corde musicali . . . . .	219
S.2 Multiprocesing . . . . .	222
<b>T Creare un eseguibile</b>	<b>227</b>
T.1 Esegibile windows . . . . .	227
T.2 Esegibile linux . . . . .	228
<b>U Bibliografia e Conclusioni</b>	<b>229</b>

# 1 Scopo, storia 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, più che altro, di dare un assaggio, un'introduzione, di quello che nella vita di un fisico capita spesso di utilizzare (gli argomenti introdotti sono molti, di vario genere, e una trattazione esaustiva richiederebbe troppo lavoro. Inoltre esiste una quantità di letteratura a riguardo da far impallidire la famosa biblioteca d'Alessandria). Non pretendo vengano letti ma se siete curiosi, loro son lì, non mordono.

Nonostante la copiosa aggiunta di argomenti come appendici, essere rimangono tali, in quanto ciò che c'è di programmazione *tout court* è (fatto salvo qualche piccola eccezione) inserito nelle "lezioni", secondo l'originale scopo di queste note. Con il tempo, in verità anche queste sono andate un allungandosi con l'aggiunta di argomenti che reputavo interessante trattare. Tant'è che in una singola lezione di un'ora e mezza può non bastare per esaurire le pagine qui scritte. Alcune di queste appendici, scelte per interesse, utilità o per semplicità sono state trattate nel "corso avanzato"; corso nato proprio in seguito a questo sproloquo di argomenti. Proprio per questa scelta soggettiva degli argomenti qui rimane annoverato tutto come appendici, cercando di dare per il lettore un filo logico e di legami di propedeuticità per gli argomenti trattati. Cosa che magari in quelli selezionati per le lezioni non si fa, dando per buono alcuni punti di parte.

Nelle note sono presenti i codici in modo che possano essere consistenti da sole e quindi il lettore possa rendersi subito conto di come implementare il tutto. Tuttavia copiare e incollare i codici potrebbe creare fastidio quando saranno eseguiti, a causa di come i vari editor leggono spazi, segni di operazioni o altro; conviene piuttosto copiarli a mano o prenderli dalla cartella dove sono presenti anche queste note: <https://github.com/Francesco-Zeno-Costanzo/4BLP> (In realtà nel momento in cui scrivo questa frase, in quella repo vi è solo questo file, ma vi è anche il link che rimanda ad un'altra repo dove i codici sono presenti). Alcuni commenti di codici sono in inglese perché si tratta di codici presenti anche in altre repository della mia pagina github. Spero, data la natura di queste note, che mi perdonerete il bilinguismo.

Queste note sono, almeno per ora, in continuo aggiornamento. A seconda se ritengo interessante inserire l'argomento.

## 2 Introduzione

Python è un linguaggio di programmazione generalista noto per la sua semplicità e leggibilità, caratteristiche che lo rendono particolarmente accessibile a noi poveri umani. Scrivere codice in Python è spesso più leggero e scorrevole rispetto a linguaggi come C o Fortran, grazie a una sintassi intuitiva e vicina al linguaggio "naturale" (i.e. il nostro). A differenza di molti altri linguaggi (e.g. C o fortran), Python è interpretato e non compilato. Questo significa che il codice viene eseguito direttamente da un interprete senza la necessità di una fase di compilazione preliminare. Tale caratteristica offre alcuni vantaggi significativi: se si verifica un errore durante l'esecuzione, l'interprete segnala immediatamente il problema e indica la linea di codice incriminata, facilitando il debug. Nei linguaggi compilati, invece, il codice viene prima tradotto in un file eseguibile indipendente, perdendo dunque ogni collegamento diretto con il sorgente originale. Di conseguenza, se si verifica un errore a tempo di esecuzione, come il famigerato *segmentation fault*, può essere molto più difficile individuare la causa. Ovviamente, come insegna la conservazione della massa, nulla si crea e nulla si distrugge: botte piena o moglie ubriaca *tertium non datur*. Il principale svantaggio di un linguaggio interpretato rispetto a uno compilato è la velocità di esecuzione. Python è sensibilmente più lento di C o Fortran, poiché il codice viene analizzato e interpretato riga per riga anziché essere eseguito direttamente in linguaggio macchina. Tuttavia, l'ampia gamma di librerie ottimizzate disponibili per Python (come NumPy, Scipy, Pandas o TensorFlow e potremmo andare avanti per molto) consente di mitigare questo problema, delegando i calcoli più pesanti a codice scritto in C o fortran appunto. Un altro punto di forza di Python è la sua versatilità. È usato in numerosi campi, calcolo scientifico, analisi dati, intelligenza artificiale, sviluppo web o anche sicurezza informatica. Quindi tanta roba da fare, avremo tempo per discuterne.



### 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 I 4 (per ora) comandamenti dell'informatica

- Se funziona quanto basta  
non toccare che si guasta.
- RTFM: Read The Fucking Manual. La documentazione on-line è il miglior posto dove trovare risposte.
- Non dite che non funziona finché non avete provato a spegnere e riaccendere.
- Il computer fa esattamente quello che voi gli dite di fare non quello che volete che faccia.

### 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 bit<sup>1</sup>. Nel caso in cui vi piaccia smanettare con i sistemi Linux, consigliamo come procedura alternativa (e più immediata) accedere al terminale e digitare i seguenti comandi:

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

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

#### 3.2 Installazione dell'interprete: Anaconda

Ora che abbiamo l'ambiente bisogna munirisi di un interprete. Tra i tanti, si consiglia Anaconda, che porta in automatico tutti i pacchetti necessari per il lavoro scientifico. Esso è reperibile al seguente indirizzo: <https://www.anaconda.com/download/>. Allo scopo di mantenere la compatibilità con il sistema Pyzo si raccomanda di scaricare la versione corrispondente a Python 3 e non Python 2. Alternativamente è possibile procurarsi Miniconda, che è una versione ridotta e più leggera di Anaconda che arriva con molti meno pacchetti, ma occupa chiaramente meno spazio in memoria. Esso è reperibile al seguente indirizzo: <https://conda.io/miniconda.html>. È fortemente consigliato installare l'interprete nella cartella di default, in modo da rendere più semplice il lavoro di riconoscimento del programma da parte di Pyzo. Una volta installato l'interprete, aprendo Pyzo dovrà 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 dovrà 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
```

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

## 4 The Zen of Python

Una volta installato pyzo, oppure python, aprete un terminale, che sia quello di pyzo, la shell di ubuntu (dopo aver digitato python3), o di anaconda, e scrivete:

```
1 >>> import this
```

ecco cosa uscirà:

The Zen of Python, by Tim Peters

Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one— and preferably only one —obvious way to do it.  
Although that way may not be obvious at first unless you're Dutch.  
Now is better than never.  
Although never is often better than \*right\* now.  
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea – let's do more of those!

Delle interessanti linee guida di certo.

## 5 Prima lezione

### 5.1 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 informazione contenute nel codice. Vediamo quindi il più classico degli esempi:

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

Bene, se siete riuscite ad eseguire questo codice siete ufficialmente dei programmatori! Giusto per voler essere pedanti i numeri che vedete a sinistra non hanno un vero e proprio significato per l'esecuzione; il loro unico scopo è indicare all'utente a che riga si trova. Questa funzione può stampare sia valori che espressioni (in gergo stringhe) [capiremo tra poco cosa queste magiche cose che vengono stampate sono effettivamente]:

```
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
```

### 5.2 Commenti

Come vedete dal codice precedente alla linea 4 ci sta un simbolo poco familiare a chi per la prima volta approccia la programmazione: "\n" chi era costui? Scritto così il codice, l'unico modo per capirlo è che voi eseguiate il codice e vedendo il risultato cerchiate di risalire al significato. Ora chiaramente questa procedura è abbastanza sbrigativa ma se dovete capire diverse parti del codice, il quale magari impiega un tempo non trascurabile a darvi un risultato, beh diciamo che non è una bella vita. Al fine dunque di rendere fruibile il codice sia ad altri o anche al voi stesso del futuro, 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 va bene  
12  
13 anche tre apici  
14 """  
15 [Output]  
16 Hello world!
```

Ricordate, un codice viene letto molte più volte rispetto a quanto viene scritto. Quindi commentate, sempre.

### 5.3 Variabili

Una variabile è un nome, un simbolo, che si da ad un certo valore. In Python non è necessario definire le variabili prima di utilizzarle specificandone il tipo, come faremmo in C o fortran, esse si creano, o meglio si inizializzano, usando il comando di assegnazione '='. Facciamo un esempio con variabile numeriche:

```
1 numerointero= 13  
2 numeroavirgolamobile= 13.  
3  
4 print('Numero intero:', numerointero, 'Numero in virgola mobile:', numeroavirgolamobile)  
5 #oppure possiamo stampare in questo modo:
```

```

6 print(f'Numero intero: {numerointero}, Numero in virgola mobile: {numeroavirgolamobile}')
7
8 [Output]
9 Numero intero: 13 Numero in virgola mobile: 13.0
10 Numero intero: 13 Numero in virgola mobile: 13.0

```

Un'altra cosa molto fondamentale, oltre i commenti, per la fruibilità del codice è il modo di dare nomi alle variabili. Nel codice di sopra il nome delle variabili è alquanto esplicativo del loro significato, ed è in genere buona norma, appunto, dare nomi che siano intuitivi. Ora non vi dico che dovete chiamare una variabile: "momentoagolaresullasseZ", che ci vogliono tre anni solo a scriverla, che nel frattempo pure il protone inizia a decadere, 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. Giusto qualche info un po' anticipata 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 cioè 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.

```

1 """
2 liste tuple e dizionari sono oggetti indicizzati che possono
3 ogni tipo di informazioni al loro interno
4 """
5 lista = [0, 1, 'lampada', [0, 23], (29, 11), {3:4, 'capra':'panca'}] # lista
6 tupla = (3, 2, "ruspa", [0, 0], (3, 9), {87:90})
7 dictz = {0:1, 1:[2, 3], 'astolfo':(2, 3), "diz":{1:2}}
8
9 # stampo tutto a schermo
10 print(lista)
11 print(tupla)
12 print(dictz)
13
14 # tramte l'indice accedo all'elemento della lista o della tupla
15 print(lista[3])
16 print(tupla[3])
17
18 # per il dizionario va invece usata la chiave
19 print(dictz['astolfo'])
20 print(dictz[1])
21
22 # modifco elemento all'indice zero nella lista e nel dizionario
23 lista[0] = (0, 1, 2, 3, 4, 5)
24 dictz[0] = "BUONA PASQUA"
25
26 print(lista)
27 print(dictz)
28
29 # se lo facessi con la tupla avrei un errore
30 tupla[0] = 1

```

```

31 [Output]
32 [0, 1, 'lampada', [0, 23], (29, 11), {3: 4, 'capra': 'panca'}]
33 (3, 2, 'ruspa', [0, 0], (3, 9), {87: 90})
34 {0: 1, 1: [2, 3], 'astolfo': (2, 3), 'diz': {1: 2}}
35 [0, 23]
36 [0, 0]
37 (2, 3)
38 [2, 3]
39 [(0, 1, 2, 3, 4, 5), 1, 'lampada', [0, 23], (29, 11), {3: 4, 'capra': 'panca'}]
40 {0: 'BUONA PASQUA', 1: [2, 3], 'astolfo': (2, 3), 'diz': {1: 2}}
41 Traceback (most recent call last):
42   File "/home/francesco/GitHub/4BLP/1 Prima Lezione/codiciL1/list_tuple_dict.py", line 30, in <module>
43     tupla[0] = 1
44 TypeError: 'tuple' object does not support item assignment

```

Concentriamoci attualmente su come fare le classiche operazioni matematiche tra variabili, numeriche ovviamente:

```

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

```

E fin qui tutto bene, tutto abbastanza normale e mi raccomando ricordate che l'elevamento a potenza si fa con doppio asterisco "``". 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. Questa precisione può comunque essere cambiata grazie alla libreria : "mpmath"; la quale permette di settare una precisione arbitraria, divertitevi a scoprirla. 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 0.3000000000000004
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. Come esercizio lasciato al lettore provate 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, vediamolo 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. Inoltre abbiamo usato la lettera "f" perché vogliamo un numero decimale, se volessimo altri formati potremmo usare: "i" o "d" per gli interi, "o" per un numero in base otto, "x" per un numero esadecimale, "e" per un numero in notazione scientifica. Infine le lettere "c" e "s" indicano rispettivamente un singolo carattere e una stringa.

Una variabile può essere ridefinita e cambiare valore, addirittura cambiate tipo, 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 operazioni varie
6
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  $x = x \text{ operazione} \text{ numero}$  possono essere abbreviati con  $x \text{ operazione=} \text{ numero}$ .

## 5.4 Operatore di Warlus

Supponiamo di avere un frammento del codice del tipo:

```

1 measure = [2, 8, 0, 1, 1, 9, 7, 7]
2
3 info_measure = {
4     "length": len(measure),
5     "sum": sum(measure),
6     "mean": sum(measure) / len(measure),
7 }
8
9 print(info_measure)
10
11 [Output]
12 {'length': 8, 'sum': 35, 'mean': 4.375}

```

Codice abbastanza chiaro, no? Abbiamo una serie di misure e creiamo un dizionario che contenga alcune informazioni sul nostro insieme di misure. Una cosa salta subito sull'occhio, le funzioni "sum" e "len" vengono chiamate due volte, cosa inutile chiaramente. Possiamo chiaramente aggiungere due righe per definire due variabili di nostro interesse in modo anche da tener traccia anche del loro valore volendo. Ma possiamo farlo in modo più compatto? Possiamo grazie all'operatore di Warlus:

```

1 measure = [2, 8, 0, 1, 1, 9, 7, 7]
2
3 info_measure = {
4     "length": (length := len(measure)),
5     "sum": (total := sum(measure)),
6     "mean": total / length,
7 }
8
9 print(info_measure)
10 print(length)
11 print(total)
12
13 [Output]
14 {'length': 8, 'sum': 35, 'mean': 4.375}
15 8
16 35

```

L'utilizzo della parentesi tonda quando si usa l'operatore di Warlus "`:=`" è necessario, e vederemo che servirà anche poi quando tratteremo le condizioni di controllo.

## 5.5 Librerie

Le librerie sono luoghi misticci 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, ma in vari ambiti, 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 angoloinradiani = math.radians(angoloingradi)
13
14 print("Angolo in gradi:", angoloingradi, "Angolo in radiani:", angoloinradiani)
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 radiani: 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 scrive 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)". Inoltre spesso si trova la sintassi:

```
1 from numpy import *
```

con numpy o con qualsiasi altra libreria. Questo ci permette di non dover scrivere ogni volta "numpy." o "np." davanti le funzioni che vogliamo utilizzare. Bisogna però stare attenti all'esistenza di funzioni con stesso nome ma che fanno cose diverse, come dicevamo prima la funzione seno ad esempio. In un contesto in cui si importano sia math che numpy usando l'asterisco ci sarà un conflitto su che funzione usare; o meglio, verrà usata la funzione della libreria più recentemente importata. il che vuol dire che se scrivete:

```
1 from numpy import *
2 from math import *
```

la funzione seno chiamata sarà quella di math e quindi avrete un errore se il possibile input non dovesse essere un numero ma un array.

## 5.6 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
        err
        b = 1/c
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.

## 5.7 Un paio di trick

Abbiamo visto che Python non necessita di punti e virgola per delimitare una linea come in C. Tuttavia è possibile usarli con lo stesso scopo, quindi se volete mettere due comandi Python sulla stessa riga vi basta separarli con un ";" e verranno eseguiti come fossero due righe diverse. Inoltre possono essere usate sulla shell per nascondere l'output del comando, dato che come premete invio sulla shell la riga appena scritta viene interpretata, esattamente come su Mathematica. Sempre su shell invece se avete bisogno di fare dei calcoli veloci stile calcolatrice potete usare l'underscore per riferirvi al risultato precedente, come quando sulla calcolatrice premete il tasto Ans.

## 6 Seconda lezione

Ripetiamo tutti insieme: Python conta da zero.

### 6.1 Gli array

Un array unidimensionale è semplicemente una sequenza ordinata di numeri; è, in sostanza, un vettore e come tale si comporta. 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
14     numpy:
15 array2 = np.append(array1, -4.)
16 print(array2)
17 #Mentre per togliere un elemento basta indicare il suo indice alla funzione remove di numpy:
18 array2 = np.delete(array2, 0)
19 print(array2)
20
21 [Output]
22 [ 1.  2.  4.  8. 18. 16.]
23 [ 1.  2.  4.  8. 18. 16. -4.]
24 [ 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]
```

Altri metodi interessanti per le liste sono "index()", "remove()", "count()", "insert()", "reverse()", "extend()", "sort()"; divertitevi a scoprire cosa ciascuno fa.

## 6.2 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 è 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
```

## 6.3 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 è la lunghezza
5 arraydizeri_1 = np.zeros(3, 'int')
6
7 #array contenente tutti uno
8 arraydiuni_0 = np.ones(5)#il numero specificato è 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 è un array il cui primo elemento è zero
16 e l'ultimo elemento è 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 questo invece e' un array il cui primo elemento e' 10^1
24 e l'ultimo elemento e' 10^2, lungo 10 e i cui elementi sono
25 equispaziati in maniera logaritmica tra i due estremi
26 """
27
28 equi_log = np.logspace(1, 2, 10)
29 print(equi_log)
30
31 [Output]
32 [0. 0. 0.]
33 [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. ]

```

## 6.4 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.]

```

Benché strano python non rifiuta gli indici negativi semplicemente perché essi indicizzano l'array al contrario ma partendo da 1. Quindi "-1" indica l'ultimo elemento, "-2" il penultimo e così via. Volendo generalizzare la spiegazione, l'operazione di slicing ha la seguente sintassi:

nome\_array[start : stop : step] .

Per cui [3 : 5] significa prendere i numeri corrispondenti agli indici dal 3 incluso al 5 escluso; [:: 2] o equivalentemente [0 :: 2] prende gli elementi da zero, fino a dove gli è possibile, a passi di 2. Questo perché se non specifico nulla in start o in stop di default viene considerato l'inizio e la fine dell'array rispettivamente. Quindi Python si regola da solo fino dove arrivare a seconda dello step che gli stiamo chiedendo. Ultimo esempio [:: -1] vuol dire vai dall'inizio alla fine a passi di meno uno, ma andare a passi negativi dato che non esiste l'indice -1, deve

partire dall'ultimo indice e quindi riavvoglie l'array; come a scambiare star e stop. Provate infatti, prendendo un certo array a vostra scelta, magari l'array "array1" definito nel codice sopra, a stampare le seguenti quantità:

- `array1[4 : 0 : -1]`,
- `array1[0 : 4 : -1]`.

Vedrete che i risultati saranno molto diversi.

Grande comodità sono poi 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 spaiato.

```

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.

## 6.5 Maschere

Vi sarete di certo accorti che per creare un array di numpy quel che facciamo è passare una lista alla funzione "np.array" e infatti prima avevamo visto un array che conteneva una stringa, e per cui tutti gli elementi erano diventate stringhe. Similmente è possibile fare un array di valori booleani, cioè vero e falso.

```

1 import numpy as np
2
3 # array che contiene valori logici, detti booleani
4 b = np.array([True, False])
5
6 print(b)
7
8 [Output]
9 [ True False]

```

E fin qui nulla di sorprendente si potrebbe dire. Possiamo fare però una cosa interessante:

```

1 import numpy as np
2
3 # array che contiene valori logici, detti booleani
4 b = np.array([True, False])
5
6 # normalissimo array numerico
7 x = np.array([32, 89])
8 y = x[b] # x in corrispondenza di indici di b
9 print(y)
10
11 [Output]
12 [32]

```

Vediamo che se passiamo come indice del nostro array un array di valori logici, otteniamo un nuovo array, il quale contiene solo gli elementi corrispondenti all'indice che coincide con i valori "True" all'interno dell'array logico. Abbiamo creato quella che si dice una maschera è può essere utili in molti casi, vediamo un altro esempio.

```

1 import numpy as np
2
3 # creo un array
4 x = np.array([5, 4, 2, 8, 3, 9, 7, 2, 6, 3, 9, 8])
5
6 # voglio selezionare solo gli elementi maggiori di una certa soglia
7 mask = x >= 4 # mask e' un array di booleani, secondo la condizione data
8 # mask vale True negli indici in cui il valore di x e' maggiore o uguale a 4
9
10 print(x)
11 print(mask)
12 print(x[mask])
13
14 [Output]
15 [5 4 2 8 3 9 7 2 6 3 9 8]
16 [ True  True False  True False  True  True False  True  True]
17 [5 4 8 9 7 6 9 8]

```

Poi se i nostri dati fossero una funzione del tempo potremmo magari creare la maschera sul tempo e applicarla ai nostri dati, magari perchè vogliamo considerare solo un certo range. Finché gli array son lunghi uguali potete fare quello che volete.

## 6.6 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 # oppure anche: matrice2 = np.array([[1, 2], [3, 4], [5, 6]])
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 # attenzione che non funziona se si usa np.array()
12 #alternativamente si potrebbe scrivere: prodottodimatrici = matrice1 @ matrice3
13 # che funziona sia con np.matrix che np.array
14 print('\nProdotto di matrici:\n', prodottodimatrici)
15
16 [Output]
17 Somma di matrici:
18 [[2. 3.]
19 [4. 5.]
20 [6. 7.]]
21
22 Prodotto di matrici:
23 [[15 18 21]
24 [33 40 47]
25 [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  $3 \times 3$ . Se questo vi sembra strano aspettate di fare meccanica quantistica e ne riparliamo.

Ora è importante far notare una cosa: l’operazione di assegnazione con gli array (o liste) è delicata:

```

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.

## 6.7 Esercizi

Ora che grazie agli array abbiamo un po’ più di carne al fuoco voglio proporvi un paio di esercizi da fare da voi per prendere familiarità con questi oggetti, o strutture dati volendo. In basso ci sarà anche la soluzione che

gradirei non guardaste prima di aver fatto almeno un tentativo.

1. Verificare che anche con le liste l'operazione di assegnazione fa coincidere gli indirizzi di memoria.
2. Creare la matrice identità  $I 3 \times 3$  e verificare che, dato un vettore  $v$  a vostra scelta,  $v^T I v$  sia uguale a "sum( $v^{**2}$ )".
3. Creare inizialmente una matrice  $L 4 \times 4$  tutta nulla ma la cui diagonale sia: "-1, 1, 1, 1", e poi creare una matrice  $B$  definita come

$$B = \begin{bmatrix} \gamma & -\beta\gamma & 0 & 0 \\ -\beta\gamma & \gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{con } \gamma = 1/\sqrt{1-\beta^2} \quad \beta \in [0, 1].$$

preso  $v$  un vettore a vostra scelta cosa succede se calcolate  $v^T L v$  e  $w^T L w$ , con  $w = Bv$ ? Confrontare tra l'utilizzo di  $L$  e quello di  $I$  (ovviamente  $4 \times 4$ ), per vari valori di beta.

4. Data una matrice di zeri  $4 \times 2$  (4 righe, 2 colonne), riempite la colonna di sinistra e poi invertite le colonne.
5. Dato un linspace (e.g. tra 0 e 20) creare due nuovi array con solo i numeri pari uno e solo i dispari l'altro.
6. Dato un array a vostro piacimento, creare una maschera, con due condizioni sempre a vostro gusto.
7. Creare una matrice che contenga sulle colonne le tabelline da 0 a 10 (non banale da fare in poche righe).
8. Creare un logspace in base 10 senza usare l'apposita funzione di numpy.
9. Dati due array, createli come volete, dovete eliminare dal primo array, tutti gli elementi che esso condivide con il secondo array.
10. Create una matrice  $3 \times 3$  con numeri a caso ma reali compresi fra 0 e 1 e stampateli prima normalmente e poi con solo due cifre decimali (cercate su internet, non morde).
11. Prendete un array a caso non ordinato di 10 elementi e sostituite il valore più grande con 7 e il più piccolo con 1/7.
12. Dato un array contenente valori ripetuti sempre creato da voi (ormai siete bravissimi a creare array dato che lo lascio sempre a voi) creare un array che contenga tutti gli elementi senza ripetizione.
13. Dato l'array  $x=(1, 2, 3)$ , visto come insieme, creare una matrice  $(9 \times 2)$  che sia il prodotto cartesiano di  $x$  con se stesso (i.e. tutte le possibili coppie che potete formare).
14. Prendete due vettori lunghi tre che rappresentino la posizione e la velocità di un oggetto puntiforme, calcolate il momento angolare.
15. Dato un array 2D, la posizione di un particella su un piano, in cartesiane passare alla scrittura in coordinate polari.
16. Data una matrice  $4 \times 3$  creata con valori a vostra scelta, create una nuova matrice che sia la trasposta di quella originale e verificate che  $A^T \times A$  risulti una matrice quadrata simmetrica.
17. Create una matrice di rotazione  $2 \times 2$  per un angolo  $\theta$  dato, applicatela a un vettore di coordinate  $(x, y)$  a vostra scelta e verificate che la lunghezza del vettore rimanga invariata.
18. Generate un array con numeri consecutivi da 1 a 20 e trasformatelo in modo che solo gli elementi in posizione pari vengano elevati al quadrato.
19. Dato un vettore che rappresenta la posizione iniziale e un array di velocità costante, calcolate la posizione finale del punto dopo 10 secondi di movimento rettilineo uniforme.
20. Dato un array  $x$  che rappresenta il dominio tra 0 e  $2\pi$ , calcolate due array  $\sin(x)$  e  $\cos(x)$  e costruite una matrice con ciascuno come colonna. Successivamente, calcolate la somma degli elementi di ciascuna colonna.
21. Dato un array di numeri a vostra scelta, accedete e stampate il terzo elemento. Poi, provate a usare un indice negativo per accedere all'ultimo elemento.
22. Create due array con 5 elementi ciascuno e concatenateli in un unico array.
23. Dato un array di numeri consecutivi (per esempio, da 1 a 10), estraete un subarray che contenga solo i primi 5 elementi e poi uno che contenga solo gli ultimi 3.

Ecco la mia soluzione di questi esercizi. non riporto l'output per brevità.

Primo:

```

1 a = [1, 2, 3, 4]
2 print(f"array iniziale: {a}, id: {id(a)}")
3
4 b = a
5 b[0] = 7
6
7 print(f"array iniziale: {a}, id: {id(a)}")
8 print(f"array finale : {b}, id: {id(b)}")
```

Secondo:

```

1 import numpy as np
2
3 I = np.zeros((3, 3))
4 idx = [0, 1, 2] # lista degli indici
5
6 I[idx, idx] = 1
7
8 v = np.array([4, 8, 2]) # vettore a caso
9 # calcolo prodotto scalare
10 print(v.T @ I @ v)
11 print(sum(v**2))

```

Terzo:

```

1 import numpy as np
2
3 L = np.zeros((4, 4))
4 I = np.zeros((4, 4))
5 B = np.zeros((4, 4))
6 idx = [0, 1, 2, 3] # lista degli indici
7
8 L[idx, idx] = [-1, 1, 1, 1]
9 I = np.eye(4)
10
11 beta = 0.1
12 gamma = 1/np.sqrt(1 - beta**2)
13 B[idx, idx] = [gamma, gamma, 1, 1]
14 B[0, 1] = B[1, 0] = -beta*gamma
15
16 v = np.array([4, 8, 2, 1]) # vettore a caso
17 print(v.T @ L @ v, v.T @ I @ v)
18 w = B @ v
19 print(w.T @ L @ w, w.T @ I @ w)

```

Quarto:

```

1 import numpy as np
2
3 A = np.zeros((4, 2))
4 A[:, 0] = 1
5 print(A)
6 # faccio lo swap cambiando l'ordine degli indici
7 print(A[:, [1, 0]])

```

Quinto:

```

1 import numpy as np
2
3 x = np.linspace(0, 20, 21, dtype=int)
4
5 pari = x[x%2 == 0] # maschera per i pari
6 disp = x[x%2 == 1] # maschera per i dispari
7 print(pari)
8 print(disp)

```

Sesto:

```

1 import numpy as np
2
3 x = np.linspace(0, 20, 21, dtype=int)
4
5 # maschera con due condizioni
6 mask = (x > 5) & (x < 15)
7 # "&" per and mentre "|" per or
8 print(x[mask])

```

Settimo:

```

1 import numpy as np
2
3 x = np.linspace(0, 10, 11, dtype=int)
4 # trasformo il vettore in una matrice N x 1
5 y = x[:, None]
6 print(x)
7 print(y)
8 print(x*y)

```

Ottavo:

```

1 import numpy as np
2
3 decade_iniziale = -3 # da 10^-3
4 decade_finale = 3 # fino a 10^3
5 base = 10 # base della scala
6 numero_punti = 15
7
8 x = np.linspace(decade_iniziale, decade_finale, numero_punti)
9 x_log = base**x
10 n_log = np.logspace(decade_iniziale, decade_finale, numero_punti)
11
12 print(x_log)
13 print(n_log)

```

Nono:

```

1 import numpy as np
2
3 a = np.array([1, 5, 2, 0, 4, 7, 8, 3, 9])
4 b = np.array([2, 4, 6])
5
6 a = np.setdiffid(a, b)
7 print(a)

```

Decimo:

```

1 import numpy as np
2
3 # Ogni entrata della matrice e' una variabile estratta da una distribuzione uniforme
4 Mat = np.random.random((3, 3))
5 print(Mat)
6
7 np.set_printoptions(precision=2)
8 print(Mat)

```

Undicesimo:

```

1 import numpy as np
2
3 x = np.random.random(10)
4 print(x)
5 x[x.argmin()], x[x.argmax()] = 1/7, 7
6 print(x)

```

Dodicesimo:

```

1 import numpy as np
2
3 x = np.array([1, 2, 2, 3, 4, 4, 5])
4 print(x)
5 unique_x = np.unique(x)
6 print(unique_x)

```

Tredicesimo:

```

1 import numpy as np
2
3 x = np.array([1, 2, 3])
4
5 # Creo due matrici che sono la copia
6 # del mio array di partenza ripetuto
7 # tre volte per righe e per colonne
8 X, Y = np.meshgrid(x, x)
9 print(X)
10 print(Y)
11 # Con ravel trasformo le matrici in vettori
12 # e con stack li attacco secondo l'asse
13 # specificato (i.e. 1 fa elemento di x elemento di y
14 # 0 fa tutti elementi x, tutti elementi di y)
15 Z = np.stack((X.ravel(), Y.ravel()), axis=1)
16 print(Z)

```

Quattordicesimo:

```

1 import numpy as np
2
3 x = np.array([1, 2, 1])
4 v = np.array([0, 1, 0])
5 L = np.cross(x, v)
6 print(L)

```

Quindicesimo:

```
1 import numpy as np
2
3 x      = np.array([1, 1])
4 r      = np.linalg.norm(x)
5 theta = np.arctan2(x[1], x[0])
6 pol   = np.array([r, theta])
7 print(pol)
```

Sedicesimo:

```
1 import numpy as np
2
3 A = np.random.rand(4, 3)
4 AT_A = A.T @ A
5 print("Matrice originale:\n", A)
6 print("Matrice trasposta per originale:\n", AT_A)
7 print("Simmetrica?", np.allclose(AT_A, AT_A.T))
```

Diciassettesimo:

```
1 import numpy as np
2
3 theta = np.radians(45) # angolo di rotazione in radianti
4 R = np.array([[np.cos(theta), -np.sin(theta)],
5               [np.sin(theta), np.cos(theta)]])
6
7 v = np.array([1, 0]) # vettore iniziale
8 v_rot = R @ v
9
10 print("Vettore originale:", v)
11 print("Vettore ruotato:", v_rot)
12 print("Lunghezza invariata?", np.isclose(np.linalg.norm(v), np.linalg.norm(v_rot)))
```

Diciottesimo:

```
1 import numpy as np
2
3 arr = np.arange(1, 21)
4 arr[1::2] = arr[1::2] ** 2
5 print(arr)
```

Diciannovesimo:

```
1 import numpy as np
2
3 pos_iniziale = np.array([0, 0, 0])
4 vel = np.array([1, 2, 3]) # velocita' costante
5 t = 10 # tempo
6 pos_finale = pos_iniziale + vel * t
7 print("Posizione finale:", pos_finale)
```

Ventesimo:

```
1 import numpy as np
2
3 x = np.linspace(0, 2 * np.pi, 100)
4
5 matrice_armoniche = np.column_stack((np.sin(x), np.cos(x)))
6 somme_colonne = matrice_armoniche.sum(axis=0)
7
8 print("Somma della colonna sin(x):", somme_colonne[0])
9 print("Somma della colonna cos(x):", somme_colonne[1])
```

Ventunesimo:

```
1 import numpy as np
2
3 arr = np.array([10, 20, 30, 40, 50]) # Esempio di array
4 terzo = arr[2]
5 ultimo = arr[-1]
6 print("Terzo elemento:", terzo)
7 print("Ultimo elemento:", ultimo)
```

Ventiduesimo:

```
1 import numpy as np
2
3 numeri = np.arange(1, 11) # Array di numeri da 1 a 10
4 subarray_primi_5 = numeri[:5]
```

```
5 subarray_ultimi_3 = numeri[-3:]
6 print("Primi 5 elementi:", subarray_primi_5)
7 print("Ultimi 3 elementi:", subarray_ultimi_3)
```

Ventitreesimo:

```
1 import numpy as np
2
3 numeri = np.arange(1, 11) # Array di numeri da 1 a 10
4 subarray_primi_5 = numeri[:5]
5 subarray_ultimi_3 = numeri[-3:]
6 print("Primi 5 elementi:", subarray_primi_5)
7 print("Ultimi 3 elementi:", subarray_ultimi_3)
```

## 7 Terza lezione

### 7.1 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 è None.
18 """
19 def procedura(a):
20     a = a+1
21
22 print(procedura(2))
23
24 """
25 Volendo si possono creare anche funzioni
26 che non hanno valori in ingresso:
27 """
28 def pigreco():
29     return 3.14
30 print(pigreco())
31
32 [Output]
33 12
34 10
35 None
36 3.14
```

Portiamo all'attenzione due fatti importanti:

- È fondamentale in Python che il corpo della funzione sia indentato, per seguire un raggruppamento logico del codice. Lo stesso vale per altri costrutti che vedremo tra poco. 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.

### 7.2 Istruzioni di controllo

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

#### 7.2.1 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

```

### 7.2.2 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 verificata:

```

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 range(1, n+1) = [1,2,..., n]. Il programma effettua l'operazione  $R = R * i$  per tutti i valori possibili che i assume in questa lista, nell'ordine. Da notare il comando range che crea una lista sulla quale iterare, ma noi abbiamo visto già le liste e gli array e abbiamo visto che presentano alcune somiglianze, un'altra somiglianza da far vedere è che entrambi sono 'iterabili' e quindi possiamo iterarci sopra:

```

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.

### 7.3 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.

### 7.4 Funzioni lambda

Esiste un particolare tipo di funzioni in Python, le cosiddette funzioni lambda, sono fondamentalmente uguali alle funzioni dichiarate dall'utente ma devono avere una singola espressione, quindi niente calcoli complicati nel mezzo. La sintassi è: lambda argomenti : espressione. Sono in genere usate come anonime, magari da passare ad altre funzioni ad esempio se bisogna fittare con una retta, (vedere lezione successiva) posso usare curve\_fit così: curve\_fit(lambda x, m, q : x\*m + q, x, y, ...); ma ciò non vieta comunque di dargli un nome.

```

1 f = lambda x : 3*x
2 print(f(2))
3
4 h = lambda x, y, z : x*y + z
5 print(h(2, 3, 4))
6
7 [Output]
8 6
9 10

```

Ovviamente esse possono anche essere usate all'interno di una funzione magari come ciò che la funzione ritorna, il che vuol dire che la funzione restituirà un'altra funzione.

```

1 def g(x):
2     ''' restituisce potenza x-esima di y
3     '''
4     return lambda y : y**x
5
6 G = g(3) # potenza cubica
7 print(G(4))
8
9 =====
10 =====
11
12 def m(f, y, x):
13     ''' passo una funzione ad una funzione
14     '''
15     return f(y) - x
16
17 print(m(lambda x : x**2, 5, 1))
18
19 [Output]
20 64
21 24

```

## 7.5 Sempre più funzioni: i decoratori

Altre funzioni interessanti da analizzare sono i decoratori, si tratta di funzioni che prendono in input altre funzioni e ne cambiano il comportamento. Questo viene fatto tramite un *wrapper*. In sostanza il decoratore è una funzione, che chiama un'altra funzione, cioè il wrapper, che incapsula la chiamata delle funzione su cui stiamo applicando il decoratore fra altre linee di codice, permettendo dunque, grazie alle linee aggiuntive, di modificare come la funzione si comporta senza però andare ad intaccare il codice della funzione originaria. Cominciamo a vedere un semplice esempio:

```

1 def decoratore(func):
2     '''
3     Funzione decoratore
4
5     Parameters
6     -----
7     func : callable
8         funzione da decorare
9
10    Return
11    -----
12    wrapper : callable
13        funzione che incapsula e modifica func
14    '''
15    def wrapper():
16        '''
17        Wrapper del decoratore
18        Ovviamente il nome e' convenzionale
19        '''
20        print("Codice eseguito prima")
21        func()
22        print("Codice eseguito dopo")
23
24    return wrapper
25
26 @decoratore
27 def f():
28     ''' Funzione che vogliamo decorare
29     '''
30     print("Funzione originale che fa cose")
31
32 f()
33
34 [Output]
35 Codice eseguito prima
36 Funzione originale che fa cose
37 Codice eseguito dopo

```

Lo ripetiamo, "wrapper" è un nome convenzionale, potete chiamarla come volete. Vedete che quindi potendo eseguire altro codice oltre alla nostra funzione iniziale, i decoratori ampiano di molto le possibilità di utilizzo delle nostre funzioni. Vediamo come cambia il codice se la nostra funzione, dovesse prendere in input dei parametri.

```
1 def decoratore(func):
```

```

2     ''
3     Funzione decoratore
4
5     Parameters
6     -----
7     func : callable
8         funzione da decorare
9
10    Return
11    -----
12    wrapper : callable
13        funzione che incapsula e modifica func
14    ,,
15    def wrapper(*args, **kwargs):
16        ,
17        Wrapper del decoratore
18        Ovviamente il nome e' convenzionale
19
20        Parameters
21        -----
22        args : tuple
23            argumets of func
24        kwargs : dict
25            keyword argumets of func
26        ,
27        print(f"Eseguendo: {func.__name__} con argomenti {args} {kwargs}")
28        result = func(*args, **kwargs)
29        print(f"Risultato: {result}")
30        return result
31
32    return wrapper
33
34
35 @decoratore
36 def op(a, b, k=1):
37     ,
38     Funzione che implementa somma e sottrazione
39
40     Parameters
41     -----
42     a, b : float
43         valori di cui calcolare somma o sottrazione
44     k : int, optional, default 1
45         flag per passare da somma a sottrazione
46     ,
47     return a + k * b
48
49 op(3, 5, k=-1)
50 op(3, 5)
51
52 [Output]
53 Eseguendo: op con argomenti (3, 5) {'k': -1}
54 Risultato: -2
55 Eseguendo: op con argomenti (3, 5) {}
56 Risultato: 8

```

Tra l'altro questo ci permette di definire un'altra caratteristica interessante delle funzioni. Infatti proprio come con `"*args"`, possiamo passare infiniti argomenti, `"**kwargs"` (che sta per keyword arguments) ci permette di passare infiniti valori come parole chiavi. Vediamo infatti che nella nostra funzione `"op"` è presente una variabile `k` che ha un valore di default, quindi a meno di dargli noi un valore, esso rimane 1. Qualora il valore di default non ci fosse avremo un `"TypeError"`. Usare `"**kwargs"` ci protegge proprio da dover scrivere nella riga dove definiamo la funzione, una serie infinita di variabili e assegnare loro valori di default e aggiornarla poi ad ogni cambiamento; in genere questo si fa con variabili non troppo importanti possiamo dire. Vediamo brevemente come:

```

1 def op(a, b, **kwargs):
2     ,
3     Funzione che implementa le 4 operazioni
4
5     Parameters
6     -----
7     a, b : float
8         valori di input
9
10    Returns

```

```

11 -----
12     float
13         risultato dell'operazione richiesta
14
15 Other Parameters
16 -----
17 type : str, optional, default 'sum'
18     tipo di operazione da eseguire
19     ''
20 # Assegno valore di default a type
21 type = kwargs.get('type', 'sum')
22 if type == 'sum':
23     return a + b
24 elif type == 'sub':
25     return a - b
26 elif type == 'mul':
27     return a * b
28 elif type == 'div':
29     return a / b
30 else :
31     return
32
33 print(op(2, 3, type='mul'))
34
35 [Output]
36 6

```

Come vedete un valore di default dobbiamo sempre darlo, ma adesso così come abbiamo fatto per "type", possiamo definire molti altri argomenti della nostra funzione che andranno passati per nome (i.e. scrivendo quando chiamiamo la funzione nome=valore). Torniamo ai nostri decoratori e vediamo qualche esempio carino. Consideriamo il seguente codice:

```

1 def fibonacci(n):
2     ''' Funzione riconosciuta di fibonacci
3     '''
4     if n <= 1:
5         return n
6
7     return fibonacci(n-1) + fibonacci(n-2)
8
9 start = time.time()
10 print(fibonacci(38))
11 end = time.time() - start
12 print(f"Tempo impiegato: {end:.2f}")
13
14 [Output]
15 39088169
16 Tempo impiegato: 11.97

```

Come vedete il calcolo lo fa ma ci mette un po'. Immagino che questo non vi sorprenda, stiamo facendo tante chiamate della funzione, per di più ogni volta è chiamata due volte. Come facciamo a contare il numero delle chiamate? Beh stiamo parlando di decoratori quindi...

```

1 def callcounter(func):
2     '''
3     Funzione decoratore
4
5     Parameters
6     -----
7     func : callable
8         funzione da decorare
9
10    Return
11    -----
12    wrapper_cc : callable
13        funzione che incapsula e modifica func
14    ''
15    def wrapper_cc(*args):
16        '''
17            Wrapper del decoratore
18
19            Parameters
20            -----
21            args : tuple
22                arguments of func
23

```

```

24     Returns
25     -----
26     valore della funzione calcolata in args
27     ''
28     wrapper_cc.ncalls += 1 # Conta il numero di chiamate
29     return func(*args)
30
31     wrapper_cc.ncalls = 0      # Inizializza il contatore per ogni funzione decorata
32     return wrapper_cc
33
34 @callcounter
35 def fibonacci(n):
36     ''' Funzione ricorsiva di fibonacci
37     '''
38     if n <= 1:
39         return n
40
41     return fibonacci(n-1) + fibonacci(n-2)
42
43 n = 38
44 start = time.time()
45 F = fibonacci(n)
46 end = time.time() - start
47 print(f"F({n}) = {F}")
48 print(f"Tempo impiegato: {end:.2f}")
49 print(f"Numero di chiamate effettuate: {fibonacci.ncalls}")
50
51 [Output]
52 F(38) = 39088169
53 Tempo impiegato: 35.22
54 Numero di chiamate effettuate: 126491971

```

Allora notiamo subito un paio di cose, in primis il tempo impiegato è quasi 3 volte tanto rispetto a prima. Questo è dovuto al fatto che per un po più di 126 milioni di volte ci siamo messi ad aggiornare una variabile. In secundis notiamo che effettivamente la funzione "fibonacci" è chiamata davvero molte volte. Riguardo a questo voglio soffermarmi un attimo su come è scritto il decoratore. Abbiamo detto che il decoratore (nella fattispecie "callcounter") deve ritornare una funzione ("wrapper") che è quella che viene effettivamente eseguita. Per sapere quindi quante chiamate sono state fatte, e per evitare di metter un "print" per ogni chiamata, ne approfittiamo per scoprire l'esistenza di un nuovo tipo di variabili: gli attributi delle funzioni. Sono normalissime variabili, semplicemente ci accediamo usando il punto: "nome\_funzione.nome\_variabile" (gli attributi torneranno meglio quando spiegheremo le classi e la programmazione a oggetti). Però voi vedete bene che la variabile "ncalls" è un attributo della funzione "wrapper"; tuttavia grazie al decoratore noi stiamo eseguendo proprio quest'ultima funzione, e quindi vi possiamo accedere. Se ricordate, nell'esempio precedente dentro la funzione "wrapper" ci stava la seguente riga: "print(f"Eseguendo: func.\_\_name\_\_ con argomenti args kwargs)". La prima variabile stampata è esattamente un attributo della funzione, nella fattispecie quello che ci da il suo nome, e infatti essendo all'interno del "wrapper" veniva stampato "op". Se stampassimo questa stessa variabile però dopo tutto il conto, vedremo che il nome della funzione "fibonacci" per quanto sa l'interprete di Python, non è fibonacci ma wrapper. Comunque detto questo siamo giunti ad un'interessante conclusione: la funzione di fibonacci ricorsiva fa tantissime chiamate. E se vi fermate a pensare un attimo, non solo sono tante, ma sono pure troppe; nel senso che molte delle chiamate sono già state fatte. Il numero di chiamate di "fibonacci(k)" in "fibonacci(n)" è:

$$C(n, k) = C(n - 1, k) + C(n - 2, k), \quad (1)$$

con  $C(n, k) = 1$  se  $n = k$  e  $C(n, k) = 0$  se  $n < k$ . Prendendo ad esempio  $n = 38, k = 5$  si ha:

$$C(38, 5) = 5702887.$$

Abbiamo quindi 5702886 di chiamate inutili, ne basta una, no?. Beh la soluzione è usare un bellissimo decoratore che conservi le chiamate già fatte in una variabile:

```

1 def cache(func):
2     ''
3     Funzione decoratore
4
5     Parameters
6     -----
7     func : callable
8         funzione da decorare
9
10    Return
11    -----
12    wrapper : callable

```

```

13     funzione che incapsula e modifica func
14     ,,
15     risultati = {}
16
17 def wrapper(*args):
18     ,,
19     Wrapper del decoratore
20
21     Parameters
22     -----
23     args : tuple
24         argumets of func
25
26     Returns
27     -----
28     risultato: int
29         valore della funzione calcolata in args
30     ,,
31     if args in risultati:
32
33         return risultati[args]
34     risultato = func(*args)
35     risultati[args] = risultato
36     return risultato
37
38 return wrapper

```

Eseguendo ora:

```

1 # non riportiamo i decoratori scritti sopra
2
3 @callcounter
4 @cache
5 def fibonacci(n):
6     ''' Funzione riconosciuta di fibonacci
7     ,,
8     if n <= 1:
9         return n
10
11    return fibonacci(n-1) + fibonacci(n-2)
12
13 n = 38
14 start = time.time()
15 F = fibonacci(n)
16 end = time.time() - start
17 print(f"F({n}) = {F}")
18 print(f"Tempo impiegato: {end:.2f}")
19 print(f"Numero di chiamate effettuate: {fibonacci.ncalls}")
20 print(f"Nome funzione: {fibonacci.__name__}")
21
22 [Output]
23 F(38) = 39088169
24 Tempo impiegato: 0.00
25 Numero di chiamate effettuate: 75
26 Nome funzione: wrapper_cc

```

Vedete che ora ci impiega molto meno tempo. Ora in verità non stiamo contando le chiamate di "fibonacci", ma le chiamate della funzione wrappata da "@cache"; infatti l'ordine dei decoratori è importante, se li invertissimo avremmo errore provando a stampare il numero di chiamate. Il numero è 75 e non 39, come ci potremmo aspettare, perché la funzione "fibonacci" chiama se stessa due volte, quindi uno potrebbe pensare ci siano 78 chiamate. Le mancanti chiamate sono dovute al fatto che avendo come argomenti  $n - 1$  ed  $n - 2$  le due funzioni arrivano a zero in momenti diversi. Ultimo caso da considerare è come passare argomenti ad un decoratore e vogliamo anche far vedere come evitare che il nome della funzione cambi come abbiamo visto sopra. Questo perché se cambia il nome, allora cambiano anche tutte le altre informazioni importanti come la docstring, e questa non è una cosa auspicabile.

```

1 from functools import wraps
2
3 def ripeti(n):
4     ,,
5     Funzione per passare argomenti al decoratore
6
7     Parameters
8     -----
9     n : int
10        numero di chiamate da fare

```

```

11     Return
12     -----
13     decoratore : callable
14         decoratore
15     '',
16     def decoratore(func):
17         '',
18         Funzione decoratore
19
20     Parameters
21     -----
22     func : callable
23         funzione da decorare
24
25     Return
26     -----
27     wrapper : callable
28         funzione che incapsula e modifica func
29     '',
30
31     # wrappo il wrapper, cosi' le informazioni
32     # originali di func non vanno perse
33     @wraps(func)
34     def wrapper():
35         ''' wrapper, esegue n volte func
36         '',
37         for _ in range(n):
38             func()
39         return wrapper
40     return decoratore
41
42 @ripeti(3)
43 def hello():
44     '',
45     Importante e complessa docstring
46     '',
47     print("Hello World!")
48
49 hello()
50 print(hello.__name__)
51 print(hello.__doc__)
52
53 [Output]
54 Hello World!
55 Hello World!
56 Hello World!
57 hello
58
59     Importante e complessa docstring

```

Vediamo quindi che per passare un argomento la fatica non è tanta, abbiamo solo messo tutto dentro un'altra funzione. Per preservare nome e docstring quello che andiamo a fare è (linea 33) decorare il wrapper con una funzione apposita "wraps" (che tra l'altro prende in input un argomento) di un "functools" che fa parte della libreria standard di Python.

## 7.6 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 verrà saltata
10 """
11
12 #se vogliamo invece che venga letto tutto come una matrice scriviamo:
13 path = 'dati.txt'
14 dati = np.loadtxt(path) # sarebbe unpack = False
15 #dati sarà 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 si capisce senza troppa fatica che usecols indica le colonne che vogliamo leggere
23 skiprows il numero di righe da saltare e delimiter indica il carattere che separa le colonne
24 """
```

Un interessante tipo di file sono i file con estensione ".npy" comodi se i dati sono di dimensioni elevate:

```
1 import time
2 import numpy as np
3
4 x = np.linspace(0, 1, int(2e6)) # dati
5
6 =====
7 # file txt
8 =====
9
10 # salviamo su file
11 start = time.time()
12 path = r'c:\Users\franc\Desktop\dati.txt'
13 f = open(path, 'w')
14 for i in x:
15     f.write(f'{i} \n')
16 f.close()
17 end = time.time() - start
18
19 print(f"tempo di scrittura txt: {end} s")
20
21 #leggiamo da file
22 start = time.time()
23 X = np.loadtxt(path, unpack=True)
24 end = time.time() - start
25
26 print(f"tempo di lettura txt: {end} s")
27
28 =====
29 # file npy
30 =====
31
32 # salviamo su file
33 start = time.time()
34 path = r'c:\Users\franc\Desktop\dati.npy'
35 np.save(path, x)
```

```

36 end = time.time() - start
37
38 print(f"tempo di scrittura npy: {end} s")
39
40 #leggiamo da file
41 start = time.time()
42 X = np.load(path, allow_pickle='TRUE')
43 end = time.time() - start
44
45 print(f"tempo di lettura    npy: {end} s")
46
47 [Output]
48 tempo di scrittura txt: 5.610752582550049 s
49 tempo di lettura    txt: 9.489601373672485 s
50 tempo di scrittura npy: 0.016022205352783203 s
51 tempo di lettura    npy: 0.015637636184692383 s

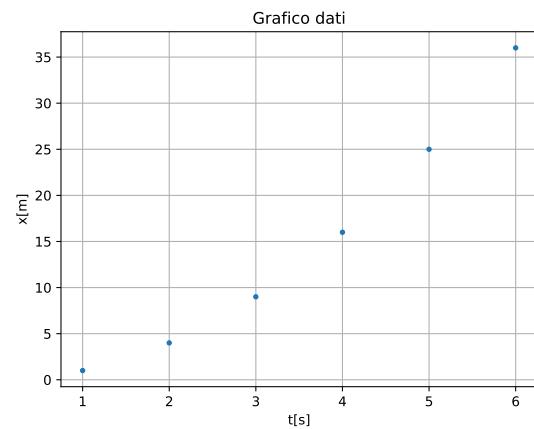
```

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). Come vediamo la differenza di tempi è abissale. Passiamo ora alla creazione di un grafico. 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 leggenda (loc='best' indica che Python la mette dove ritiene più consono, in modo che non rischi magari di coprire porzioni di grafico). Ovviamente è consigliata una lettura della documentazione per conosce tutti gli altri comandi possibili per migliorare/abbellire il grafico da adre alle funzioni già presenti. Altre funzioni utili possono essere: "plt.axis(...)" che imposta il range da visualizzare su entrambi gli assi; il comando "plt.xscale(...)" che permette di fare i grafici con una scala, magari logaritmica o altro sull'asse x (analogo sarà sulle y mutatis mutandis).

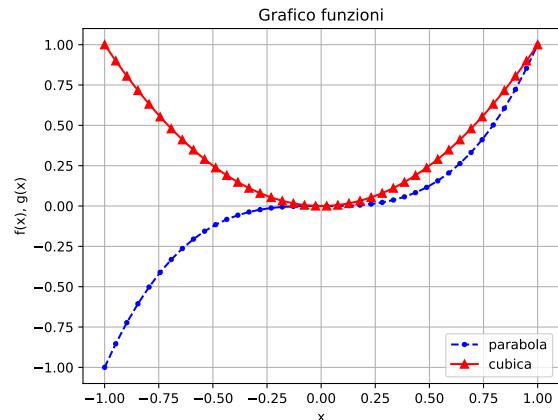
### 7.6.1 Iistogrammi

Altra menzione da fare sono gli istogrammi:

```

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 gaussiana')
18 plt.xlabel('x')
19 plt.ylabel('p(x)')
20
21 """
22 lista di numeri distribuiti gaussianamente con media 0 e varianza 1
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.normal(0, 1) for _ in range(int(1e5))]
29 plt.hist(z, bins=50, density=True, histtype='step')
30 plt.minorticks_on() # tick piccoli sugli assi
31
32 plt.show()

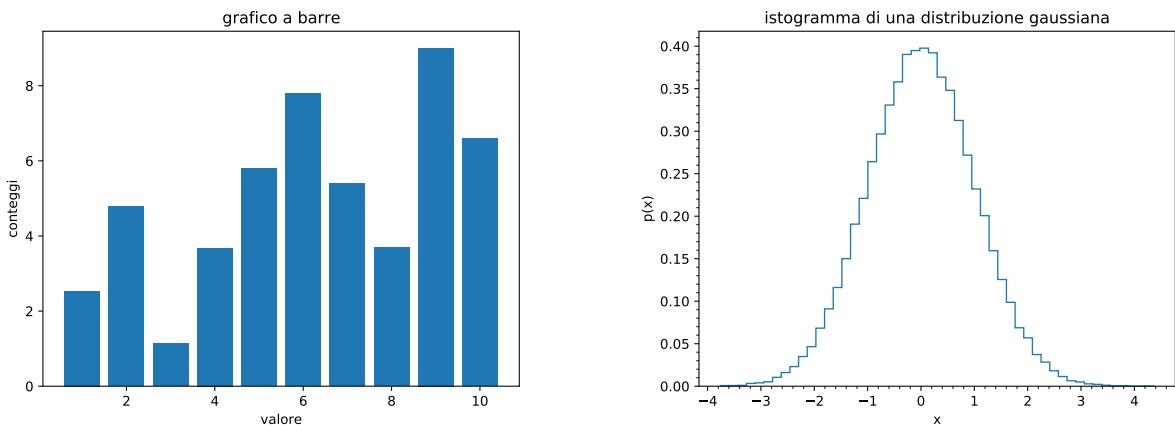
```



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 50 sulla fiducia). Se tale parametro viene fornito come un intero (pari a  $n$  per esempio), suddivide i nostri dati in  $n$  barre equispaziate<sup>2</sup>, tuttavia se gli viene passata una sequenza (una lista o, equivalentemente, una tupla) oppure come un altro tipo di dato (per esempio come strin-

<sup>2</sup>Cosa accade se avete meno dati dei bins inseriti? Banalmente si creano delle colonne vuote

ga) presenta dei comportamenti differenti: invitiamo dunque il lettore a leggere la documentazione relativa a "matplotlib.pyplot.hist".



### 7.6.2 I colori

Ci sta ora da trattare un tema importante. I colori. E no, non è solo una questione di gusto. Tranquilli non andremo a fare una dissertazione per distinguere il bello dal sublime, probabilmente non ne siamo capaci e il sublime ci apparirebbe terribile. Quando andiamo a fare un plot e plottiamo diverse linee, lì possiamo fare un po' come ci pare, Basta un pizzico di buon senso per capire che se ci sono molte linee ci deve essere un buon contrasto, e ricordarsi che ogni tanto che al mondo esistono i daltonici. Quando però non parliamo più di linee ma di superfici le cose in genere si complicano. Perché molto spesso non si fa un plot in 3D, se ne fa uno in 2D e si affida la terza dimensione ad una mappa di colori. capite bene che quindi una scelta sbagliata può farci male interpretare i dati. vediamo subito qualche esempio. Consideriamo la seguente funzione:

$$f(x, y) = e^{-(x^2+y^2)/2}(\sin(3x) + \cos(3y) + 2),$$

Immagino riuscite ad immaginarla ma per semplicità vediamo il plot 3D:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(-2, 2, 1000)
5 y = np.linspace(-2, 2, 1000)
6 X, Y = np.meshgrid(x, y)
7
8 Z1 = np.exp(-(X**2 + Y**2)/2) * (np.sin(3 * X) + np.cos(3 * Y) + 2)
9
10 fig = plt.figure()
11 ax = fig.add_subplot(111, projection='3d')
12 ax.plot_surface(X, Y, Z1)
13 plt.show()
```

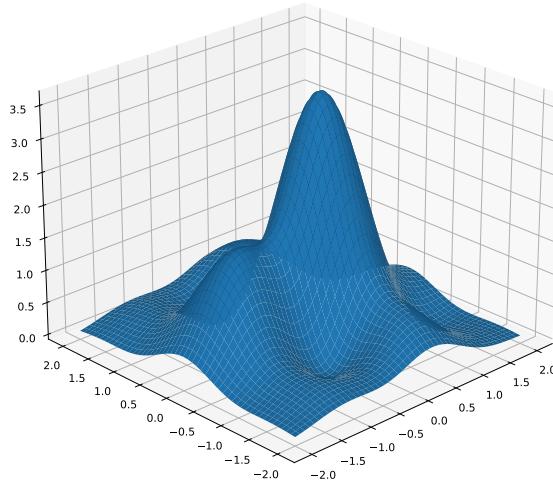


Figura 1: Una funzione intenta a funzionare

Adesso vi mostro la stessa funzione ma con un plot 2D con due colormap diverse:

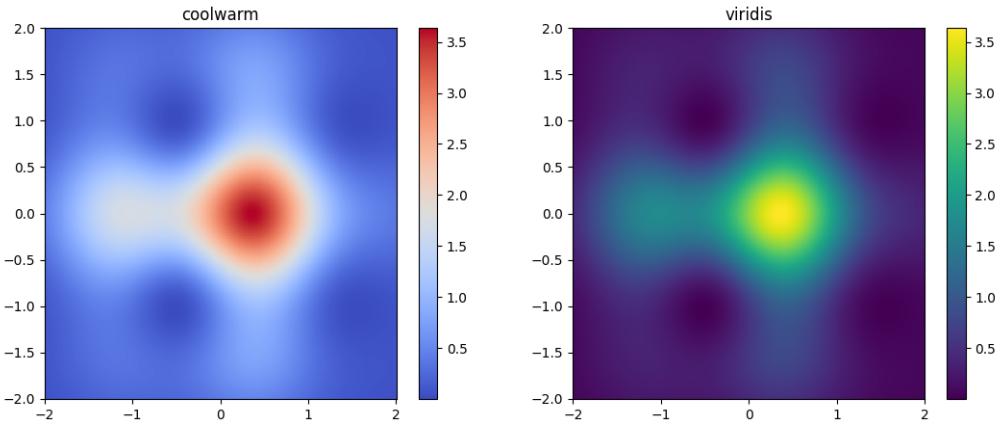


Figura 2: Funzione precedente plottata in 2D con due colormap diverse (titolo del grafico).

Dunque, secondo voi: una delle due colormap è meglio dell'altra? Se si, quale? Beh partiamo prima di tutto dalla nostra funzione, vediamo che sembra ragionevolmente continua ed è sempre maggiore di zero. Quindi la nostra colormap deve farci intuire queste caratteristiche. Vediamo il primo grafico 2D, quello con la mappa "coolwarm"; vediamo subito che si passa da blu a rosso, con uno stacco di bianco. Questo cambio di colori ci colpisce subito, potremmo pensare che zone rosse e blu ci sia forse un avvallamento o simili. Il bianco delinea in maniere abbastanza netta una separazione fra le due regioni. Con la "viridis" invece questa cosa non si verifica; è tutto molto più liscio, la transizione è lineare fra un colore scuro che pian piano sfuma in uno più chiaro e acceso. La prima mappa è chiamata *divergente*, mentre la "viridis" è una mappa *percettiva uniforme sequenziale*. Vete che quindi, già dal nome, la prima è migliore nei casi in cui i dati si stiano allontanando da un punto centrale, ad esempio lo zero, mentre la seconda per casi di dati totalmente positivi o negativi con che variano con continuità. Vediamo ora un altro caso:

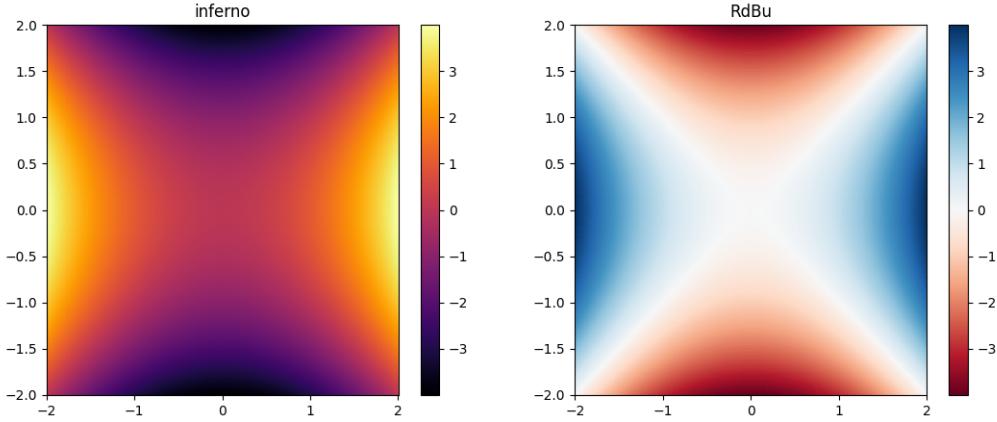


Figura 3: Funzione  $f(x, y) = x^2 - y^2$  plottata in 2D con due colormap diverse (titolo del grafico).

La funzione è una tranquillissima:  $f(x, y) = x^2 - y^2$ . Stesse domande di prima: chi è meglio e perché? Le due mappe sono dello stipo di sopra, la "inferno" è come la "viridis" e "RdBu" e come "coolwarm". Immagino che data l'inversione delle due non sia difficile immaginare quella giusta. Infatti ora avendo una funzione a valori sia negativi che positivi la mappa "inferno" ci rende più difficile intuire questo passaggio, mentre la "RdBu" con il suo forte stacco bianco fra il rosso e il blu ci fa ben distinguere le due zone di interesse. Ora di mappe di colori ce ne sono varie e se ne potrebbe parlare molto, interessante è la pagina di matplotlib in cui trovate l'argomento esposto in maniera più dettagliata: <https://matplotlib.org/stable/users/explain/colors/colorbars.html>. Trattiamo adesso un ultimo esempio:

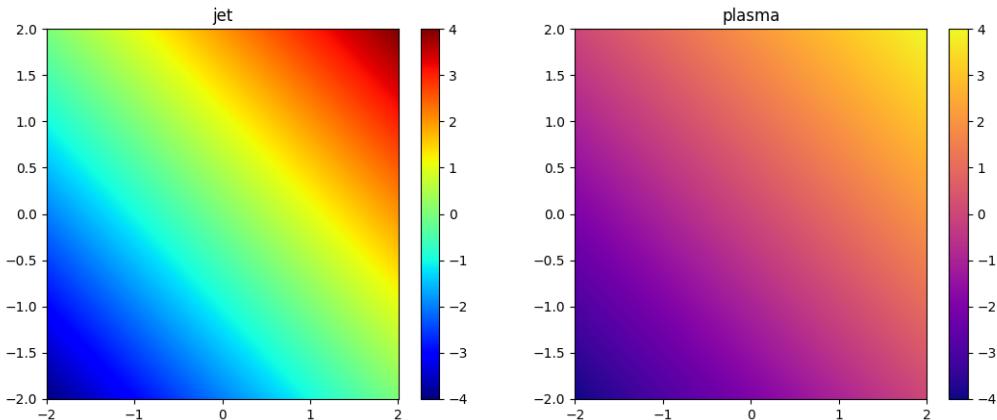


Figura 4: Funzione  $f(x, y) = x + y$  plottata in 2D con due colormap diverse (titolo del grafico).

Sta volta la domanda neanche ve la faccio, la "jet" o "rainbow" la possiamo prendere e toglierla dalla nostra memoria abbastanza tranquillamente. Stiamo plottando un piano nulla di troppo astruso, e la "jet" ci mostra zone di vari colori facendoci intuire che stia succedendo qualcosa, che ci siano dei cambiamenti repentini, quando in realtà in realtà è tutto lineare. La mappa "plasma", della stessa categoria di "inferno" e "viridis" invece ci fa capire molto meglio quale sia il trend dei dati. Quindi mi raccomando, quando scegliete una colormap va fatto *cum grano salis*, per riuscire a creare un grafico che sia ben leggibile e chiaro.

## 7.7 Standard input

È (a proposito, la È si fa premendo Alt+0200 sul tastierino, quantomeno su windows) inoltre possibile dare al codice che abbiamo scritto degli input da shell. Esistono due modi per farlo: l'uso della funzione "input()", oppure utilizzare "argparse" per dare al codice ciò che serve direttamente da linea di comando su shell, ad esempio se fossimo su una partizione linux senza un editor stile pyzo. Cominciamo con la prima possibilità:

```

1 """"
2 Programma per calcolare il trinagolo di tartaglia

```

```

3 """
4 import numpy as np
5
6 # leggo da input un valore e lo rendo intero
7 # la stringa verrà stampata su shell
8 n = int(input("Ordine del triangolo: "))
9 a = np.zeros((n, n), dtype=int) # matrice per i coefficienti
10
11 # calcolo i coefficienti del triangolo
12 a[0,0] = 1
13 for i in range(1, n):
14     a[i, 0] = 1
15     for j in range(1, i):
16         a[i, j] = a[i-1, j-1] + a[i-1, j]
17     a[i, i] = 1
18
19 # stampo a schermo
20 for i in range(n):
21     for j in range(i+1):
22         # solo per fare la forma a piramide
23         if j == 0 : # non funziona con numeri a due cifre
24             print(*[""]*(n-i), a[i, j], end=',')
25         else:
26             print("", a[i, j], end=',')
27     print()
28
29 [Output]
30 Ordine del triangolo: 5
31     1
32     1 1
33     1 2 1
34     1 3 3 1
35     1 4 6 4 1

```

Vediamo ora come usare argparse. Ora però il codice è più comodo eseguirlo su una shell, che sia quella di anaconda o quella della vostra distro linux è uguale. Le modifiche al codice sono veramente poche:

```

1 """
2 Programma per calcolare il triangolo di tartaglia
3 """
4 import argparse
5 import numpy as np
6
7 description='Programma per calcolare il triangolo di tartaglia leggendo le informazioni da
8     linea di comando'
9 # descrizione accessibile con -h su shell
10 parser = argparse.ArgumentParser(description=description)
11 parser.add_argument('dim', help='Dimensione della matrice, ovvero potenza del binomio')
12 args = parser.parse_args()
13 n = int(args.dim) # accedo alla variabile tramite il nome messo a linea 10
14
15 a = np.zeros((n, n), dtype=int) # matrice per i coefficienti
16
17 # calcolo i coefficienti del triangolo
18 a[0,0] = 1
19 for i in range(1, n):
20     a[i, 0] = 1
21     for j in range(1, i):
22         a[i, j] = a[i-1, j-1] + a[i-1, j]
23     a[i, i] = 1
24
25 # stampo a schermo
26 for i in range(n):
27     for j in range(i+1):
28         # solo per fare la forma a piramide
29         if j == 0 : # non funziona con numeri a due cifre
30             print(*[""]*(n-i), a[i, j], end=',')
31         else:
32             print("", a[i, j], end=',')
33     print()

```

Vi metto uno screen della shell per capire cosa è successo (un po' sgranata ma pazienza):

```

C:\Windows\System32\Windc > + -
(base) PS C:\Users\franc\Desktop\Nuova cartella\3 Terza Lezione\codiciL3> python tartaglia_argparse.py -h
usage: tartaglia_argparse.py [-h] dim

Programma per calcolare il trinagolo di tartaglia leggendo le informazioni da linea di comando

positional arguments:
  dim            Dimensione della matrice, ovvero potenza del binomio

optional arguments:
  -h, --help    show this help message and exit
(base) PS C:\Users\franc\Desktop\Nuova cartella\3 Terza Lezione\codiciL3> python tartaglia_argparse.py 5
   1
  1 1
 1 2 1
 1 3 3 1
 1 4 6 4 1
(base) PS C:\Users\franc\Desktop\Nuova cartella\3 Terza Lezione\codiciL3>

```

## 7.8 Prestazioni: *pure Python* vs librerie

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 conterrà il risultato
9 N = int(5e6) # numero di iterazioni da fare = 5 x 10**6
10
11 #faccio il conto a 'mano'
12 for i in range(N):
13     a1 += np.sqrt(i)
14
15 #finisco di misurare il tempo
16 end = time.time()-start
17
18 print(end)
19
20 #inizio a misurare il tempo
21 start = time.time()
22
23 #stesso conto ma fatto tramite le librerie di python
24 a2 = sum(np.sqrt(np.arange(N)))
25
26 #finisco di misurare il tempo
27 end = time.time()-start
28
29 #sperabilmente sara' minore del tempo impiegato prima
30 print(end)
31
32 [Output]
33 11.588378429412842
34 0.8475463390350342

```

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

## 7.9 Prestazioni: globale vs locale

Dedicato a Mattia che non vuole usare le funzioni. Rimanendo però nell'ambito di *pure Python*, si può comunque migliorare le prestazioni del codice. Infatti un codice scritto all'interno di una funzione esegue più velocemente rispetto alle stesse righe scritte in globale.

```

1 import timeit
2
3 start = timeit.default_timer()
4
5 for i in range(int(1e8)):
6     pass
7
8 end = timeit.default_timer() - start
9

```

```

10 print(f"Tempo in gloable = {end}")
11
12 def f():
13     for i in range(int(1e8)):
14         pass
15
16 start = timeit.default_timer()
17 f()
18 end = timeit.default_timer() - start
19
20 print(f"Tempo in locale = {end}")
21
22 [Output]
23 Tempo in gloable = 2.915754556655884
24 Tempo in locale = 1.5078275203704834

```

Se prima la scusa era l'utilizzo delle librerie scritte in C ora dov'è la magagna? Adiamo a fare una cosa molto poco capibile, disassembliamo il codice. Ovvero vediamo il bytecode corrispondente al nostro codice. Infatti l'interprete di Python è una macchina virtuale che esegue il bytecode. Per farlo usiamo la libreria "dis".

```

1   5          0 LOAD_NAME               0 (range)
2       2 LOAD_NAME               1 (int)
3       4 LOAD_CONST              0 (100000000.0)
4       6 CALL_FUNCTION           1
5       8 CALL_FUNCTION           1
6      10 GET_ITER
7      >> 12 FOR_ITER              2 (to 18)
8      14 STORE_NAME              2 (i)
9
10    6      16 JUMP_ABSOLUTE        6 (to 12)
11
12    5      >> 18 LOAD_CONST            1 (None)
13    20 RETURN_VALUE

```

```

1   14         0 LOAD_GLOBAL             0 (range)
2       2 LOAD_GLOBAL             1 (int)
3       4 LOAD_CONST              1 (100000000.0)
4       6 CALL_FUNCTION           1
5       8 CALL_FUNCTION           1
6      10 GET_ITER
7      >> 12 FOR_ITER              2 (to 18)
8      14 STORE_FAST              0 (i)
9
10    15      16 JUMP_ABSOLUTE        6 (to 12)
11
12    14      >> 18 LOAD_CONST            0 (None)
13    20 RETURN_VALUE
14 None

```

Le righe sopra riportate sono rispettivamente il bytecode delle righe 5 e 6 del codice soprastante prima e della funzione f poi. Vedete che salta subito all'occhio una differenza: per il codice globale alla riga 8 del bytecode c'è scritto STORE\_NAME, mentre per la funzione abbiamo STORE\_FAST. Cosa vuol dire quindi questa differenza? Al di là del fast del nome il punto è che in una funzione le variabili locali vengono salvate in un array di dimensione fissa, non in un dizionario (come avviene con quelle globali). Per cui ci si accede direttamente tramite un indice, rendendo l'operazione molto rapida. Ricordando la scrittura in C a livello base di Python, si tratta semplicemente di una ricerca del puntatore nell'elenco e di un aumento del conteggio dei riferimenti di quello che è la lista, ovvero un PyObject al livello di C, entrambe operazioni altamente efficienti. Le variabili globali, invece vengono memorizzate in un dizionario. Per cui quando si accede a una variabile globale, Python deve eseguire una ricerca nella tabella hash, che implica il calcolo di un hash e quindi il recupero del valore ad esso associato (non sto adesso a spiegarvi cos'è una tabella hash ma come penso possiate intuire non è altro che una struttura dati a livello di C che permette di utilizzare una corrispondenza chiave-valore). E tutto ciò è più lento.

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

```

Abbiamo fatto una cosa molto brutta, nemmeno Dio può dividere per zero (al più possiamo appellarcia 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:

### 7.10.1 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

```

Anche qui è fondamentale indentare il blocco delle istruzioni.

### 7.10.2 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)
25 Exception: il valore scelto risulta insensato in quanto nulla supera 7, misura massima di ogni
cosa

```

## 7.11 Logging

Supponiamo voi abbiate un certo codice, che fa delle certe cose. Per verificare che tutto stia andando bene quello che in genere facciamo e mettere dei "print" in giro per il codice facendoci stampare qualche quantità per vedere se il suo valore sia qualcosa dotato di una parvenza di senso. Possiamo però anche fare qualcosa di un pochetto più sofisticato. Avete presente che quando scrivete in latex, dopo aver compilato, vi compaiono diversi file? Avete mai notato che uno di questi ha l'estensione ".log"? Quel file contiene tutte le informazioni di ciò che è successo durante la compilazione, che siano errori, warning o che tutto sia andato liscio. La libreria

”logging” ci per mette di fare una cosa analoga; e questo ci aiuta nella gestione degli errori, nel debug, nel verificare che tutto funzioni come deve. Iniziamo con il dire che in logging esistono 5 livelli:

- DEBUG: quando servono informazioni dettagliate per fare diagnostica.
- INFO: quando vogliamo conferma che tutto sta andando come deve.
- WARNING: quando succede qualcosa di non molto grave, il codice comunque può continuare ad andare.
- ERROR: c’è un errore, qualche comando non funziona.
- CRITICAL: veramente grave il codice non può continuare a funzionare.

```

1 import logging
2
3 # configuro il formato del logging; voglio sapere: il tempo, il nome del logger, il livello, e
4 # il messaggio
5 logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')
6
7 def divisione(x, y):
8     """ Funzione particolarmente complicata
9     """
10    return x / y
11
12 # codice particolarmente complicato
13 x_1 = 17
14 x_2 = 4
15
16 quoziante = divisione(x_1, x_2)
17
18 # controllo che tutto sia andato bene via print
19 print(f"{x_1} / {x_2} = {quoziante}")
20
21 # Controllo che tutto sia andato bene via logging, la stringa sara' il messaggio
22 logging.info(f"{x_1} / {x_2} = {quoziante}")
23
24 [Output]
25 17 / 4 = 4.25
26 2024-01-11 14:42:43,998 - root - INFO - 17 / 4 = 4.25

```

Vediamo che succede. Cominciamo con il dire che dei 5 livelli elencati prima, quello di default è WARNING. Il che implica che eventuali ”logging.debug/logging.info” non verebbero stampati. A linea 4 noi settiamo il livello ad INFO, in modo che ”logging.info” funzioni. Specifichiamo poi un formato del messaggio con alcune caratteristiche che possono essere di nostro interesse. Possiamo anche aggiungere altre informazioni che volendo trovate sulla documentazione. Se vogliamo rimandare l’output su file basta aggiungere due voci a ”basic.Config”:

```

1 import time
2 import logging
3
4 # configuro il formato del logging; voglio sapere:
5 # il tempo, il nome del logger, il livello, e il messaggio
6 # Inoltre piuttosto che su shell deve essere stampato su un file,
7 # che deve essere sovrascritto ad ogni esecuzione del codice
8 logging.basicConfig(level=logging.INFO,
9                     format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
10                    filename="file.log", filemode="w") # filemode di default e' a (append)
11
12 def divisione(x, y):
13     """ Funzione particolarmente complicata
14     """
15    return x / y
16
17 # codice particolarmente complicato
18 for i, j in zip(range(15, 30), range(1, 16)):
19
20    quoziante = divisione(i, j)
21
22    # Controllo che tutto sia andato bene via logging, la stringa sara' il messaggio
23    logging.info(f"{i} / {j} = {quoziante}")
24    time.sleep(1)

```

Avrete un file di questo tipo:

```

1 2024-01-11 15:01:27,827 - root - INFO - 15 / 1 = 15.0
2 2024-01-11 15:01:28,829 - root - INFO - 16 / 2 = 8.0
3 2024-01-11 15:01:29,830 - root - INFO - 17 / 3 = 5.6666666666666667
4 2024-01-11 15:01:30,831 - root - INFO - 18 / 4 = 4.5
5 2024-01-11 15:01:31,833 - root - INFO - 19 / 5 = 3.8

```

```

6 2024-01-11 15:01:32,835 - root - INFO - 20 / 6 = 3.3333333333333335
7 2024-01-11 15:01:33,837 - root - INFO - 21 / 7 = 3.0
8 2024-01-11 15:01:34,837 - root - INFO - 22 / 8 = 2.75
9 2024-01-11 15:01:35,839 - root - INFO - 23 / 9 = 2.5555555555555554
10 2024-01-11 15:01:36,841 - root - INFO - 24 / 10 = 2.4
11 2024-01-11 15:01:37,843 - root - INFO - 25 / 11 = 2.272727272727273
12 2024-01-11 15:01:38,844 - root - INFO - 26 / 12 = 2.1666666666666665
13 2024-01-11 15:01:39,846 - root - INFO - 27 / 13 = 2.076923076923077
14 2024-01-11 15:01:40,847 - root - INFO - 28 / 14 = 2.0
15 2024-01-11 15:01:41,849 - root - INFO - 29 / 15 = 1.9333333333333333

```

Quindi capite bene che è un ottimo modo per verificare il funzionamento del codice. Magari se il codice richiede tempo è meglio che l'output sia su shell. Per finire supponiamo che voi abbiate due codici che usate insieme, importando uno nell'altro magari (vedere l'inizio della prossima lezione), e che entrambi usino il logging. Utilizzare semplicemente "logging.basicConfig" potrebbe causare problemi in quanto non si specifica il nome del logger, ed entrambi hanno root di default. Quindi in sostanza potrebbe capitare che su shell, o su file, solo le informazioni di un codice vengano scritte. Vediamo quindi un altro modo di settare il logger.

```

1 import time
2 import logging
3
4 # Creo il logger, __name__ e' il nome del codice ed e' __main__ se viene eseguito, __module__
5 # se importato
6 logger = logging.getLogger(__name__)
7
8 # Settiamo il livello ad INFO, vogliamo controllare che vada tutto bene
9 logger.setLevel(logging.INFO)
10
11 # Settiamo il formato del messaggio di log
12 formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
13
14 # Creiamo il gestore del file
15 file_handler = logging.FileHandler('error.log', mode="w")
16 file_handler.setLevel(logging.ERROR) # Settiamo un livello diverso per il file
17 file_handler.setFormatter(formatter) # sul file ci saranno solo i messaggi di errore
18
19 # Vogliamo vedere tutto anche su shell
20 stream_handler = logging.StreamHandler()
21 stream_handler.setFormatter(formatter)
22
23 # Aggiungiamo tutto al nostro logger
24 logger.addHandler(file_handler)
25 logger.addHandler(stream_handler)
26
27 #===== main del codice =====
28
29 def divisione(x, y):
30     """ Funzione particolarmente complicata
31     """
32     try :
33         q = x / y
34     except ZeroDivisionError:
35         # logger.error restituisce solo il messaggio scritto da noi
36         #logger.error('Qualcosa e' andato storto si stava per verificare il second impact')
37
38         # logger.exception ci resistuisce anche tutto il traceback
39         logger.exception("Qualcosa stava andando storto si stava per verificare il second
40         impact")
41     else :
42         return q
43
44 # codice particolarmente complicato
45 for i, j in zip(range(15, 30), range(0, 15)):
46
47     quoziente = divisione(i, j)
48
49     logger.info(f"{i} / {j} = {quoziente}")
50     time.sleep(1)
51
52 [Output]
53 2024-01-11 22:05:25,178 - __main__ - ERROR - Qualcosa stava andando storto si stava per
54     verificare il second impact
55 Traceback (most recent call last):
      File "/home/francesco/GitHub/4BLP/3 Terza Lezione/logging3.py", line 32, in divisione

```

```

56     q = x / y
57 ZeroDivisionError: division by zero
58 2024-01-11 22:05:25,178 - __main__ - INFO - 15 / 0 = None
59 2024-01-11 22:05:26,180 - __main__ - INFO - 16 / 1 = 16.0
60 2024-01-11 22:05:27,181 - __main__ - INFO - 17 / 2 = 8.5
61 2024-01-11 22:05:28,182 - __main__ - INFO - 18 / 3 = 6.0
62 2024-01-11 22:05:29,183 - __main__ - INFO - 19 / 4 = 4.75
63 2024-01-11 22:05:30,184 - __main__ - INFO - 20 / 5 = 4.0
64 2024-01-11 22:05:31,185 - __main__ - INFO - 21 / 6 = 3.5
65 2024-01-11 22:05:32,187 - __main__ - INFO - 22 / 7 = 3.142857142857143
66 2024-01-11 22:05:33,188 - __main__ - INFO - 23 / 8 = 2.875
67 2024-01-11 22:05:34,190 - __main__ - INFO - 24 / 9 = 2.6666666666666665
68 2024-01-11 22:05:35,192 - __main__ - INFO - 25 / 10 = 2.5
69 2024-01-11 22:05:36,194 - __main__ - INFO - 26 / 11 = 2.3636363636363638
70 2024-01-11 22:05:37,196 - __main__ - INFO - 27 / 12 = 2.25
71 2024-01-11 22:05:38,197 - __main__ - INFO - 28 / 13 = 2.1538461538461537
72 2024-01-11 22:05:39,199 - __main__ - INFO - 29 / 14 = 2.0714285714285716

```

Vediamo quindi che ora il logger si chiama `__main__` e che tutto viene stampato correttamente su shell. Notiamo che c'è un errore ma il codice poi continua ad eseguire, quindi magari il messaggio sparirà poi, però noi lo abbiamo scritto anche sul file. Quindi il file "error.log" contiene le prime righe dell'output, così possiamo andare a vedere ne capire che succede e poi risolvere in caso. Giusto per chiarire: noi abbiamo scritto `"logging.getLogger(__name__)"` per distinguere tra codice ed eventuale modulo, ma nulla ci vieta di chiamare il logger in modo diverso (magari se abbiamo più codici e vogliamo distinguere la sorgente del messaggio); basta passargli una stringa: `"logging.getLogger('Ajeje')"`.

## 7.12 Generatori

Abbiamo visto che strutture dati quali liste, tuple, array eccetera sono degli iterabili e possiamo quindi usarli per scorrierci sopra. Tutto ciò può, però essere fatto con i generatori. Ovvero delle funzioni che piuttosto avere la keyword "return" hanno "yield". Scrittà così, una funzione dopo aver fatto i suoi conti non termina ma viene sospesa, finchè non servirà chiamarla nuovamente per ottenere il valore successivo. Queste funzioni si chiamano appunto funzioni generatore o generatori iteratori. Per accedere agli elementi che essa restituisce possiamo ciclarci sopra oppure usare la funzione "next()" che vedremo più avanti. Vediamo un piccolo esempio:

```

1 """
2 Codice esempio di utilizzo yield per un generatore
3 """
4
5 def generatore():
6     ''' funzione generatore
7     '''
8     yield 1
9     yield 2
10    yield 3
11
12 gen = generatore()
13
14 for val in gen:
15     print(val)
16
17 [Output]
18 1
19 2
20 3

```

Vedete che quindi è come fosse uno di quei range che usiamo in genere nei cicli for. Se infatti provate a stampare la variabile "gen" vedrete che essa non stamperà: 1, 2, 3. Vediamo ora appunto il confronto con range:

```

1 """
2 Codice esempio di utilizzo yield e confronto con range
3 """
4
5 def generatore(N):
6     '''
7     Funzione generatore
8
9     Parameter
10    -----
11    N : int
12        limite fino a cui arriviare
13    '''
14    n = 0
15    while n < N:

```

```

16     yield n
17     n += 1
18
19 for val_g, val_r in zip(generatore(10), range(10)):
20     print(val_g, val_r)
21
22 [Output]
23 0 0
24 1 1
25 2 2
26 3 3
27 4 4
28 5 5
29 6 6
30 7 7
31 8 8
32 9 9

```

Vedete che le due funzioni si comportano allo stesso modo. Ma perchè usare le funzioni con "yield"? Il motivo sta in quello che abbiamo detto prima, la funzione non ritorna una lista o altro di simile contennente tutti i valori che ci interessano ma ci da un solo valore alla volta senza terminare l'esecuzione della funzione ma semplicemente mettendola in pausa. Vedete che quindi ciò permette ai nostri codici di allocare molta meno ram. Vediamo un effettivo esempio di quanta ram si utilizzi in questi contesti:

```

1 """
2 Codice per verificare il consumo di ram
3 """
4
5 import sys
6
7 N = int(2e8) # quanti numeri generare 2 x 10**8
8
9 def gen(N):
10     """
11         Funzione generatore
12
13     Parameter
14     -----
15     N : int
16         limite fino a cui arriviare
17     """
18     for i in range(N):
19         yield i
20
21 # Conservo una tutto in una lista
22 n_l = list(gen(N))
23
24 # Definisco semplicemente il generatore
25 n_g = gen(N)
26
27 def size(obj):
28     """
29         Funzione per calcolare tutta la memoria di un oggetto
30
31     Parameter
32     -----
33     obj : list, tuple, set, dict
34         python object
35     """
36     Size = sys.getsizeof(obj)
37
38     # se e' una lista tupla o set
39     if isinstance(obj, (list, tuple, set)):
40         for el in obj:
41             Size += size(el)
42
43     # se e' un dizionario devo considerare sia chiave che valore
44     if isinstance(obj, dict):
45         for k, v in obj.items():
46             Size += size(k)
47             Size += size(v)
48
49     return Size
50
51 print(f"Dimensione della lista: {size(n_l)} byte")
52

```

```

53 del n_l # elimino la variabile dalla memoria in quanto incredibilmente pesante
54
55 print(f"Dimensione del generatore: {size(n_g)} byte")
56
57 [Output]
58 Dimensione della lista: 7293045240 byte
59 Dimensione del generatore: 208 byte

```

Analizziamo il codice che abbiamo appena mostrato: abbiamo definito la nostra funzione generatore come nel programma precedente e gli passiamo un valore fino a cui iterare molto grande:  $2 \times 10^8$ . La variabile `n_g` è semplicemente il generatore che possiamo usare come sopra, mentre in `n_l`, grazie all'utilizzo della funzione `list`, stiamo conservando una lista contenente tutti i numeri su cui potremmo voler iterare (da 0 a  $2 \times 10^8$ ) quindi stiamo allocando dello spazio. La funzione "size" ci dice quanto pesa un oggetto nella sua totalità. Vediamo quindi che la lista pesa 7.2 Gigabyte mentre il generatore sono pochi byte, ma entrambe ci permettono di fare le stesse cose. Quindi ecco, vedete che abbiamo un enorme risparmio di ram. Un uso di tali funzioni può essere utile se magari stiamo leggendo riga per riga un file di grosse dimensioni. Impariamo poi una nuova keyword ovvero "del" che ci permette di eliminare una variabile. Qui viene usata per pulire tutta quella ram allocata ed evitare di bloccarmi il pc. Poi come il mio computer sia riuscito a gestire 7.2 giga in ram con una ram da 6 è dovuto alle magie di Zram, cercate se vi interessa.

### 7.12.1 Problema delle 8 regine

Vediamo un'applicazione carina dell'utilizzo di "yield". Il problema delle otto regine consta nel trovare le disposizioni possibili in cui collocare 8 regine in una scacchiera  $8 \times 8$  senza che esse si minaccino vicenda. Vediamo come tutto ciò può essere fatto facilmente grazie a "yield":

```

1 """
2 Codice per risolvere il problema delle N regine in una scacchiera N x N
3 """
4 import matplotlib.pyplot as plt
5
6
7 def queens(n, i=0, col=[], diag_sup=[], diag_inf=[]):
8     """
9         Codice che genera le configurazioni delle N regine,
10        la funzione e' un generatore quindi non restituisce
11        tutte le configurazioni ma le calcola man mano
12
13    Parameters
14    -----
15    n : int
16        quante regine vanno piazzate
17    i : int
18        numero di regine piazzate
19    col : list
20        lista che contiene la posizione della regina
21        nelle varie colonne.
22    diag_sup : list
23        lista per controllare le diagonali in salita
24    diag_inf : list
25        lista per controllare le diagonali in discesa
26    ...
27
28    # Finche' non sono piazzate N regine
29    if i < n:
30        # Ciclo sulle posizioni
31        for j in range(n):
32            # se la regina non e' nella colnna e non ci stanno delle regine
33            # a minacciare la casella lungo le diagonali, chiamo ricorsivamente
34            if j not in col and i + j not in diag_sup and i - j not in diag_inf:
35                # richiamo avendo fissato una regina nella posizione j
36                yield from queens(n, i + 1, col + [j], diag_sup + [i + j], diag_inf + [i - j])
37    else:
38        yield col
39
40
41 def plot_queens(board):
42     """
43         Funzione per plottare la scacchiera
44
45     Parameter
46     -----
47     board : list

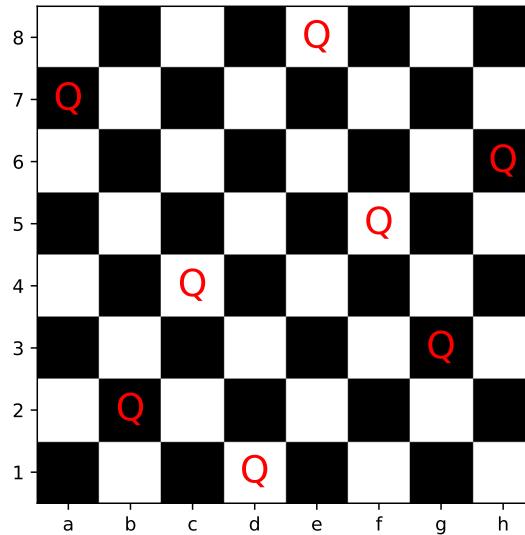
```

```

48     lista delle posizioni, output della funzione queens
49     '',
50     n = len(board)
51
52     # Creo la scacchiera
53     chessboard = [[(i + j) % 2 for i in range(n)] for j in range(n)]
54     plt.imshow(chessboard, cmap='binary')
55
56     # Metto le regine
57     for i in range(n):
58         plt.text(i, board[i], 'Q', color='red', ha='center', va='center', fontsize=20)
59
60     plt.yticks(range(n), [i for i in range(n, 0, -1)])
61     plt.xticks(range(n), [chr(i) for i in range(97, 97+n)])
62     plt.show()
63
64
65 def select(N):
66     '',
67     funzione per selezionare l'N-esima configurazione
68     '',
69     Q = queens(8)
70     for i in range(N-1):
71         next(Q)
72
73     return next(Q)
74
75 B = select(11)
76 plot_queens(B)

```

Vediamo qui inoltre l'utilizzo della funzione "next" a cui accennavamo prima. Per selezionare la configurazione di nostro interesse mandiamo avanti con "next" lo stato del generatore per quanto ci interessa. Vediamo anche il plot che è carino.



## 7.13 Esercizi

Anche qui voglio lasciarvi qualche esercizio per farvi prendere confidenza con quanto imparato finora. Ci saranno però delle modifiche. Infatti di alcuni esercizi non vi darò la soluzione ma solo un file che testa la vostra proposta di soluzione. Tale test vi darà un paio di info sul vostro codice, ad esempio la memoria che esso utilizza, il tempo che impiega, ed anche implementato il test tramite pylint. Si tratta di un pacchetto molto interessante che analizza la sintassi del vostro codice secondo quelle che sono le regole canoniche di scrittura, vi evidenzia gli errori e vi da una valutazione. Mi raccomando se nel testo dell'esercizio vi dico di chiamare una funzione in un certo modo, fatelo, così il codice test può fare i confronti del caso. Tale file di test sarà un codice python chiamato "test.py" che potete eseguire normalmente evi si aprirà la finestra. I più coraggiosi posso cimentarsi nel leggere e capire tale codice volendo "*audentes fortuna iuvat*". Vi faccio vedere sotto, alla

soluzione del primo esercizio, come appare la finestra del test.

- Scrivere una funzione "area(l, n)" che calcoli l'area di un poligono regolare di lato "l" e numero di lati "n". Scrivere poi una funzione "pitagora(lista.lati, n)" che prenda in input una lista di tre elementi (una terna pitagorica) e un numero di lati "n" e che restituisca la somma delle aree delle figure sui cateti a cui sottraete l'area della figura sull'ipotenusa. Fate lo magari per vari valori. Di seguito una formula che potrebbe tornarvi utile:

$$A = \frac{1}{2}(nl) \frac{l \cot(\pi/n)}{2}$$

- Fare con un ciclo più plot su uno stesso grafico, dove la funzione deve dipendere dalla variabile su cui si cicla (e.g.  $x^i$  con x un array di un certo range e i la variabile del ciclo).
- Stessa cosa di sopra ma ora ogni curva deve avere un colore e uno linestyle diverso e una legenda.
- Creare una funzione che legga da input un numero intero con la condizione che esso sia maggiore di zero e che dia la possibilità di inserirlo nuovamente finché la condizione non è verificata.
- Sovrapporre i plot di un istogramma e della funzione di distribuzione associata, a vostra scelta, e aggiungere al grafico tutte le bellurie del caso.
- Scrivere una funzione "osservabile(data)" che dato un array "data" ne restituisca la media e la deviazione standard in un array.

$$\mu = \sum_i^n x_i \quad \sigma = \sqrt{\frac{1}{n(n-1)} \sum_i^n (x_i - \mu)^2}$$

- Scrivere una funzione "pi\_greco\_for(N)" che usi il problema di basilea per stimare il valore di  $\pi$ , che deve essere il return della funzione, e deve utilizzare un ciclo for. Scrivere poi la funzione "pi\_greco\_vec(N)" che faccia la stessa cosa ma vettorialmente, quindi senza cicli. (Come N prendete qualcosa del tipo  $10^6$ ).

$$\frac{\pi}{6} \approx \sum_{n=1}^N \frac{1}{n^2}$$

- Scrivere una funzione "decimal\_to\_binary(n)" che converta un numero intero "n" in base 10 in un numero in base 2; il numero in base due deve essere una stringa (i.e.  $10 \rightarrow '1010'$ ).
- Scrivere una funzione "binary\_to\_decimal(nb)" che prenda una stringa "nb" rappresentante un numero in base due e lo converta in base 10 (l'inverso del precedente esercizio).
- Scrivere una funzione "trova\_primi(n)" che dato un numero intero "n" trovi tutti i numeri primi minori di n e li restituisca in un array.
- Scrivere una funzione "palindromo(n)" che controlli se un numero intero "n" sia palindromo restituendo True se lo è False altrimenti.
- Scrivere una funzione "cesare(msg, key, enc)" che prenda in input: una stringa "msg" che è il testo da cifrare o da decifrare (con il cifrario di cesare appunto), un intero "key" che è la chiave con cui criptare il messaggio, e una variabile booleana "enc" per stabilire se la funzione debba cifrare o decifrare il messaggio in input, deve restituire in output una stringa che sia il messaggio cifrato o decifrato a seconda. Il perchè della variabile booleana è data dal fatto che se un testo è stato cifrato con la chiave 13 ad esempio, storico valore usato, esso può venir decifrato usando come chiave -13. Come hint sappiate che esistono delle funzioni di python che possono tornarvi utili "ord()", "char()".
- Scrivere una funzione "dec\_to\_hex(n)" che converta un numero intero "n" in base 10 in un numero in base 16; il numero in base 16 deve essere una stringa (i.e.  $158 \rightarrow '9E'$ ).
- Scrivere una funzione "hex\_to\_dec(ne)" che prenda una stringa "ne" rappresentante un numero in base 16 e lo converta in base 10 (l'inverso del precedente esercizio).
- Scrivere una funzione "clean\_data(x)" che prenda in input un array "x" contenente dei nan (potete scriverlo come  $x=np.array([3, np.nan, 8])$ ) e che restituisca l'array pulito con degli zeri al posto dei nan (nell'esempio precedente  $[3, 0, 8]$ ).
- Scrivere una funzione "eq(a, b, c)" che prenda in input tre numeri reali "a", "b", "c" corrispondenti ai coefficienti di un polinomio  $p(x)$  di secondo grado e che restituisca gli zeri di tale polinomio in un array. Si consideri  $p(x) = ax^2 + bx + c$ .
- Scrivere una funzione "fattori(n)" che prenda in input un numero intero "n" e restituisca una lista contenente la sua scomposizione in fattori primi, con la loro molteplicità (i.e.  $20 \rightarrow [2, 2, 5]$ ).
- Scrivere una funzione "goldbach(n)" che prenda in input un numero intero "n" e restituisca una lista di tuple, ciascuna delle quali contenente due numeri primi che sommati danno "n" (i.e.  $10 \rightarrow [(3, 7), (5, 5)]$ ).

19. Esiste un semplice algoritmo per il calcolo della radice di un numero. Si chiama algoritmo di Newton e fornisce un regola iterativa per approssimare la radice data una certa tolleranza. Vediamo tale regola:

- 1) prendo  $n =$  un certo numero di cui voglio calcolare la radice
- 2) pongo  $x = n$
- 3) calcolo  $x_n = 0.5(x - n/x)$
- 4) se  $|x - x_n| <$  una certa tolleranza
- 5) altrimenti pongo  $x = x_n$  e riparto dal punto 3)

Scrivere quindi una funzione "newton\_sqrt(n, tol)" che prenda un numero reale "n", numero di cui calcolare la radice, e un numero reale "tol" che rappresenta la tolleranza dell'algoritmo (e.g.  $10^{-5}$ ). La funzione deve restituire la radice calcolata.

20. (righe, colonne, Sudoku!) Questo esercizio è guidato a "scaletta" ed è volto a costruire qualche funzione utile che potrebbe essere utilizzata per risolvere un Sudoku.

- (a) Presa una matrice  $9 \times 9$  scrivere una funzione "check\_row(sudoku, n, i)" che prende in ingresso, oltre la matrice (qui chiamata sudoku), un numero i (che rappresenta la riga in cui ci troviamo) e un intero  $1 \leq n \leq 9$  e verificare che n non sia già presente sulla riga. *Hint:* bisogna fare un ciclo su...
  - (b) Presa la stessa matrice di sopra costruire una funzione "check\_col(sudoku, n, j)" che prende in ingresso un numero j (che rappresenta la colonna in cui ci troviamo) e un intero  $1 \leq n \leq 9$  che verifica che n non sia già presente sulla colonna. *Hint:* come prima, ma va fatto su...
  - (c) Vogliamo adesso costruire una funzione "check\_box(sudoku, n, i, j)" che, preso in ingresso un numero  $1 \leq n \leq 9$  e una possibile posizione che vogliamo dare (all'interno del Sudoku) ad n data nella forma i, j che indicano rispettivamente la riga e la colonna in cui si sta cercando di metterlo, verificare che la cella del Sudoku dove vogliamo mettere n non ce l'abbia già al suo interno. *Hint:* la parte più difficile dell'esercizio, se vi siete cimentati, è sicuramente trovare i bordi della cella (a meno che non facciate caso per caso) in una maniera elegante. Se ci pensate bene, potete sempre trovare la riga x e la colonna y che identificano l'angolo in alto a sinistra della cella in cui ci troviamo approssimando per difetto una divisione e moltiplicandolo per lo stesso numero che avete diviso (pensate a com'è fatto il Sudoku e a quante celle ci sono lungo una riga/colonna)
  - (d) Infine creare una funzione "is\_safe(sudoku, n, i, j)" che prende in ingresso un intero  $1 \leq n \leq 9$  e una posizione i, j (rispettivamente riga e colonna della matrice), come nella funzione precedente, ma che verifica se un numero può essere inserito o meno all'interno della cella presa (tramite un True o False). *Hint:* bisogna rimettere insieme quanto scritto nei precedenti punti.
21. Scrivere una funzione "EMCD(a, b)" che dati due numeri interi "a" e "b" restituisca in un array: il massimo comun divisore di "a" e "b", l'inverso moltiplicativo di "a" modulo b (che chiamiamo X) e l'inverso moltiplicativo di "b" modulo a (che chiamiamo Y). Ovvero vale la seguente identità (di Bézout):

$$aX + bY = \text{MCD}(a, b)$$

Per farlo utilizzate l'algoritmo esteso di euclide. Questa volta l'algoritmo non ve lo spiego io ma vi lascio ad una facile ricerca su internet.

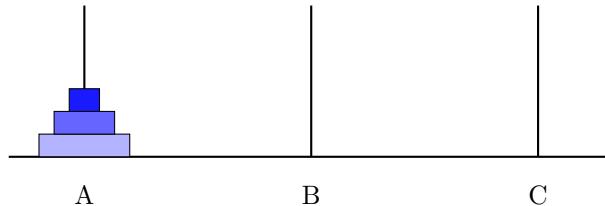
22. Scrivere una funzione "somma\_cond(arr, soglia)" che calcoli la somma degli elementi di "arr" che superano un certo valore di soglia (maggiore o uguale).
23. Scrivere una funzione "converti\_secondi(s)" che prenda un numero intero "s" di secondi e restituisca una stringa con il corrispondente numero di ore, minuti e secondi (e.g. 3622 → "01:20:02").
24. Create una funzione "grandi\_N(N)" che prenda in input un numero intero "N" e crei una sequenza di "N" numeri casuali da una distribuzione uniforme tra 0 e 1. Fatto ciò calcolate la media progressiva (cioè la media fino a ogni punto della sequenza) e restituite l'ultimo valore (che deve venire circa 0.5).
25. Scrivere una funzione che calcoli la probabilità che dato un gruppo di "n" persone almeno due abbiano la stessa data di compleanno. Fate poi il grafico di questa probabilità in funzione del numero "n" di persone.
26. Veloce definizione: un numero è detto automorfico se il suo quadrato termina con lo stesso numero. Per esempio,  $76^2 = 5776$  è un numero automorfico. Scrivere una funzione "automorfico(arr)" che prenda in input un array di numeri interi, in un certo range scelto da voi e che restituisca un array contenente tutti i numeri automorfici in quell'intervallo.
27. Altra definizione: la persistenza moltiplicativa di un numero è il numero di moltiplicazioni necessarie per ridurlo a una singola cifra moltiplicando tra loro le sue cifre. Facciamo un esempio:

$$49 \rightarrow 4 \times 9 = 36 \rightarrow 3 \times 6 = 18 \rightarrow 1 \times 8 = 8,$$

quindi 49 ha una persistenza moltiplicativa di 3. Scrivere dunque un funzione "persistenza(n)" che prenda un intero "n" e restituisca la sua persistenza moltiplicativa.

28. Le torri di Hanoi sono un gioco in cui si ha un insieme di "n" dischi di diverse dimensioni, impilati in ordine decrescente su un piolo. Lo scopo è spostare tutti i dischi da un piolo iniziale a uno finale utilizzando un piolo ausiliario, rispettando le seguenti regole:

- Si può spostare solo un disco alla volta.
- Un disco più grande non può mai essere posizionato sopra un disco più piccolo.



- Scrivete una funzione "hanoi(n, p1, p2, p3)" che prenda in input un numero intero "n" che è il numero dei dischi e tre variabili "p1", "p2" e "p3" che labellano, in ordine, i pioli (possono essere stringhe che interi, fate come volete), che risolva il problema delle torri di Hanoi. L'output deve essere una lista, la quale contenga delle tuple e per ogni tupla il primo elemento deve essere il disco che state spostando, il secondo il piolo di partenza e il terzo il piolo di arrivo.
29. Come immagino saprete l'ellisse non è una curva rettificabile; non c'è una formula in termini di funzione elementari che ci consenta di calcolarne il perimetro. Scrivete quindi una funzione "p\_ellisse(a, b)" che prenda due numeri reali "a" e "b", rispettivamente semiasse maggiore e minore e che restituisca un'approssimazione del periodo. Come suggerimento vi dico: scrivete la curva in forma parametrica in modo da avere due array che rappresentano le coordinate x e y dell'ellisse e calcolate con pitagora la distanza tra due punti vicini e poi sommate.

```

Test Esercizio
Nome del file da testare: esercizio_1.py Run Tests

Function: area
Execution time: 0.000038 seconds
Memory usage: Current=0.000024 MB; Peak=0.000208 MB
Result: 4.000000000000001

Function: pitagora
Execution time: 0.0000330 seconds
Memory usage: Current=0.000024 MB; Peak=0.000582 MB
Result: 0.0

Function: pylint
Result: **** Module esercizio_1
esercizio_1.py:6:12: W0621: Redefining name 'n' from outer scope (line 30) (redefined-outer-name)
esercizio_1.py:17:21: W0621: Redefining name 'n' from outer scope (line 30) (redefined-outer-name)
-----
Your code has been rated at 8.67/10 (previous run: 8.67/10, +0.00)

Function: Script intero
Execution time: 0.168999 seconds
Memory usage: Current=0.001613 MB; Peak=0.060718 MB
Result:
A1+A2=25.000000000000007, A3=25.000000000000004
3.552713678800591e-15
A1+A2=64.9519052838329, A3=64.9519052838329
0.0
A1+A2=120.71067811865476, A3=120.71067811865476
0.0

test functions ( __main__ .TestEsercizio)
Test for every function inside the code ... ok
test pylint ( __main__ .TestEsercizio)
PyLint's test ... ok
test script ( __main__ .TestEsercizio)
Test for the all script ... ok
-----
Ran 3 tests in 2.641s
OK

```

Figura 5: Quando eseguirete il test vi si aprirà questa finestra. Voi dovete inserire in nome del file su cui avete svolto l'esercizio e poi premere il tasto sulla finestra dove c'è scritto "Run Tests". Analizziamo ora l'output che ho ottenuto con la mia soluzione: I test delle due funzioni separatamente sono andati a buon fine; Il test di pylint ha riportato dei warning per quanto riguarda la variabile n alla linea 30, ho comunque preso un onesto punteggio 8.67 su 10. Più in basso il test esigue lo script per intero e stampa il risultato che, se eseguissi lo script, verrebbe stampato su shell. Dopodichè la finestra ci dice che il teste delle funzioni e il test dello script è andato bene; quello di pylint sarà sempre ok perchè tanto la cosa importante sono i warning che ha dato sopra e il punteggio.

```

Test Esercizio
Nome del file da testare: esercizio_1.py Run Tests

Function: area
Error: 8.000000000000002 != 4 within 6 places (4.000000000000002 difference) : Output errato per area: per (2, 4) il risultato atteso è 4, ma ho ottenuto 8.000000000000002

Function: pitagora
Execution time: 0.0000298 seconds
Memory usage: Current=0.000024 MB; Peak=0.000582 MB
Result: 0.0

Function: pylint
Result:
-----
Your code has been rated at 10.00/10 (previous run: 8.67/10, +1.33)

Function: Script intero
Execution time: 0.133604 seconds
Memory usage: Current=0.001611 MB; Peak=0.060718 MB
Result:
A1+A2=50.000000000000014, A3=50.000000000000001
7.05427357601062e-15
A1+A2=129.993105676658, A3=129.993105676658
0.0
A1+A2=241.4213562373095, A3=241.4213562373095
0.0

test functions ( __main__ .TestEsercizio)
Test for every function inside the code ... test pylint ( __main__ .TestEsercizio)
PyLint's test ... ok
test script ( __main__ .TestEsercizio)
Test for the all script ... ok
=====
FAIL: test functions ( __main__ .TestEsercizio) (function='area')
Test for every function inside the code
-----
Traceback (most recent call last):
  File "/home/francesco/GitHub/4BLP/3 Terza Lezione/test.py", line 77, in test.functions
    self.assertAlmostEqual(result, expected, 6, msg)
AssertionError: 8.000000000000002 != 4 within 6 places (4.000000000000002 difference) : Output errato per area: per (2, 4) il risultato atteso è 4, ma ho ottenuto 8.000000000000002
-----
Ran 3 tests in 2.595s
FAILED (failures=1)

```

Figura 6: Qui invece veete come appare il caso di errore. Ho corretto le lagne di pylint ma per sbaglio ho dimenticato un due nella formula dell'area per cui ottengo un errore.

Secondo:

```
1 power = [0.5, 1, 2]          # potenze
2 x = np.linspace(0, 1, 1000) # range sulle x
3
4 plt.figure(1)
5 for p in power:
6     plt.plot(x, x**p) # un plot alla volta sulla stessa figura
7
8 # bellurie
9 plt.grid()
10 plt.title("Esercizio 2")
11 plt.xlabel("x")
12 plt.ylabel("f(x)")
13 plt.show()
```

Terzo:

```
1 power = [0.5, 1, 2]          # potenze
2 color = ['k', 'r', 'b']        # colore di ogni curva
3 lnsty = ['-', '--', '-.']
4 label = [r'$\sqrt{x}$', r'$x$', r'$x^2$'] # nome della curva
5 x = np.linspace(0, 1, 1000)
6
7 plt.figure(1)
8 for p, c, ls in zip(power, color, lnsty, label):
9     # un plot alla volta sulla stessa figura
10    plt.plot(x, x**p, c=c, linestyle=ls, label=lb)
11
12 #bellurie
13 plt.grid()
14 plt.title("Esercizio 3")
15 plt.xlabel("x")
16 plt.ylabel("f(x)")
17 plt.legend(loc='best')
18 plt.show()
```

Quarto:

```
1 def read():
2     """
3         funzione che legge da input un numero con la condizione che esso
4             sia maggiore di zero e che dia la possibilita' di inserirlo
5                 nuovamente finche' la condizione non e' verificata.
6             Volendo si puo' generalizzare il codice passando la condizione come input
7     """
8
9     while True: # Il codice deve runnare finche' non inserisco un numero buono
10
11         try: # provo a leggere il numero e a renderlo intero
12             x = int(input("Iserisci un numero: "))
13
14         except ValueError: # se non riesco sollevo l'eccezione
15             print(f"Fra ti ho chiesto di mettere un numero") # messaggio di errore
16             continue # questo comando fa ripartire il ciclo da capo
17
18         if x > 0: # se la lettura e' andata a buon fine verifico la condizione
19             return x # se e' verificata ritorno il numero
20         else :
21             # altrimenti stampo un messaggio di errore
22             print("In numero inserito e' minore di zero, sceglierne un altro.")
23             continue # e faccio ripartire il ciclo da capo
24
25
26 x = read()
27 print(f"Il numero letto e': {x}")
```

Quinto:

```
1 # Gaussiana
2 m = 0
3 s = 1
4 z = [np.random.normal(m, s) for _ in range(int(1e5))]
5
6 # Plot dati
7 plt.figure(1)
8 plt.hist(z, bins=50, density=True, histtype='step', label='dati')
9 plt.grid()
10 plt.xlabel("x")
```

```

11 plt.ylabel("P(x)")
12 plt.title("Distribuzione gaussiana")
13
14 # Plot curva
15 x = np.linspace(-5*s + m, 5*s + m, 1000)
16 plt.plot(x, np.exp(-(x-m)**2 / (2*s**2))/np.sqrt(2*np.pi*s**2), 'b', label=f"N({m}, {s})")
17 plt.legend(loc='best')
18 plt.show()

```

Venticinquesimo:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def compleanno(n, prove):
5     """
6         Funzione che calcola la probabilita' che almeno due persone
7         in un gruppo di n persone facciano il compleanno insieme.
8         Il calcolo e' eseguito generando prove numero di gruppi e
9         verificando se ci sono coincidenze.
10
11     Parameters
12     -----
13     n : int
14         numero di persone nel gruppo
15     prove : int
16         numero di prove da fare
17
18     Returns
19     -----
20     p : float
21         casi favorevoli / casi totali
22     """
23     s = 0
24
25     for _ in range(prove):
26         # Generazione casuale di n compleanni (numeri da 1 a 365)
27         c = np.random.randint(1, 365 + 1, n)
28
29         # Verifica se ci sono duplicati nei compleanni
30         if len(c) != len(set(c)):
31             s += 1
32
33     # Calcolo della probabilita' stimata
34     p = s / prove
35     return p
36
37 N      = 70
38 p      = int(3e4)
39 all_n  = range(2, N + 1)
40 P      = [compleanno(n, p) for n in all_n]
41
42 plt.figure(1)
43 plt.plot(all_n, P, label="Probabilita' stimata")
44 plt.xlabel("Numero di persone nell gruppo")
45 plt.ylabel("Probabilita' di coincidenza di compleanni")
46 plt.title("Paradosso del Compleanno")
47 plt.grid()
48 plt.show()

```

## 8 Quarta lezione

### 8.1 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 perché 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. Per capire meglio quello che stiamo dicendo con quell'if possiamo usare la funzione 'locals()' la quale restituisce tutte le variabili che sono definite nel nostro codice:

```
1 x = 256
2 y = x /2
3 print(locals())
4
5 [Output]
6 {'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <
7     _frozen_importlib_external.SourceFileLoader object at 0x7f5a033dd2d0>, '__spec__': None, '8
8     __annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__': '/home/
9     francesco/GitHub/tmp/3 Terza Lezione/prova.py', '__cached__': None, 'x': 256, 'y': 128.0}
```

Vediamo quindi che "locals()" restituisce un dizionario con chiave il nome della variabile e come valore il valore della variabile appunto. Vediamo che ci sono le due variabili da noi definite e poi tutta una serie di variabili "private" chiamate con il doppio underscore. La prima variabile è proprio "\_\_name\_\_" che è il nome del codice, "\_\_main\_\_" in questo caso; poi ci sta "\_\_doc\_\_" che è la documentazione, ovvero il commento con tre apici che spesso avete visto all'inizio di molti codici, che ora è assente e quindi è None; altro interessante è "\_\_file\_\_" che contiene il path del file. Se ora importassimo questo codice come abbiamo fatto sopra con "elevamento" queste istruzioni verrebbero eseguite, e quindi il dizionario verrebbe stampato, ma ora alla voce "\_\_name\_\_" corrisponderebbe la stringa "prova" (questo perché come vedete dalla voce "\_\_file\_\_" ho chiamato il codice "prova.py") più in genere compare nome del codice, provate a farlo.

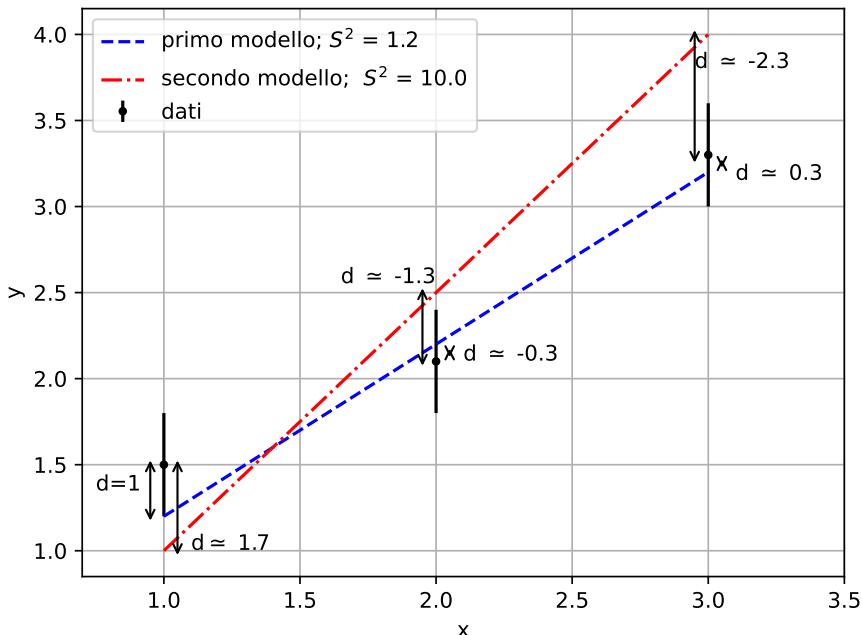
### 8.2 Fit

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.

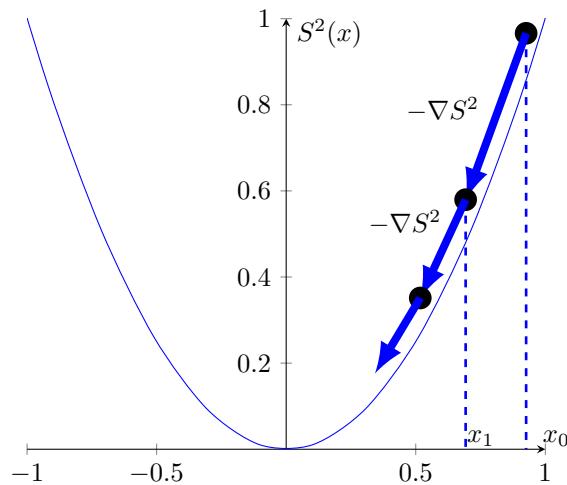
Nel caso di regressione lineare la procedura da eseguire non è troppo complicata, mentre per la regressione non lineare le cose si fanno parecchio complicate e si utilizzano algoritmi di ottimizzazione. Se noi abbiamo quindi un modello teorico che ci dice che un corpo cade con una legge oraria della 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). In ogni caso comunque l'idea di ciò che va fatto è trovare il minimo della seguente funzione:

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

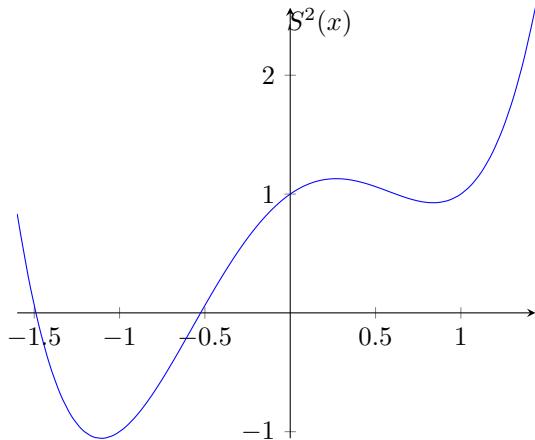
che nel caso in cui il termine dentro la somma sia distribuito in modo gaussiano allora la quantità  $S^2$  è distribuita come un chiquadro, e da qui si potrebbe fare tutta una discussione sulla significatività statistica di quello che andiamo a fare, che ovviamente noi non facciamo. Analizziamo un attimo questa formula:  $S^2$  è in linea di principio una funzione a molte variabili e che restituisce un numero reale. Il termine dentro la somma rappresenta la distanza tra valore del dato e valore della funzione in unità della barra d'errore del dato. Facciamo un esempio visivo per rendere più chiaro il concetto. Consideriamo giusto a titolo di esempio tre punti e due possibili rette che noi possiamo pensare che più o meno approssimino i dati.



Nel grafico  $d$  è proprio la distanza del modello dal dato in unità di barre di errore, e quindi la somma di tutte queste quantità elevate al quadrato è il valore di  $S^2$  che è riportato nella legenda. Notiamo quindi che effettivamente la retta che presenta un valore di  $S^2$  minore è quella che ad occhio meglio approssima i dati. Ora uno potrebbe pensare che quindi basta calcolare  $S^2$  su una griglia e vedere dove assume il valore più piccolo. Questo è un metodo abbastanza brute force e il linea di principio funziona, ma quello che in genere si fa è un po' diverso. In linea di principio per capire come funziona un algoritmo di minimo basta pensare ad una pallina che cade in una ciotola. Precisiamo che si vuole fare tutta questa trattazione per far capire che la parte più delicata di questa procedura è scegliere quello che noi chiameremo nel codice "init" e che esso violentemente aggiusta o complica la nostra situazione. Consideriamo per semplicità, didattica e grafica, una funzione di una singola variabile.



Quel che noi facciamo è scegliere un  $x_0$ , (il nostro init) ed aggiornare questa posizione considerando la pendenza della funzione, che altro non sarebbe che la derivata della funzione che vogliamo considerare. Concedetemi, per maggiore generalità, di sostituire il termine derivata con il termine gradiente, indicato dal simbolo  $\nabla$ . Quindi quello che il codice fa è dirci analogo ad una pallina che si muove sotto l'azione di un potenziale, che sarebbe  $S^2$  e la sua derivata, il suo gradiente, non è altro che la forza che la pallina sente. Facendo così troviamo una serie di  $x_i$  iterativamente, fino ad arrivare al minimo dove il gradiente, la forza esterna, è zero. Questo metodo è chiamato gradiente discendente. Finché abbiamo un solo minimo quindi va tutto bene. Lo troviamo senza problemi a prescindere da dove partiamo. Supponiamo ora una situazione più brutta:



In questo caso vediamo subito che abbiamo due punti in cui la derivata è nulla, quindi due minimi (questo sarebbe il caso di una regressione non lineare, a differenza di quella lineare di sopra, un solo minimo), ma quello a cui siamo interessati noi è il minimo assoluto. Se utilizzassimo il metodo precedente è facile vedere che se partiamo per esempio con  $x > 1$  ci incastriamo nel minimo locale. Le uniche zone buone sono soltanto quelle con  $x < 0$ . Vedete quindi che una piccola complicazione riduce di molto le nostre possibilità e dobbiamo quindi selezionare il nostro punto di partenza con delicatezza. Questo perché una volta arrivato al minimo locale la nostra "pallina" non ci arriva con una velocità come accadrebbe nella realtà e quindi non riesce a scavallare la collinetta. Fondamentalmente per migliorare la cosa dobbiamo spiegare al computer il concetto di inerzia e anche di attrito (se l'energia si conservasse la pallina oscillerebbe all'infinito e il codice non terminerebbe). Un esempio di ciò, chiamato gradiente discendente con momento, e anche di quanto visto sopra è disponibile in una delle appendici. Inoltre in questa stessa lezione andremo a vedere cosa fa effettivamente "curve.fit" che è un po' diverso. Un caso ancora peggiore lo vediamo adesso con un problema fisico, dove ora non abbiamo un semiasse da poter scegliere, ma solo una piccola e precisa zona, dovuto al fatto che ora non abbiamo due minimi ma molti di più. 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

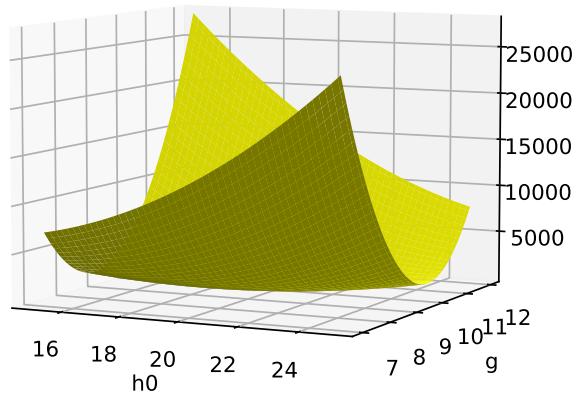


Figura 7: modello lineare  $y(t) = h_0 - \frac{1}{2}gt^2$ . Unico minimo, qualunque punto iniziale va bene.

Chiquadro regressione non-lineare

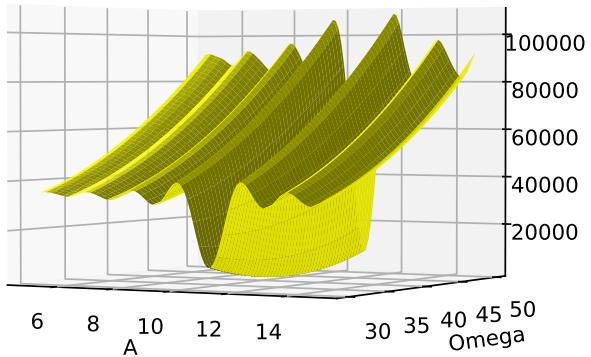


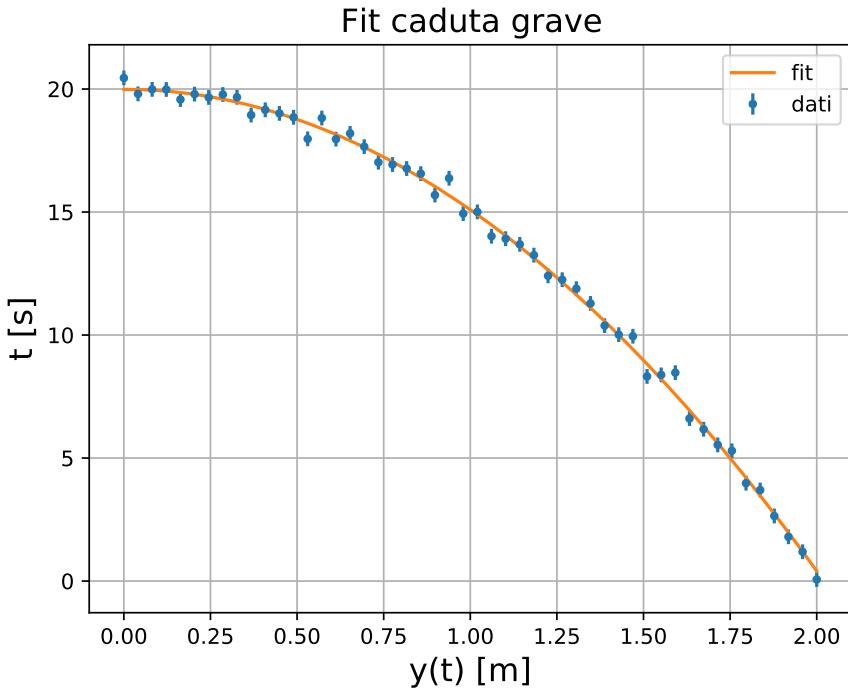
Figura 8: modello non lineare  $y(t) = A\cos(\omega t)$ . Tanti minimi locali bisogna stare attenti a dove partire altrimenti l'algoritmo si blocca su soluzioni non fisiche. Solo una piccola regione va bene come valori iniziali.

I codici per generare i grafici che abbiamo visto non sono riportati per brevità ma sono presenti nella cartella. 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 velocità iniziale nulla
10    """
11    return h0 - 0.5*g*t**2
12
13 """
14 dati misurati:
15 xdata : fisicamente 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 affetta
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 affetti da errore
31 ydata = y + y_noise
32 dydata = np.array(ydata.size*[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 gravità g = {g:.3f} +- {dg:.3f}')
46
47 #grafico 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.title('Fit caduta grave', fontsize=15)
53 plt.xlabel('y(t) [m]', fontsize=15)
54 plt.ylabel('t [s]', fontsize=15)
55 plt.legend(loc='best')
56 plt.show()
57
58 [Output]
59 Altezza iniziale h0 = 19.988 +- 0.065
60 Accelerazione di gravità g = 9.790 +- 0.071

```



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. Provate a fittare il modello non lineare visto sopra e vi accorgerete come solo una piccola regione dei parametri conduca alla soluzione corretta e che basti spostarvi di poco per ottenere risultati poco sensati.

### 8.2.1 Init

Spero che abbiate capito, arrivati a questo punto, che è fondamentale mettere dei parametri iniziali sensati. Ma la domanda che sorge è come li determiniamo? Un po' come volete. Si possono fare tante cose, a seconda di che tipo di dati avete poi e da che processo fisico essi derivano. Io qui voglio solo fornirvi un piccolo codice che usando delle particolarità (i widget) di matplotlib permette di scrivere in un box la funzione che volete plottare, in codice python ovviamente, e dopo aver premuto invio essa viene plottata sui dati, in modo che voi abbiate sempre il grafico sottocchio (senza ogni volta chiudere il grafico, cambiare valori, ed eseguire di nuovo il codice).

```

1 """
2 Codice Per plottare i dati con una funzione per capire
3 i valori dei parametri ottimali da passare a curve_fit
4 """
5
6 import numpy as np
7 import matplotlib.pyplot as plt
8 from matplotlib.widgets import TextBox
9
10 # Leggo i dati
11 #x_data, y_data, dy_data = np.loadtxt('...', unpack=True)
12
13 # Qui per comodita' li simulo
14 x_data = np.linspace(0, 10, 60)
15 y_data = 10*np.cos(2.5*x_data + np.pi/4) + 3
16
17 # Un po' di rumore quanto basta
18 rng = np.random.default_rng(seed=69420)
19 dy = 1
20 y_noise = dy * rng.normal(size=x_data.size)
21 y_data += y_noise
22 dy_data = np.array(x_data.size*[dy])
23
24 # Creazione della figura
25 plt.figure(figsize=(8, 8))
26 plt.title("TITOLO")
27 plt.xlabel("t", fontsize=15)
28 plt.ylabel("F(t)", fontsize=15)
29 plt.subplots_adjust(bottom=0.2)
```

```

30 plt.errorbar(x_data, y_data, dy_data, c='k', fmt='.', label='data')
31
32 # Testo da scrivere inizialmente sulla barra per spiegare
33 text = "Insert here function, e.g. np.cos(t) or 3*t - 2 then press enter"
34
35 # Linspace per il plot e definiamo la variabile l che e' l'output del grafico
36 # ci servira' in quanto noi andremo a sovrascrivere questa variabile in modo
37 # che sia sempre tutto associato a questo grafico. Di default si plotta una
38 # retta alla media dei dati
39 t = np.linspace(np.min(x_data), np.max(x_data), 1000)
40 l, = plt.plot(t, np.mean(y_data)*np.ones(t.size), 'b', label='fit law')
41 plt.legend(loc='best')
42 plt.grid()
43
44 def submit(text):
45     """
46     Funzione che valuta l'espressione e la plotta
47
48     Parameter
49     -----
50     text : string
51         Espressione da valutare scritta in python
52     """
53     ydata = eval(text) # valuto l'espressione
54     l.set_ydata(ydata) # aggiorno la variabile del plot
55     plt.draw()          # Disegno il plot aggiornato
56
57 # Box per prendere l'input
58 axbox = plt.axes([0.15, 0.05, 0.75, 0.075])
59 text_box = TextBox(axbox, 'F(t)=', initial=text)
60 text_box.on_submit(submit)
61
62 plt.show()

```

### 8.3 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  $\theta$ , il quale fondamentalmente è un vettore di  $\mathbb{R}^m$ . Possiamo espandere  $f$  in serie di taylor intorno ad un valore dei nostri parametri:

$$f(x_i, \theta_j + \delta_j) \simeq f(x_i, \theta_j) + J_{ij}\delta_j \quad (3)$$

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

$$J_{ij} = \frac{\partial f(x_i, \theta_j)}{\partial \theta_j} = \begin{bmatrix} \frac{\partial f(x_1, \theta_1)}{\partial \theta_1} & \dots & \frac{\partial f(x_1, \theta_m)}{\partial \theta_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(x_n, \theta_1)}{\partial \theta_1} & \dots & \frac{\partial f(x_n, \theta_m)}{\partial \theta_m} \end{bmatrix} \quad (4)$$

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  $\delta$  espandiamo la (2):

$$\begin{aligned} S^2(\theta + \delta) &\simeq \sum_{i=1}^n \frac{(y_i - f(x_i, \beta) - J_{ij}\delta_j)^2}{\sigma_{y_i}^2} \\ &= (y - f(x, \theta) - J\delta)^T W (y - f(x, \theta) - J\delta) \\ &= (y - f(x, \theta))^T W (y - f(x, \theta)) - (y - f(x, \theta))^T W J \delta - (J\delta)^T W (y - f(x, \theta)) + (J\delta)^T W (J\delta) \\ &= (y - f(x, \theta))^T W (y - f(x, \theta)) - 2(y - f(x, \theta))^T W J \delta + \delta^T J^T W (J\delta) \end{aligned} \quad (5)$$

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

$$\frac{\partial S^2(\theta + \delta)}{\partial \delta} = -2(y - f(x, \theta))^T W J + 2\delta^T J^T W J = 0 \quad (6)$$

per cui facendo il trasposto a tutto otteniamo:

$$(J^T W J)\delta = J^T W (y - f(x, \theta)) \quad (7)$$

La quale si risolve per  $\delta$ . Per migliorare la convergenza del metodo si introduce un parametro di damping  $\lambda$  e l'equazione diventa:

$$(J^T W J - \lambda \text{diag}(J^T W J))\delta = J^T W (y - f(x, \theta)) \quad (8)$$

Il valore di  $\lambda$  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:

$$\begin{aligned} \rho(\delta) &= \frac{S^2(x, \theta) - S^2(x, \theta + \delta)}{|(y - f(x, \theta) - J\delta)^T W (y - f(x, \theta) - J\delta)|} \\ &= \frac{S^2(x, \theta) - S^2(x, \theta + \delta)}{|\delta^T (\lambda \text{diag}(J^T W J)\delta + J^T W (y - f(x, \theta)))|} \end{aligned} \quad (9)$$

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

$$R1 = \max(|J^T W (y - f(x, \theta))|) \quad (10)$$

$$R2 = \max(|\delta/\theta|) \quad (11)$$

$$R3 = |S^2(x, \theta)/(n-m) - 1| \quad (12)$$

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:

$$\text{Cov} = (J^T W J)^{-1} \quad (13)$$

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,  $y \leq f(x, \{\theta\})$ 
28     x0 : 1darray
29         initial guess
30     sigma : None or 1darray
31         the uncertainty on y, if None  $\sigma = \text{np.ones}(\text{len}(y))$ 
32     tol : float
33         required tollerance, the algorithm stop if one of this quantities
34          $R_1 = \text{np.max}(\text{abs}(J.T @ W @ (y - func(x, *x0))))$ 
35          $R_2 = \text{np.max}(\text{abs}(d/x0))$ 
36          $R_3 = \text{sum}(((y - func(x, *x0))/dy)**2)/(N - M) - 1$ 
37         is smaller than tol
38
39     dense_output : bool, optional default False
40         if true all iteration are returned
41     absolute_sigma : bool, optional default False
42         If True, sigma is used in an absolute sense and
43         the estimated parameter covariance pcov reflects
44         these absolute values.
45          $\text{pcov}(\text{absolute\_sigma=False}) = \text{pcov}(\text{absolute\_sigma=True}) * \text{chisq}(\text{popt})/(M-N)$ 
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

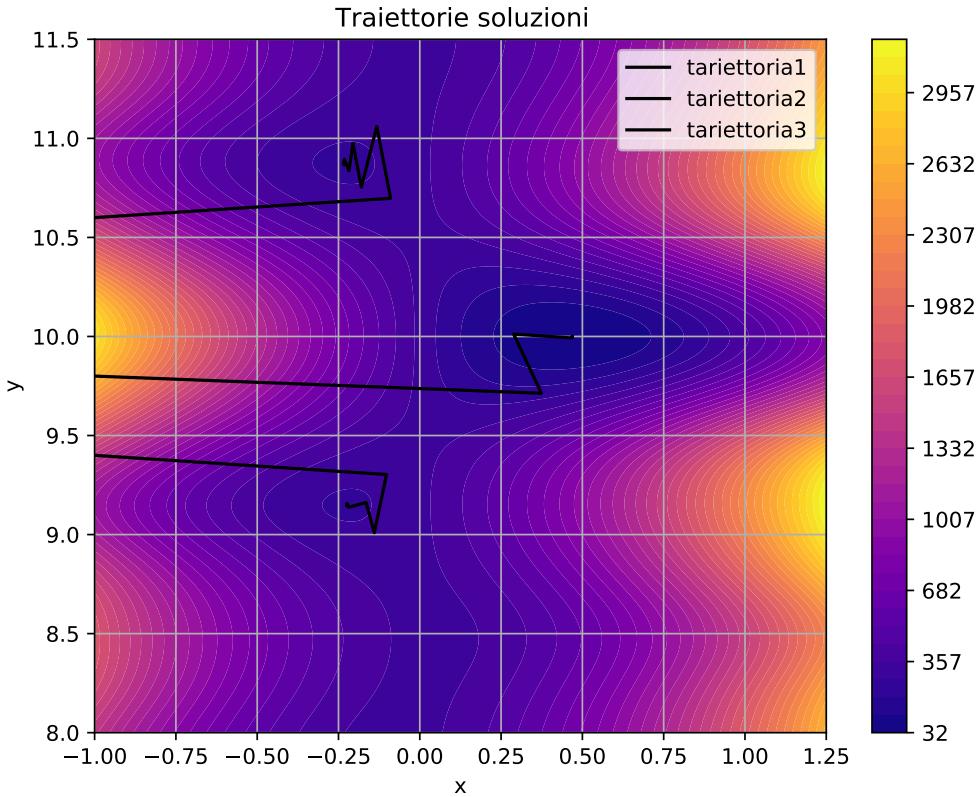
```

```

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
92         variables
93
94         JtJ = J.T @ W @ J           #matrix multiplication, JtJ is an MxM matrix
95         dia = np.eye(M)*np.diag(JtJ)
96         res = (y - func(x, *x0))   #residuals
97         b = J.T @ W @ res          #ordinate or dependent variable values of
98
99         system
100        d = np.linalg.solve(JtJ + l*dia, b)      #system solution
101        x_n = x0 + d                         #solution at new time
102
103        # compute the metric
104        chisq_v = sum((res/dy)**2)
105        chisq_n = sum(((y - func(x, *x_n))/dy)**2)
106
107        rho = chisq_v - chisq_n
108        den = abs(d.T @ (l*np.diag(JtJ)@d + J.T @ W @ res))
109        rho = rho/den
110
111        # acceptance
112        if rho > eps_1 :                 #if i'm closer to the solution
113            x0 = x_n                      #update solution
114            l /= f                        #reduce damping factor
115        else:
116            l *= f                      #else magnify
117
118        # Convergence criteria
119        R1 = np.max(abs(J.T @ W @ (y - func(x, *x0))))
120        R2 = np.max(abs(d/x0))
121        R3 = abs(sum(((y - func(x, *x0))/dy)**2)/(N - M) - 1)
122
123        if R1 < eps_2 or R2 < eps_3 or R3 < eps_4:          #break condition
124            break
125
126        iter += 1
127
128        if dense_output:
129            X.append(x0)
130
131        #compute covariance matrix
132        pcov = np.linalg.inv(JtJ)
133
134        if not absolute_sigma:
135            s_sq = sum(((y - func(x, *x0))/dy)**2)/(N - M)
136            pcov = pcov * s_sq
137
138        if not dense_output:
139            return x0, pcov, iter
140        else :
141            X = np.array(X)
142            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 incasti in minimo locali. L'asse y corrisponde a  $\omega$  mentre l'asse x corrisponde ad  $A$ . Vediamo ora di testare i risultati del codice fissando 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_Maq 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.)

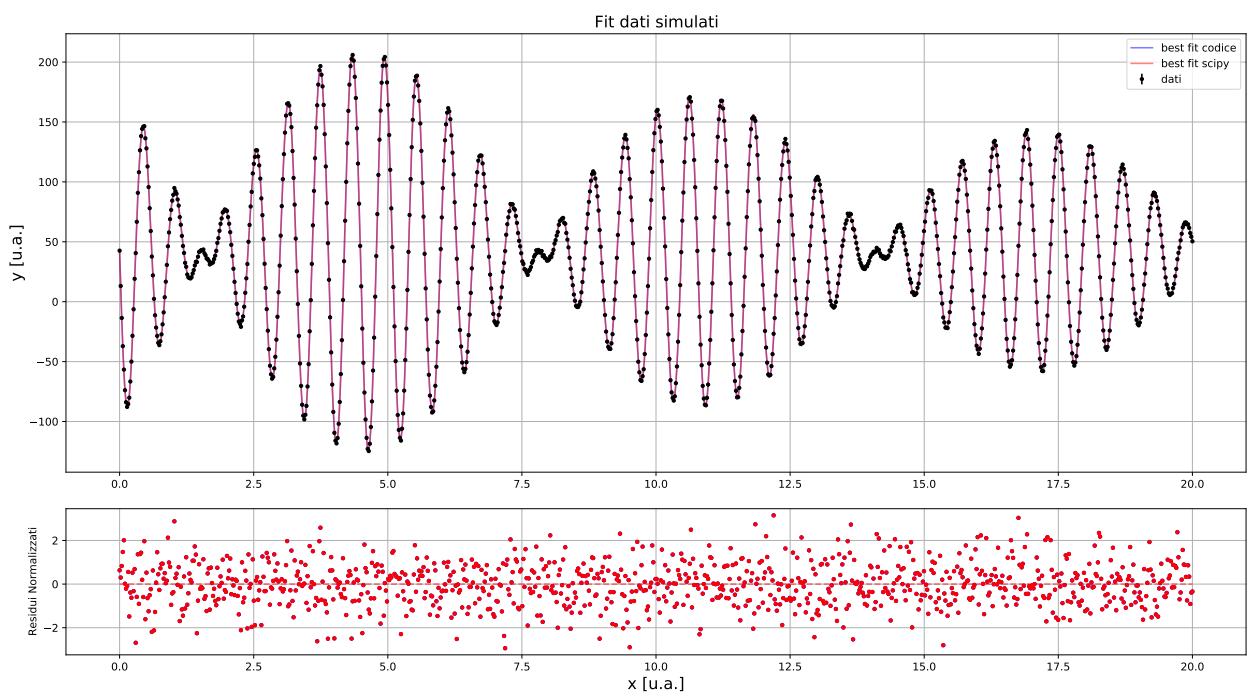
```

```

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=15)
61 plt.ylabel('y [u.a.]', fontsize=15)
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
       fit
68 plt.plot(t, f(t, *pars2), color='red', alpha=0.5, label='best fit scipy') #grafico del best
       fit scipy
69 plt.legend(loc='best')#inserisce la legenda nel posto migliore
70
71 #Parte inferiore contenente i residui
72 frame2=fig1.add_axes((.1,.1,.8,.2))
73
74 #Calcolo i residui normalizzati
75 ff1 = (y - f(x, *pars1))/dy
76 ff2 = (y - f(x, *pars2))/dy
77 frame2.set_ylabel('Residui Normalizzati')
78 plt.xlabel('x [u.a.]', fontsize=15)
79
80 plt.plot(t, 0*t, color='red', linestyle='--', alpha=0.5) #grafico la retta costantemente zero
81 plt.plot(x, ff1, '.', color='blue') #grafico i residui normalizzati
82 plt.plot(x, ff2, '.', color='red') #grafico i residui normalizzati scipy
83 plt.grid()
84 plt.show()
85
86 [Output]
87      -----codice-----|-----scipy-----
88 pars0 = 199.85504 +- 0.17712 ; 199.85504 +- 0.17712
89 pars1 = 10.50005 +- 0.00009 ; 10.50005 +- 0.00009
90 pars2 = 0.49990 +- 0.00008 ; 0.49990 +- 0.00008
91 pars3 = 1.57040 +- 0.00087 ; 1.57040 +- 0.00087
92 pars4 = 0.78579 +- 0.00067 ; 0.78579 +- 0.00067
93 pars5 = 41.92350 +- 0.03125 ; 41.92350 +- 0.03125
94 pars6 = 24.99194 +- 0.05652 ; 24.99194 +- 0.05652
95 numero di iterazioni = 6
96 chi quadro codice = 969.017 (993 dof)
97 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:



## 9 Lezione Bonus: Version control

Ok che le lezioni dovrebbero essere 4, ma come avete potuto vedere dall'indice ci sono una serie svariata di appendici e quindi tanto vale aggiungere questa lezione bonus che potrebbe essere molto utile. Tra l'altro potete capire da quanto appena scritto che questa sezione è stata aggiunta molto dopo. Non sappiamo nemmeno se sarà effettivamente una lezione che verrà tenuta. Però nella convinzione che possa essere utile io comunque provo a spiegarvelo. Come sapete queste note e i vari codici sono disponibili su un sito internet chiamato GitHub. GitHub è un sito che serve proprio per queste tipo di cose e per interagirci si può usare git o più semplicemente l'applicazione github desktop. Spieghiamo ora brevemente cos'è il controllo versione. Fondamentalmente immaginate di dover lavorare con più persone ad un progetto, che sia un codice software o una relazione di laboratorio. Quindi è necessario che tutti abbiano le stesse informazioni sui più recenti sviluppi fatti. Per quanto riguarda lo scrivere le relazione e basta magari uno può dire che esistono modi per scrivere e condividere il sorgente, ad esempio overleaf; ma non c'è solo la relazione, c'è anche il codice di analisi dati che sarebbe giusto tutti abbiano. Quindi quello che facciamo è creare un luogo remoto, ad esempio una repository su GitHub, in cui sarà caricato il codice e tutte le altre informazioni. Ogni membro potrà quindi farne una copia, tramite Git o github desktop, sul suo pc, quindi una copia locale della repository, che potrà modificare all'occorrenza. Tale copia si fa con il comando "**git clone ...**" vedremo poi il perché dei tre puntini. Una volta fatta le varie modifiche voi potete, allo stesso modo con cui avete scaricato la versione originale, potete cariarla dalla vostra repository locale sulla repository in remoto sul server. Vediamo un semplice schemino per rendere chiare le idee.

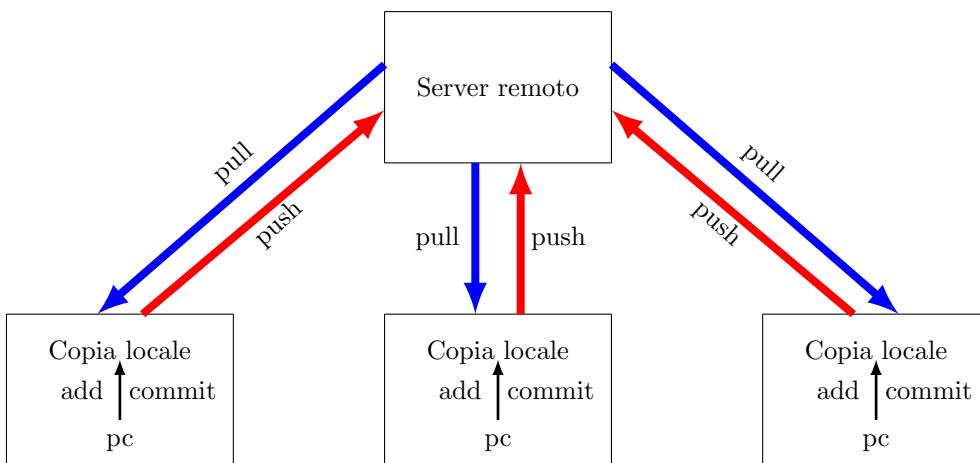


Figura 9: Schema controllo versione distribuito.

I nomi che vedete accanto alle frecce sono i nomi delle operazioni, banalmente i comandi da dare a git. Spieghiamo un attimo: avete un certo file e lo modificate; dovete quindi aggiornare il vostro repository locale facendo "**git add nomefile.estensione**" e "**git commit nomefile.estensione**" (su github desktop vi basterà premere il tasto commit). Fatto ciò per spedire la copia in remoto vi basterà fare "**git push**". Se invece la copia locale è indietro rispetto alla copia remota, per aggiornare vi basterà fare "**git pull**". Ovviamente tutti questi comandi vanno scritti su shell, su github desktop vi basta preme i bottoni in cui ci sta scritto push, pull o che altro sia; abbastanza semplice. Buona norma è aggiungere un commento quando si fa il commit; commento che sarà visibile sulla repository locale. da shell basta fare "**git commit nomefile.estensione -m "commento da inserire"**". Ciò è importante per far capire cosa avete fatto; infatti questo sistema di collaborazione tiene traccia di tutte le versioni di ciò su cui state lavorando, come una vera e propria cronologia, in modo che sia semplice risalire a vecchie versioni in caso di errori. Ora in linea di principio anche dentro il server remoto possono succedere cose, tra **fork**, **pull request**, **issue** e **merge**. Spieghiamo brevemente di che si tratta: Sia A la repository sull'account GitHub di persona A (ho molta fantasia). Voi potete premere dove ci sta scritto **fork** e creare così una copia della repository sul vostro account e fare tutto come detto sopra, creando un ramo secondario. Una volta fatto tutto potete fare una **pull request** cioè chiedere che la vostra modifica entri nel ramo principale dalla quale avete eseguito il **fork**; se chi gestisce la repository (persona A) è d'accordo allora lui farà il **merge**. Tutto ciò può anche essere fatto su una vostra repository ovviamente. Infine un **issue** è una cosa che voi su GitHub aprite dicendo al mondo: sta cosa non funge. Tutto ciò nella speranza che qualcuno veda il vostro problema e attraverso tutti i meccanismi di sopra, o semplicemente con un commento vi aiuti a risolverlo o vi dia consigli sul da farsi. Veniamo all'ultima questione lasciata in sospeso: "**git clone ...**" cosa ci va al posto dei tre puntini? Potete fare due cose: potete banalmente mettere l'HTTPS, prendendo l'url, ma lo trovate anche nella repository su GitHub premendo dove c'è un bottone verde con scritto "Code". Lì trovate anche l'SSH, altra cosa che potete utilizzare, secondo me più comodo perché una

volta che configurate l'SSH dopo aver creato il vostro account GitHub potrete fare tutto quello che volete senza che git vi chieda varie ed eventuali password. Per configurare la chiave SSH potete seguire i due link nell'ordine: prima lui <https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent> e poi lui <https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account>. Con github desktop non avrete questi problemi nel clone e nel resto.

# *Hic Sunt Dracones*

Q *ui ci sono i dragi.* Così sulle mappe medievali venivano indicati luoghi pericolosi da esplorare. Che queste appendici lo siano? Spererei di no. Però data la vastità dei temi trattati direi che certamente chi per di qua si avventura ha scelto la pillola rossa e vuole iniziale a scoprire quanto è profonda la tana del bian coniglio. Beh che la forza sia con voi, d'altronde *Audentes Fortuna Iuvat*.

Gli argomenti che seguono saranno riportati in quello che forse, almeno credo, è l'ordine migliore per esporli, in modo da non rendere pesanti eventuali richiami e per far sì che quanto non spiegato nell'appendice sia però presente in una delle precedenti. Con questa logica capiterà dunque di vedere magari delle applicazioni ad esempi fisici di ciò che viene spiegato in altri capitoli che riguardano quell'esempio. Quindi, benché nella sezione dei sistemi lineari vediamo la spiegazione del SOR la sua implementazione per risolvere l'equazione di Laplace si troverà nella sezione dedicata alle pde; quando però risolveremo un'equazione differenziale con una rete neurale, trattandosi di meccanismi più complessi troveremo tutto nello stesso capitolo.

አመገኘ የሚያስተካክለ ማርያም እና የሚያስተካክለ ማርያም እና የሚያስተካክለ ማርያም

## A Sistemi lineari

Spesso capita, come abbiamo visto sopra con il caso dei fit e come vedremo più avanti, di dover risolvere dei sistemi di equazioni o di invertire una matrice. Vogliamo vedere qui qualche metodo rimandando le applicazioni a successive sezioni. Abbiamo un sistema del tipo:

$$Ax = b, \quad (14)$$

dove

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

Come troviamo  $x$ ?

### A.1 Metodo Gauss–Seidel

Un primo metodo è quello di Gauss–Seidel. Ditta  $L$  una matrice triangolare inferiore e  $U$  matrice triangolare superiore, con diagonale nulla tale che  $A = L + U$ :

$$L = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}. \quad (15)$$

Allora abbiamo:

$$Ax = b \quad (16)$$

$$(L + U)x = b \quad (17)$$

$$Lx + Ux = b \quad (18)$$

$$Lx = b - Ux, \quad (19)$$

e quindiabbiamo in maniera iterativa:

$$x^{k+1} = L^{-1}(b - Ux^k), \quad (20)$$

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

$$x_i^{k+1} = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^n a_{ij}x_j^k \right). \quad (21)$$

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). Benchè nella raccolta di codici sarà presente l'implementazione di questo metodo, qui mostriamo solo l'implementazione del SOR.

### A.2 Successive over-relaxation

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

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

Introducendo il parametro  $\omega$  di overrelaxation l'algoritmo diventa:

$$x_i^{k+1} = (1 - \omega)x_i^k + \frac{\omega}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^n a_{ij}x_j^k \right). \quad (23)$$

Vediamo quindi che per  $\omega = 1$  recuperiamo il metodo precedente. Vediamo ora un esempio di codice e vedremo che due esecuzioni con  $\omega$  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
```

### A.3 Metodo del gradiente coniugato

Un altro modo per risolvere sistemi lineari, e che ci permette 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:

$$x_{k+1} = x_k + \alpha_k p_k \quad (24)$$

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  $\alpha_k$  è la grandezza del passo,  $\frac{p_k^T r_k}{p_k^T A p_k}$ . L'idea di muoversi lungo le direzioni ortogonali di  $A$ , quindi via vettori coniugati, viene perché data la forma quadratica:

$$Q(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T A \mathbf{x} - \mathbf{x}^T b, \quad (25)$$

si vede che la soluzione del nostro sistema è il minimo di questa forma quadratica, che esiste ed è unico, poiché l'essiana è esattamente  $A$ . Tratteremo più avanti i problemi di minimo, anche se ne abbiamo già parlato nella quarta lezione; qui ne usiamo solo l'idea sta tutto sotto il tappeto. Per il calcolo delle direzioni coniugate quello che si va a fare è sostanzialmente una ortogonalizzazione di Gram-Schmidt. 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) #initial guess

```

```

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 con matrici molto grosse.

#### A.4 Matrici tridiagonali

Capita spesso di avere a che fare con delle matrici, e infatti più avanti saranno ricorrenti, che sono principalmente zero e le cui uniche entrate non nulle sono la diagonale e le due diagonali adiacenti. Queste si chiamano matrici tridiagonali. Un sistema tridiagonale è quindi un sistema lineare  $A\mathbf{x} = \mathbf{b}$ , dove  $A$  è:

$$A = \begin{bmatrix} d_1 & c_1 & 0 & 0 & \cdots & 0 \\ a_1 & d_2 & c_2 & 0 & \cdots & 0 \\ 0 & a_2 & d_3 & c_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & a_{N-1} & d_N & c_{N-1} \\ 0 & 0 & 0 & 0 & a_N & d_N \end{bmatrix}. \quad (26)$$

Data quindi questa importante caratteristica della nostra matrice è facile vedere che l'eliminazione di Gauss si riduce a  $\mathcal{O}(N)$  come complessità. Si applica infatti la riduzione eliminando gli elementi della sottodiagonale mediante sostituzioni

$$d'i = d_i - \frac{a_{i-1}c_{i-1}}{d'_{i-1}}, \quad \text{per } i = 2, \dots, N. \quad (27)$$

Analogamente, si aggiorna il termine noto:

$$b'i = b_i - \frac{a_{i-1}b'_{i-1}}{d'_{i-1}}, \quad \text{per } i = 2, \dots, N. \quad (28)$$

Una volta terminata la fase di eliminazione, rimaniamo che l'ultima riga della matrice ha un solo elemento quindi è facile trovare  $x_N$

$$x_N = \frac{b'_N}{d'_N}. \quad (29)$$

Fatto questo si risolve il sistema con la sostituzione all'indietro:

$$x_i = \frac{b'_i - c_i x_{i+1}}{d'_i}, \quad \text{per } i = N-1, \dots, 1. \quad (30)$$

Quindi noi abbiamo "percorso" soltanto dei vettori lunghi  $N$  avanti e indietro. Vediamo qui un esempio di come implementarla, più avanti ci saranno altre applicazioni. Ecco il codice di esempio:

```

1 import time
2 import numpy as np
3
4 def solve_tridiagonal(diag, sup_diag, inf_diag, b):
5     """
6         Function to solve a tridiagonal system using Gaussian elimination.
7
8     Parameters
9     -----
10    diag : 1darray
11        A(i, i) = diag
12    sup_diag : 1darray
13        A(i, i + 1) = sup_diag
14    inf_diag : 1darray
15        A(i - 1, i) = inf_diag
16    b : array

```

```

17     RHS of the equation
18
19 Returns
20 -----
21 x : 1darray
22     solution of the sistem
23 ,,
24
25 N      = len(diag)
26 d_ii   = np.copy(diag)
27 d_up   = np.copy(sup_diag)
28 d_lo   = np.copy(inf_diag)
29 b      = np.copy(b)
30 x      = np.zeros(N)
31
32 # Forward elimination
33 for i in range(1, N):
34     if d_ii[i-1] == 0.0:
35         raise ValueError("Division by zero, non-invertible matrix")
36     factor = d_lo[i-1] / d_ii[i-1]
37     d_ii[i] -= factor * d_up[i-1]
38     b[i]   -= factor * b[i-1]
39
40 # Back substitution
41 if d_ii[-1] == 0.0:
42     raise ValueError("Division by zero, non-invertible matrix")
43
44 x[-1] = b[-1] / d_ii[-1]
45 for i in range(N-2, -1, -1):
46     b[i] -= d_up[i] * x[i+1]
47     if d_ii[i] == 0.0:
48         raise ValueError("Division by zero, non-invertible matrix")
49     x[i] = b[i] / d_ii[i]
50
51 return x
52
53
54 if __name__ == "__main__":
55     np.random.seed(69420)
56     N   = 1000
57     d1 = np.random.normal(size=[N])
58     d2 = np.random.normal(size=[N-1])
59     d3 = np.random.normal(size=[N-1])
60     A  = np.diag(d2, -1) + np.diag(d1, 0) + np.diag(d3, 1)
61     b  = np.random.normal(size=[N])
62
63     start = time.time()
64     x1 = solve_tridiagonal(d1, d3, d2, b)
65     print(f"Elapsed time = {time.time()-start}")
66
67     start = time.time()
68     x2 = np.linalg.solve(A, b)
69     print(f"Elapsed time = {time.time()-start}")
70
71     d = np.sqrt(np.sum((x1 - x2)**2))
72     print(f'difference with numpy = {d}')
73
74 [Output]
75 Elapsed time = 0.0034050941467285156
76 Elapsed time = 0.05156588554382324
77 difference with numpy = 3.294673424535412e-11

```

Tra l'altro se notate, grande vantaggio, abbiamo allocato solo 3 vettori. Quindi possiamo risparmiare molto non solo in tempo ma anche in memoria.

## B 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" (capiremo poi il perchè del nome) ma qui vogliamo fare le cose a mano.

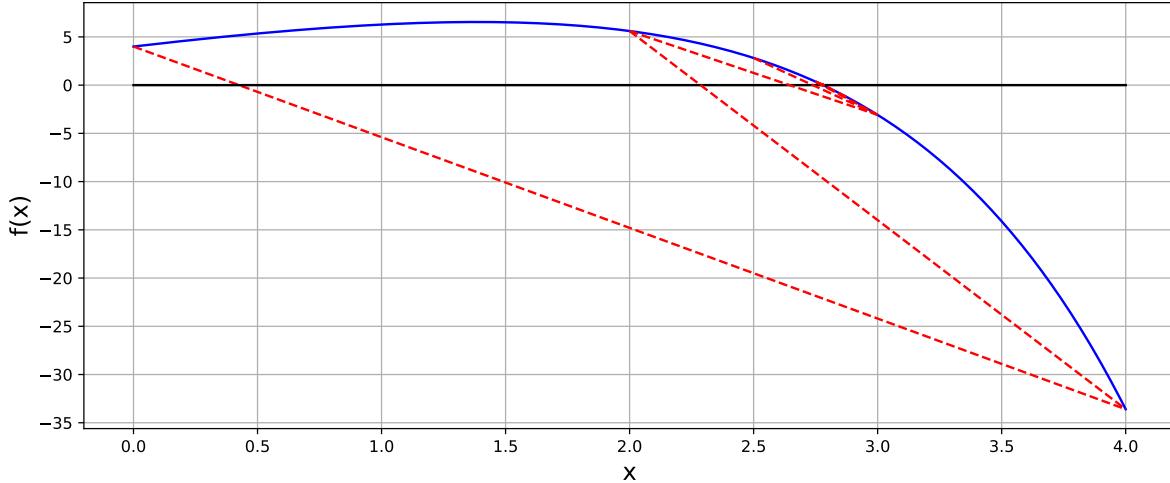
### B.1 Bisezione

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:

Costuzione metodo di bisezione



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', fontsize=15)
10 plt.xlabel('x', fontsize=15)
11 plt.ylabel('f(x)', fontsize=15)
12 plt.plot(x, f(x), 'b')
13 plt.plot([a, b],[f(a), f(b)], linestyle='--', c='r')
14 plt.plot(x, x*0, 'k')
15
16 iter = 1
17 #fai finche l'intervallo e' piu' grande della tolleranza
18 while (b-a) > t:
19     c = (a+b)/2.0 #punto medio
20     fc = f(c)
21     #se hanno lo stesso segno allora c e' piu' vicino allo zero che a
22     if fc*fa > 0:
23         a = c
24     #altrimenti e' b che e' piu' lontano
25     else:
26         b = c
27     iter += 1
28     plt.plot([a, b],[f(a), f(b)], linestyle='--', c='r')
29
30 plt.grid()
31 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...

## B.2 Metodo di Newton

Il metodo di Newton, o delle tangenti, è in genere più comune e molto usato, vediamo come funziona: Se si considera un punto  $x_0$  molto vicino alla soluzione possiamo espandere in serie di taylor la nostra funzione e ottenere:

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

Ora se parto da più lontano  $s$  non sarà la soluzione, ma possiamo usare il valore trovato come nuovo punto di partenza e rifare il conto. Abbiamo quindi la regola per questo 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 qualora non sia fattibile la derivata va calcolata numericamente, coi metodi visti prima. Facciamo quindi una breve digressione per conoscere l'utilizzo di questa nuova libreria.

### B.2.1 Calcolo simbolico

Il calcolo simbolico è fondamentalmente quello che ognuno di voi fa con carta e penna. Per esempio  $\frac{4}{6}$  se lo calcolate e stampate il valore troverete qualcosa del tipo 0.6666... mentre voi con carta e penna dite che  $\frac{4}{6} = \frac{2}{3}$  e vi fermate qui, non fate altre sostituzioni o conti. Ecco un codice che usa il calcolo simbolico facciamo quest'ultima cosa. Vediamolo praticamente:

```

1 import sympy as sp
2
3 x = sp.Symbol('x')
4 y = sp.Symbol('y')
5
6 z = x/y
7 print(z)
8 print(z.subs(x, 4).subs(y, 6))
9 print(4/6)
10
11 [Output]
12 x/y
13 2/3
14 0.6666666666666666

```

Per utilizzare il calcolo simbolico ci avvaliamo di una libreria "Sympy". Analizziamo il codice: Per prima cosa definiamo le due variabili, i due simboli, "x" e "y", dopo di che creiamo la variabile che rappresenta il rapporto e vedrete subito che se stampata su shell otteniamo proprio " $x/y$ ". Per avere il risultato numerico dobbiamo sostituire ai due simboli i valori numerici e o facciamo attraverso la funzione "subs" che come vedete non è di difficile intuizione, il primo argomento è il simbolo, il secondo è il valore numerico. Vediamo ora esempi più interessanti:

```

1 import sympy as sp
2
3 #=====
4 # Equations
5 #=====
6 print(sp.solve(x**2 - 2, x))
#=====
7 # Limit
8 #=====
10 print(sp.limit(sp.log(x)*x, x, 0))
#=====
12 # Derivatives
13 #=====
14 f = sp.sin(x)*sp.exp(x)
15 dfdx = sp.diff(f, x)
16 print(dfdx)
#=====
18 # Integrals
19 #=====
20 I = sp.integrate(sp.exp(-x**2), (x, -sp.oo, sp.oo))
21 print(I)
22 I = sp.integrate(sp.sin(x), x)
23 print(I)
#=====
25 # Differential equations
26 #=====
27 t = sp.Symbol('t')
28 y = sp.Function('y')
29 eq = sp.Eq(y(t).diff(t, t) + y(t), 0) # y'' + y = 0
30 sol = sp.dsolve(eq, y(t))
31 print(sol)
32
33 [Output]
34 [-sqrt(2), sqrt(2)]
35 0
36 exp(x)*sin(x) + exp(x)*cos(x)
37 sqrt(pi)
38 -cos(x)
39 Eq(y(t), C1*sin(t) + C2*cos(t))

```

Quindi vedete bene che possiamo risolvere equazione algebriche, calcolare limiti, derivate, integrali e anche risolvere equazioni differenziali. Per i nostri attuali scopi ci serve soltanto fare le derivate. Torniamo ora quindi al metodo di newton e vediamo il codice:

```

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
partenza')
19
20    return x0, iter
21
22 #valore iniziale da cui partire
23 init = 4.4
24
25 xs, iter = tangenti(init, t)
26
27 print(iter , " iterazioni necessarie")
28
29 print("xs= %.15f" %xs)
30
31 print("|f(xs)|= %e" %abs(f.subs(x,xs)))
32
33 [Output]
34 7 iterazioni necessarie
35 xs= 4.493409457909064
36 |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:

```

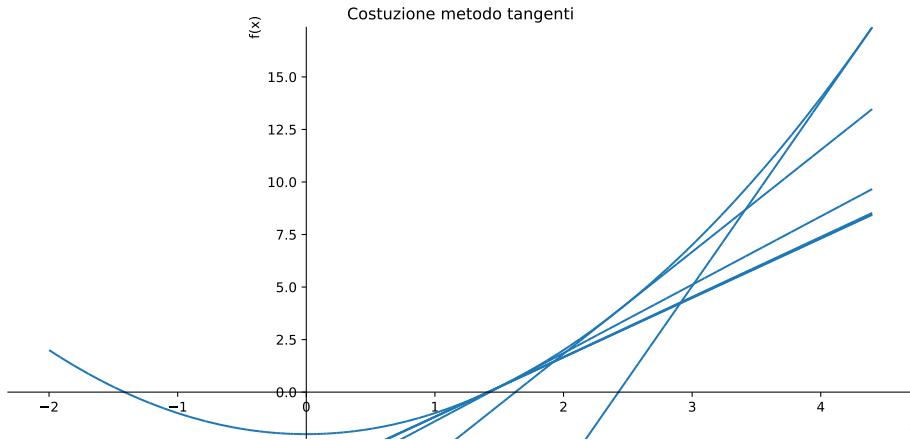
1 import sympy as sp
2 from sympy.plotting import plot
3
4 x = sp.Symbol('x')
5 f = x**2 - 2 #funzione di cui trovare gli zeri
6 df = sp.diff(f, x) #derivata della funzione f
7 t = 1e-13 #tolleranza
8 init = 4.4
9 P1 = plot(f, (x, -2, init), ylim=(-2.2, f.subs(x,init)), show=False, title='Costuzione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

```

```

30
31 print(iter , " iterazioni necessarie")
32 print("xs= %.15f" %xs)
33 print("|f(xs)|= %e" %abs(f.subs(x,xs)))
34
35 P1.show()
36
37 [Output]
38 7  iterazioni necessarie
39 xs= 1.414213562373095
40 |f(xs)|= 4.440892e-16

```



Purtroppo questo metodo può presentare problemi dipendenti dalla concavità della funzione, provate a risolvere l'equazione  $x^3 - 2x + 2 = 0$  vi accorgerete che a seconda di dove partite succedono cose strane... (Hint: non partire da zero o da uno).

### B.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  $\mathbf{x}$  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:

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

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. La funzione che implemeneteremo vale per un sistema di  $N$  equazioni, e per semplicità qui usaremos le funzioni di numpy per risolvere il sistema.

```

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 def newton(f, start, tol, args=(), dense_output=False, max_it=1000):
12     """
13     Generalizzation of newton method for n equations
14
15     Parameters
16     -----
17     f : callable
18         A vector function to find a root of.
19     start : ndarray
20         Initial guess. The method is very sensitive to this
21         must be chosen carefully
22     tol :float, optional, default 1e-8
23         required tollerance
24     args : tuple, optional
25         Extra arguments passed to f
26     dense_output : bool, optional
27         true for full and number of iteration
28     max_it : int, optional, default 1000
29         after max_it iteration the code stop raising an exception
30
31     Return
32     -----
33     x0 : 1darray
34         solution of the system
35         if dense_outupt=True all iteration are returned
36         in a matrix called X and also the number of iteration
37         '',
38
39     # initial guess
40     x0 = start
41     f0 = f(x0, *args)
42     # for the computation of jacobian
43     nd = len(x0)
44     df = np.zeros((nd, nd))
45     h = 1e-8
46     s = np.zeros(nd)
47     #for full output
48     X = []
49     if dense_output : X.append(x0)
50     # count
51     n_iter = 0
52
53     while True:
54
55         # compute jacobian using symmetric derivative
56         for i in range(nd):          # loop over functions
57             for j in range(nd):      # loop over variables
58                 s[j] = 1
59                 xr, xl = x0 + h*s, x0 - h*s
60                 df[i, j] = (f(xr, *args) - f(xl, *args))[i]/(2*h)
61                 s[:] = 0
62
63         # update solution
64         delta = np.linalg.solve(df, f0)
65         x0 = x0 - delta
66         f0 = f(x0, *args)
67
68         if dense_output:
69             n_iter += 1
70             X.append(x0)
71
72         # stop condition
73         if all(abs(f0) < tol):
74             break
75         # check iterations
76         if n_iter > max_it :
77             err_msg = 'too many iteration, failure to converge, change initial guess'
78             raise Exception(err_msg)
79
80         if dense_output:
81             return np.array(X), n_iter
82         else:
83             return x0
84
85
86

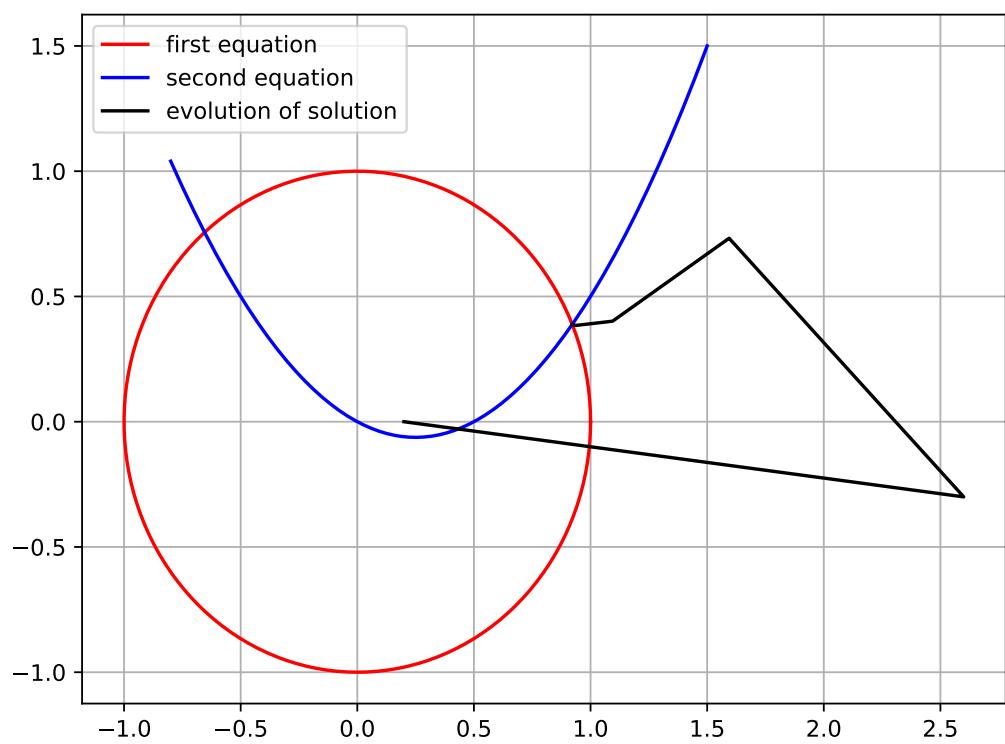
```

```

87 #=====
88 # System to solve
89 #=====
90
91 def system(V):
92     x1, x2 = V
93
94     r1 = x1**2 + x2**2 - 1#x3
95     r2 = x2 - x1**2 + x1/2 #+ x3/5
96     #r3 = x3**2 + 5*x2 - 7
97
98     R = np.array([r1, r2])#, r3])
99     return R
100
101 #=====
102 # Solution and compare with scipy
103 #=====
104
105 init = np.array([0.2, 0.0])
106 tol = 1e-12
107 sol, n_iter = newton(system, init, tol=tol, dense_output=True)
108 xs, ys = sol.T
109 print("Solution with newton: ", *sol[-1], "in", n_iter, "iterations")
110
111 sol = root(system, init, method='hybr', tol=tol)
112 print("Solution with root: ", *sol.x)
113
114 sol = fsolve(system , init, xtol=tol)
115 print("Solution with fsolve: ", *sol)
116
117 #=====
118 # Plot
119 #=====
120
121 t = np.linspace(0, 2*np.pi, 1000)
122 z = np.linspace(-0.8, 1.5, 1000)
123 plt.figure(1)
124 plt.grid()
125 plt.plot(np.cos(t), np.sin(t), 'r', label='first equation')
126 plt.plot(z, z**2- z/2, 'b', label='second equation')
127 plt.plot(xs, ys, 'k', label='evolution of solution')
128 plt.legend(loc='best')
129 plt.show()
130
131 [Output]
132 Solution with newton:  0.9214908788160613  0.38840000033316596 in 7 iterations
133 Solution with root:    0.9214908788160611  0.388400000333166
134 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 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".



## C Risolvere numericamente le ODE: IVP

In questa sezione ci occuperemo delle eqazioni differenziali ordinarie (ODE), nella fattispecie dei così-detti problemi ai valori iniziali (initial value problem: IVP).

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, 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, possiamo dire. Sia:

$$\begin{cases} f'(t) = g(t, f(t)) \\ x(t_0) = x_0 \end{cases} \quad \begin{array}{l} \text{equazione differenziale} \\ \text{condizione iniziale} \end{array} \quad (31)$$

il problema ai valori iniziali da risolvere. Il primo passo è quello di scegliere il range in cui vogliamo risolvere la nostra equazione, ad esempio fra  $t_0$  e  $t_1$  e dividiamo questo intervallo in  $N$  punti. Questa sarà la nostra griglia, che nel limite di distanza fra due punti adiacenti che tende a zero ritorna il nostro intervallo continuo. Abbiamo quindi che tutte le varie quantità andranno calcolate su questa griglia (o comunque a partire da essa).

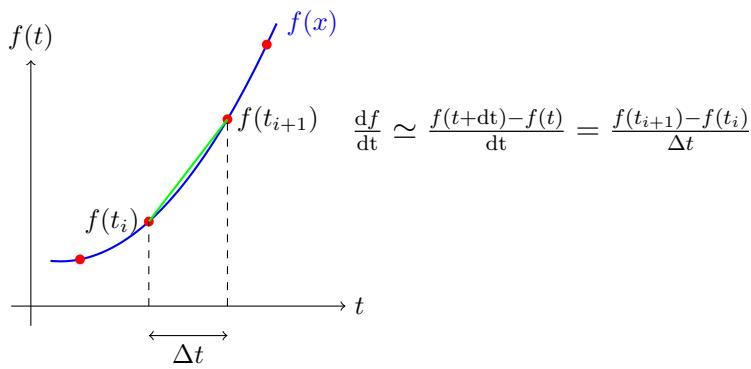


Figura 10: Rappresentazione della discretizzazione e del calcolo della derivata discreta.

Presi quindi due punti della griglia potremo calcolare la derivata della nostra funzione usando la definizione, ovvero il rapporto incrementale. Abbiamo dunque:

$$f' = \frac{df}{dt} \xrightarrow{\text{sulla griglia}} \frac{f(t_{i+1}) - f(t_i)}{\Delta t} \quad (32)$$

Sapendo quindi la forma funzionale della derivata, ovvero la  $g(t, f(t))$ , data dall'equazione differenziale, possiamo ottenere la soluzione dell'equazione per passi:

$$f(t_{i+1}) = f(t_i) + \Delta t g(t_i, f(t_i)) \quad (33)$$

quindi possiamo trovare la soluzione al tempo  $t_{i+1}$ , sapendo quella al tempo  $t$ , e andando a ritroso, grazie alla condizione iniziale possiamo avere tutta la soluzione. Inoltre nella  $g$  non compare la dipendenza da  $f(t_{i+1})$  ma solo da  $f(t_i)$ , per questo il metodo è chiamato esplicito (ovvero al momento del calcolo abbiamo già tutte le informazioni necessarie per eseguirlo). Importante notare che la discretizzazione che abbiamo deciso di usare è del tutto arbitraria (la più semplice possibile, che ci fornisce il metodo di Eulero), e anche il passo  $\Delta t$  è un argomento delicato, vediamo perché.

### C.1 Calcolo delle derivate

Vediamo un esempio di come varia l'errore nel calcolo di una derivata numerica al variare dell'incremento:

```

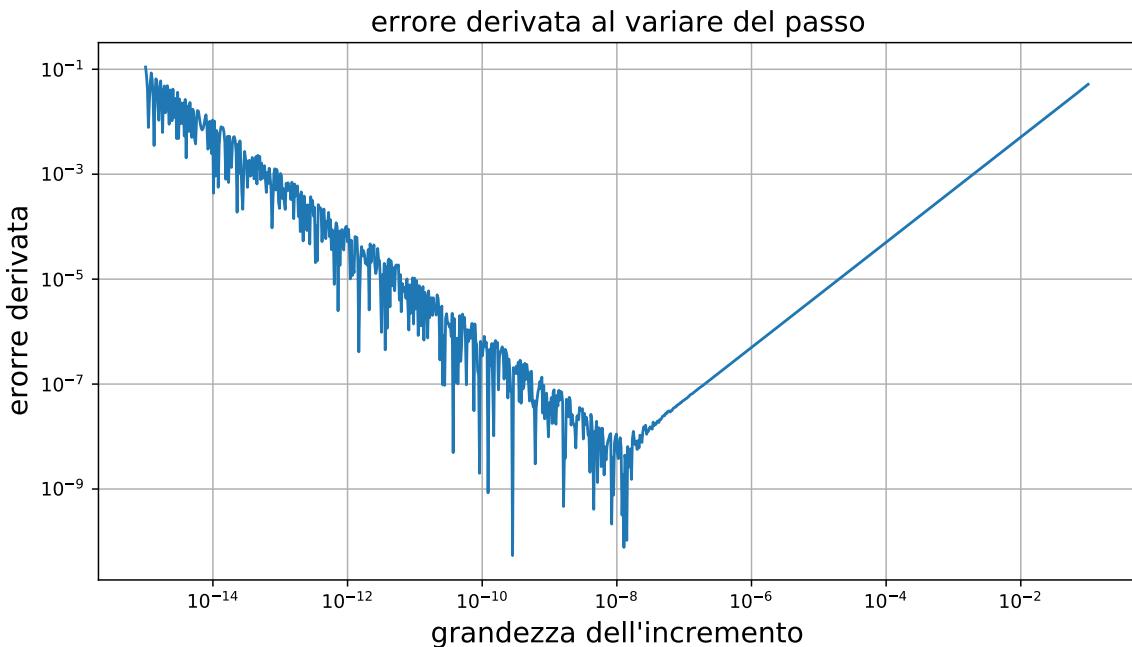
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def f(x):
5     """
6         funzione di cui calcolare la derivata
7     """
8     return np.exp(x)
9

```

```

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', fontsize=15)
22 plt.ylabel('errore derivata', fontsize=15)
23 plt.xlabel("grandezza dell'incremento", fontsize=15)
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  $10^{-2}$ , questo è fondamentalmente dovuto al fatto che il computer ha difficoltà con le operazioni. 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:

$$f' = \frac{f(x+h) - f(x)}{h} \quad (34)$$

$$f' = \frac{f(x) - f(x-h)}{h} \quad (35)$$

$$f' = \frac{f(x+h) - f(x-h)}{2h} \quad (36)$$

$$f' = \frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h} \quad (37)$$

$$f' = \frac{3f(x) - 4f(x-h) + f(x-2h)}{2h} \quad (38)$$

$$f' = \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h} \quad (39)$$

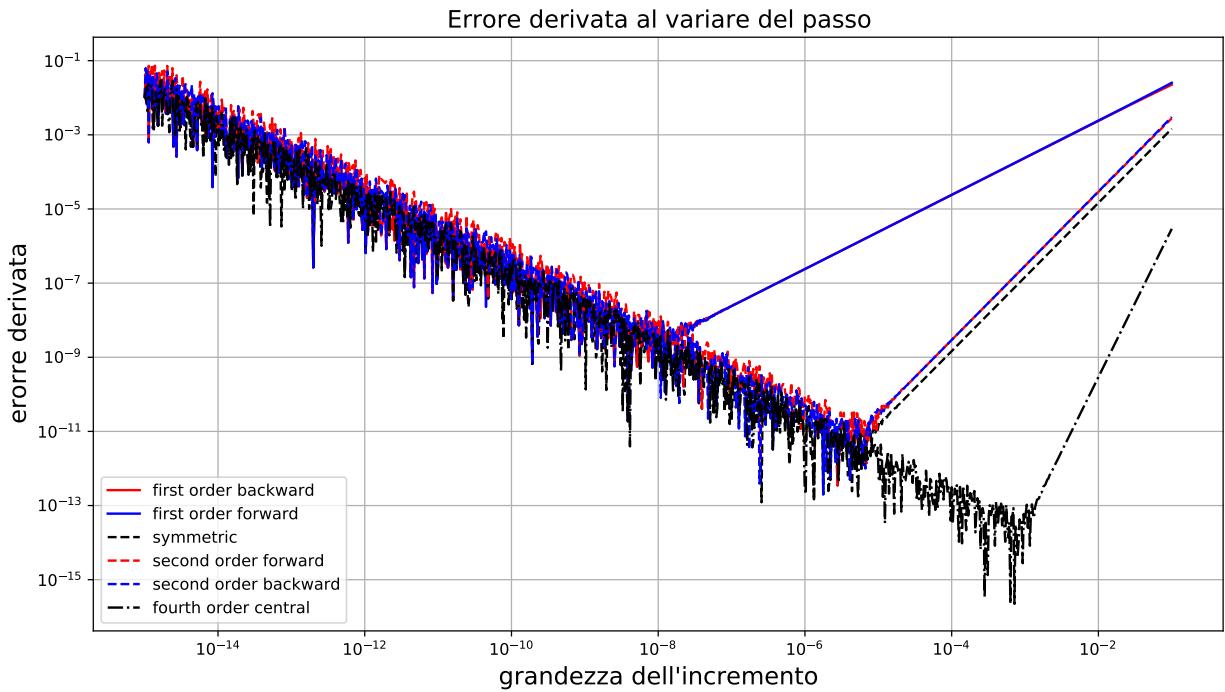
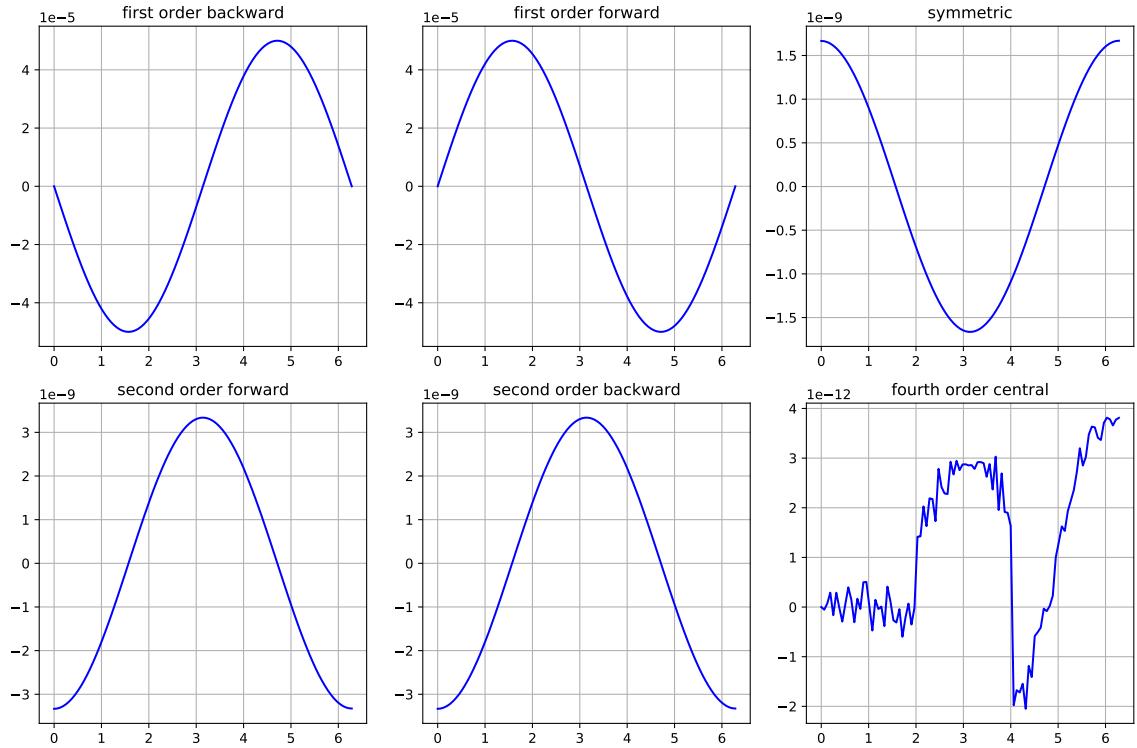
Rispettivamente i primi due due primo ordine, i seguenti tre del secondo ordine e il terzo del quarto ordine. Vediamo un piccolo codice nel quale vi potete divertire a cambiare la grandezza passo per rendervi conto di quanto dicevamo:

```

1 """
2 Code that compiles some method of calculating a derivative to various orders of accuracy
3 """
4
5 import numpy as np
6 import matplotlib.pyplot as plt
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()

```

Se prima abbiamo visto la derivata calcolata in un singolo punto, qui vediamo la derivata calcolata su tutta la nostra griglia con un passo di discretizzazione fisso però. E possiamo vedere come gli errori siano dello stesso ordine di grandezza separatamente per i metodi al primo e al secondo ordine. Più interessante è però provare a vedere come appare ora il grafico fatto sopra. Si tratta di poche righe che lascio a voi in modo che facciate le vostre bellurie.



Potete divertirvi a prendere le parti finali delle curve, quelle non rumorose, e calcolare i coefficienti angolari, se c'è giustizia a questo mondo, essi saranno parenti di 1, 2 e 4. (Tra l'altro nel caso del secondo ordine si vede ad occhio in quanto passando due decadi, da  $10^{-4}$  a  $10^{-2}$ , la y passa da  $10^{-9}$  a  $10^{-5}$ , 4 decadi, quindi proprio quadratico).

## C.2 ODE: caso esponenziale

Torniamo ora alle equazioni differenziali vere e proprie. 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.

```

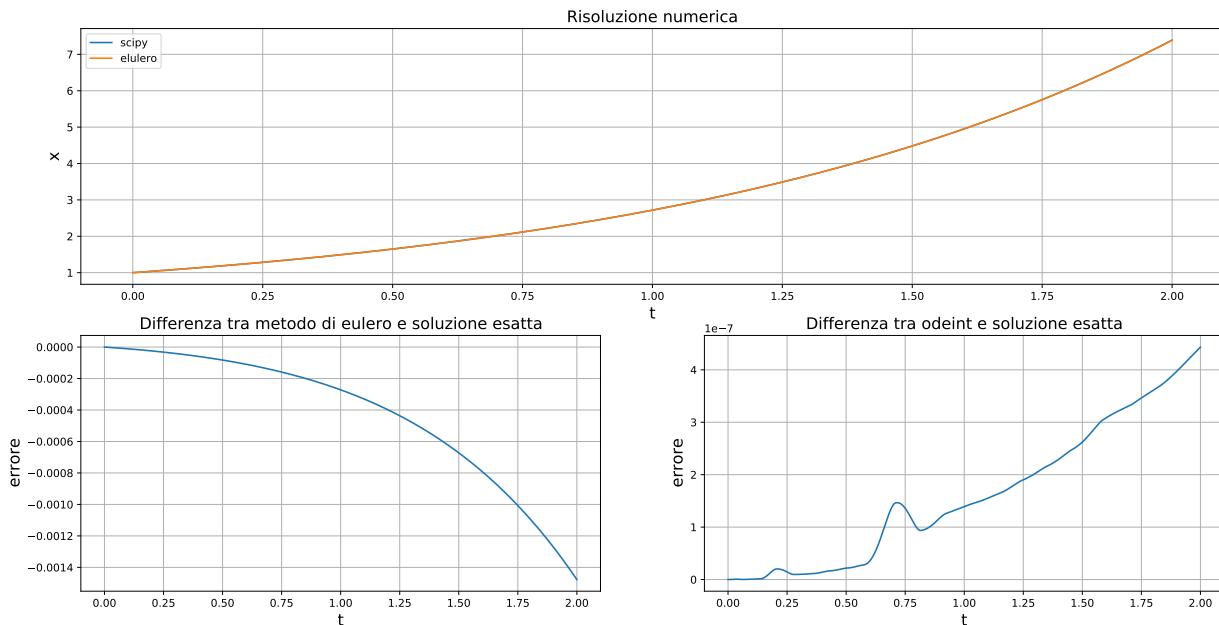
1 import numpy as np
2 import scipy.integrate
3 import matplotlib.pyplot as plt
4
5 #parametri
6 x0 = 1      #condizione iniziale
7 tf = 2      #fino a dove integrare
8 N = 10000   #numero di punti
9
10 #odeint
11 def ODE_1(y, t):
12     """
13         equazione da risolvere per odeint
14     """
15     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 egualiarla
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', fontsize=15)
59 ax1.set_xlabel('t', fontsize=15)
60 ax1.set_ylabel('x', fontsize=15)
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', fontsize=15)
68 ax2.set_xlabel('t', fontsize=15)
69 ax2.set_ylabel('errore', fontsize=15)
70 ax2.plot(t, x_eulero-np.exp(t))
71 ax2.grid()
72

```

```

73 ax3 = plt.subplot(224)
74 ax3.set_title('Differenza tra odeint e soluzione esatta', fontsize=15)
75 ax3.set_xlabel('t', fontsize=15)
76 ax3.set_ylabel('errore', fontsize=15)
77 ax3.plot(t, x_scipy-np.exp(t))
78 ax3.grid()
79
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. (Inoltre potremmo dire che questa equazione differenziale rientra nelle equazioni stiff quindi una casistica particolare, in cui metodi impliciti funzionano meglio che metodi esplicativi ma va beh). 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;

### C.3 Sistema di ODE: pendolo semplice

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

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

È la famosa equazione del pendolo semplice che approssimata dà luogo all'oscillatore armonico ovvero a tutta la fisica. Noi andremo a risolvere il secondo sistema, in modo da non dover modificare l'algoritmo visto. Vediamo come si modifica il codice di sopra ora:

```

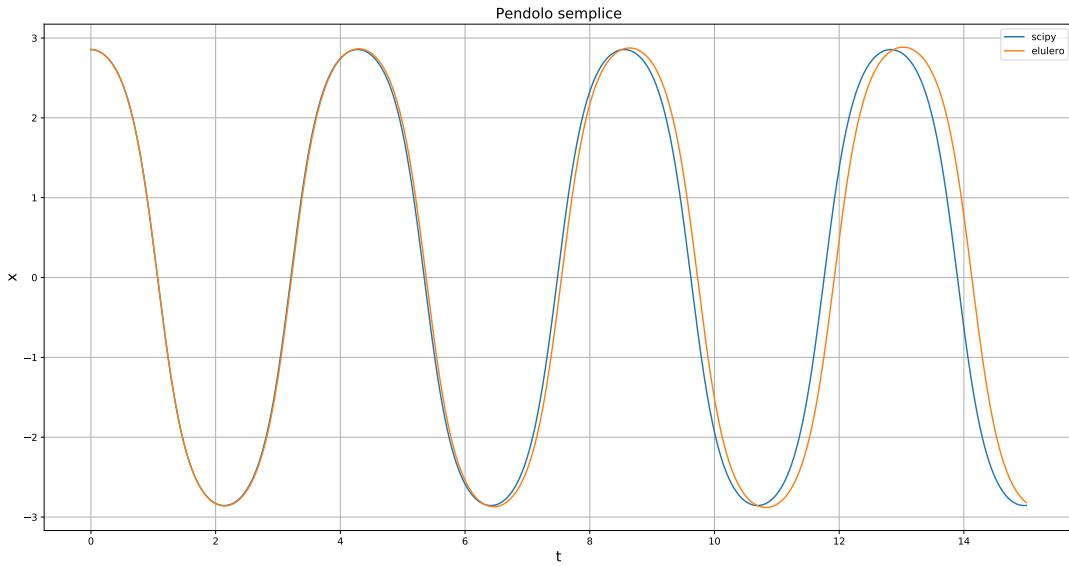
1 import numpy as np
2 import scipy.integrate
3 import matplotlib.pyplot as plt
4
5 #parametri
6 N = 100000      #numero di punti
7 l = 1            #lunghezza pendolo
8 g = 9.81         #accelerazione di gravita'
9 o0 = g/l        #frequenza piccole oscillazioni
10 v0 = 0           #condizioni iniziali 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 egualiarla
44         perche dx/dt = g(x(t)) quindi (x[i+1]-x[i])/dt = g(x[i])
45         da cui si isola x[i+1]. Dove x e' pero' un array contenente
46         sia le posizioni sia le velocita'; nel codice pero' per
47         semplicita' si e' scelto di dividere le due grandezze.
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 eulero non è delle migliori perché è un metodo del primo ordine e per tempi lunghi quindi ci sono dei problemi. Anche se non è l'unica causa, in un linguaggio che useremo più avanti possiamo dire è dovuto al fatto che la trasformazione non è canonica; per ora però non preoccupiamoci. Come ultima cosa menzioniamo l'esistenza di algoritmi adattivi, ovvero algoritmi dove il passo di integrazione cambia man mano che l'integrazione va avanti, in modo da mantenere l'errore di integrazione sotto una certa soglia.

## C.4 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 gravità
10 o0 = g/l       #frequenza piccole oscillazioni
11 v0 = 0          #condizioni iniziali velocità
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()

```

Il codice è abbastanza auto esplicativo, le poche cose importanti sono che a linea 47 scriviamo "line1," perché "plt.plot" restituisce una lista con un singolo elemento e quindi la virgola ci permette di spacchettare la lista. Inoltre la funzione "animate" deve restituire un'iterabile. Provate da voi ad eseguire il codice e vedrete il pendolo oscillare.

## D Risolvere numericamente le ODE: BVP

In questa sezione si vuole, invece, riprendendo le equazioni differenziali ordinarie introdurre i problemi con condizioni al bordo, (boundary value problem BVP). Questo tipo di problemi sono un po' più delicati in quanto la soluzione non è detto che sia unica. Due metodi famosi per risolvere questi problemi sono il metodo di shooting, utile anche per equazioni come l'equazione di Schrödinger, e il metodo di rilassamento.

### D.1 Shooting

Supponiamo di avere il seguente problema al bordo:

$$\begin{cases} \ddot{y}(t) = f(t, y(t), \dot{y}(t)) \\ y(t_0) = y_0 \\ y(t_1) = y_1 \end{cases} \quad (40)$$

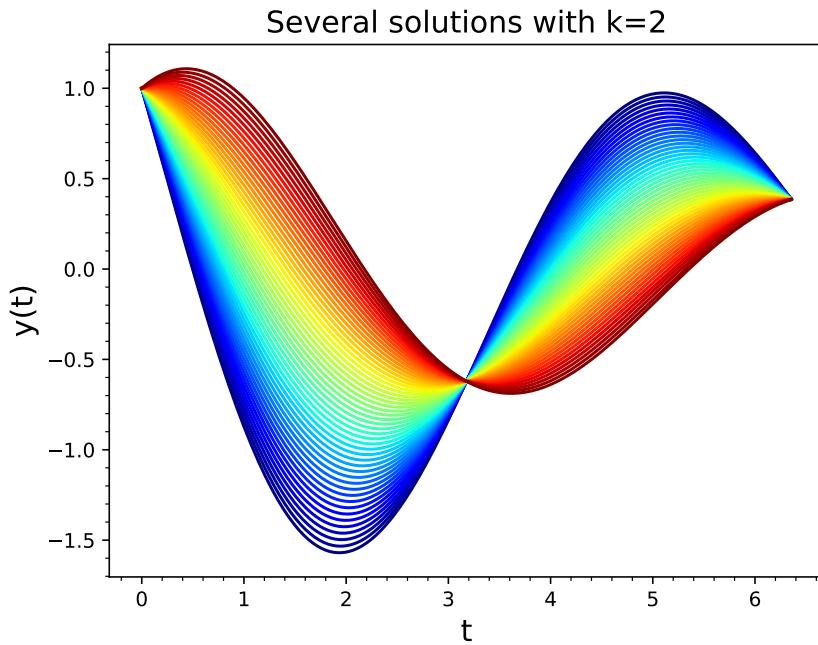
Quel che si può fare è trasformare tale problema in un problema ai valori iniziali dipendenti da un parametro, chiamiamolo ' $s$ ':

$$\begin{cases} \ddot{y}(t) = f(t, y(t), \dot{y}(t)) \\ y(t_0) = y_0 \\ \dot{y}(t_0) = s \end{cases} \quad (41)$$

Per cui detta  $F(s) = y(t_1, s) - y_1$ , ci basta trovare lo zero di questa funzione e avremo la condizione iniziale  $s$  che ci dà il valore al bordo da noi desiderato. Consideriamo la seguente equazione differenziale:

$$\ddot{y}(t) + \gamma \dot{y}(t) + \omega_0^2 y(t) = 0 \quad , \quad (42)$$

Si può vedere che le soluzioni di questa equazione al variare della condizione iniziale sulla derivata prima assumono lo stesso valore in dati istanti di tempo:  $t_k = k\pi/\omega$  dove  $k$  è un generico numero naturale e  $\omega^2 = \omega_0^2 - (\gamma/2)^2$ . Vediamo queste soluzioni con  $k = 2$ :



Mi spiace per gli amici daltonici, ma ammetto che questo grafico è stato fatto solo perché bellino vederlo arcobaleno. Non riportiamo tutto il codice ma solo le due funzioni principali, sia perché il resto del codice non è così istruttivo, sia per invogliarvi a scrivere da voi. In ogni caso però nella cartella sarà presente tutto.

Abbiamo detto quindi che ci siamo ricondotti a risolvere una ODE come nella sezione di sopra; sfruttiamo l'occasione per introdurre un nuovo metodo di integrazione: un predittore correttore del quarto ordine, Adams-Bashforth-Moulton. Esso fondamentalmente è l'unione di un metodo esplicito, con cui si fa una prima stima della soluzione, la predizione, e poi si inserisce il valore all'interno di un metodo隐式 per ottenere il valore

finale, la correzione. Detta  $f$  la funzione dell'equazione differenziale:

$$\bar{y}_{i+1} = y_i + \frac{h}{24}(55f(t_i, y_i) - 59f(t_{i-1}, y_{i-1}) + 37f(t_{i-2}, y_{i-2}) - 9f(t_{i-3}, y_{i-3})) \quad \text{A.B. predico,} \quad (43)$$

$$y_{i+1} = y_i + \frac{h}{24}(9f(t_{i+1}, \bar{y}_{i+1}) + 19f(t_i, y_i) - 5f(t_{i-1}, y_{i-1}) + f(t_{i-2}, y_{i-2})) \quad \text{A.M. correggo.} \quad (44)$$

Può ora sorgere un dubbio, come otteniamo i primi tre valori necessari per fare il primo passo di predizione? Dato che si tratta di un integratore di quarto ordine usiamo per i primi tre un Runge-Kutta di ordine 4 che brevemente riportiamo:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad (45)$$

dove:

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{1}{2}k_1 h\right) \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{1}{2}k_2 h\right) \\ k_4 &= f(t_n + h, y_n + k_3 h) \end{aligned}$$

Volendo potremmo usare direttamente un Runge-Kutta del quarto ordine come integratore. Il vantaggio di usare un predittore correttore come quello sopra esposto nasce dal fatto che ad ogni iterazione necessitiamo di una sola chiamata della funzione  $f$  cioè quando calcoliamo  $f(t_{i+1}, \bar{y}_{i+1})$  in quanto gli altri termini sono già stati calcolati e quindi li conserviamo per non calcolarli nuovamente. Se si vuole usare un Runge-Kutta che ha invece più chiamate alla funzione possiamo "aggirare" il problema modificando l'algoritmo in modo da dargli un passo adattivo. Quindi sempre tante chiamate alle funzioni ma almeno ora avremo un errore piccolo. Passiamo ora al codice:

```

1 #=====
2 # Integration: Adams-Bashforth-Moulton predictor and corrector of order 4
3 #=====
4
5 def AMB4(num_steps, t0, tf, f, init, args=()):
6     """
7         Integrator with Adams-Bashforth-Moulton
8         predictor and corrector of order 4
9
10    Parameters
11    -----
12        num_steps : int
13            number of point of solution
14        t0 : float
15            lower bound of integration
16        tf : float
17            upper bound of integration
18        f : callable
19            function to integrate, must accept vectorial input
20        init : 1darray
21            array of initial condition
22        args : tuple, optional
23            extra arguments to pass to f
24
25    Return
26    -----
27        X : array, shape (num_steps + 1, len(init))
28            solution of equation
29        t : 1darray
30            time
31        """
32        #time steps
33        dt = tf/num_steps
34
35        X = np.zeros((num_steps + 1, len(init))) #matrice delle soluzioni
36        t = np.zeros(num_steps + 1)                 #array dei tempi
37
38        X[0, :] = init                            #condizioni iniziali
39        t[0] = t0
40
41        #primi passi con runge kutta
42        for i in range(3):

```

```

43         xk1 = f(t[i], X[i, :], *args)
44         xk2 = f(t[i] + dt/2, X[i, :] + xk1*dt/2, *args)
45         xk3 = f(t[i] + dt/2, X[i, :] + xk2*dt/2, *args)
46         xk4 = f(t[i] + dt, X[i, :] + xk3*dt, *args)
47         X[i + 1, :] = X[i, :] + (dt/6)*(xk1 + 2*xk2 + 2*xk3 + xk4)
48         t[i + 1] = t[i] + dt
49
50     # Adams-Bashforth-Moulton
51     i = 3
52     ABO = f(t[i], X[i, :], *args)
53     AB1 = f(t[i-1], X[i-1, :], *args)
54     AB2 = f(t[i-2], X[i-2, :], *args)
55     AB3 = f(t[i-3], X[i-3, :], *args)
56
57     for i in range(3,num_steps):
58         #predico
59         X[i + 1, :] = X[i, :] + dt/24*(55*AB0 - 59*AB1 + 37*AB2 - 9*AB3)
60         t[i + 1] = t[i] + dt
61         #correggo
62         AB3 = AB2
63         AB2 = AB1
64         AB1 = ABO
65         ABO = f(t[i+1], X[i + 1, :], *args)
66
67         X[i + 1, :] = X[i, :] + dt/24*(9*AB0 + 19*AB1 - 5*AB2 + AB3)
68
69     return X, t
70
71 #=====
72 # Binary research to find the right solution with shooting method
73 #=====
74
75 def SH(N, x0, start, xi, xf, step, x1, tau, f, args=()):
76     '''
77         Function that calculates zeros with the bisection method
78         Parameters
79         -----
80         N : Integer
81             number of integration steps.
82         x0 : float
83             initial condition on position.
84         start : float
85             initial condition on speed.
86         xi : float
87             initial time of integration.
88         xf : float
89             final time of integration.
90         step : float
91             start increment
92         x1 : float
93             boundary condition of solution
94         tau : float
95             tollerance on find value
96         f : callable
97             function to integrate, must accept vectorial input
98         args : tuple, optional
99             extra arguments to pass to f
100        Returns
101        -----
102        m : float
103            ideal intial condition for speed
104        sol : one dimensional array
105            solution of the equation
106    '''
107    a = start
108    sol = AMB4(N, xi, xf, f, init=(x0, a), args=args)
109    k = sol[0][-1, 0] - x1
110    while True:
111        b = a + step
112        sol = AMB4(N, xi, xf, f, init=(x0, b), args=args)
113        D = sol[0][-1, 0] - x1
114        if (k*D)<0.0:
115            break
116        k = D
117        a = b
118        while abs(a - b)>tau:

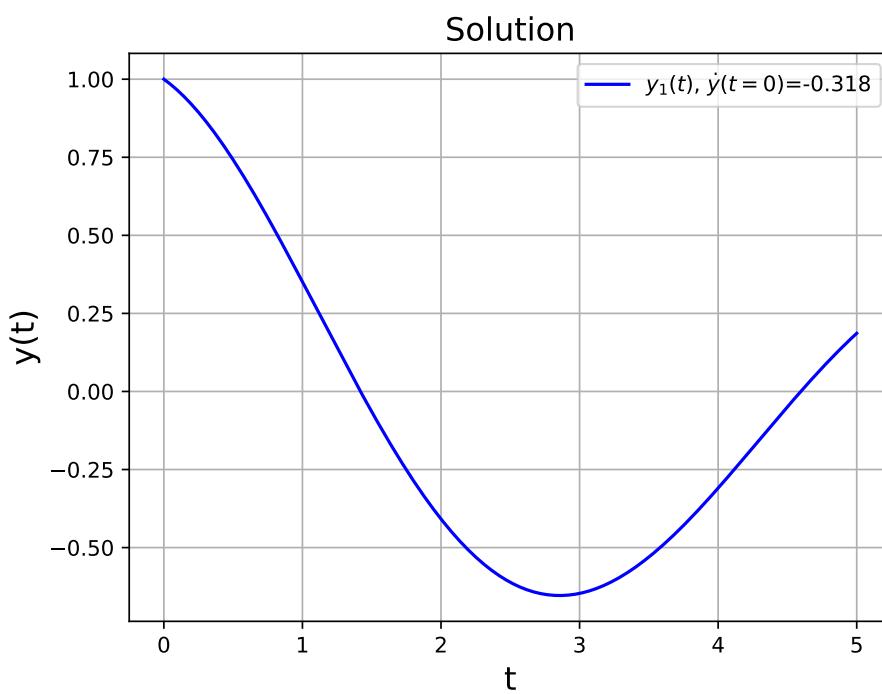
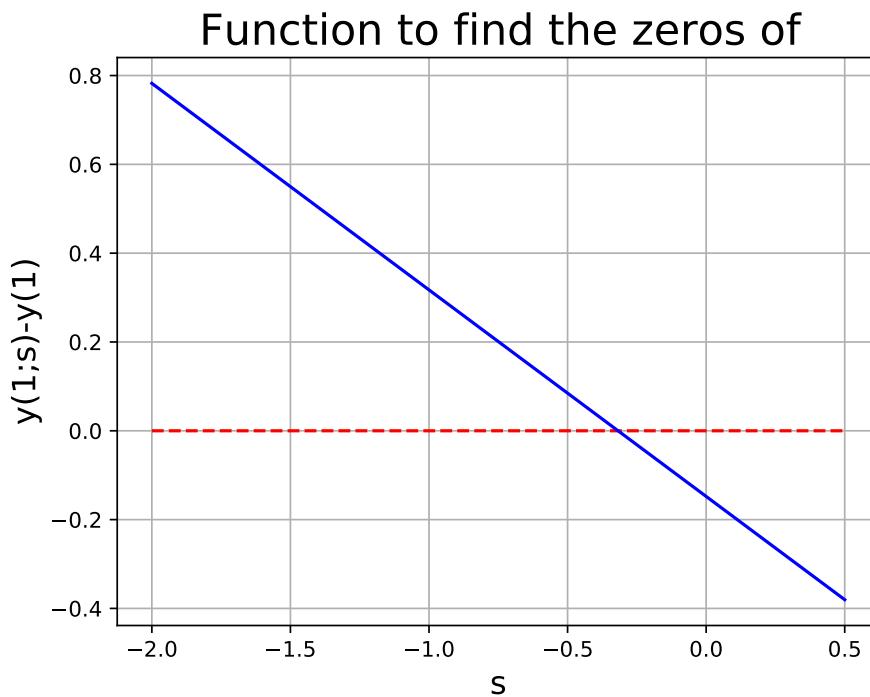
```

```

119     m = (a + b)/2.0
120     sol = AMB4(N, xi, xf, f, init=(x0, m), args=args)
121     M = sol[0][-1, 0] - x1
122     if (M*k)>0 :
123         k = M
124         a = m
125     else :
126         D = M
127         b = m
128
return m, sol

```

scegliendo ora come tempo finale  $t_f = 5$  e il valore al bordo  $x_1 = 0.1862$ , in modo che la soluzione sia unica, otteniamo:



## D.2 Relaxation

Un altro metodo per risolvere equazioni di questo tipo è il metodo di rilassamento, consideriamo come prima, un esempio del tipo precedente:

$$\begin{cases} \ddot{y}(t) = f(t, y(t), \dot{y}(t)) \\ y(t_0) = y_0 \\ y(t_1) = y_1 \end{cases} \quad (46)$$

Discretizzando la prima equazione su una griglia di  $N$  punti nella variabile indipendente  $t$  abbiamo:

$$\frac{y[i-1] - 2y[i] + y[i+1]}{h^2} = f(x[i], y[i], \frac{y[i-1] - y[i+1]}{2h}) \quad , \quad (47)$$

dove usiamo la derivata simmetrica per avere tutto al secondo ordine. Ora però non andiamo a trattare il singolo punto sulla griglia, ma li consideriamo tutti insieme; abbiamo dunque che il nostro problema diventa la soluzione di un sistema. Imponendo le condizioni al bordo otteniamo il seguente:

$$P\mathbf{y} = f(\mathbf{y}) - b \quad , \quad (48)$$

dove  $P$  è la matrice che contiene i coefficienti per il calcolo della derivata seconda in forma espansa diventa:

$$\frac{1}{h^2} \begin{pmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & 1 & -2 & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} y[0] \\ y[1] \\ \vdots \\ y[N-1] \\ y[N-2] \end{pmatrix} = \begin{pmatrix} f[0] \\ f[1] \\ \vdots \\ f[N-1] \\ f[N-2] \end{pmatrix} - \begin{pmatrix} y_0/h^2 \\ 0 \\ \vdots \\ 0 \\ y_1/h^2 \end{pmatrix} . \quad (49)$$

Però  $f$  è una generica funzione quindi fondamentalmente abbiamo un sistema di equazioni non lineare da risolvere nella forma:

$$H(\mathbf{y}) = 0 \quad \text{con} \quad H(\mathbf{y}) = P\mathbf{y} - f(\mathbf{y}) + b \quad , \quad (50)$$

quindi sappiamo bene che la soluzione la possiamo ottenere linearizzando  $H$  con il metodo di newton, si arriva dunque ad un'espressione iterativa:

$$\mathbf{y}^{n+1} = \mathbf{y}^n - \left[ P - \frac{\partial f_i}{\partial y_j} \right]^{-1} [P\mathbf{y} - f(\mathbf{y}) + b] \quad , \quad (51)$$

dove  $\frac{\partial f_i}{\partial y_j}$  è lo jacobiano di  $f$ . Come guess iniziale possiamo provare una funzione generica in linea di principio, certamente più siamo vicini, più il metodo converge senza problemi. La cosa più semplice è una retta che passa per i punti al bordo:

$$y_{\text{initial guess}} = (t - t_0) \frac{y_1 - y_0}{t_1 - t_0} + y_0 \quad . \quad (52)$$

Dove  $t$  è il nostro array che va da  $t_0$  a  $t_1$  in  $N$  passi. Scelta la guess, il sistema piano piano rilassa verso la soluzione cercata. Come criterio di stop calcoliamo la distanza tra le soluzioni ad ogni iterazione:

$$R = \sqrt{\sum (\mathbf{y}^{n+1} - \mathbf{y}^n)^2} \quad , \quad (53)$$

se questa quantità è minore di una certa tolleranza allora il programma termina. Vediamo ora il codice:

```

1 import numpy as np
2 from scipy.sparse import diags
3 import matplotlib.pyplot as plt
4
5 #=====
6 # Function for the solution of boundary value problem via relaxation
7 #=====
8
9 def relax(f, y0, y1, x, init, args=(), tol=1e-8, max_iter=100, dense_output=False):
10     """
11         Implementation of relaxation method for ODE 2pt-BVP.
12         The equation must be in the form: y''(x) = f(x, y, y')
13
14     Parameters
15     -----
16     f : callable
17         A vector function of differential equation like: y'' = f

```

```

18     y0, y1 : float
19         required value of solution at boundary
20     x : 1darray
21         array of position, or time, independent variable
22     init : 1darray
23         Initial guess.
24     args : tuple, optional
25         Extra arguments passed to f
26     tol :float, optional, default 1e-8
27         required tollerance
28     max_iter : int, optional, default 100
29         after max_it iteration the code stop raising an exception
30     dense_output : bool, optional, default False
31         true for full and number of iteration
32
33     Return
34     -----
35     yo : 1darray
36         solution of differential equation
37         if dense_outupt=True all iteration are returned
38         in a matrix called Y and also the number of iteration
39         '',
40
41     # parameter of discretizzation
42     N = len(x)
43     h = np.diff(x)[0]
44     # second derivative matrix
45     d2 = diags([1, -2, 1], [-1, 0, 1], shape=(N, N)).toarray()
46     d2 = d2/h**2
47     # bound values required
48     yb = [y0/h**2] + [0]*int(N-2) + [y1/h**2]
49     yb = np.array(yb)
50     # init guess from imput
51     yo = init
52     # interation count
53     it = 0
54     #for full output
55     Y = []
56     if dense_output : Y.append(yo)
57
58     while True:
59         # for jacobian computation
60         df = np.zeros(d2.shape)
61         s = np.zeros(N)
62
63         for i in range(N):
64             s[i] = 1
65             yr, yl = yo + h*s, yo - h*s
66             df[i, :] = (f(x, yr, h, *args) - f(x, yl, h, *args)) / (2*h)
67             s[:] = 0
68
69         yn = yo - np.linalg.solve(d2-df, d2@yo - f(x, yo, h, g, o02) + yb)
70         # residual
71         R = np.sqrt(np.sum((yn-yo)**2))
72         if R < tol:
73             yo = yn
74             break
75
76         if it > max_iter:
77             raise Exception("to many iteration")
78
79         #update
80         yo = yn
81         it = it + 1
82         if dense_output : Y.append(yo)
83
84     if dense_output:
85         return Y, it
86     else:
87         return yo
88
89 #=====#
90 # RHS of differential equations y' = f
91 #=====#
92 def f(t, y, h, g, o02):
93     ''

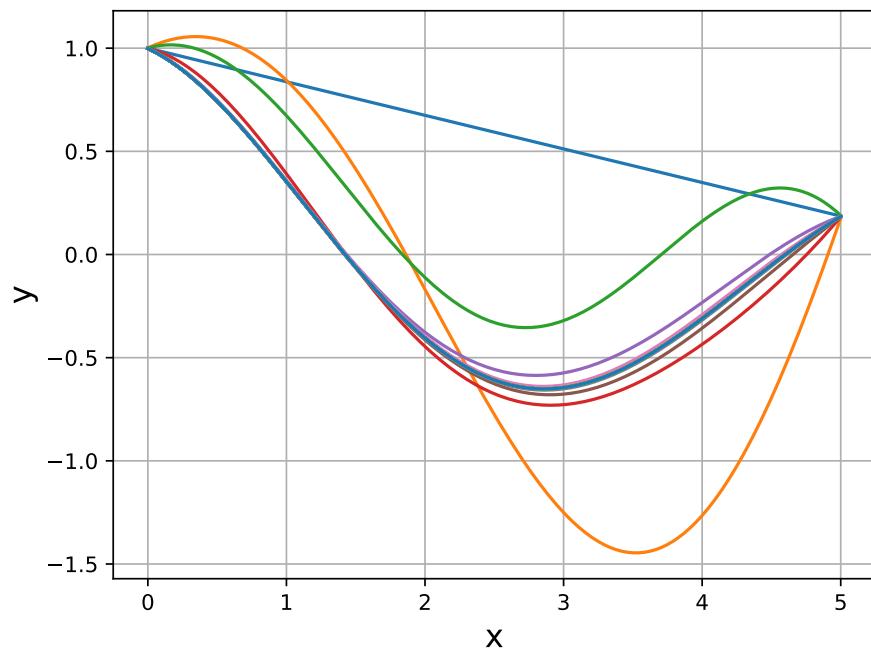
```

```

94     RHS of differential equations  $y'' = f$ 
95      $f$  can be a function non only of  $y$  but also  $y'$  so
96     we use a second order approximation to compute  $y'$ 
97
98     Parameter
99     -----
100    t : 1darray
101        independent variable
102    y : 1darray
103        solution or guess of solution
104    h : float
105        step's size for derivative computation
106    g, o02 : float
107        parameter of our differential equation
108
109    Return
110    -----
111    y_ddot : float
112        RHS of equation computed on a grid
113        ,
114        y_dot =  $(y[2:] - y[:-2])/(2*h)$  # second order derivative
115        y_dot_0 =  $(-3*y[0] + 4*y[1] - y[2])/(2*h)$  # second order derivative on left bound
116        y_dot_n =  $(3*y[-1] - 4*y[-2] + y[-3])/(2*h)$  # second order derivative on right bound
117        # join everything together to get the second order derivative
118        y_dot = np.insert(y_dot, 0, y_dot_0)
119        y_dot = np.insert(y_dot, len(y_dot), y_dot_n)
120
121        # equation to solve
122        y_ddot = -g*y_dot - o02*y
123
124    return y_ddot
125
126 #####
127 # Main code and plot
128 #####
129
130 g = 0.3      # damping factor
131 o02 = 1       # proper frequency squared
132 xi = 0        # left end of the interval
133 xf = 5        # right end of the interval
134 N = 1000      # number of points
135 y0 = 1         # boundary condition on xi
136 y1 = 0.1862   # boundary condition on xf
137
138 x = np.linspace(xi, xf, N)
139 y = (x - xi) * (y1 - y0)/(xf - xi) + y0 # linear guess
140
141 Y, n = relax(f, y0, y1, x, y, args=(g, o02), dense_output=True)
142 print(f"{n} iterations required")
143
144 for y in Y:
145     plt.plot(x, y)
146
147 plt.title("Solution via relaxation method", fontsize=15)
148 plt.xlabel("x", fontsize=15)
149 plt.ylabel("y", fontsize=15)
150 plt.grid()
151 plt.show()
152
153 [Output]
154 30 iterations required

```

Solution via relaxation method



## E 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 vogliamo illustrare un metodo di quadratura gaussiana, il metodo di Gauss-Legendre. Questo metodo ci permette di approssimare l'integrale con una somma pesata:

$$\int_{-1}^1 f(x) \simeq \sum_{i=1}^n w_i f(x_i) , \quad (54)$$

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:

$$w_i = \frac{2}{(1 - x_i^2)(P'_n(x_i))^2} . \quad (55)$$

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

$$I_{ab} = \int_a^b f(x) \simeq \frac{b-a}{2} \sum_{i=1}^n w_i f\left(x_i \frac{b-a}{2} + \frac{b+a}{2}\right) . \quad (56)$$

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 e rifacciamo. Possiamo così attribuire facilmente un errore al nostro integrale, useremo infatti 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 division
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 :
87         pass
88
89     Inte = 0
90     dxddxi = (b - a)/2
91     roots, weights = roots_legendre(n)
92
93     for x_i, w_i in zip(roots, weights):
94         Inte += w_i * f(x_i * dxddxi + (b + a)/2, *args)
95
96     Inte *= dxddxi
97
98     return Inte
99
100
101 def test():
102     """ little test
103     """
104     def h(x):
105         """sine
106         """
107         return math.sin(x)
108     def f(x, a1):
109         """gaussian
110         """
111         return math.exp(-x**2/(2*a1))/(math.sqrt(2*math.pi)*a1)
112     def g(x, a1, a2):
113         """fermi dirac
114         """
115         return math.exp(-(x - a1)/a2)/(1 + math.exp(-(x - a1)/a2))
116
117     I, dI = adaptive_gaussleg_quadrature(h, 0, math.pi)
118     print('Integral value is', I, '+-', dI)
119     I, dI = adaptive_gaussleg_quadrature(f, -100, 100, args=(1,))
120     print('Integral value is', I, '+-', dI)
121     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 1.0000000051542988 +- 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 sono tabulati (più di 400 linee di codice che trovate nella cartella). 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).

## F Diagonalizzazione

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

$$A\mathbf{v} = \lambda\mathbf{v} \quad . \quad (57)$$

A volte siamo interessati a tutti gli autovalori, a volte solo ai più grandi o a i più piccoli, dipende un po' dai casi, quindi vogliamo un po' far vedere qualche metodo per calcolare la decomposizione spettrale di una data matrice.

### F.1 Metodo delle potenze

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

$$\mathbf{v}_{k+1} = \frac{A\mathbf{v}_k}{\|A\mathbf{v}_k\|} = \frac{A^k\mathbf{v}_0}{\|A^k\mathbf{v}_0\|} \quad . \quad (58)$$

Dove  $\mathbf{v}_0$  è un certo vettore iniziale che sceglio a caso. L'idea è abbastanza semplice, se  $A$  è diagonalizzabile allora possiamo decomporre un vettore generico nella base degli autovettori di  $A$ :

$$\mathbf{v}_0 = \alpha_1 v_1 + \cdots + \alpha_n v_n \quad , \quad (59)$$

Quindi applicando la potenza ennesima di  $A$  a  $v_0$  si ottiene:

$$A^k\mathbf{v}_0 = \alpha_1 A^k v_1 + \cdots + \alpha_n A^k v_n \quad (60)$$

$$= \alpha_1 \lambda_1^k v_1 + \cdots + \alpha_n \lambda_n^k v_n \quad (61)$$

$$= \lambda_1^k \left( \alpha_1 v_1 + \cdots + \alpha_n \left( \frac{\lambda_n}{\lambda_1} \right)^k v_n \right) \quad . \quad (62)$$

Quindi se gli autovalori sono tutti diversi e sono ordinati in ordine decrescente  $\lambda_1 > \cdots > \lambda_n$  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 (di per sé Gram–Schmidt non è la scelta migliore, introduce rumore numerico, ma per i nostri scopi va più che bene). 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:

$$R_{1,2} = \|\mathbf{v}_{k+1} \pm \mathbf{v}_k\|, \quad R_3 = \sqrt{|\lambda_{k+1} - \lambda_k|} \quad , \quad (63)$$

dove il  $\pm$  deriva dal fatto che sia  $+\mathbf{v}$  che  $-\mathbf{v}$  sono autovettori. Quindi quando una di queste quantità è minore di una certa tolleranza che sceglio noi l'algoritmo termina.

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. In genere questo in questo metodo non si inverte direttamente  $A$ ; si usa invece una routine per risolvere un sistema (implementeremo questa caratteristica quando affronteremo la scrittura di un codice matrix-less). Per il momento decidiamo di invertire direttamente la matrice di partenza senza farci troppi problemi. 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 eigenvariates 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 tollerance
22     magnitude : string, optional, default small
23         if magnitude == 'small' the smallest eigenvalues
24         and thei relative eigenvectors will be computed
25         if magnitude == 'big' the biggest eigenvalues
26         and thei 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 number 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
66             # Orthogonalization respect
67             # all eigenvectors find previously
68             for i in range(len(eigvec)):
69                 v_p = v_p - np.dot(eigvec[i], v_p) * eigvec[i]
70
71             #eigenvalue of v_p, A @ v_p = l_v * v_p
72             #multiplying by the transposed => (A @ v_p) @ v_p.T = l_v
73             #using v_p @ v_p.T = 1
74             l_v = np.dot(np.dot(A, v_p), v_p)
75
76             R1 = np.sqrt(sum((v_p - v_o)**2))
77             R2 = np.sqrt(sum((v_o + v_p)**2))
78             R3 = np.sqrt(abs(l_v - l_o)) # In eigenvalues the convergence is quadratic
79
80             Iter += 1
81             if R1 < tol or R2 < tol or R3 < tol:
82                 break
83
84             eigvec.append(v_p)
85             eigval.append(l_v)
86             counts.append(Iter)
87
88     if magnitude == 'small':
89         eigval = 1/np.array(eigval)
90         eigvec = np.array(eigvec).T

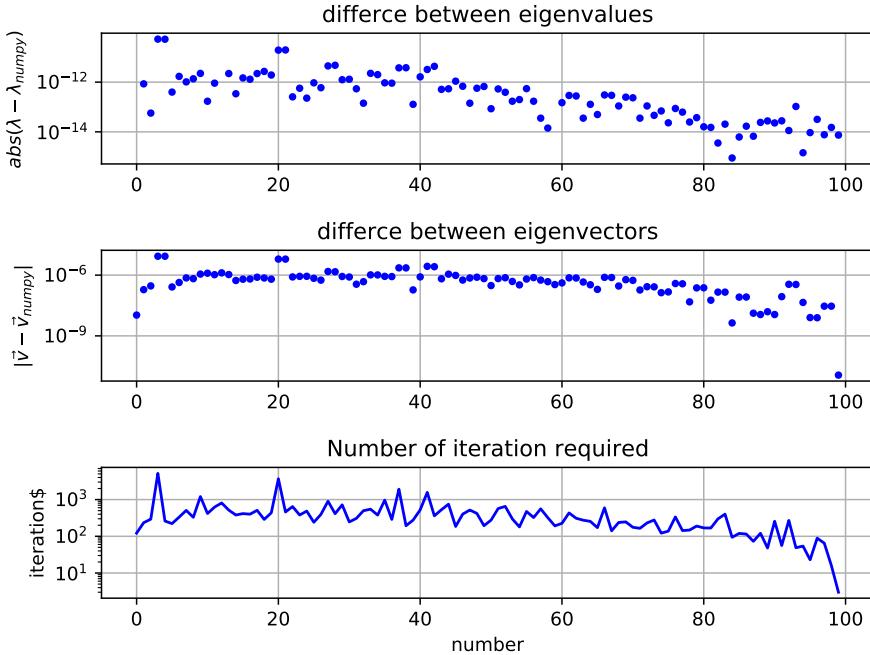
```

```

91     if magnitude == 'big':
92         eigvec = np.array(eigvec).T
93         eigval = np.array(eigval)
94
95     return eigvec, eigval, counts

```

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  $H$ , prendiamo  $n = 100$ , 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: 14.240 secondi noi contro 0.001 di numpy, e ci guarda pure con aria di superiorità perché gli facciamo schifo. Comunque il risultato è soddisfacente.

## F.2 Equazione di Schrödinger

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

$$H = -\frac{1}{2h^2} \begin{pmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & 1 & -2 & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{pmatrix} + \begin{pmatrix} V(x_1) & 0 & 0 & \cdots & 0 \\ 0 & V(x_2) & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & 0 & V(x_{N-1}) & 0 \\ 0 & \cdots & 0 & 0 & V(x_N) \end{pmatrix} \quad (64)$$

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 prima matrice scritta sopra non è altro che la discretizzazione di una derivata seconda come avevamo già visto prima; mentre la seconda è una semplice matrice diagonale che contiene i valori di una certa funzione calcolata su una griglia di  $N$  punti. Si tratta dell'equazione di Schrödinger (non stiamo qui a ricavarla, lo vedrete nei corsi di meccanica quantistica, prendetela per buona per il momento) non dipendente dal tempo :

$$H\psi = \left( -\frac{1}{2} \frac{\partial^2}{\partial x^2} + V(x) \right) \psi = E\psi \quad (65)$$

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 (e quindi le energie più alte). Discretizzare così infatti significa mettersi dentro una scatola di lato fissato ma chiaramente noi vorremmo la scatola infinita (ovvero risolvere il problema su tutto l'asse reale), quindi per i livelli più bassi, dove la maggior parte dei valori

non nulli della soluzione è intorno a zero, l'approssimazione è buona. Se prendiamo alti livelli energetici abbiamo che molto del significato fisico è anche vicino al bordo e di questo l'algoritmo se ne accorge. Un modo carino per vederlo è mettere un potenziale  $V(x) = 0$ ; la soluzione teorica che ci potremmo aspettare è un'onda piana, ma in realtà otterremo la soluzione all'interno di una scatola infinita. Qui abbiamo scelto ( $h = L/N$ )  $L=20$  e  $N=1000$ . Per semplicità consideriamo il caso dell'oscillatore armonico  $V(x) = x^2/2$  e troviamo i dieci autovalori più bassi (analiticamente e in unità naturali  $E = n + 1/2$ ). Grafichiamo anche per completezza e bellezza tre autovettori, o autofunzioni, con il caveat che ogni autovettore va diviso per  $\sqrt{h}$ . Questo perché la soluzione è una densità di probabilità:

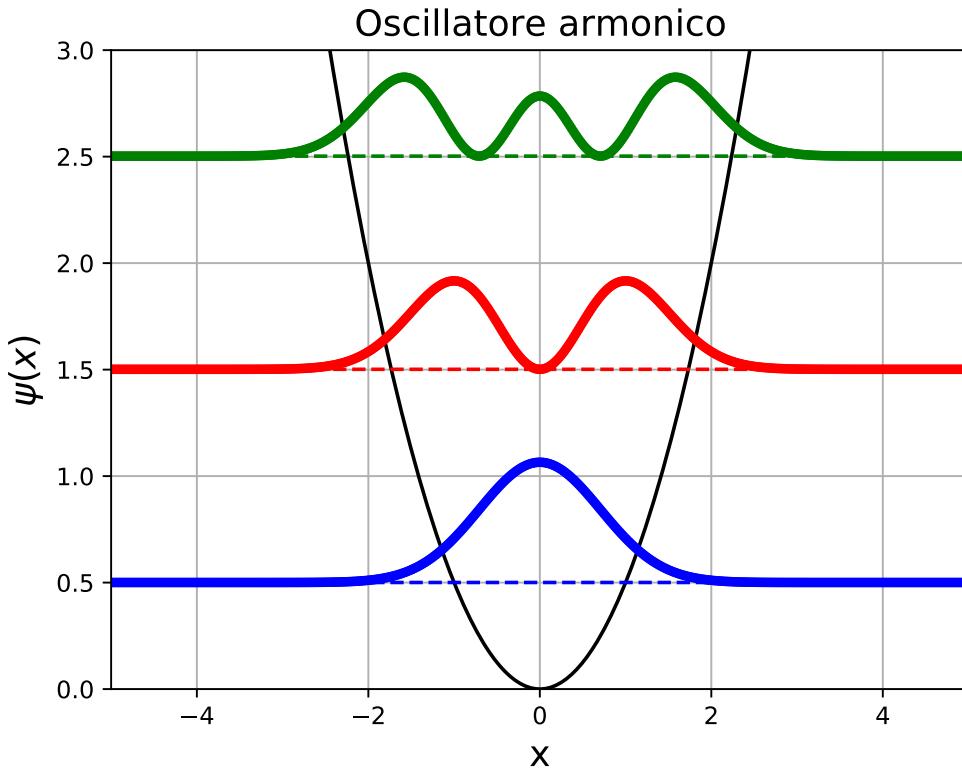
$$\int_V |\psi(x)|^2 dx = 1 \rightarrow \sum_i |\psi(x_i)|^2 h = 1. \quad (66)$$

La costante per normalizzare si calcola così:

$$\sum_i |\alpha\psi(x_i)|^2 h = 1 \rightarrow \alpha = \frac{1}{\sqrt{h \sum_i |\psi(x_i)|^2}} = \frac{1}{\sqrt{h}} \quad (67)$$

Dove abbiamo usato che  $\sum_i |\psi(x_i)|^2 = 1$  in quanto la routine normalizza i vettori. Come sopra 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 0.77814 secondi. Precisiamo infine che benché essa sia un'equazione differenziale ordinaria non può essere risolta come tale in quanto non è noto a priori il valore dell'energia. Volendo si può adottare una strategia simile usando il metodo di shooting, ovvero si risolve l'equazione come fosse una normale ode usando una guess per l'energia e poi si usa l'energia come parametro di shooting; Si cerca quindi tramite un algoritmo di root finding un valore di  $E$  (parametro di shooting) che dia un buon risultato (dove buono si intende che c'è una variazione di una qualche minore di una tolleranza da noi settata).

### F.3 Algoritmo QR

Vediamo adesso un altro algoritmo che può essere interessante trattare. Come si può intuire dal nome questo algoritmo si basa sulla scomposizione  $QR$  di una matrice, dove  $Q$  è una matrice ortogonale  $Q^T = Q^{-1}$  e  $R$  è una matrice triangolare superiore. Data quindi la nostra matrice  $A$  da diagonalizzare quello che si fa è, chiamando  $A_0 = A = Q_0 R_0$ :

$$A_{k+1} = R_k Q_k = Q_k^{-1} Q_k R_k Q_k = Q_k^T A_k Q_k \quad . \quad (68)$$

Vedete quindi che procedendo per trasformazioni ortogonali tutte le  $A_k$  sono simili. Avremo quindi che gli autovalori saranno gli elementi sulla diagonale della matrice  $A_{k+1}$ . Mentre per gli autovettori quello che si fa è calcolare il prodotto delle varie  $Q_k$  con loro stesse:

$$V = Q_0 Q_1 \dots Q_k \quad , \quad (69)$$

$V$  sarà la matrice che conterrà gli autovettori di  $A$ . Tutto ciò viene eseguito un certo tot di volte che scegliamo noi da input. Per completare la spiegazione andiamo a vedere come si esegue la decomposizione  $QR$ . Fondamentalmente si tratta di applicare Gram-Schmidt alle colonne di  $A$ .

$$A = [\mathbf{a}_1 | \mathbf{a}_2 | \dots | \mathbf{a}_N] \quad (70)$$

$$\begin{aligned} \mathbf{v}_1 &= \mathbf{a}_1, & \mathbf{e}_1 &= \frac{\mathbf{v}_1}{\|\mathbf{v}_1\|} \\ \mathbf{v}_2 &= \mathbf{a}_2 - (\mathbf{a}_2 \cdot \mathbf{e}_1) \mathbf{e}_1, & \mathbf{e}_2 &= \frac{\mathbf{v}_2}{\|\mathbf{v}_2\|} \\ &\vdots & &\vdots \\ \mathbf{v}_k &= \mathbf{a}_k - \sum_{j=1}^{k-1} (\mathbf{a}_k \cdot \mathbf{e}_j) \mathbf{e}_j, & \mathbf{e}_k &= \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|}. \end{aligned}$$

Dunque possiamo arrivare a dire che:

$$A = [\mathbf{e}_1 | \mathbf{e}_2 | \dots | \mathbf{e}_N] \begin{pmatrix} \mathbf{a}_1 \cdot \mathbf{e}_1 & \mathbf{a}_2 \cdot \mathbf{e}_1 & \dots & \mathbf{a}_N \cdot \mathbf{e}_1 \\ 0 & \mathbf{a}_2 \cdot \mathbf{e}_2 & \dots & \mathbf{a}_N \cdot \mathbf{e}_2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \mathbf{a}_N \cdot \mathbf{e}_N \end{pmatrix} = QR \quad . \quad (71)$$

Questo metodo, a differenza del precedente, con il quale potevamo scegliere quanti autovalori e autovettori farci calcolare, restituisce tutti gli autovalori e gli autovettori della nostra matrice. Siamo in Python, probabilmente per matrici grandi ci vorrà tanto. Passiamo quindi adesso al codice:

```

1 #=====
2 # Function to compute QR decomposition
3 #=====
4
5 def QR_decomp(A):
6     """
7     Comupute QR decomposition of a matrix A
8
9     Parameters
10    -----
11    A : 2darray
12        N x N matrix
13
14    Returns
15    -----
16    Q, R : 2darray
17        A = Q @ R
18
19    # we give different names because in principle the QR
20    # decomposition also applies to non-square matrices
21    n, m = A.shape
22
23    Q = np.zeros((n, n)) # Initialize matrix Q
24    R = np.zeros((n, m)) # Initialize matrix R
25    v = np.zeros((n, n)) # Initialize matrix v: for Gram-Schmidt
26
27
```

```

28     u[:, 0] = A[:, 0]
29     Q[:, 0] = v[:, 0] / np.sqrt(sum(v[:, 0]**2))
30
31     for i in range(1, n):
32
33         v[:, i] = A[:, i]
34         for j in range(i):
35             v[:, i] -= (A[:, i] @ Q[:, j]) * Q[:, j]
36
37         Q[:, i] = v[:, i] / np.sqrt(sum(v[:, i]**2))
38
39
40     for i in range(n):
41         for j in range(i, m):
42             R[i, j] = A[:, j] @ Q[:, i]
43
44     # uncomment for scipy comparison
45     #D = np.diag(np.sign(np.diag(Q)))
46     #Q[:, :] = Q @ D
47     #R[:, :] = D @ R
48
49     return Q, R
50
51 #####
52 # Function to solve eigensystem via QR iterartion
53 #####
54
55 def QR_eig(A, maxiter=100):
56     """
57     Find the eigenvalues of A using QR iteration.
58
59     Parameters
60     -----
61     A : 2darray
62         N x N matrix
63     maxiter : int, optional, default 100
64         number of iterations to do
65
66     Returns
67     -----
68     eigval : 1darray
69         array of eigenvalues
70     eigvec : 2darray
71         N x N matrix, the column eigvec[:, i] is the
72         ormalized eigenvector corresponding to the
73         eigenvalue eigval[i]
74     """
75     A_new, A_old = [np.copy(A)]*2
76
77     eigvec = np.eye(A.shape[0])
78
79     for i in range(maxiter):
80
81         A_old[:, :] = A_new
82         Q, R = QR_decomp(A_old)
83
84         A_new[:, :] = R @ Q
85         eigvec = eigvec @ Q
86
87         eigval = np.diag(A_new)
88
89     return eigval, eigvec

```

Facendo la prova con la stessa matrice a bischero di prima infatti otteniamo un tempo di 717.6 secondi, usando le iterazioni di default. I risultati sono gli stessi che quelli mostrati in precedenza. Attenzione ora però all'ordine degli autovalori, dovrebbero essere ordinati dal più grande al più piccolo.

## F.4 Lanczos

Un'ultima cosa carina da spiegare è l'iterazione di Lanczos, caso particolare della più generale iterazione di Arnoldi; ma per i casi di interesse fisico le ipotesi sono sempre ragionevoli: la matrice da diagonalizzare deve essere hermitiana. L'idea è di trovare due matrici per cui valga:

$$A = Q^\dagger H Q. \quad (72)$$

$H$  sarà un amatrice tridiagonale simmetrica reale e sarà quella che andremo a diagonalizzare. Essa sarà una matrice più piccola rispetto ad  $A$ , la dimensione è un parametro che sceglieremo noi, quindi verrà più leggero il conto, ma non siamo in grado di recuperare tutti gli autovalori, e autovettori di  $A$ . L'unico modo per farlo è imporre che  $H$  abbia la stessa dimensione di  $A$ .  $Q$  invece sarà una certa matrice con colonne ortonormali; se poi  $H$  ha la stessa dimensione di  $A$  allora  $Q$  sarà unitaria. Tornando a noi diagonalizzando  $H$  avremo un certo numero di autovalori, che sono anche autovalori di  $A$ , non necessariamente tutti sequenziali. Per gli autovettori invece, se vale  $Hy = \lambda y$  allora  $Qy$  è autovalore di  $A$ . Vediamo subito il codice:

```

1 def lanczos(A, n):
2     """
3         Lanczos iteration for a matrix A
4
5     Parameter
6     -----
7     A : 2darray
8         Hermitian N x N matrix
9     n : int
10        dimension of Krylov subspace
11        e.g. dimension of H
12
13    Return
14    -----
15    Q, H : 2darray
16        A = Q.T H Q
17        ,
18    m      = A.shape[0]
19    Q      = np.zeros((m, n+1))
20    alpha = np.zeros(n)
21    beta  = np.zeros(n)
22    b      = np.random.randn(m)
23
24    Q[:,0] = b / np.sqrt(sum(b**2))
25
26    for i in range(n):
27        v       = np.dot(A, Q[:,i])
28        alpha[i] = np.dot(Q[:,i], v)
29
30        if i == 0:
31            v = v - alpha[i] * Q[:, i]
32        else :
33            v = v - beta[i-1] * Q[:, i-1] - alpha[i] * Q[:, i]
34
35        beta[i]   = np.sqrt(sum(v**2))
36        Q[:,i+1] = v / beta[i]
37
38    H = Q.T @ A @ Q
39
40    return Q, H

```

Calcolata  $H$  la passiamo quindi a una delle funzioni sopra scritte e vediamo che succede. Per riuscire a prendere dei buoni risultati per i primi livelli eccitati usiamo come  $n = 300$ , quindi l'algoritmo QR ora sarà un po' più spicchio. Vediamo cosa esce sempre per il caso dell'oscillatore armonico.

teorico	QR	errore	potenze	errore
0.5	0.50068	6.80e-04	0.50068	6.80e-04
1.5	1.50146	1.46e-03	1.50145	1.45e-03
2.5	2.50321	3.21e-03	2.50257	2.57e-03
3.5	3.59515	9.51e-02	3.59459	9.46e-02
4.5	4.61829	1.18e-01	4.61261	1.13e-01
5.5	6.60135	1.10e+00	6.59104	1.09e+00
6.5	7.76801	1.27e+00	7.74208	1.24e+00
7.5	10.35934	2.86e+00	10.36662	2.87e+00
8.5	12.02403	3.52e+00	12.04071	3.54e+00
9.5	13.48530	3.99e+00	13.49227	3.99e+00

Vedete quindi come per i primi 5 livelli vada tutto bene, poi gli altri autovalori sono diversi o un po' sbagliati: ad esempio 10.35 è un po' lontano mentre 13.49 è già un migliore risultato (Ovviamente non va confrontato con 9.5 ma con il fatto che sappiamo che esiste l'autovalore 13.5). Come dicevamo prima vedete poi che ci sono degli autovalori che sono spariti in quanto  $H$  è solo  $300 \times 300$ . Dal punto di vista dei tempi QR impiega circa 48 secondi, mentre il metodo delle potenze circa 0.1 secondi.

#### F.4.1 Matrix-less

Prima avevamo detto, con il metodo delle potenze, che per diagonalizzare una matrice, in linea di principio, per il metodo delle potenze inverso, non è prassi invertire da subito la matrice; piuttosto si usa una routine per risolvere un sistema lineare. Vogliamo approfittare di quest'altra implementazione per far vedere qualcosa di nuovo. Proveremo a fare un codice matrix-less, ovvero non avremo bisogno da nessuna parte di allocare dello spazio per conservare la matrice da diagonalizzare. Quello che verrà fatto è calcolare direttamente il prodotto tra la matrice e il vettore, sapendo quali sono le caratteristiche della matrice. Questo ci aiuterà ad esempio a poter aumentare il numero di punti senza troppi problemi di ram. La matrice è la solita matrice a bischero di prima, sempre l'oscillatore armonico quantistico. Andiamo quindi a vedere come diventa il codice:

```

1 """
2 Code for diagonalizing a Hamiltonian of the form H = P^2 + V(x).
3 The code uses the power method using the conjugate gradient as a method
4 for inverting a matrix.
5 Everything is implemented in the matrixless form, i.e. no matrices are allocated
6 instead whenever necessary the result of the matrix-vector product is calculated.
7 """
8 import time
9 import numpy as np
10 from math import factorial
11 import matplotlib.pyplot as plt
12 from scipy.special import eval_hermite
13
14
15 def conj_grad_matrixless(mult, b, tol=1e-6):
16     """
17         Matrix-free implementation of conjugate gradient method for solving M v = b.
18
19         Parameters
20         -----
21         mult : callable
22             A function that returns the product of the matrix with a vector.
23         b : 1darray
24             Ordinate or "dependent variable" values.
25         tol : float, optional
26             Required tolerance (default 1e-6).
27
28         Return
29         -----
30         x : 1darray
31             Solution of the system.
32         err : float
33             Error of the solution.
34     """
35
36     N = len(b)
37     x = np.zeros(N) # Initial guess
38
39     r = b - mult(x) # Residuals
40     p = r # Descent direction
41     r2 = sum(r*r) # Norm^2 of residuals
42
43     iter = 0
44
45     while True:
46         Ap = mult(p) # Compute matrix-free product M @ p
47         alpha = r2 / (p @ Ap) # Descent step
48
49         x = x + alpha * p # Update position
50         r = r - alpha * Ap # Update residuals
51
52         r2_new = sum(r*r) # Norm^2 of new residuals
53         beta = r2_new / r2 # Compute step for p
54
55         r2 = r2_new # Update norm
56
57         if np.sqrt(r2_new) < tol: # Break condition
58             break
59
60         p = r + beta * p # Update descent direction
61         iter += 1
62
63     err = np.sqrt(r2_new)

```

```

65     return x, err
66
67
68 def eig_matrixless(mult, N, k=None, tol=1e-10, magnitude='small', inverse_solver='cg'):
69     """
70     Compute the eigenvalue decomposition of a matrix using power iteration
71     or inverse iteration in a matrix-free manner, avoiding storing the matrix.
72
73     Parameters
74     -----
75     mult : Callable
76         A function that returns the product of the matrix with a vector (matrix-free approach)
77
78     k : int or None
79         Number of eigenvalues and eigenvectors to find. If None, find all.
80     tol : float
81         Tolerance for convergence.
82     magnitude : string
83         'small' to find smallest eigenvalues, 'big' to find largest eigenvalues.
84     inverse_solver : string
85         Method for solving linear systems. Default is 'cg' (Conjugate Gradient).
86
87     Returns
88     -----
89     eigval : 1darray
90         Eigenvalues found.
91     eigvec : 2darray
92         Corresponding eigenvectors.
93     counts : 1darray
94         Number of iterations for each eigenvalue.
95
96     # Handle whether we want the smallest or largest eigenvalues
97     if magnitude == 'small':
98         # Define a matrix-free inverse operator using your Conjugate Gradient implementation
99         def mult_inv(v):
100             if inverse_solver == 'cg':
101                 # Use custom CG solver to solve M*x = v (instead of np.linalg.solve)
102                 x, _ = conj_grad_matrixless(mult, v)
103                 return x
104             else:
105                 raise NotImplementedError(f"Inverse solver '{inverse_solver}' not implemented")
106
107             mat = mult_inv
108         else:
109             # Normal power iteration
110             mat = mult
111
112         if k is None:
113             k = N
114
115         eigvec = []
116         eigval = []
117         counts = []
118
119         for _ in range(k):
120
121             # Initialization
122             v_p = np.random.randn(N)
123             v_p = v_p / np.linalg.norm(v_p)
124             l_v = np.random.random()
125             Iter = 0
126
127             while True:
128                 l_o = l_v
129                 v_o = v_p                      # Update vector
130                 v_p = mat(v_p)                # Compute new vector
131                 v_p /= np.linalg.norm(v_p)    # Normalization
132
133                 # Orthogonalization respect
134                 # all eigenvectors find previously
135                 for i in range(len(eigvec)):
136                     v_p = v_p - np.dot(eigvec[i], v_p) * eigvec[i]
137
138                 # Eigenvalue of v_p, A @ v_p = l_v * v_p

```

```

139     # Multiplying by the transposed => (A @ v_p) @ v_p.T = l_v
140     # Using v_p @ v_p.T = 1
141     l_v = np.dot(v_p, mat(v_p))
142
143     R1 = np.linalg.norm(v_p - v_o)
144     R2 = np.linalg.norm(v_o + v_p)
145     R3 = abs(l_v - l_o)           # In eigenvalues the convergence is quadratic
146
147     Iter += 1
148     if R1 < tol or R2 < tol or R3 < tol:
149         break
150
151     eigvec.append(v_p)
152     eigval.append(l_v)
153     counts.append(Iter)
154
155 if magnitude == 'small':
156     eigval = 1 / np.array(eigval)
157 else:
158     eigval = np.array(eigval)
159
160 eigvec = np.array(eigvec).T
161 return eigval, eigvec, counts
162
163
164
165 if __name__ == "__main__":
166
167     np.random.seed(69420)
168
169 ##### Computational parameter #####
170 #####
171 #####
172
173     k = 10                      # how many levels compute
174     n = 1000                     # size of matrix
175     xr = 10                      # bound
176     xl = -10                     # bound
177     L = xr - xl                 # dimension of box
178     h = (xr - xl)/(n)            # step size
179     tt = np.linspace(0, n, n)    # array form 0 to n
180     xp = xl + h*tt              # array of position
181
182 #####
183 # Hamiltonian
184 #####
185
186 def H(psi, V, x):
187     """
188         This function computes the product M @ v without storing M.
189         In this case we have a simple tridiagonal matrix of the form:
190
191             [ 2 -1  0  0  0 ]      [ V0 0  0  0  0 ]
192             [-1  2 -1  0  0 ]      [ 0  V1 0  0  0 ]
193             1/(2*h^2) [ 0 -1  2 -1  0 ] + [ 0  0  V2 0  0 ]
194             [ 0  0 -1  2 -1 ]      [ 0  0  0  V3 0 ]
195             [ 0  0  0 -1  2 ]      [ 0  0  0  0  V4]
196
197     Parameters
198     -----
199     psi : 1darray
200         vector to multiply to the matrix
201     V : callable
202         potential of the sistem
203     x : 1darray
204         grid of our discretized sistem
205
206     Return
207     -----
208     H_psi : 1darray
209         result of H @ psi
210     ,
211     N = len(x)
212     h = np.diff(x)[0]
213     H_psi = np.zeros(N)
214

```

```

215     # Kinetic term (tridiagonal part)
216     factor = - 1/ (2 * h**2)
217
218     for i in range(1, N-1): # Loop through interior points only
219         H_psi[i] = psi[i - 1] - 2 * psi[i] + psi[i + 1]
220
221     # Boundary conditions
222     H_psi[0] = -2 * psi[0] + psi[1]
223     H_psi[N-1] = -2 * psi[N-1] + psi[N-2]
224
225     H_psi *= factor
226
227     # Potential term (diagonal part)
228     H_psi += V(x) * psi
229
230     return H_psi
231
232 def Potential(x):
233     return 0.5 * x**2
234
#=====
# Computation
#=====
238
239     start = time.time()
240     eigval, eigvec, Iter = eig_matrixless(lambda v: H(v, Potential, xp), n, k, tol=1e-5,
241     magnitude='small')
242
243     end = time.time() - start
244     print(f'Elapsed time      = {end:.5f}\n')
245
246     print("Theoretical      Computed            error")
247     print("-----")
248     for i in range(k):
249         print(f'{i+0.5} \t \t {eigval[i]:.5f} \t {eigval[i]-(i+0.5):.2e}')
250
251     psi = eigvec/np.sqrt(h)
252
[Output]
253 Elapsed time      = 46.41249
254
255 Theoretical      Computed            error
256 -----
257 0.5              0.49999          -1.22e-05
258 1.5              1.49995          -5.34e-05
259 2.5              2.49988          -1.23e-04
260 3.5              3.49979          -2.11e-04
261 4.5              4.49945          -5.47e-04
262 5.5              5.49968          -3.21e-04
263 6.5              6.49952          -4.75e-04
264 7.5              7.49884          -1.16e-03
265 8.5              8.49886          -1.14e-03
266 9.5              9.49695          -3.05e-03

```

Non riportiamo il grafico perchè tanto è il medesimo.

## 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 esempio da tempo a frequenze o da spazio a numero d'onda. Data una funzione:

$$f(t) \quad \text{tale che} \quad \int_{-\infty}^{\infty} |f(t)| dt < \infty \quad (73)$$

Definiamo trasformata e anti trasformata di  $f$  come:

$$\tilde{f}(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt = \mathcal{F}\{f\}(\omega) \quad (74)$$

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \tilde{f}(\omega) e^{i\omega t} d\omega = \mathcal{F}^{-1}\{\tilde{f}\}(t) \quad (75)$$

$$(76)$$

e gode di varie proprietà:

$$\mathcal{F}(D^k f) = (-i\omega)^k \mathcal{F}(f) \quad \mathcal{F}(f(t-a)) = e^{i\omega a} \mathcal{F}(f(t)) \quad \mathcal{F}(e^{i\omega a} f(t)) = \tilde{f}(\omega - a) \quad \mathcal{F}((it)^k f) = D^k \mathcal{F}(f) \quad (77)$$

Qui abbiamo in realtà aggiunto altre ipotesi su  $f$  (e.g.  $f \in C^k$ ) ma va beh. Vediamo un piccolo grafico che meglio ci consente di capire, in maniera intuitiva, cosa vuol dire questo cambio di base:

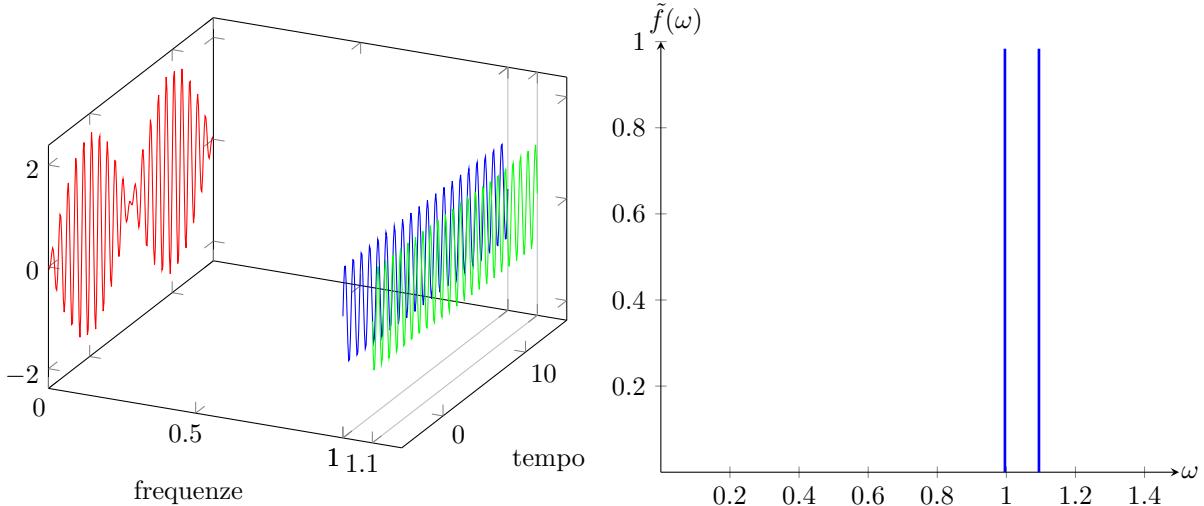


Figura 11: Allora, capisco che i due grafici sopra, in special modo il grafico di sinistra, siano magari non di immediata comprensione ma cerchiamo di descriverli e capirli. Partiamo da quello di sinistra: si tratta di un immagine pittorica per vedere quella che è la serie di Fourier, ovvero una funzione scritta come somma di seni e/o coseni (armoniche). La funzione che vedete plottata sul piano a frequenza zero è:  $\sin(2\pi t_1) + \sin(2\pi t_{1.1})$  (ovviamente quella funzione non ha una omega nulla, è solo plottata lì a titolo espositivo). Tale funzione è formata da due seni che sono i due plottati a parte (blu e verde); ognuno giace sul piano alla propria frequenza, rispettivamente  $\omega = 1, 1.1$  (purtroppo per avere i battimenti le  $\omega$  devono essere vicine). Questo plot ci aiuta a capire cosa fa la trasformata di Fourier, perché vediamo in funzione di omega solo due segnali, quindi ci aspettiamo solo due valori di frequenza non nulli. Ciò che la trasformata ci restituisce è il grafico a destra: ovvero un grafico che ci fa capire le quali sono le armoniche che compongono la nostra funzione. Nella fattispecie sono due delta di Dirac (in linea teorica); nella pratica saranno delle gaussiane più o meno strette a seconda dei dati.

Andiamo ora nel discreto; abbiamo:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn}, \quad k = 0, \dots, N-1 \quad (78)$$

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

$$X_k = W_{kn} x_n \quad W_{kn} = e^{-\frac{2\pi i}{N} kn} \quad (79)$$

e l'anti trasformata non sarà altro che:

$$X_k = \frac{1}{N} W_{kn}^{-1} x_n \quad W_{kn}^{-1} = e^{\frac{2\pi i}{N} kn} \quad (80)$$

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
5 import time
6 import numpy as np
7 import matplotlib.pyplot as plt
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:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \quad (81)$$

$$= \sum_{n=0}^{N/2-1} x_{2n} e^{-\frac{2\pi i}{N} k(2n)} + \sum_{n=0}^{N/2-1} x_{2n+1} e^{-\frac{2\pi i}{N} k(2n+1)} \quad (82)$$

$$= \sum_{n=0}^{N/2-1} x_{2n} e^{-\frac{2\pi i}{N/2} kn} + e^{-\frac{2\pi i}{N} k} \sum_{n=0}^{N/2-1} x_{2n+1} e^{-\frac{2\pi i}{N/2} kn} \quad (83)$$

$$= DFT(x_{2n}) + e^{-\frac{2\pi i}{N} k} DFT(x_{2n+1}) \quad (84)$$

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 Implementation 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 : ndarray
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: abbiamo detto che la DFT è  $\mathcal{O}(N^2)$ . Quindi al primo passo come visto sopra abbiamo 2 DFT e un prodotto di due vettori  $\mathcal{O}(N)$  per cui:

- prima divisione:  $\frac{N}{2} \rightarrow 2 \underbrace{\left(\frac{N}{2}\right)^2}_{\text{DFT}} + N = \frac{N^2}{2} + N$
- seconda divisione:  $\frac{N}{4} \rightarrow 2 \left(2 \underbrace{\left(\frac{N}{4}\right)^2}_{\text{DFT}} + \frac{N}{2}\right) + N = \frac{N^2}{4} + 2N$

Capendo l'antifona otteniamo che l'ultima divisione è:  $\frac{N}{2^p} \rightarrow \frac{N^2}{2^p} + pN = \frac{N^2}{N} + \ln_2(N)N \rightarrow \mathcal{O}(N \ln_2(N))$   
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 : ndarray
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.2.1 Standard FFT iterativa

C'è una piccola postilla che va precisata. Infatti se vi dovesse capitare fra le mani una spiegazione di come viene fatta una FFT iterativa non troverete quanto scritto sopra. Quanto appena presentato è solo una maniera iterativa e ottimizzata della funzione precedente. Infatti la "classica" FFT iterativa non solo è più veloce di una FFT ricorsiva, ma ha anche il vantaggio che, dal punto di vista della memoria allocata, si occupa poco spazio. Concentrandosi su quanto abbiamo scritto noi è facile vedere che avendo a che fare con due array ad ogni chiamata ricorsiva si ha una complessità spaziale di  $\mathcal{O}(2N \log(N))$ , con  $N$  lunghezza dell'array da trasformare. In genere però la versione iterativa di una FFT, facendo solo operazione "in place" alloca solo l'array iniziale per cui si ottiene:  $\mathcal{O}(N)$  (più in verità un paio di variabili che però sono  $\mathcal{O}(1)$  e per cui non sono un problema). Vediamo come solitamente si implementa questa versione. La prima cosa da fare riordinare l'array di partenza in un modo chiamato "bit reversal":

Indice	Binario	Inversione	Nuovo indice
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Abbiamo mostrato qui il caso da 8 elementi che è un caso classico. Prendiamo quindi l'indice del nostro array, lo scriviamo in binario, invertiamo l'ordine dei bit, riconvertiamo in decimale e otteniamo l'indice dell'array di output. Vediamo ora un diagramma che ci permette di capire che operazioni vanno fatte per il calcolo di questa FFT iterativa:

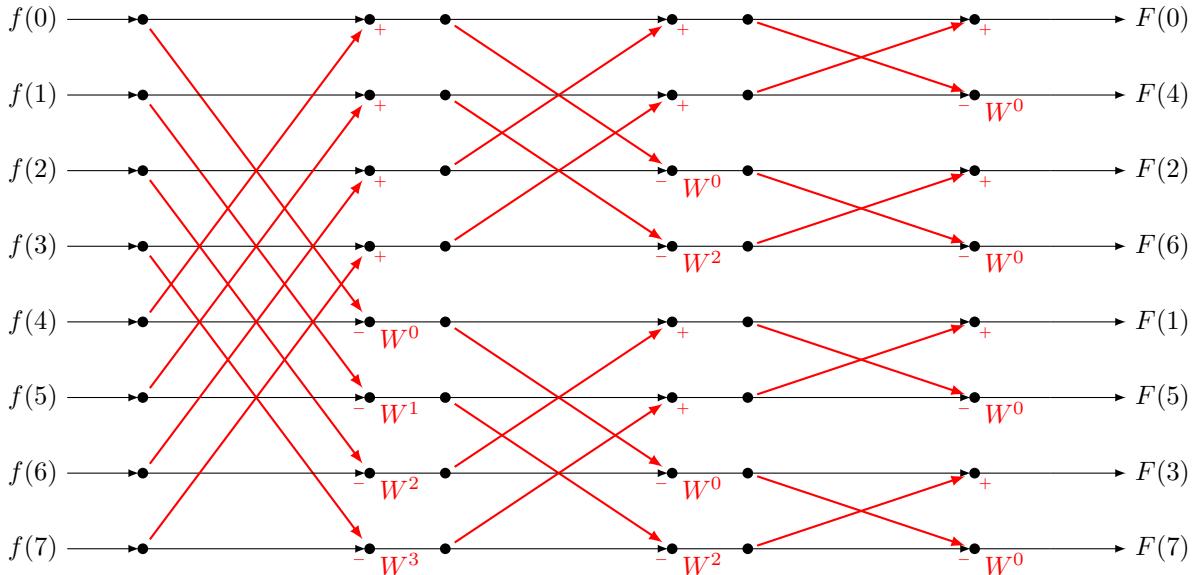


Figura 12: Diagramma a farfalla per una fft iterativa

Analizziamolo insieme per capire cosa succede. Iniziamo con il dire che prima avevamo definito la matrice  $W_{kn} = e^{-\frac{2\pi i}{N} kn}$ , ma fondamentalmente questa matrice potete pensarla come  $W_N^{kn}$ , dove  $W_N = e^{-\frac{2\pi i}{N}}$  sono le radici dell'unità. Tutto ciò lo diciamo per far vedere due interessanti proprietà:  $W_N^2 = W_{N/2}$  e  $W_N^{k+N/2} = -W_N^k$ . Ora se ricordate sopra quando abbiamo l'idea di dividere l'array in pari e dispari, la parte dispari era moltiplicata proprio da  $W_N^k$ ; quindi ora per brevità e grazie alle proprietà sopra elencate, avendo scelto  $N = 8$  (omettiamo il valore del pedice), rimaniamo solo con 4 valori per  $W: W^0, W^1, W^2, W^3$ . Cominciamo ora il percorso per il calcolo: abbiamo in nostro array iniziale  $f$  di lunghezza 8. Prendiamo il primo elemento  $f(0)$  seguendo la freccia rossa arriviamo sulla linea nera di  $f(4)$ , vediamo che ci sta un segno  $-$  e un  $W^0$ , questo ci indica che in quel punto dobbiamo calcolare  $f(0) - W^0 f(4)$ . Se invece fossimo partiti con  $f(4)$  allora seguendo la freccia rossa avremmo dovuto calcolare  $f(0) + f(4)$ . Se consideriamo quindi tutte le componenti del nostro vettore vediamo che al primo passo si formano due nuovi vettori lunghi 4:

$$y = \begin{pmatrix} f(0) + f(4) \\ f(1) + f(5) \\ f(2) + f(6) \\ f(3) + f(7) \end{pmatrix}, \quad z = \begin{pmatrix} f(0) - W^0 f(4) \\ f(1) - W^1 f(5) \\ f(2) - W^2 f(6) \\ f(3) - W^3 f(7) \end{pmatrix}.$$

Con la stessa logica applichiamo ugualmente il procedimento separatamente ai due vettori ed otteniamo quindi 4 vettori lunghi 2:

$$y' = \begin{pmatrix} y_0 + y_2 \\ y_1 + y_3 \end{pmatrix}, \quad y'' = \begin{pmatrix} y_0 - W^0 y_2 \\ y_1 - W^2 y_3 \end{pmatrix}, \quad z' = \begin{pmatrix} z_0 + z_2 \\ z_1 + z_3 \end{pmatrix}, \quad z'' = \begin{pmatrix} z_0 - W^0 z_2 \\ z_1 - W^2 z_3 \end{pmatrix}.$$

Iteriamo ancora una volta e siamo giunti alla fine:

$$\begin{aligned} F(0) &= y'_0 + y'_1, \quad F(1) = y'_0 - W^0 y'_1, \quad F(2) = y''_0 + y''_1, \quad F(3) = y''_0 - W^0 y''_1, \\ F(4) &= z'_0 + z'_1, \quad F(5) = z'_0 - W^0 z'_1, \quad F(6) = z''_0 + z''_1, \quad F(7) = z''_0 - W^0 z''_1. \end{aligned}$$

Vedete bene che quindi tutti questi conti non dobbiamo farli effettivamente creando nuovi vettori ausiliari, basta sovrascrivere il vettore iniziale, poiché al secondo passo l'informazione in esso contenuta non è più necessaria. Si ottiene quindi che usiamo un solo array e quindi lo spazio occupato è  $\mathcal{O}(N)$ . Vediamo come poter implementare questa versione:

```

1 def traditional_iterative_fft(x, anti=-1):
2     """
3         Compute the FFT of a 1D array using an iterative in-place approach.
4         Optimized to minimize memory usage.
5
6     Parameters:
7     -----
8     x : np.array
9         Input signal (length must be a power of 2).

```

```

10     anti : int, optional, default -1
11         -1 trasform
12         1 anti trasform
13
14     Returns:
15     -----
16     x : np.array
17         The transformed array.
18     """
19     N = len(x)
20     if np.log2(N) % 1 > 0:
21         raise ValueError("The input size must be a power of 2.")
22
23     x = np.array(x, dtype=np.complex128) # Ensure complex type
24
25     # Step 1: Bit-reversal reordering
26     j = 0
27     for i in range(1, N):
28         bit = N >> 1      # bit-shift right
29         while j & bit:    # bitwise and
30             j ^= bit      # xor
31             bit >>= 1     # bit-shift right
32         j ^= bit          # xor
33         if i < j:
34             x[i], x[j] = x[j], x[i] # Swap elements in-place
35
36     # Step 2: Iterative FFT computation
37     m = 2
38     while m <= N:
39         wm = np.exp(anti * 2j * np.pi / m) # Root of unity
40         for k in range(0, N, m):
41             w = 1
42             for j in range(m // 2):
43                 t = w * x[k + j + m // 2] # Odd term
44                 u = x[k + j]           # Even term
45                 x[k + j] = u + t       # update in-place
46                 x[k + j + m // 2] = u - t # update in-place
47                 w *= wm            # Update twiddle factor
48             m *= 2 # Double the segment size
49
50     if anti == 1:
51         x /= N # Normalize for inverse FFT
52
53     return x

```

Ci soffermiamo un attimo, dato che credo sappiate cosa sia un xor, a spiegare cosa sia il bit-shift. Detto semplice " $N >> 1$ " significa che  $N$  viene convertito in binario e i bit sono shiftati di uno a destra (i.e.  $N=8$  in binario è 1000, se shiftiamo di 1 otteniamo 0100 che è 4).

### 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 metà 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         For the forward transform (anti=-1):
8         -----
9         1) The real signal is converted into a complex sequence by combining
10            even and odd samples: z[n] = x[2n] + j * x[2n+1].
11         2) A standard FFT is applied to this reduced sequence of length N/2.
12         3) The symmetric spectrum required for the real transform is reconstructed.
13         4) Take only the first N/2+1 values of the spectrum, for positive frequencies.
14
15         For the inverse transform (anti=1):
16         -----
17         1) Given only the RFFT coefficients, the full spectrum is reconstructed
18            using conjugate symmetry.
19         2) The even and odd components of the signal in the frequency domain are separated.

```

```

20     3) An iFFT of length N/2 is applied to obtain a complex signal Z.
21     4) The real part of Z gives the even-indexed samples, the imaginary part the odd ones.
22
23     Parameters
24     -----
25     x : ndarray
26         data to transform
27     anti : int, optional
28         -1 trasform
29         1 anti trasform
30
31     Return
32     -----
33     rfft : 1d array
34         rfft or anti rfft of x
35     '',
36
37     if anti == -1 :
38         z = x[0::2] + 1j * x[1::2] # Splitting odd and even
39         Zf = FFT(z)
40         Zc = np.array([Zf[-k] for k in range(len(z))]).conj()
41         Zx = 0.5 * (Zf + Zc)
42         Zy = -0.5j * (Zf - Zc)
43
44         N = len(x)
45         W = np.exp(- 2j * np.pi * np.arange(N//2) / N)
46         Z = np.concatenate([Zx + W*Zy, Zx - W*Zy])
47
48     return Z[:N//2+1]
49
50
51     if anti == 1 :
52
53         N = 2 * (len(x) - 1) # Length of the original signal
54         k = np.arange(N//2) # Index until N/2-1
55
56         # Reconstruction of the full spectrum
57         X_full = np.zeros(N, dtype=complex)
58         X_full[:len(x)] = x
59         X_full[len(x):] = np.conj(x[-2:0:-1]) # Simmetria coniugata
60
61         Xe = 0.5 * (X_full[k] + np.conj(X_full[N//2 - k]))
62         Xo = 0.5 * (X_full[k] - np.conj(X_full[N//2 - k])) * np.exp(2j * np.pi * k / N)
63         # N/2 IFFT
64         Z = Xe + 1j * Xo
65         z = FFT(Z, anti=1)
66
67         # Reconstruction of the original signal
68         x_n = np.zeros(N, dtype=float)
69         x_n[0::2] = np.real(z)
70         x_n[1::2] = np.imag(z)
71
72     return x_n

```

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

```

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

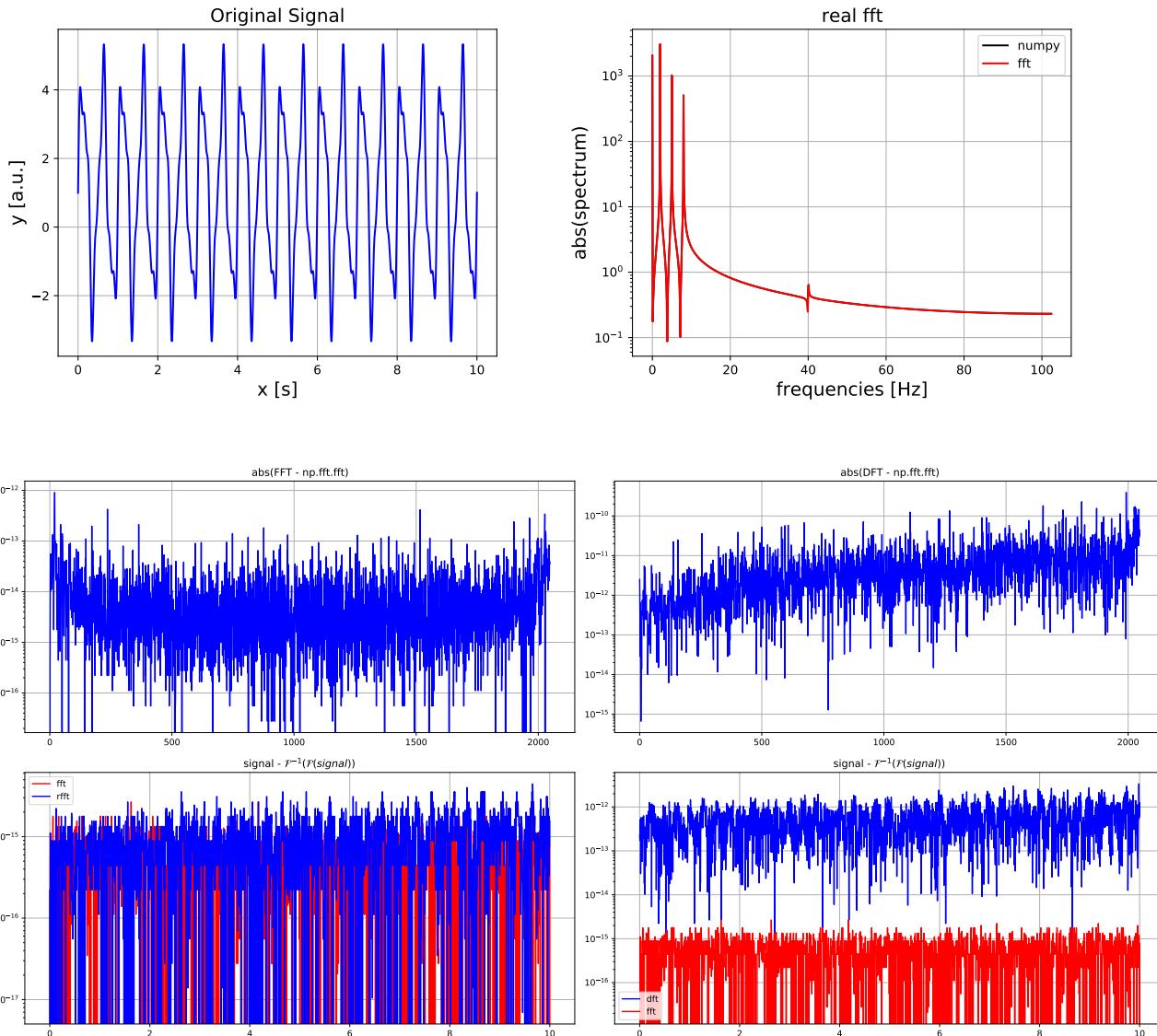
```

```

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.



## G.4 Applicazioni delle FFT

### G.4.1 Derivate con FFT

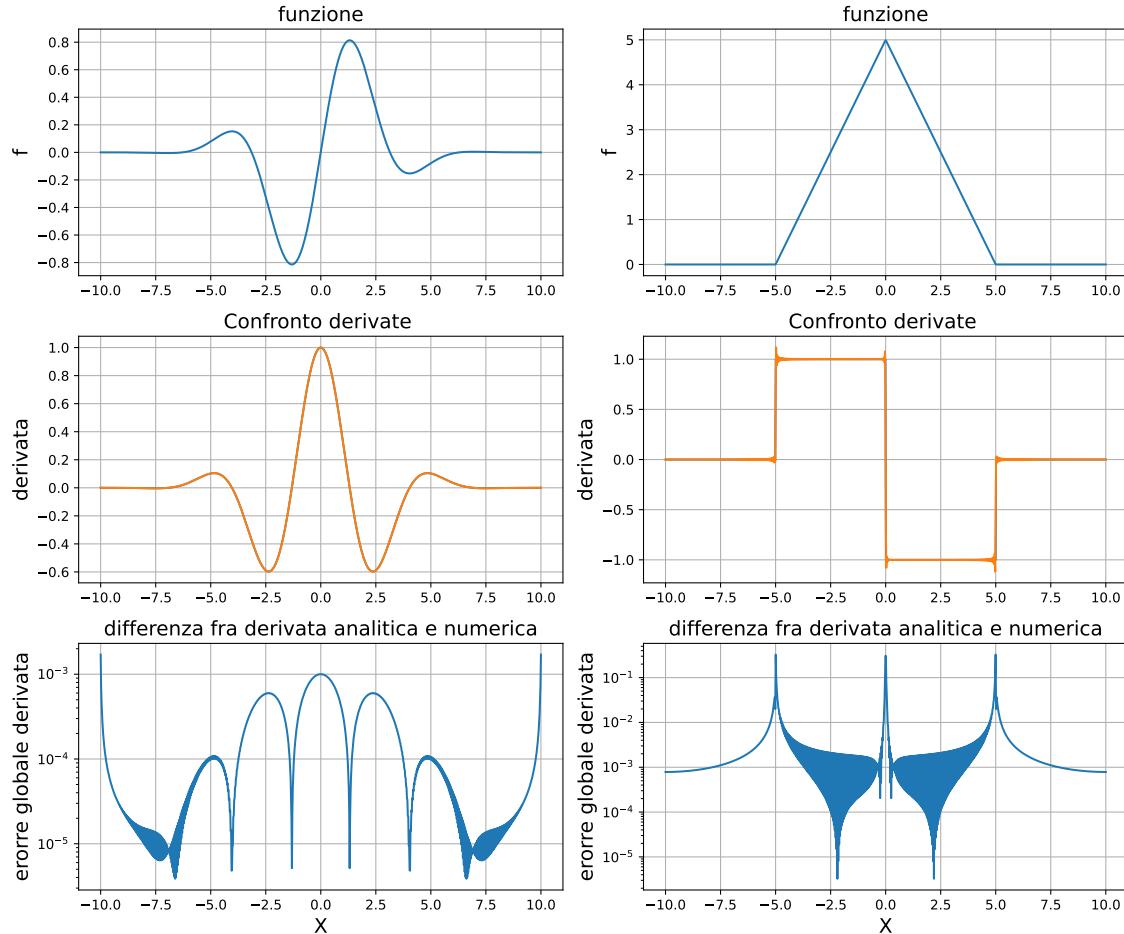
Vediamo ora una comoda applicazione delle fft, ovvero il calcolo delle derivate. In linea di principio quanto stiamo qui per fare si può fare con ogni tipo di polinomio con il quale possiamo sviluppare una certa funzione, ad esempio Legendre o Chebyshev; questo è chiamato metodo dei punti di collocazione. In serie di Fourier la cosa è piuttosto semplice, grazie alla sue proprietà matematiche, una derivata nello spazio (o nel tempo) corrisponde ad una moltiplicazione per  $ik$  ( $i\omega$ ) nello spazio degli impulsi (frequenze). Vediamo come:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 xi = 10          # estremo sinistro
5 xf = -xi         # estremo destro
6 N = 1000         # numero punti
7 dx = (xf-xi)/N # spaziatura punti
8
9 k = 2*np.pi*np.fft.fftfreq(N, dx) # vettore d'onda
10
11 ##### CASO TRANQUILLO #####
12 x = np.linspace(xi, xf, N)          # array posizioni
13 f = np.sin(x)*np.exp(-x**2/10)      # funzione di cui calcolare la derivata
14 g = np.cos(x)*np.exp(-x**2/10) - 2/10*x*f # derivata analitica
15
16 f_hat = np.fft.fft(f)              # trasformo con fourier
17 df_hat = 1j*k*f_hat             # moltiplico per l'impulso
18 df = np.fft.ifft(df_hat) # derivata nello spazio
19
20 plt.figure(1)
21
22 plt.subplot(321)
23 plt.title('funzione', fontsize=15)
24 plt.ylabel('f', fontsize=15)
25 plt.plot(x, f)
26 plt.grid()
27
28 plt.subplot(323)
29 plt.title('Confronto derivate', fontsize=15)
30 plt.ylabel('derivata', fontsize=15)
31 plt.plot(x, g)
32 plt.plot(x, df)
33 plt.grid()
34
35 plt.subplot(325)
36 plt.title('differenza fra derivata analitica e numerica', fontsize=15)
37 plt.ylabel('errore globale derivata', fontsize=15)
38 plt.xlabel("X", fontsize=15)
39 plt.plot(x, abs(g-df))
40 plt.yscale('log')
41 plt.grid()
42 ##### CASO MENO TRANQUILLO #####
43
44 def f():
45     '''funzione
46     '''
47     y = []
48     for xi in x:
49         if xi < -5 :
50             y.append(0)
51         if xi > -5 and xi < 0:
52             y.append(xi + 5)
53         if xi > 0 and xi < 5:
54             y.append(-xi + 5)
55         if xi > 5:
56             y.append(0)
57     return np.array(y)
48
49 def g():
50     ''' derivata analitica
51     '''
52     y = []
53     for xi in x:
54         if xi < -5 :
55             y.append(0)
56         if xi > -5 and xi < 0:
```

```

67         y.append(1)
68     if xi > 0 and xi < 5:
69         y.append(-1)
70     if xi > 5:
71         y.append(0)
72 return np.array(y)
73
74 f = f()
75 g = g()
76
77 f_hat = np.fft.fft(f)          # trasformo con fourier
78 df_hat = 1j*k*f_hat          # moltiplico per l'impulso
79 df      = np.fft.ifft(df_hat) # derivata nello spazio
80
81 plt.subplot(322)
82 plt.title('funzione', fontsize=15)
83 plt.ylabel('f', fontsize=15)
84 plt.plot(x, f)
85 plt.grid()
86
87 plt.subplot(324)
88 plt.title('Confronto derivate', fontsize=15)
89 plt.ylabel('derivata', fontsize=15)
90 plt.plot(x, g)
91 plt.plot(x, df)
92 plt.grid()
93
94 plt.subplot(326)
95 plt.title('differenza fra derivata analitica e numerica', fontsize=15)
96 plt.ylabel('errore globale derivata', fontsize=15)
97 plt.xlabel("X", fontsize=15)
98 plt.plot(x, abs(g-df))
99 plt.yscale('log')
100 plt.grid()
101
102 plt.show()

```



Si vede facilmente come nel caso della funzione a tratti la questione sia più delicata, già ad occhio vediamo dei piccoli spike ai bordi dove la funzione cambia e la derivata ha un gradino, inoltre l'errore è maggiore. Precisiamo che trattandosi di un'integrale, un punto nello spazio diretto ( $x$ ) prende informazione da tutto lo spazio reciproco ( $k$ ).

#### G.4.2 Altre applicazioni

Anche se non le esponiamo è interessante dire come la fft sia molto utile anche nell'analisi dati. Ad esempio se voglia filtrare un segnale si tratterebbe di fare una convoluzione tra segnale di input e il vostro filtro, ma le convoluzioni nello spazio della trasformata sono semplici prodotti. O magari anche capire se vi è o meno una componente debole ad una data frequenza nel vostro segnale: prendete ad esempio il segnale con cui abbiamo fatto sopra i test; se gli aggiungete del rumore, la componente a 40 Hz nello spettro magari non si vede. Ma se il segnale è a media zero vedrete che vi è un picco in negativo molto importante a frequenza nulla nella trasformata; perciò vi basta moltiplicare tutto il segnale per, un seno ad esempio, alla stessa frequenza della componente che cercate; ciò creerà un battimento e una delle due omega sarà nulla. Per cui nella FFT il profondo picco negativo sarà notevolmente ridotto, e ciò vi fa quindi capire la presenza di quell'armonica. Non so quanto sono stato chiaro, magari più il là fornirò degli esempi di codice. Intanto magari potete provare a scriverli da voi.

## H Presa dati da foto

Ora facciamo una breve pausa e mostriamo un paio di codici semplici che possono tornare utili nei laboratori, almeno per primo e terzo anno. Può capitare infatti 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 (Codice scritto per laboratorio 1, tornato utile per laboratorio 3).

```
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\\\\eOverm\\\\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 plt.show()
```

Ogni riga di questo codice permette di percepire il peso del tempo. Potrebbe essere il caso che lui salpi verso occidente per approdare sulle coste di Valinor.

# I Fit

La nostra pausa continua, e come potete capire l'output del codice dell'appendice precedente è diventato l'input di questi (Questi codici sono stati scritti invece durante laboratorio tre, quindi il peso del tempo è minore). Nell'ultima lezione del corso base avevamo visto come fare i fit cercando di spiegare cosa stesse succedendo. In quel caso si trattava di fittare una funzione di una variabile. Qui vogliamo invece mostrare come si possono fittare dei dati che seguono un cerchio o un'elisse.

## I.1 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>.

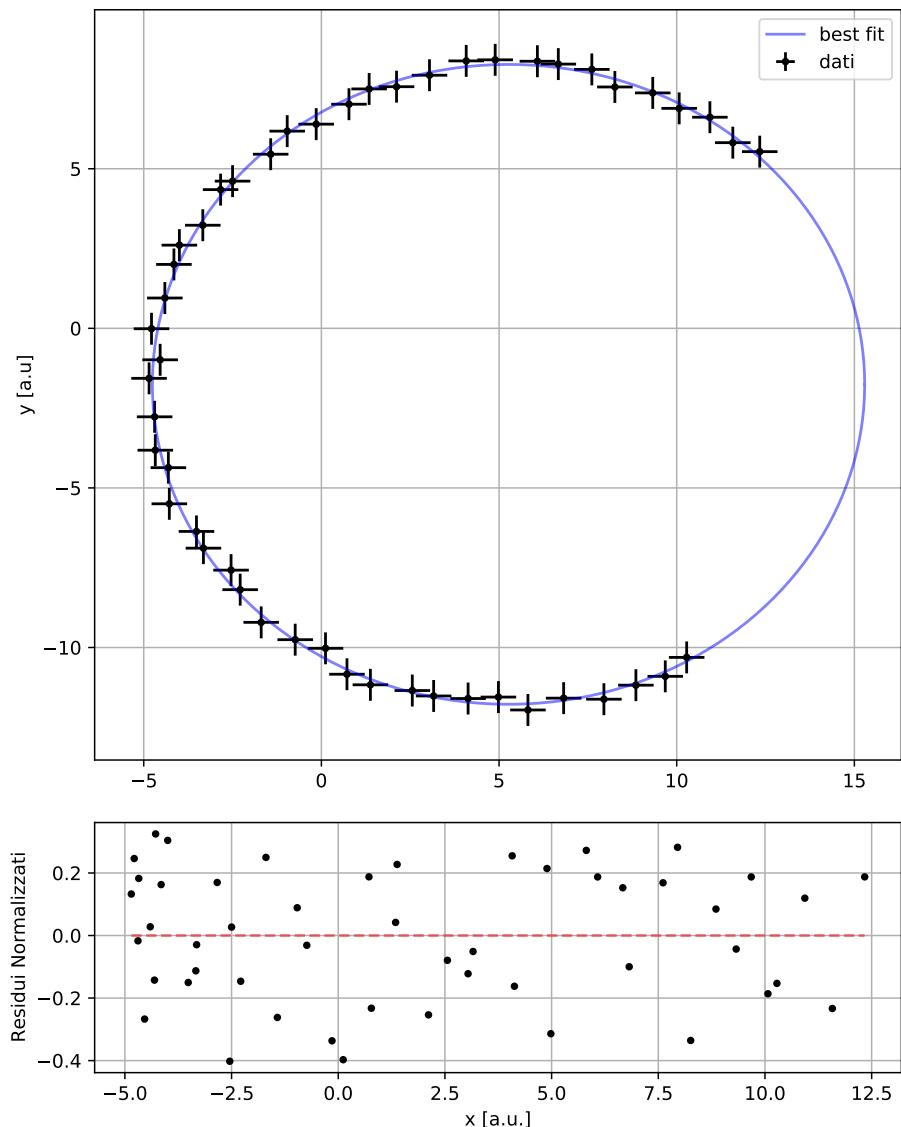
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 def cerchio(xc, yc, r, N, phi_min=0, phi_max=2*np.pi):
6     """
7         Restituisce un cerchio di centro (xc, yc) e di raggio r
8         phi e' il parametro di percorrenza del cerchio
9     """
10
11    phi = np.linspace(phi_min, phi_max, 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.        ]]

```

Fit dati simuli



## I.2 Fit di un'ellisse, metodo di Halir e Flusser

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

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

```

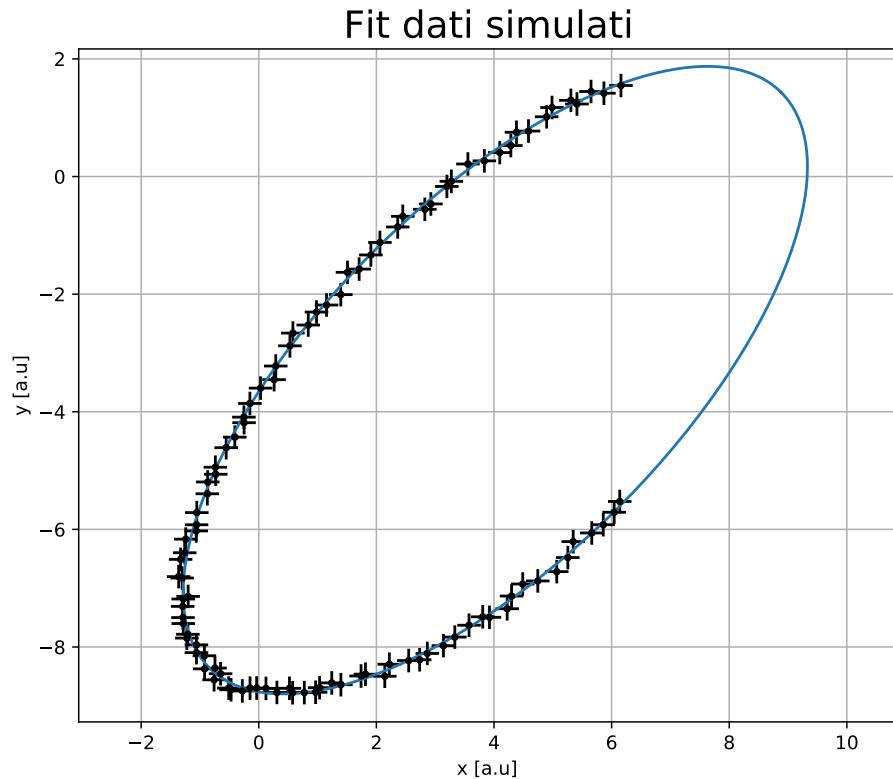
72     phi = np.arctan((2*b)/(a - c))/2
73     if a > c:
74         phi += np.pi/2
75
76     if not M_gt_m :
77         phi += np.pi/2
78
79 #periodicità della rotazione
80 phi = phi % np.pi
81
82 return x0, y0, semi_M, semi_m, e, phi
83
84
85 def fit_ellisse(x, y):
86     """
87     Basato sull'articolo di Halir and Flusser,
88     "Numerically stable direct
89     least squares fitting of ellipses".
89
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'ellisse
116     x0, y0 = 4, -3.5
117     semi_M, semi_m = 7, 3
118     phi = np.pi/4
119     #eccentricità 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:.4f}, {y0:.4f}, semi_M:{semi_M:.4f}, semi_m:{semi_m:.4f}, phi:{phi:.4f}, e:{e:.4f}')
140     x0, y0, semi_M, semi_m, e, phi = cartesiano_a_polaris(coef_cart)
141     print('valori fittati')
142     print(f'{x0:.4f}, {y0:.4f}, semi_M:{semi_M:.4f}, semi_m:{semi_m:.4f}, phi:{phi:.4f}, e:{e:.4f}')
143
144     #plot
145     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

```



## J 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.

### J.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:

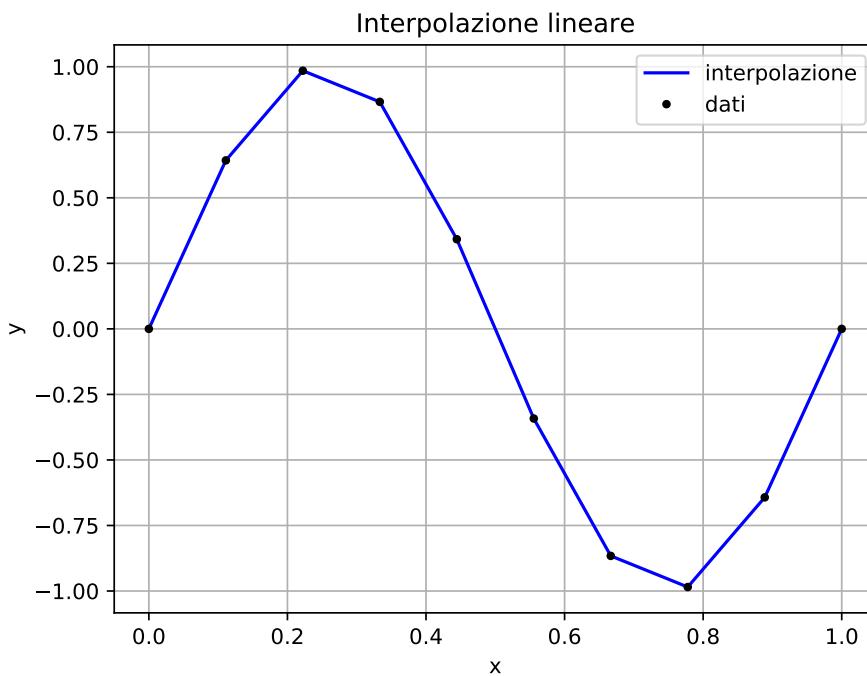
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def f(x, xx, yy):
5     """
6         restituisce l'interpolazione dei punti xx yy
7         x puo' essere un singolo valore in cui calcolare
8         la funzione interpolante o un intero array
9     """
10    #proviamo se x e' un array
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        #controllo 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            #controllo 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 da errore se si calcola la lunghezza di un numero. Vi è poi il sollevamento di un'eccezione in caso i valori di interesse siano fuori dagli estremi della tabella dove non si può dire nulla quindi il codice si interrompe. Vediamo il risultato:



## J.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:

```

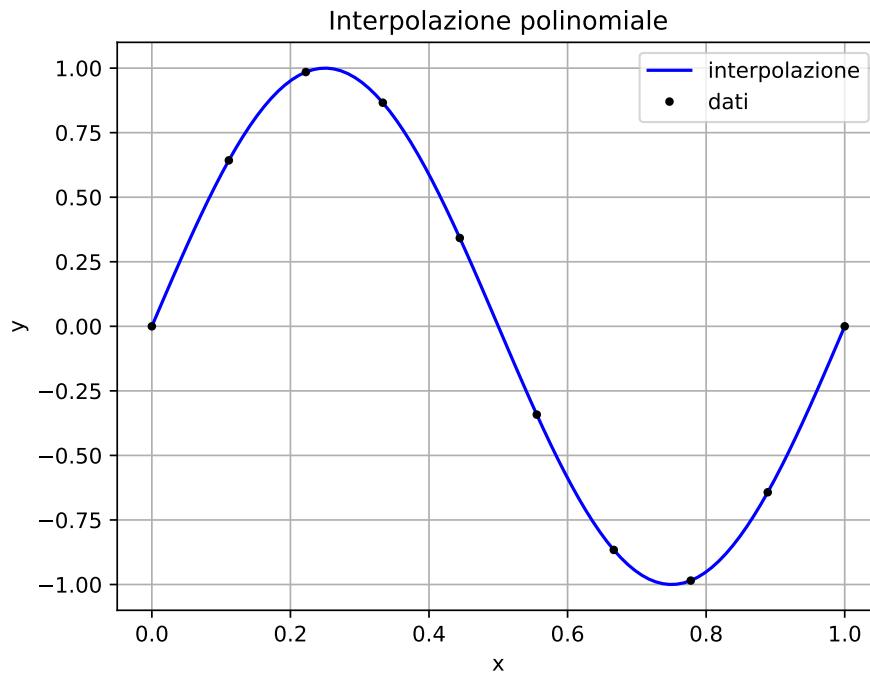
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 N = 10
5 x = np.linspace(0, 1, N)
6 y = np.sin(2*np.pi*x)
7
8 #Matrice di Vandermonde

```

```

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()

```



### J.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).

```

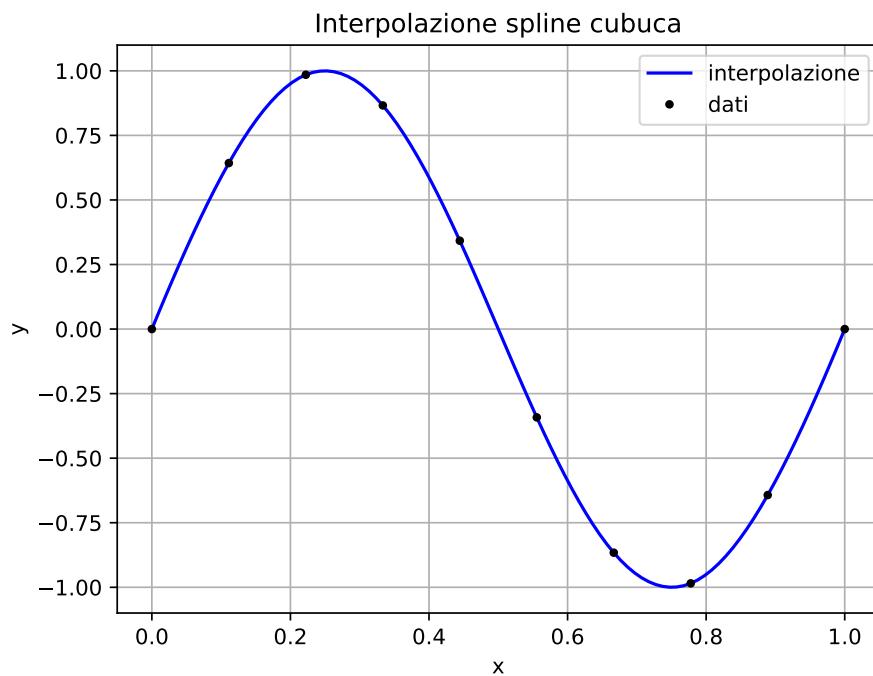
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.interpolate import InterpolatedUnivariateSpline
4
5 N = 10
6 x = np.linspace(0, 1, N)
7 y = np.sin(2*np.pi*x)
8
9 #interpolazione con una, spline cubica (k=3)

```

```

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()

```



# K Programmazione a oggetti

## K.1 Problema N-body

Python è un linguaggio che permette la programmazione ad oggetti. Molto di Python stesso è scritto con programmazione orientata ad oggetti. Lo scopo è quello di illustrare un semplice esempio di utilizzo creando una piccola simulazione ad N corpi. Per programmare ad oggetti si utilizza quelle che sono chiamate classi; fondamentalmente creare una classe vuol dire definire un oggetto. All'interno della classe è possibile definire delle funzioni che verranno chiamati metodi. Alcuni metodi sono particolari e sono contrassegnati da doppi underscore, e.g: `__init__`, `__iter__`, `__next__`, `__call__`; qui ci limiteremo al primo, in quanto necessario, e l'ultimo poiché può essere utile. I due nel mezzo permettono di costruire un "iterabile", parola che non dovrebbe esservi nuova; così come l'ultimo permette di costruire un oggetto "chiamabile" ("callable" anche questo non dovrebbe sembrarvi nuovo). Cominciamo quindi, nell'ottica della nostra simulazione a N corpi, a definire la nostra classe che ci permetterà di costruire i protagonisti della nostra simulazione (ci limiteremo per semplicità a vincolare la dinamica su un piano):

```
1 import numpy as np
2 import random as rn
3 import matplotlib.pyplot as plt
4 from matplotlib import animation
5
6 class Body:
7     """
8         Classe che rappresenta una pallina
9         intesa come oggetto puntiforme
10    """
11
12     def __init__(self, x, y, vx, vy, m=1):
13         """
14             costruttore della classe, verra' chiamato
15             quando creeremo l'istanza della classe, (i.e. b=Body()
16             b e' chiamata istanza).
17             In input prende la posizione, la velocita' e massa
18             che sono le quantita' che identificano il corpo
19             che saranno gli attributi della classe;
20             il costruttore e' un particolare metodo della
21             classe per questo si utilizzano gli underscore.
22             il primo parametro che passiamo (self) rappresenta
23             l'istanza della classe (self e' un nome di default)
24             questo perche' la classe e' un modello generico che
25             deve valere per ogni corpo.
26         """
27         #posizione
28         self.x = x
29         self.y = y
30         #velocita'
31         self.vx = vx
32         self.vy = vy
33         #massa
34         self.m = m
35
36         #aggiornamento posizione e velocita' con eulero
37     def n_vel(self, fx, fy, dt):
38         """
39             ad ogni metodo della classe viene passato
40             come primo argomento self , quindi l ' istanza
41             date le componenti della forza e il passo temporale
42             aggiorno le componenti della velocita'
43         """
44         self.vx += fx*dt
45         self.vy += fy*dt
46
47     def n_pos(self, dt):
48         """
49             dato il passo temporale aggiorno le posizioni
50         """
51         self.x += self.vx*dt
52         self.y += self.vy*dt
```

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. Dunque ora abbiamo qualcosa che ci permette di creare i nostri oggetti, ognuno con caratteristiche (valori degli attributi) diversi. Abbiamo poi due metodi che ci permettono di aggiornare le quantità fisiche e dare una dinamica a tutti i nostri corpi. Vediamo ora, sempre

grazie ad un'altra classe come implementare la nostra simulazione. Useremo per evitare divergenze in caso di incontri ravvicinati (del terzo tipo) quella che è la tecnica del softening, cioè il potenziale kepleriano sarà:

$$V(r) = -\frac{1}{\sqrt{r^2 + \epsilon}} \quad (85)$$

con  $\epsilon$  chiamato appunto parametro di softening. Così anche se i pianeti sono molto vicini la forza tra essi non diverge. Se vogliamo simulare una normale orbita possiamo sempre settare a zero questo parametro.

```

1 class Sistema:
2     """
3         Classe per evoluzione del sistema.
4         Viene utilizzata la tecnica del softening per impedire
5         divergenze nella foza, sp e' il parametro di softening
6
7     Parameters
8     -----
9     corpi : list
10        lista di oggetti della classe Body
11    G : float
12        Costante di gravitazione universale (=1)
13    sp : float, optional, default 0
14        parametro di softening
15        ,
16
17    def __init__(self, corpi, G, sp=0):
18        self.corpi = corpi
19        self.G = G
20        self.sp = sp
21
22    def evolvo(self, dt):
23        """
24            chiamata ad ogni passo temporale, fa evolvere il sistema
25            solo di uno step dt, la forza e' calcolata secondo la
26            legge di gravitazione universale;
27        ,
28
29        for corpo_1 in self.corpi:
30
31            fx = 0.0
32            fy = 0.0
33
34            for corpo_2 in self.corpi:
35                if corpo_1 != corpo_2:
36
37                    dx = corpo_2.x - corpo_1.x
38                    dy = corpo_2.y - corpo_1.y
39
40                    d = np.sqrt(dx**2 + dy**2 + self.sp)
41
42                    fx += self.G * corpo_2.m * dx / d**3
43                    fy += self.G * corpo_2.m * dy / d**3
44
45                    corpo_1.n_vel(fx, fy, dt)
46
47            for corpo in self.corpi:
48                corpo.n_pos(dt)

```

Fondamentalmente la nostra classe ci permette quindi di integrare le varie equazioni del moto. Importante notare il metodo con cui esse sono integrate: aggiorniamo prima tutte le velocità e poi tutte le posizioni. Questo è chiamato metodo di euler simplettico. Ora mostriamo la parte conclusiva del codice per avviare la nostra simulazione:

```

1 #=====
2 # Creating bodies and the system and computational parameters
3 #=====
4
5 rn.seed(69420)
6 dt = 1/20000
7 T = int(2/dt)
8 E = np.zeros(T)
9 L = np.zeros(T)
10 G = 1
11
12 # Number of body, must be even
13 N = 10

```

```

14 C = []
15 for n in range(N//2):
16     '',
17     two bodies are created at a time
18     with equal and opposite velocity
19     to keep the total momentum of the system zero
20     '',
21     v_x = rn.uniform(-0.5, 0.5)
22     v_y = rn.uniform(-0.5, 0.5)
23     C.append(Body(rn.uniform(-0.5, 0.5), rn.uniform(-0.5, 0.5), v_x, v_y))
24     C.append(Body(rn.uniform(-0.5, 0.5), rn.uniform(-0.5, 0.5), -v_x, -v_y))
25
26
27 X = np.zeros((2, T, N)) # 2 because the motion is on a plane
28
29 # Creation of the system
30 soft = 0.01
31 sist = System(C, G, soft)
32
33 =====
34 # Evolution
35 =====
36
37 start = time.time()
38
39 for t in range(T):
40     sist.update(dt)
41     for n, body in enumerate(sist.bodies):
42         X[:, t, n] = body.x, body.y
43
44 print("--- %s seconds ---" % (time.time() - start))
45
46 =====
47 # Plot and animation
48 =====
49
50 fig = plt.figure(0)
51 plt.grid()
52 plt.xlim(np.min(X[::2, :])-0.5, np.max(X[::2, :])+0.5)
53 plt.ylim(np.min(X[1::2,:])-0.5, np.max(X[1::2,:])+0.5)
54 colors = ['b']*N#plt.cm.jet(np.linspace(0, 1, N))
55
56 dot = np.array([]) # for the planet
57
58 for c in colors:
59     dot = np.append(dot, plt.plot([], [], 'o', c=c))
60
61 def animate(i):
62
63     for k in range(N):
64
65         dot[k].set_data((X[0, i, k],), (X[1, i, k],))
66
67     return dot
68
69 anim = animation.FuncAnimation(fig, animate, frames=np.arange(0, T, 50), interval=1, blit=True,
70 , repeat=True)
71
72 plt.title('N body problem', fontsize=20)
73 plt.xlabel('X(t)', fontsize=20)
74 plt.ylabel('Y(t)', fontsize=20)
75
76 # Uncomment to save the animation, extra_args for .mp4
77 #anim.save('N_body.gif', fps=50)# extra_args=['-vcodec', 'libx264'])
78
79 plt.show()

```

Oltre ad n palline a caso, creiamo anche un sistema, un pianeta che orbita intorno a due stelle:

```

1 # creation of body
2 C1 = Body(0.5, 0, 0, 20, int(1e3))
3 C2 = Body(-0.5, 0, 0, -20, int(1e3))
4 C3 = Body(-1.5, 0, 0, 40, int(1e1))
5 C = [C1, C2, C3]
6 N = len(C)

```

Ora in linea di principio la simulazione finisce qui. Tuttavia risulta interessante calcolare l'energia e il momento angolare del nostro sistema e vedere se esse, come ci si aspetteremmo, sono effettivamente conservate durante l'evoluzione del sistema. Mostreremo i risultati ma senza mostrare il codice che comunque trovate disponibile. Tra l'altro questa analisi la vogliamo fare mostrando soltanto il caso di tre corpi che si orbitano attorno in quanto nei due casi che mostreremo (due integratori diversi) le differenze sono apprezzabili. Usando N corpi creati a caso la conservazione dell'energia non si manifesta mai, a differenza, e vedremo perché, della conservazione del momento angolare. Per quanto riguarda la conservazione dell'energia probabilmente ciò è dovuto al fatto che, quelle scelte, non sono delle buone condizioni iniziali: in genere l'inizializzazione è affrontata in maniera più delicata (a noi interessava solo vedere l'animazione carina). Da qui e dal tipo di integratore usato, che abbiamo citato prima nasce tutta un'interessante discussione sulla meccanica classica, che brevemente vogliamo trattare.

## K.2 Breve compendio di meccanica Hamiltoniana

Allora come prima cosa diamo un paio di definizioni, cercando comunque di non essere troppo matematici (non dimostriamo nulla qui, lasciamo tutto all'eventuale curiosità dell'eventuale lettore):

Consideriamo inizialmente una 2-forma  $\omega$  differenziale su una varietà liscia  $M$ . La 2-forma  $\omega$  è chiamata simplettica se è chiusa e non degenere. Chiusa significa che la sua derivata esterna è nulla  $d\omega = 0$ ; non vogliamo perdere tempo a spiegare cosa sia la derivata esterna, ma la potete vedere come una derivata che sia antisimmetrica (cfr. tensore elettromagnetico: dato  $A$  potenziale vettore  $dA = \partial_i A_j - \partial_j A_i = F_{ij}$ ). Non degenere significa invece che, per ogni punto  $p \in M$ , considerando lo spazio tangente al punto  $T_p M$  (che è uno spazio vettoriale), si ha che per ogni vettore  $x$  non nullo su questo spazio esiste un vettore  $y$  tale che:  $\omega(x, y) = 0$  solo per  $y = 0$ . Dunque la coppia  $(M, \omega)$  è una varietà simplettica.

Consideriamo ora una funzione liscia  $H : M \rightarrow \mathbb{R}$  su una varietà simplettica  $(M, \omega)$ . Abbiamo allora che il campo vettoriale  $X_H$  definito da  $\omega(X_H, Y) = dH(Y)$  è chiamato campo vettoriale hamiltoniano generato da  $H$ . La terna  $(M, \omega, H)$  definisce un sistema hamiltoniano.

Inoltre è importante menzionare il seguente teorema: Sia  $(M, \omega, H)$  un sistema hamiltoniano e sia  $\Phi : \mathbb{R} \times M \rightarrow M$  il flusso generato dal campo vettoriale  $X_H$ . Allora  $\forall t \in \mathbb{R} \Phi_t$  è simplettico. Questo significa che il flusso non solo preserva la struttura simplettica, in particolare esso preserva la forma volume, il che vuol dire che preserva il volume dello spazio delle fasi. Inoltre il flusso esatto della nostra hamiltoniana è reversibile  $\Phi_t^{-1} = \Phi_{-t}$ . Detto ciò vediamo di concretizzare un po'; supponiamo di avere una hamiltoniana della forma:

$$H(p, q) = T(p) + V(q) . \quad (86)$$

Se definiamo  $x = (p, q)$  l'insieme di tutte le nostre variabili allora sappiamo che l'equazione del moto sono date dalla parentesi di poisson:

$$\dot{x} = X_H = \{x, H(x)\} , \quad (87)$$

la cui soluzione è:

$$x(\tau) = e^{\tau X_H} x(0) . \quad (88)$$

Abbiamo detto che vogliamo dunque integrare le nostre equazioni del moto. Assumiamo che dato un certo  $n$  esistano certi coefficienti  $c_1, \dots, c_k$  e  $d_1, \dots, d_k$  tale che:

$$e^{\tau X_H} = e^{\tau(X_T + X_V)} = \prod_{i=1}^k e^{\tau c_i X_T} e^{\tau d_i X_V} + \mathcal{O}(\tau^{n+1}) , \quad (89)$$

dove i  $c_i$  e  $d_i$  devono rispettare certi vincoli a seconda del valore di  $n$ , e a seconda del quale inoltre si ottengono diversi algoritmi: per  $n = 1$  abbiamo eluero simplettico (quello implementato sopra) mentre per  $n = 2$  abbiamo il leapfrog. Non ci metremo ora a calcolare questi coefficienti. Inoltre si dimostra, ma noi non lo facciamo, che trovare i  $c_i$  e  $d_i$  che soddisfano la (89) è equivalente a trovare dei coefficienti  $c_i$  e  $d_i$  tali che:

$$\Phi_\tau = \prod_{i=1}^k e^{\tau c_i X_T} e^{\tau d_i X_V} = e^{\tau(X_T + X_V) + \mathcal{O}(\tau^{n+1})} . \quad (90)$$

É inoltre interessante notare che se consideriamo l'integratore di eulero:  $\Phi_h = e^{hX_T} e^{hX_V}$  esso è soluzione esatta di un altro sistema hamiltoniano scritto come perturbazioni di  $H$ , che non è detto abbia le stesse simmetrie dell'hamiltoniana di partenza:

$$\bar{H} = H + hH_2 + h^2H_3 + \dots \quad (91)$$

dove gli  $H_i$  si calcolano tramite parentesi di poisson espandendo in serie con la formula di Baker-Campbell-Hausdorff (BCH):

$$\begin{aligned}\overline{X_H} &= \frac{1}{h} \ln(e^{hX_T} e^{hX_V}) \\ &= \underbrace{X_T + X_V}_{H_1} + h \underbrace{\frac{1}{2}[X_T, X_V]}_{H_2} + h^2 \underbrace{\frac{1}{12}([[X_T, X_V], X_V] + [[X_V, X_T], X_T])}_{H_3} .\end{aligned}\quad (92)$$

Più in generale per un integratore della forma:  $\Phi_h = \prod_{i=1}^k e^{hc_i X_T} e^{hd_i X_V}$  esso è soluzione di:

$$\overline{H} = H + h^n H_{n+1} + \mathcal{O}(h^{n+1}) , \quad (93)$$

dove ora i vari  $H_i$  saranno diversi ma il modo di calcolarli è sempre lo stesso. Tornando a noi è stato implementato per fare un confronto anche un medoto simplettico del 4 ordine noto con il nome di Yoshida; scrivendo in coordinate lagrangiane la regola iterativa è:

$$v_{i+1} = v_i + d_i a(x_i) dt \quad (94)$$

$$x_{i+1} = x_i + c_i v_{i+1} dt \quad (95)$$

dove i coefficienti valgono:

$$c_1 = c_4 = \frac{1}{2(2 - 2^{1/3})} \quad c_2 = c_3 = \frac{1 - 2^{1/3}}{2(2 - 2^{1/3})}, \quad (96)$$

$$d_1 = d_3 = \frac{1}{2 - 2^{1/3}}, \quad d_2 = -\frac{2^{1/3}}{2 - 2^{1/3}}, \quad d_4 = 0. \quad (97)$$

Ultima cosa da far notare, ma molto interessante è che ogni integratore simplettico conserva il momento angolare, o più in generale è conservato qualsiasi generatore di trasformazioni puntuali lineari che sia una simmetria di  $H$  (e.g. rotazioni, dilatazioni, traslazioni). Per trasformazioni del tipo:

$$q_i \rightarrow q_i + \epsilon A_i(q) = q_i + \epsilon(\Omega_{ij} q_j + n_i), \quad (98)$$

$$p_i \rightarrow p_i - \epsilon \Omega_{ij} p_j , \quad (99)$$

dove  $\Omega$  è una matrice  $N \times N$  e  $n$  è un vettore di coefficienti costanti. L'invarianza di  $H$  si scrive come:

$$\frac{\partial H}{\partial q_i} \delta q_i + \frac{\partial H}{\partial p_i} \delta p_i = \epsilon \left( \frac{\partial H}{\partial q_i} n_i + \frac{\partial H}{\partial q_i} \Omega_{ij} q_j - \frac{\partial H}{\partial p_i} \Omega_{ij} p_j \right) = 0 . \quad (100)$$

Potete divertirvi a sostituire l'espressione data per le nuove coordinate date dall'integratore e far vedere che il generatore  $G(p, q) = A_i p_i = \Omega_{ij} p_i q_j + n_i p_i$  è conservato.

Un modo magari più familiare per vedere tutto quel che abbiamo detto è quello dell'utilizzo delle trasformazioni canoniche.

Ricordiamo intanto che:  $Q(q, p, t)$  e  $P(q, p, t)$  sono trasformazioni canoniche se per una qualsiasi hamiltoniana  $H(q, p, t)$  si può trovare un'hamiltoniana  $K(Q, P, t)$  tale per cui le equazioni del moto per  $p$  e  $q$  data da  $H$ , si trasformano in quelle date da  $K$  in termini di  $Q$  e  $P$ .

In termini di  $x$ , definito sopra, abbiamo che:

$$\dot{x} = \Gamma \frac{\partial H}{\partial x} \quad \text{con} \quad \Gamma = \begin{pmatrix} 0 & -\text{Id}_{N \times N} \\ \text{Id}_{N \times N} & 0 \end{pmatrix} \quad (101)$$

Facciamo ora una trasformazione  $X(x)$  (si generalizza a anche a trasformazioni dipendenti dal tempo). Abbiamo quindi che:

$$\dot{X}_i = \frac{\partial X_i}{\partial x_j} \dot{x}_j = \frac{\partial X_i}{\partial x_j} \Gamma_{jl} \frac{\partial H}{\partial x_l} = \frac{\partial X_i}{\partial x_j} \Gamma_{jl} \frac{\partial H}{\partial X_k} \frac{\partial X_k}{\partial x_l} = J_{ij} \Gamma_{jl} J_{lk}^T \frac{\partial H}{\partial X_k} = (J \Gamma J^T)_{ik} \frac{\partial H}{\partial X_k} \quad (102)$$

Per cui si arriva a:

$$J \Gamma J^T = \Gamma, \quad (103)$$

equazione che, in termini di matrici, definisce il gruppo simplettico. Diciamo questo perché tra simpletticità e canonicità c'è un legame stretto:  $X(x, t)$  è canonica se la sua matrice jacobiana è simplettica a tutti i tempi a meno di una costante cioè:  $J \Gamma J^T = \alpha \Gamma$ . Inoltre se abbiamo che:

$$\{A, B\}_x = \{A, B\}_X, \quad (104)$$

la nostra trasformazione è simplettica. Ma sappiamo anche che verificare le parentesi di Poisson ci dà la canonicità della trasformazione:

$$X_i = \{X_i, H\}_x = \{X_i, H\}_X = \Gamma_{ij} \frac{\partial H}{\partial X_j} \quad (105)$$

Facciamo quindi un esempio semplice per far vedere come il metodo di Eulero non sia simplettico ma quello da noi implementato sopra lo sia. Per semplicità consentitemi di considerare (piuttosto che  $N$  corpi) la seguente:

$$H = \frac{p^2}{2} - \frac{1}{q}. \quad (106)$$

Il metodo di Eulero prevede:

$$p(t + \delta t) = p(t) - \delta t \frac{\partial H(q(t), p(t))}{\partial q} \quad (107)$$

$$q(t + \delta t) = q(t) + \delta t \frac{\partial H(q(t), p(t))}{\partial p} \quad (108)$$

Nel nostro caso abbiamo:

$$p(t + \delta t) = p(t) - \delta t \frac{1}{q^2(t)} \quad (109)$$

$$q(t + \delta t) = q(t) + \delta t p(t) \quad (110)$$

Per verificare la canonicità va dunque calcolata la:

$$\begin{aligned} \{q(t + \delta t), p(t + \delta t)\} &= \{q(t), p(t)\} - \delta t \{q(t), q^{-2}(t)\} + \delta t \{p(t), p(t)\} - \delta t^2 \{p(t), q^{-2}(t)\} \\ &= 1 - \delta t^2 2q^{-3}(t). \end{aligned} \quad (111)$$

Vediamo dunque che la trasformazione non è canonica a tutti gli ordini in  $\delta t$ . Il metodo di Eulero simplettico invece ci dice:

$$p(t + \delta t) = p(t) - \delta t \frac{\partial H(q(t), p(t + \delta t))}{\partial q} \quad (112)$$

$$q(t + \delta t) = q(t) + \delta t \frac{\partial H(q(t), p(t + \delta t))}{\partial p} \quad (113)$$

Che nel nostro caso diventa:

$$p(t + \delta t) = p(t) - \delta t \frac{1}{q^2(t)} \quad (114)$$

$$q(t + \delta t) = q(t) + \delta t p(t + \delta t) = q(t) + \delta t p(t) - \delta t^2 \frac{1}{q^2(t)} \quad (115)$$

Per cui abbiamo:

$$\begin{aligned} \{q(t + \delta t), p(t + \delta t)\} &= \{q(t), p(t)\} - \\ &\quad - \delta t (\{q(t), q^{-2}(t)\} - \{p(t), p(t)\}) - \\ &\quad - \delta t^2 (\{p(t), q^{-2}(t)\} + \{q^{-2}(t), p(t)\} - \{q^{-2}(t), q^{-2}(t)\}) \\ &= 1 \end{aligned} \quad (116)$$

Si vede bene dunque che la trasformazione è canonica per ogni  $\delta t$ ; dato che il termine che prima rompeva la canonicità ora si cancella con un altro termine, mentre gli altri sono tranquillamente nulli. Potete se volete verificare che Eulero simplettico è dato dalla seguente funzione generatrice:

$$F_2(q(t), p(t + \delta t)) = q(t)p(t + \delta t) + \delta t H(q(t), p(t + \delta t)). \quad (117)$$

Il problema di questo metodo è però che non è reversibile. Si può però combinare con la variante in cui si usa  $q(t + \delta t)$  per aggiornare  $p(t)$ , dando così vita ad un metodo del secondo ordine che è il leapfrog citato prima. Il quale è reversibile. Vediamo ora i risultati, dal punto di vista delle conservazioni, di due integratori (Eulero simplettico e Yoshida del quarto ordine):

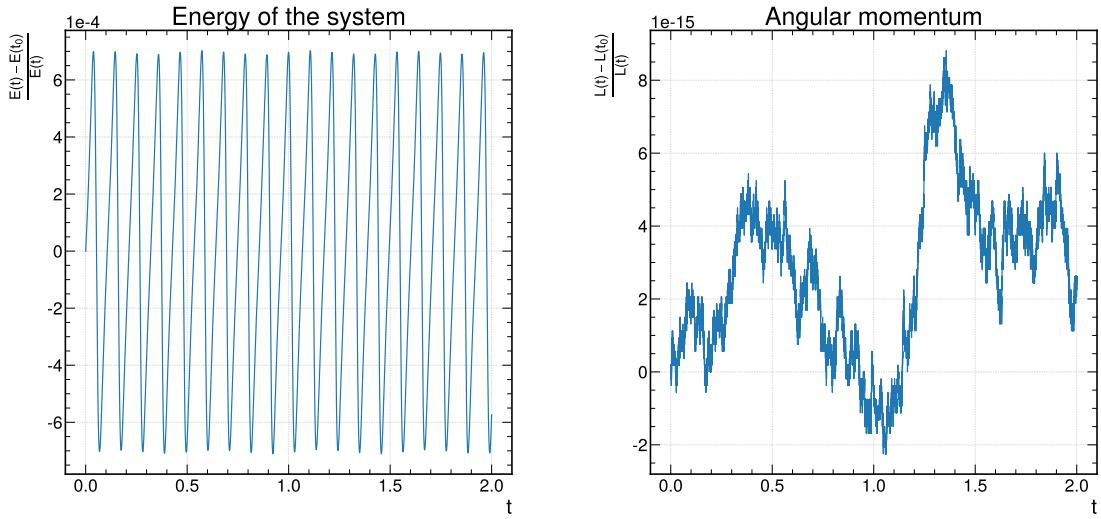


Figura 13: Conservazione di energia e momento angolare con l'integratore di Eulero simplettico.

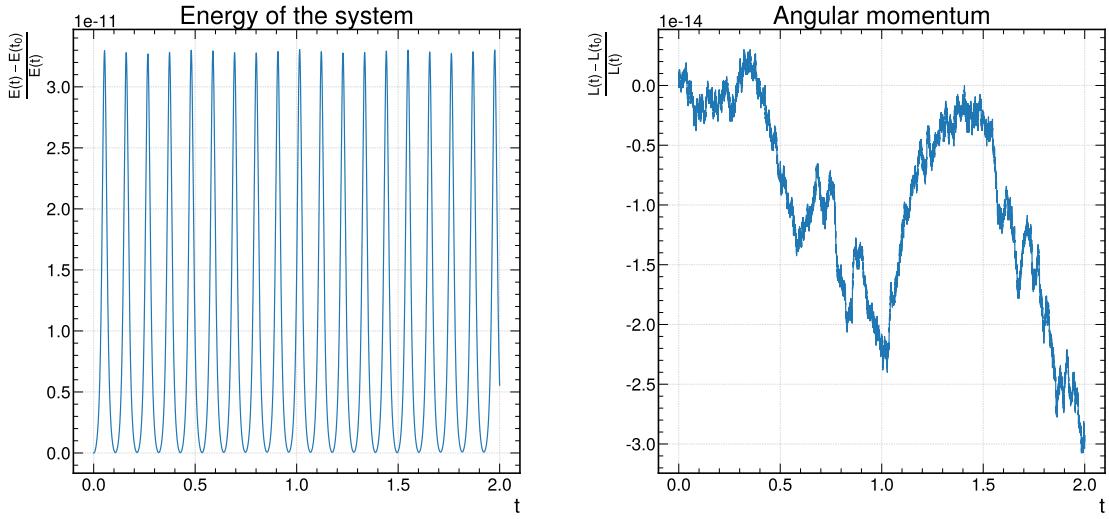


Figura 14: Conservazione di energia e momento angolare con l'integratore di Yoshida del quarto ordine.

Vediamo dunque come la conservazione dell'energia sia molto migliore nel caso di un integratore di ordine superiore, mentre la variazione di momento angolare è sempre dello stesso ordine circa. Avrete notato che lo stile dei plot è leggermente diverso, questo è dovuto al fatto che, per ottenere una migliore leggibilità, si è usato:

```
1 import mplhep
2 plt.style.use(mplhep.style.CMS)
```

Ok che è pallettaro ma onore al merito.

### K.3 Più che una funzione

Le classi sono molto utili perché 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. Se ricordate, nell’appendice precedente è presente un codice che esegue un’interpolazione lineare; se andiamo a vederlo e ci fermiamo un attimo a pensare notiamo subito che quella funzione ogni volta che viene chiamata rifà tutto il conto necessario per trovare i coefficienti del polinomio in tutto l’intervallo. 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, un po’ più complicato ma più interessante, di una spline cubica (anche se a mali estremi un’interpolazione lineare funziona sempre, ordini

più alti possono dare problemi). Spieghiamo intanto cos'è una spline, si tratta di una funzione del tipo:

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3 \quad . \quad (118)$$

Stiamo quindi connettendo i vari punti del grafico con un polinomio, nella fattispecie di terzo grado. Richiediamo quindi che siano soddisfatte le seguenti condizioni.

- $S_i(x_i) = y_i = S_{i-1}(x_i)$
- $S_0(x_0) = y_0$
- $S_{n-1}(x_n) = y_n$
- $S'_i(x_i) = S'_{i-1}(x_i)$
- $S''_i(x_i) = S''_{i-1}(x_i)$
- $S''(x_0) = S''_{n-1}(x_n) = 0$  .

Le prime tre, quelle di sinistra, impongono il passaggio della spline per i punti. Le restanti tre sono, in ordine: continuità della derivata prima e seconda nel punto e infine la richiesta di spline naturale, ovvero fuori dall'intervallo dei dati la spline sarà una linea retta. Spesso si usa un polinomio di grado tre in quanto è il minimo grado necessario per minimizzare la curvatura e inoltre minore è il grado maggiormente si evitano oscillazioni. Noi quindi implementeremo una spline naturale:

```

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(self.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

```

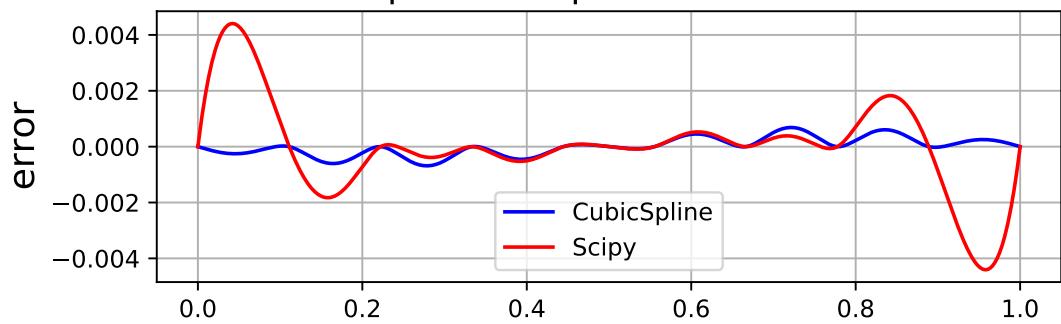
```

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
144 #=====#
145 # Cfr scipy and our spline
146 #=====#
147
148 from scipy.interpolate import InterpolatedUnivariateSpline
149 s3 = InterpolatedUnivariateSpline(x, y, k=3)
150 plt.figure(2)
151 plt.subplot(211)
152 plt.title("Spline interpolation N = 10", fontsize=15);
153 plt.ylabel("error", fontsize=15)
154 plt.plot(z, G(z)-np.sin(2*np.pi*z), 'b', label='CubicSpline');
155 plt.plot(z, s3(z)-np.sin(2*np.pi*z), 'r', label='Scipy');
156 plt.grid();plt.legend(loc='best');
157 plt.subplot(212);
158 plt.title("Spline interpolation N = 30", fontsize=15)
159
160 x = np.linspace(0, 1, 30)
161 y = np.sin(2*np.pi*x)
162 G = CubicSpline(x, y)
163 s3 = InterpolatedUnivariateSpline(x, y, k=3)
164
165 plt.ylabel("error", fontsize=15)
166 plt.xlabel('x', fontsize=15)
167 plt.plot(z, G(z)-np.sin(2*np.pi*z), 'b', label='CubicSpline');
168 plt.plot(z, s3(z)-np.sin(2*np.pi*z), 'r', label='Scipy');
169 plt.grid();plt.legend(loc='best')
170 plt.show()
171
172 [Output]
173 0.9508316728694627

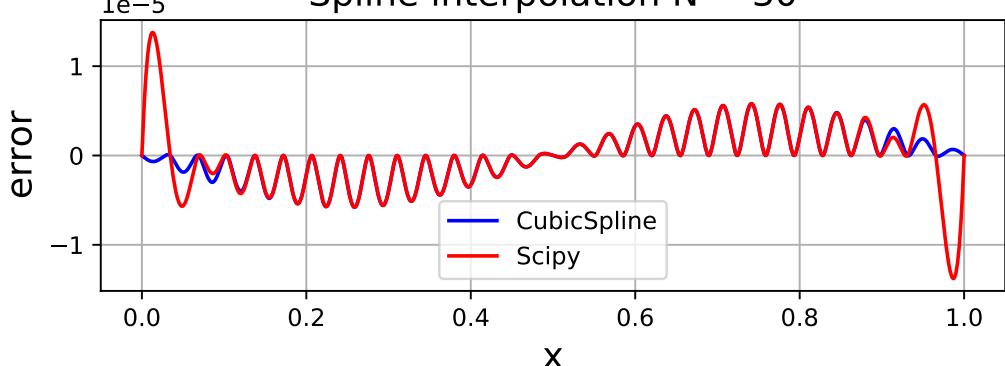
```

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 che abbiamo usato da `scipy` e che abbiamo visto sopra; Per vedere cosa cambia calcoliamo la differenza fra l'interpolazione calcolata in  $z$  (l'array nel codice) e i valori che restituisce `np.sin(2*np.pi*z)`. 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.

Spline interpolation N = 10



Spline interpolation N = 30



## L Propagazione errori

Può capitare spesso, sempre, che vadano propagati degli errori. 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.

### L.1 Propagarli a mano

Come primo tentativo potremmo scrivere noi un codice che faccia le derivate e tutto il resto quindi ci serve sapere come fare una cosa chiamata calcolo simbolico. Avevamo visto nella sezione per gli zeri di una funzione la libreria sympy, e qui la riutilizzeremo. Vediamo come potrebbe dunque essere il codice:

```
1 """
2 Code to calculate errors on a generic function
3 """
4 import numpy as np
5 import sympy as sp
6
7 def compute_error(f, variables, data, error):
8     """
9         Function to propagate the error of an arbitay function f.
10        For example if f is f(x, y, z) the error, df, will be:
11
12        df = sqrt( (df(x,y,z)/dx * dx)**2 + (df(x,y,z)/dy * dy)**2 + (df(x,y,z)/dz * dz)**2 )
13
14    Parameters
15    -----
16    f : sympy.core.(something)
17        function, written with sympy symbols,
18        of which to compute the error
19    variables : list
20        list of variables of f, each element
21        mu be a sympy.core.symbol.Symbol
22    data : list
23        list of the numerical value of the variables
24    error : list
25        list of the error on data
26
27    Returns
28    -----
29    central_value : float
30        value of f comuted on data
31    error : float
32        error on central value
33
34    Example
35    -----
36    >>> x = sp.Symbol('x')
37    >>> y = sp.Symbol('y')
38    >>> X = [x, y]
39    >>> f = ((1/x) + (1/y))**(-1)
40    >>> compute_error(f, X, [1.3, 2], [0.1, 0.1])
41    >>> (0.7499999999999999, 0.044393977016173036)
42    ''
43
44    # Compute the central value i.e. f(data)
45    # We use func beacuse we sobsistute on variable at time
46    func = f
47    for var, val in zip(variables, data):
48        func = func.subs(var, val)
49
50    central_value = float(func)
51
52    # Compute the error on f
53    # First of all we compute al derivatives of f respect each variable
54    d_func = []
55    for var in variables:
56        d_func.append(sp.diff(f, var))
57
58    # Secondly we compute the numerical value of all the derivatives
59    d_func_v = []
60    for deriv in d_func:
61        for var, val in zip(variables, data):
62            deriv = deriv.subs(var, val)
63        d_func_v.append(deriv)
```

```

64      # Finally we compute the error adding the squares of derivative and error
65      d_func_v_2 = [(df*err)**2 for df, err in zip(d_func_v, error)]
66      error     = np.sqrt(float(sum(d_func_v_2)))
67
68      return central_value, error

```

Per completezza presentiamo anche una libreria che fa tutto questo al posto nostro. Libreria che useremo fra poco per testare il codice appena scritto.

## L.2 Uncertainties

Ecco un semplice esempio di questa bellissima libreria:

```

1 from uncertainties import ufloat
2 import uncertainties.umath as um
3
4 #il pimo 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

```

## L.3 Unit test

Un buon modo per fare dei test è usare i test unitari o unit test appunto. Ovvero dei test che ci permettono di verificare singole parti del codice che sono autonome, come delle funzioni magari. Ciò facilita il debug perché magari in un codice composto da tante funzioni che si incrociano, mischiano, ballano la macarena e tanto altro, essendo ognuna testata singolarmente possiamo risalire più facilmente ad eventuali problemi vedendo subito quale funzione non passa i test. Vediamo come scrivere un test per la nostra funzione di propagazione:

```

1 import unittest
2 import sympy as sp
3 from uncertainties import ufloat
4 import uncertainties.umath as uu
5
6 from errori import compute_error
7
8 class TESTS(unittest.TestCase):
9
10     def test_0(self):
11         ''' Little test
12         '''
13         x = sp.Symbol('x')
14         y = sp.Symbol('y')
15
16         X = [x, y]
17         f = x + y
18
19         data = [1.2, 2]
20         error = [0.1, 0.15]
21
22         prop_err = compute_error(f, X, data, error)
23
24         x = ufloat(data[0], error[0])
25         y = ufloat(data[1], error[1])
26
27         f = x + y
28
29         print("prop_code:", prop_err)
30         print("prop_lib : ", f)
31
32         self.assertAlmostEqual(prop_err[0], f.n, 10)
33         self.assertAlmostEqual(prop_err[1], f.s, 10)
34
35     def test_1(self):

```

```

36     ''' Litte test
37     '''
38     x = sp.Symbol('x')
39     y = sp.Symbol('y')
40
41     X = [x, y]
42     f = ((1/x) + (1/y))**(-1)
43
44     data = [1.2, 2]
45     error = [0.1, 0.15]
46
47     prop_err = compute_error(f, X, data, error)
48
49     x = ufloat(data[0], error[0])
50     y = ufloat(data[1], error[1])
51
52     f = ((1/x) + (1/y))**(-1)
53
54     print("prop_code:", prop_err)
55     print("prop_lib :", f)
56
57     self.assertAlmostEqual(prop_err[0], f.n, 10)
58     self.assertAlmostEqual(prop_err[1], f.s, 10)
59
60 def test_2(self):
61     ''' Litte test
62     '''
63     x = sp.Symbol('x')
64     y = sp.Symbol('y')
65     z = sp.Symbol('z')
66
67     X = [x, y, z]
68     f = sp.sin(x*z)*sp.log(y/z) + sp.tan(x**y)
69
70     data = [1.2, 2, 1.5]
71     error = [0.1, 0.15, 0.2]
72
73     prop_err = compute_error(f, X, data, error)
74
75     x = ufloat(data[0], error[0])
76     y = ufloat(data[1], error[1])
77     z = ufloat(data[2], error[2])
78
79     f = uu.sin(x*z)*uu.log(y/z) + uu.tan(x**y)
80
81     print("prop_code:", prop_err)
82     print("prop_lib :", f)
83
84     self.assertAlmostEqual(prop_err[0], f.n, 10)
85     self.assertAlmostEqual(prop_err[1], f.s, 10)
86
87
88 if __name__ == '__main__':
89     unittest.main()
90
91 [Output]
92 prop_code: (3.2, 0.18027756377319948)
93 prop_lib : 3.20+/-0.18
94 .prop_code: (0.7499999999999999, 0.044393977016173036)
95 prop_lib : 0.75+/-0.04
96 .prop_code: (7.881984566729017, 14.300828247837693)
97 prop_lib : 8+/-14
98
99 .
100 -----
101 Ran 3 tests in 0.126s
102
103 OK

```

Immagino che non vi sia difficile capire che la sintassi "f.n" ed "f.s" stia rispettivamente per valore centrale ed errore, mentre "assertAlmostEqual(x, y, 10)" confronti i valori di x e y fino alla decima cifra decimale. Vediamo che a riga 8 quando definiamo la classe lo facciamo in maniera diversa da quanto avevamo fatto sopra. Scriviamo infatti "class TESTS(unittest.TestCase)": stiamo fondamentalmente passando una classe ad un'altra classe. Ciò è chiamata ereditarietà, divertitevi a scoprire cos'è.

Vediamo ora un riassunto della propagazione degli errori:

PRECISE NUMBER + PRECISE NUMBER = SLIGHTLY LESS PRECISE NUMBER

PRECISE NUMBER × PRECISE NUMBER = SLIGHTLY LESS PRECISE NUMBER

PRECISE NUMBER + GARBAGE = GARBAGE

PRECISE NUMBER × GARBAGE = GARBAGE

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

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

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

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

GARBAGE - GARBAGE = MUCH WORSE GARBAGE

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

GARBAGE × 0 = PRECISE NUMBER

# M 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).

## M.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(el)/(M-1) for el 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.

## M.2 Calcolo di Pi greco

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

```

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

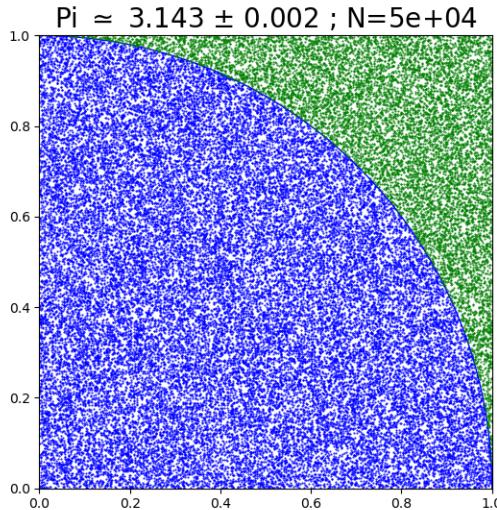
```

```

50 plt.errorbar(X_out, Y_out, fmt='.', markersize=1, color='green')
51 ax = plt.gca()
52 ax.set_aspect('equal')
53 plt.show()
54
55 print("--- %s seconds ---" % (time.time() - start_time))
56
57 [Output]
58 3.142720 +- 0.001835
59 3.141592653589793
60 0.0003588455075227156
61 --- 0.19469761848449707 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.



### M.3 Radice di N

Facciamo un piccolo codice per verificare quanto detto sopra, ovvero che l'errore non dipenda dalla dimensione, ma sia sempre  $1/\sqrt{N}$ . Supponiamo di voler integrare un paraboloido in dimensioni diverse nell'ipercubo unitario detto brevemente vogliamo calcolare i seguenti integrali:

$$\int_0^1 x^2 dx \quad \int_0^1 \int_0^1 x^2 + y^2 dxdy \quad \int_0^1 \int_0^1 \int_0^1 x^2 + y^2 + z^2 dxdydz \quad \text{ecc.} \quad (119)$$

Avevamo visto prima i metodi di quadratura per calcolare gli integrali, qui per semplicità faremo il confronto il metodo dei rettangoli:

$$\int_a^b f(x) dx \approx \sum_{i=0}^{N-1} f(x_i) \cdot \Delta x, \quad (120)$$

Dove chiaramente le  $x_i$  appartendo al nostro intervallo  $[a, b]$  e  $\Delta x = (b - a)/N$ . Immagino non vi sconvolgerà sapere che in caso di dimensione arbitraria  $D$ , volendo fare un'integrazione con un numero totale di punti  $N$ , avremo:

$$\int_{a_1}^{b_1} \cdots \int_{a_D}^{b_D} f(x_1, \dots, x_D) dx_1 \cdots dx_D \approx \sum_{i_1=0}^{N^{1/D}-1} \cdots \sum_{i_D=0}^{N^{1/D}-1} f(x_{i_1}, \dots, x_{i_D}) \cdot \Delta V, \quad (121)$$

Dove  $\Delta V = \Delta x_1 \cdot \Delta x_2 \cdots \Delta x_D$  e ciscun  $\Delta x_i = 1/N^{1/D}$  per cui  $\Delta V = 1/N$ . Vedete che quindi, avendo lo stesso errore su ogni dimensione abbiamo che l'errore totale di integrazione andrà come  $D/N^{1/D}$ . Per quanto riguarda il metodo montecarlo, invece il procedimento è analogo a quanto mostrato prima nel caso del calcolo di  $\pi$ . In genere con metodi montecarlo si vuole calcolare integrali del tipo:

$$\langle f \rangle = \int dx p(x) f(x), \quad (122)$$

ovvero valori medi di una certa funzione  $f$ , che è funzione di variabili stocastiche che sono distribuite secondo la distribuzione  $p(x)$ . Vediamo che se prendiamo come distribuzione una uniforme, otteniamo proprio l'integrale

di  $f$ . Quindi quello che facciamo è estrarre  $N$  variabili casuali dalla nostra distribuzione e calcoliamo la funzione in queste. La media di tutti i valori ottenuti sarà il nostro integrale:

$$\bar{f} = \frac{1}{N} \sum_{i=1}^N f(x_i). \quad (123)$$

Ora la cosa bella è che il termine generale della somma  $f(x_i)$ , da ora chiamato  $f_i$ , è a sua volta una variabile casuale, così come lo è  $\bar{f}$ ; ma essendo quest'ultima una somma di variabili stocastiche, sappiamo che per grandi  $N$  essa sarà distribuita come una gaussiana. Un ringraziamento speciale al teorema centrale del limite. Stiamo supponendo in verità che le  $f_i$  abbiano varianza definita, ma in genere è una supposizione più che ragionevole. Avremo perciò:

$$P(\bar{f}) = \frac{1}{\sqrt{2\pi\bar{\sigma}^2}} e^{-\frac{\bar{f}-\langle f \rangle}{2\pi\bar{\sigma}^2}}, \quad (124)$$

dove  $\bar{\sigma}^2 = \sigma^2/N = (\langle f^2 \rangle - \langle f \rangle^2)/N$ . Puntualizziamo che i valori fra parentesi angolate sono i "valori veri" che noi non possiamo conoscere, ma di cui ne possiamo ottenere solo una stima. Per cui alla fine l'errore sul nostro valor medio sarà:

$$\sigma_f = \sqrt{\frac{1}{N(N-1)} \sum_{i=1}^N (f_i - \bar{f})^2}. \quad (125)$$

Vediamo dunque il codice:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 np.random.seed(69420)
5
6 def f(coords):
7     return np.sum(coords**2, axis=1)
8
9 def mc_integration(f, D, N):
10    """
11        Function for a montecarlo integration
12
13    Parameters
14    -----
15    f : callable
16        function to integrate
17    D : int
18        number of dimensions
19    N : int
20        total number of points
21
22    Return
23    -----
24    integral : float
25        integral of f
26    error : float
27        error over mean value of f
28    """
29
30    # N random points in D dimensions
31    points    = np.random.rand(N, D)
32    values    = f(points)
33    integral  = np.mean(values)
34    error     = np.sqrt(np.sum((values - integral)**2)/(N*(N-1)))
35
36    return integral, error
37
38 def rectangular_integration(f, D, N):
39    """
40        Function for rectangular integration
41
42    Parameters
43    -----
44    f : callable
45        function to integrate
46    D : int
47        number of dimensions
48    N : int
49        total number of points
50

```

```

51     Return
52     -----
53     integral : float
54         integral of f
55     ,
56
57     points_per_dim = np.linspace(0, 1, N)
58
59     grid    = np.meshgrid(*[points_per_dim]*D)
60     coords  = np.stack(grid, axis=-1).reshape(-1, D)
61
62     integral = np.mean(f(coords))
63
64     return integral
65
66
67 dimensions = np.array([i for i in range(1, 8 + 1)])
68 N = int(2.5e7)
69
70 errors_rectangular = []
71 errors_monte_carlo = []
72 diff_mc             = []
73
74 for D in dimensions:
75     exact_value = D/3
76
77     # rectangular
78     rect_integral = rectangular_integration(f, D, int(N**((1/D))))
79     errors_rectangular.append(abs(exact_value - rect_integral))
80
81     # Montecarlo
82     mc_integral, e = mc_integration(f, D, N)
83     errors_monte_carlo.append(e)
84     diff_mc.append(abs(exact_value - mc_integral))
85
86
87 plt.figure(figsize=(10, 6))
88 plt.plot(dimensions, errors_rectangular, label='rect', marker='.')
89 plt.plot(dimensions, diff_mc, label="MC diff", marker='.')
90 plt.plot(dimensions, errors_monte_carlo, label='MC error', marker='.')
91 plt.plot(dimensions, np.ones(len(dimensions))/np.sqrt(N), 'b--', label=r"$1/\sqrt{N}$")
92 plt.plot(dimensions, 0.2*dimensions/(N**((1/dimensions))), 'k--', label=r'$0.2*D/N^{1/D}$')
93 plt.xlabel('Dimensions (D)')
94 plt.ylabel('|analytical - numerical|')
95 plt.title('Error vs Dimension')
96 plt.yscale('log')
97 plt.legend()
98 plt.grid()
99 plt.show()

```

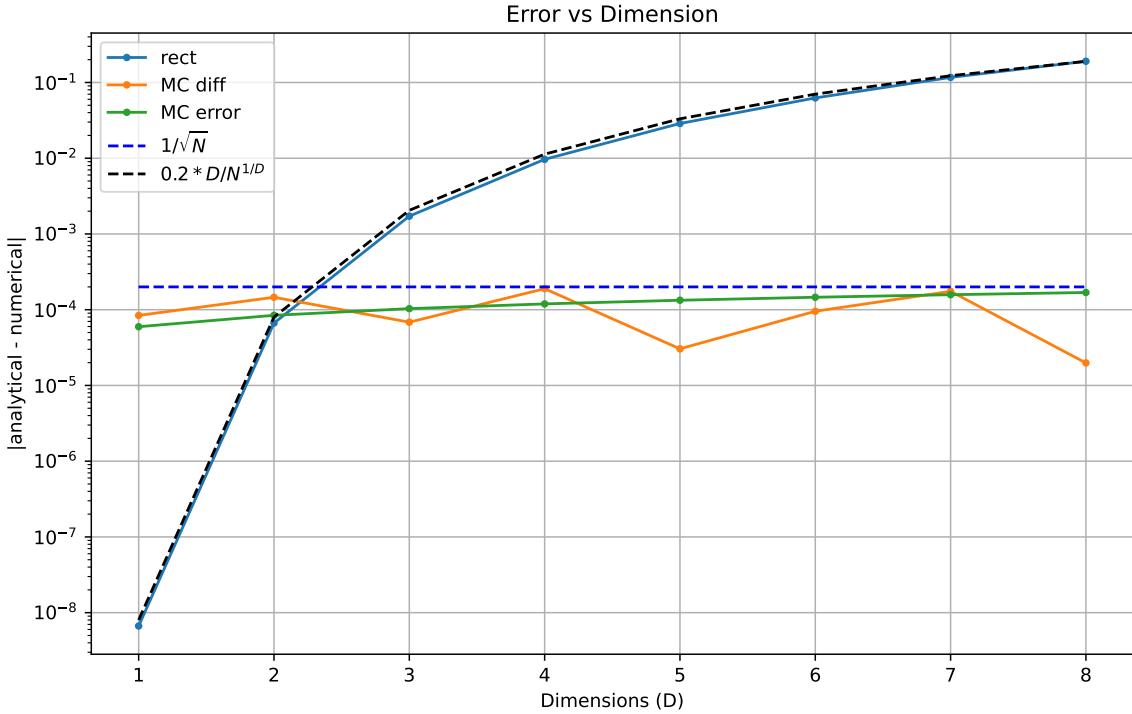


Figura 15: Errore al variare della dimensione, con numero di punti fisso. ”MC error” è l’errore calcolato come spiegato sopra mentre ”MC diff” è la differenza, in valore assoluto, fra valore numerico e valore analitico. Il fattore moltiplicativo 0.2, per il metodo dei rettangoli, è puramente empirico e circostanziale.

Vediamo come l’errore del metodo dei rettangoli cambia al crescere della dimensione, in dimensione 1 è davvero un buon risultato, ma peggiora abbastanza velocemente; l’errore del metodo montecarlo invece varia poco con la dimensione ed è abbastanza parente di  $1/\sqrt{N}$ .

#### M.4 Importance sampling

Attualmente abbiamo visto solo generatori di numeri casuali distribuiti in maniera uniforme in un certo intervallo. Però immagino sappiate che esistono altre distribuzioni, quindi la domanda è: come possiamo estrarre variabili stocastiche che siano distribuite secondo altre distribuzioni? Se abbiamo quindi delle variabili casuali  $x_i$  distribuite secondo una certa distribuzione  $p(x)$  (uniforme nella fattispecie), e abbiamo una certa funzione  $y(x)$  che deve essere invertibile, avremo che ciò che estraiamo fra  $x$  e  $x + dx$  coincideranno con quelle fra  $y(x)$  e  $y(x + dx)$ , per iniettività. Per cui:

$$\bar{p}(y)dy = p(x)dx. \quad (126)$$

Quindi se  $p(x)$  è la nostra distribuzione uniforme, per ottenere variabili in  $y$ , dobbiamo fare un cambio di variabili integrando l’equazione precedente e poi invertire la funzione trovata; cosa che analiticamente non si può fare sempre, ma numericamente sì. Procediamo prima analiticamente. Prendiamo per esempio  $y(x) = \exp(-x)$ ; come intervallo prendiamo  $[0, +\infty)$  così non abbiamo problemi di normalizzazione. Abbiamo quindi:

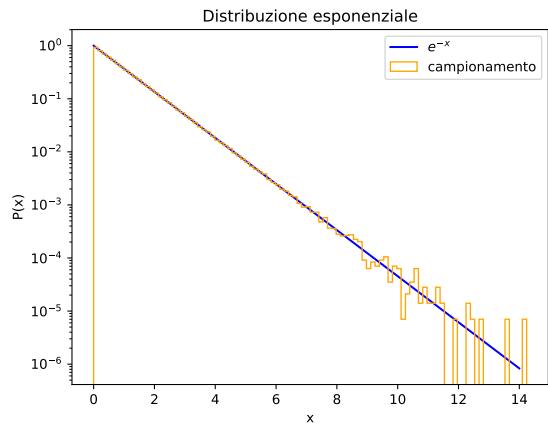
$$\begin{aligned} \int_0^y \bar{p}(t)dt &= \int_0^x p(t)dt, \\ \int_0^y \exp(-t)dt &= \int_0^x dt = x, \\ 1 - \exp(-y) &= x, \\ y &= -\ln(1 - x). \end{aligned} \quad (127)$$

Quindi se prendiamo delle variabili casuali fra zero e uno e ne calcoliamo  $-\ln(1 - x)$  e facciamo l’istogramma di queste nuove variabili sarà facile vedere come queste siano distribuite come l’esponenziale. Quello che abbiamo fatto in paroloni è calcolare la funzione di distribuzione cumulativa (CDF) e poi invertirla calcolando la funzione quantile. Quindi la funzione quantile calcolata in variabili stocastiche uniforme ci da variabili distribuite secondo la distribuzione con cui la quantile è stata catturata. Vediamo intanto un semplice codice per il caso esponenziale.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 np.random.seed(69420)
4
5 p = lambda x : np.exp(-x)
6 q = lambda x : -np.log(1-x)
7 N = int(1e6)
8 x_i = np.random.rand(N)
9 y_i = q(x_i)
10 x = np.linspace(0, 14, 1000)
11
12 plt.title("Distribuzione esponenziale")
13 plt.ylabel("P(x)", fontsize=10)
14 plt.xlabel("x", fontsize=10)
15 plt.plot(x, p(x), 'b-', label=r'$e^{-x}$')
16 plt.hist(y_i, bins=100, histtype='step',
17           density=True, color='orange', label=
18           'campionamento')
19 plt.legend()
20 plt.show()

```



Fin qui tutto moto bello. Però come dicevamo non si può sempre fare tutto analiticamente. Le vie possiamo dire sono due: rifare numericamente quanto abbiamo appena fatto, oppure affidarci ad altri algoritmi. Procediamo per la prima stra nel frattempo, la seconda la vedremo poi. Consideriamo una distribuzione che non sappiamo integrare analiticamente:

$$y(x) = x^x. \quad (128)$$

Si non ho molta fantasia ma va bene (e la gaussiana in realtà si integra mettendoci in 2 dimensioni). Vediamo quindi il codice:

```

1 import numpy as np
2 import scipy.integrate as si
3 import matplotlib.pyplot as plt
4 from scipy.interpolate import InterpolatedUnivariateSpline
5
6 # Distribution to be sampled, as if they were experimental data
7 x = np.linspace(0, 1.5, 50000)
8 y = x**x
9
10 # Compute the integral to normalize it, the normalization will
11 # ensure that we can always use the uniform generator between 0 and 1
12 Norm = si.simpson(y, x=x)
13 y = y/Norm
14
15 # Interpolations
16 s3 = InterpolatedUnivariateSpline(x, y, k=3)
17
18 # Compute cumulative ditribution function
19 cdf = np.array([s3.integral(x[0], i) for i in x])
20
21 # Remove possible equal values, otherwise the subsequent interpolation would not work
22 xq, iq = np.unique(cdf, return_index=True)
23
24 # Swap x with y to invert the cumulative function and get the quantile function
25 yq = x[iq]
26 quantile = InterpolatedUnivariateSpline(xq, yq, k=3)
27
28 Y=quantile(np.random.uniform(size=int(1e6)))
29
30
31 plt.figure(1)
32 plt.grid()
33 plt.title('Sampling with quantile function')
34 plt.xlabel('x', fontsize=10)
35 plt.ylabel('p(x)', fontsize=10)
36 plt.plot(x, y, '.', label='original data')
37 plt.plot(x, s3(x), 'k', label='spline')
38 plt.hist(Y, 100, histtype='step', density=True, label='sampling')
39 plt.legend(loc='best')
40 plt.show()

```

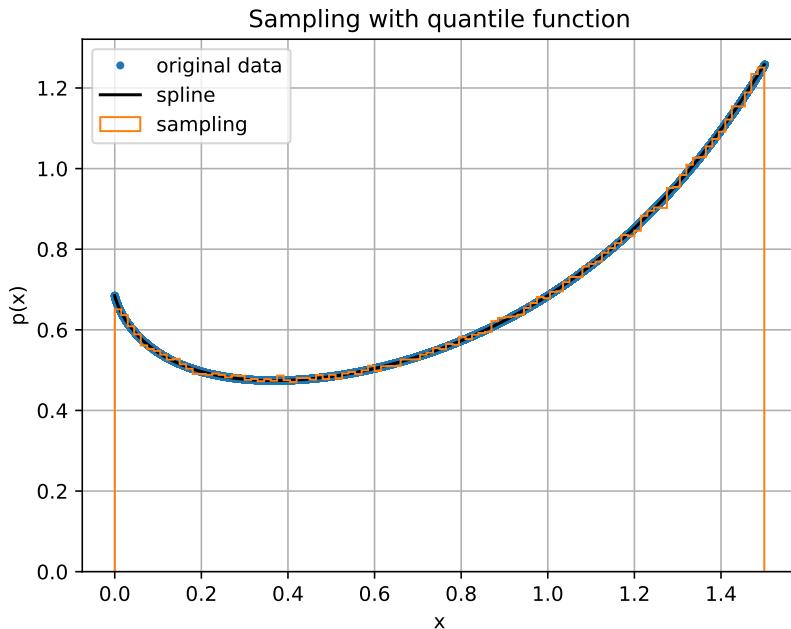


Figura 16: campionamento di una distribuzione usando la funzione quantile

Il vantaggio di questa implementazione è che anche se non sappiamo fare l'integrale lo possiamo fare numericamente, e per invertire una funzione numericamente basta scambiare di posto i due array che corrispondono alle  $x$  e alle  $y$ . Vi basti provare a plottare `"plt.plot(x, y)"` e `"plt.plot(y, x)"`; nel primo caso vedrete la funzione, nel secondo la sua inversa.

#### M.4.1 Importance vs Simple

Ora immagino che giustamente vi starete chiedendo il perché di tutto questo. Sempre per il calcolo degli integrali. Infatti esiste una tecnica, chiamata del ripesamento, che si basa sull'importance sampling e ci permette di ottenere vantaggi rispetto al simple sampling. Dovevamo calcolare:

$$\langle f \rangle_p = \int dx p(x) f(x), \quad (129)$$

che possiamo però riscrivere così:

$$\langle f \rangle_p = \int dx \frac{p(x)f(x)}{g(x)} g(x) = \langle fp/g \rangle_g. \quad (130)$$

Quindi ora non andremo più a estrarre variabili secondo  $p(x)$ , ma secondo  $g(x)$ , ripesando però la funzione integranda per ottenere risultati coerenti. La questione è però delicata, infatti è vero che una scelta oculata di  $g$  ci permette di migliorare le cose rispetto al simple sampling, ma una cattiva scelta le può peggiorare. Vediamo un semplice esempio. Vogliamo calcolare:

$$\int_0^2 e^{-x^2/0.1} \simeq 0.28$$

E usiamo per il simple sampling una distribuzione uniforme fra 0 e 2; mentre per l'importance sampling useremo un esponenziale. Vediamo il codice:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.integrate import quad
4
5 np.random.seed(69420)
6
7 def mc_integration(f, dist, N, **kwargs):
8     """
9         Function for a montecarlo integration
10    
```

```

11 Parameters
12 -----
13 f : callable
14     function to integrate
15 dist : callable
16     distribution from which to
17     extract the random variables
18 N : int
19     total number of points
20
21 Return
22 -----
23 integral : float
24     integral of f
25 error : float
26     error over mean value of f
27
28 Other parameters
29 -----
30 args_f : tuple
31     extra arguments to pass to f
32 args_dist : tuple
33     extra arguments to pass to dist
34 ,,
35 # Default values
36 args_f = kwargs.get("args_f", ())
37 args_dist = kwargs.get("args_dist", ())
38
39 points = dist(N, *args_dist)
40 values = f(points, *args_f)
41 integral = np.mean(values)
42 error = np.sqrt(np.sum((values - integral)**2)/(N*(N-1)))
43
44 return integral, error
45
46 def uniform(N, x_min=0, x_max=1):
47     return np.random.uniform(x_min, x_max, N)
48
49 def exp_quantile(N):
50     x = np.random.uniform(0, 1, N)
51     return -np.log(1 - x)
52
53 def f(x):
54     return np.exp(-x**2/0.1)
55
56 def weighed_f(x):
57     return f(x)/np.exp(-x)
58
59 # Numerical value for comparison
60 I_exact, _ = quad(f, 0, 2)
61
62 # Valutiamo per diversi N
63 N_values = np.logspace(1, 7, 20, dtype=int)
64 errors_simple_cfr = []
65 errors_import_cfr = []
66 errors_simple = []
67 errors_import = []
68
69 for N in N_values:
70     I_simple, e_simple = mc_integration(f, uniform, N, args_dist=(0, 2))
71     I_import, e_import = mc_integration(weighed_f, exp_quantile, N)
72
73     errors_simple_cfr.append(abs(I_simple - I_exact))
74     errors_import_cfr.append(abs(I_import - I_exact))
75     errors_simple.append(e_simple)
76     errors_import.append(e_import)
77
78 # Plot della funzione e delle distribuzioni
79 plt.figure(1)
80 plt.loglog(N_values, errors_simple_cfr, 'o-', label="abs err simple", color='b')
81 plt.loglog(N_values, errors_import_cfr, 's-', label="abs err importance", color='r')
82 plt.loglog(N_values, errors_simple, 'd--', label="mc err simple", color='g')
83 plt.loglog(N_values, errors_import, '---', label="mc err importance", color='c')
84 plt.loglog(N_values, 1/np.sqrt(N_values), 'k:', label=r"$1/\sqrt{N}$")
85 plt.xlabel("$N$", fontsize=15)

```

```

87 plt.ylabel("Error", fontsize=15)
88 plt.legend()
89
90 plt.tight_layout()
91 plt.show()

```

Abbiamo leggermente modificato la funzione usata precedentemente per poterla generalizzare. Vediamo il grafico e commentiamolo:

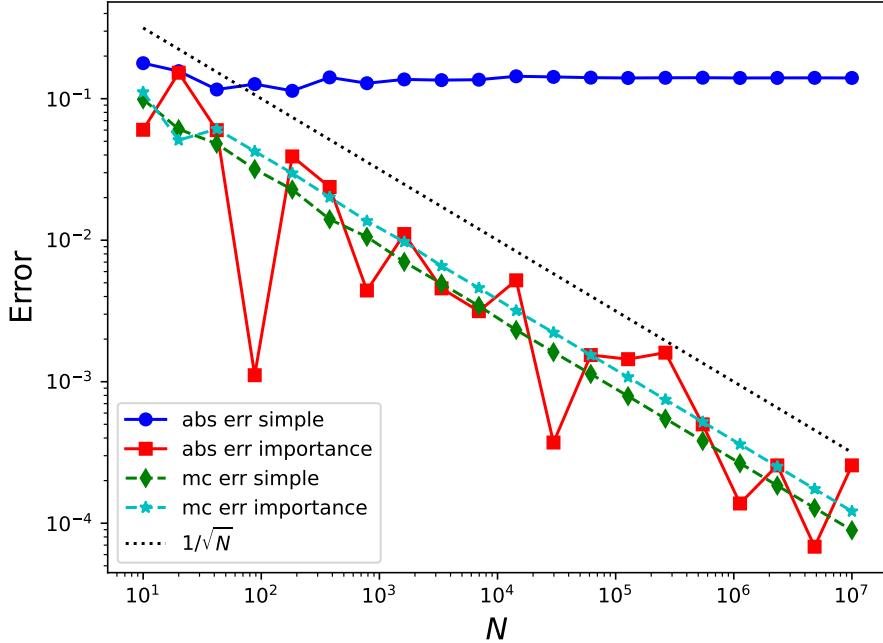


Figura 17: Andamento degli errori per simple e importance samplig.

Allora intanto vediamo cosa è stato plottato. La linea punitata nera è  $1/\sqrt{N}$ , giusto per vedere il confronto. Le due linee il cui nome inizia per "abs" stanno a rappresentare la differenza in valore assoluto fra il valore calcolato con montecarlo e il valore calcolato con "quad". Mentre le due linee il cui nome inizia per "mc" stanno a rappresentare la varianza e cioè l'errore associato al nostro integrale. Vediamo subito quindi che, benché tutti e due gli errori vadano giù come ci aspetteremo, fornendoci un risultato preciso al crescere di  $N$ , per il simple sampling il risultato non è molto accurato nonostante  $N$  cresca, mentre per l'importance sampling abbiamo risultati più accurati. Questo è dovuto al fatto che la maggior parte dell'area della funzione integranda è vicina a zero. Quindi con la distribuzione uniforme fra 0 e 2, molti punti sono estratti in zone che contribuiscono poco all'integrale. Con la distribuzione esponenziale invece, molti punti sono estratti nella zona dove il contributo all'integrale è maggiore, consentendoci quindi una miglior approssimazione del risultato.

## M.5 Metropolis (Hastings)

Vogliamo adesso introdurre un nuovo metodo per campionare una distribuzione. Quello che abbiamo appena visto con la funzione quantile funziona tutto bene fino a che siamo in una dimensione, in dimensioni maggiori immagino che non sia per voi difficile capire che le situazioni peggiorino. Sia per trovare un cambio di variabili analitico sia per dover calcolare integrali e poi interpolazioni di superfici N-dimensional. Possiamo tentare due approcci: l'Accept-Reject o Metropolis-Hastings. Il primo consiste nel prendere una certa distribuzione  $h(x)$  che sappiamo campionare e tale che valga:  $p(x) < ch(x)$ . Dove  $p(x)$  è la distribuzione che vogliamo calcolare e  $c$  è un qualche numero reale positivo. L'idea è: si campiona una variabile  $x_i$  secondo  $ch(x)$  e si calcola  $y_i = ch(x_i)$ . Dopo di che si estrarre uniformemente una variabile stocastica fra 0 e  $y_i$  e se vale  $y_i < p(x_i)$  si conserva la  $x_i$  e si riparte, altrimenti va scartata e si riparte. Se ci fermiamo un attimo a pensare vedere che non sembra troppo diverso da quello che abbiamo fatto per calcolare  $\pi$  solo che questa volta dobbiamo tenere solo i punti dentro il quarto di cerchio. Però penso che a questo punto vi siate già accorti di una cosa: lo sforzo numerico è elevato perché molti punti estratti vanno rigettati, e se la dimensione cresce, o facciamo una scelta infelice per la nostra  $h(x)$ , questo effetto si accentuerà. Quindi per ovviare a questo vogliamo mostrare un algoritmo più efficace, ovviamente non esente da problemi, che è il Metropolis-Hastings. Consideriamo quindi la nostra funzione di distribuzione  $p(x)$  che noi sappiamo calcolare ma di cui non conosciamo la costante di integrazione (sembra poco ma è importante e in genere il calcolo specialmente in alte dimensioni non è banale). Estraiamo da una

distribuzione  $g(x)$  (chiamata distribuzione di proposta, che in italiano è orribile in inglese poposal distributions suono molto meglio) una variabile  $x_0$ . Concedetemi da qui di scrivere la  $g(x)$  come  $g(x|y)$ ; dove questa scrittura sta semplicemente ad indicare che la funzione  $g$  ci restituisce un certo  $x$  dato un certo  $y$ . Sempre per semplicità diciamo che  $g(x|y) = g(y|x)$ . Questo significa che la distribuzione è simmetrica, inoltre così facendo ci stiamo riducendo all'algoritmo Metropolis, è semplicemente un caso particolare del Metropolis-Hastings, ma tutto ciò che andremo a dire varrà anche se la  $g$  non fosse simmetrica. Detto ciò se  $i$  è l'indice che labella le iterazioni del nostro algoritmo qui che va fatto è:

1. Dato un seed iniziale  $x_t$ , generiamo un candidato  $x'$  distribuito secondo la proposal  $g(x'|x_t)$ .
2. Definiamo  $\alpha = p(x')/p(x_t)$ , vedete che grazie a questo rapporto non ci si deve preoccupare della costante di normalizzazione.
3. Se  $\alpha > 1$  allora il candidato iniziale viene accettato e si pone  $x_{t+1} = x'$ ; se così non fosse però, invece che rigettare il candidato lo ci accetta con probabilità  $\alpha$ . Questo significa che se genero un numero uniformemente distribuito fra 0 e 1 e tale numero è minore di  $\alpha$  il candidato è accettato. Altrimenti si rimane da dove siamo partiti e si pone  $x_{t+1} = x_t$ .
4. Si riparte dal punto 1.

Ora vi sarete di certo accorti che in maniera più compatta possiamo dire che possiamo definire, per regolare l'accettanza, la variabile:

$$A = \min \left( 1, \frac{p(x')}{p(x_t)} \right). \quad (131)$$

Quindi questa fondamentalmente è la probabilità di passare da  $x_t$  a  $x'$ . Consentitemi per un attimo di rinominare alcune cose è dire che:

$$A_{i \rightarrow j} = \min \left( 1, \frac{p_j}{p_i} \right). \quad (132)$$

Questa è quindi la probabilità di transire dallo "stato"  $i$  allo "stato"  $j$ . Questo algoritmo tende ad un equilibrio che è proprio quello dato dalla nostra distribuzione. Un modo facile per vederlo è il seguente, rimanendo con i nostri stati  $i$  e  $j$  abbiamo che:

$$\Delta n_i = n_j A_{j \rightarrow i} - n_i A_{i \rightarrow j} \quad (133)$$

Dove il primo termine è la probabilità di essere in  $j$  per la probabilità di transire da  $j$  a  $i$  mentre il secondo è la probabilità di essere in  $i$  per la probabilità di andarsene da  $i$  in  $j$ . Raccogliendo si ha:

$$\Delta n_i = n_j A_{i \rightarrow j} \left( \frac{A_{j \rightarrow i}}{A_{i \rightarrow j}} - \frac{n_i}{n_j} \right). \quad (134)$$

Ora è facile vedere che il rapporto delle  $A$  è sempre uguale a quello della nostra densità di probabilità. Infatti:

- Se  $\frac{p_j}{p_i} < 1$  si ha che  $A_{i \rightarrow j} = \frac{p_j}{p_i}$  mentre  $A_{j \rightarrow i} = 1$ . Per cui  $\frac{A_{j \rightarrow i}}{A_{i \rightarrow j}} = \frac{p_i}{p_j}$ .
- Se  $\frac{p_j}{p_i} > 1$  si ha che  $A_{i \rightarrow j} = 1$  mentre  $A_{j \rightarrow i} = \frac{p_i}{p_j}$ . Per cui  $\frac{A_{j \rightarrow i}}{A_{i \rightarrow j}} = \frac{p_i}{p_j}$ .

Questo cosa vuol dire? Vuol dire che all'equilibrio avremo:

$$\frac{n_i}{n_j} = \frac{p_i}{p_j}. \quad (135)$$

Quindi gli stati saranno stati visitati proporzionalmente alla loro probabilità data dalla nostra distribuzione. Nel caso generale del Metropolis-Hastings è tutto analogo solo che la probabilità è in realtà:  $P_{i \rightarrow j} = g_{i \rightarrow j} A_{i \rightarrow j}$ . Fondamentalmente quanto detto sopra non è altro che il principio del bilancio dettagliato. Principio che una catena di Markov (i.e. la nostra sequenza di  $x_t$  che generiamo) deve soddisfare per far sì che essa tenda alla distribuzione desiderata. Spiegazione molto riduttiva ma diciamo che quanto di importante spero sia passato. Prima di passare al codice fatemi fare una precisazione: nei codici non lavoreremo con le probabilità ma con i loro logaritmi; questo aiuta a evitare problemi di underflow o overflow. Vediamone quindi una breve implementazione. Come distribuzione da campionare prederemo una gaussiana (che in realtà si può campionare con un cambio di variabili analitico).

```

1 import numpy as np
2
3 def target(x, mu=0.0, sigma=1.0):
4     """
5         Target distribution, i.e. the distribution
6         from which we want to sample
7
8     Parameters
9     -----
10    x : float
11        independent variable

```

```

12     mu, sigma : float, optional, default 0, 1
13         Gaussian parameters
14
15     Returns
16     -----
17     log_g : float
18         logarithm of the target distribution
19     ''
20
21     log_g = ((x - mu)/sigma)**2
22
23     return -0.5*log_g
24
25
26 def uniform_proposal(x0, rng, step=0.5):
27     ''
28     Distribution that we know how to sample
29
30     Parameters
31     -----
32     x0 : float
33         point of the chain at 'time' i
34     rng : Generator(PCG64)
35         random number generator from numpy
36     step : float, optional, default 0.5
37         step of the uniform distribution
38
39     Returns
40     -----
41     uniform distribution centered in x0
42     ''
43
44     return x0 + step*rng.uniform(-1, 1)
45
46
47 def metropolis_hastings(target, proposal, rng, n=int(1e3), step=0.5, args=()):
48     ''
49     Metropolis hastings algorithm
50
51     Parameters
52     -----
53     target : callable
54         Distribution to sampling
55     proposal : callable
56         Distribution that we know how to sample
57     rng : Generator(PCG64)
58         random number generator from numpy
59     n : int
60         number of iteration
61     step : float, optional, default 0.5
62         step for proposal distribution
63     args : tuple, optional, default ()
64         extra arguments for target distribution
65
66     Returns
67     -----
68     samples : 1darray
69         array of montecarlo chain
70     ''
71
72     samples = np.zeros(n)
73
74     x0      = rng.uniform(-10, 10)
75     logP0  = target(x0, *args)
76
77
78     for i in range(n):
79
80         x_pr = proposal(x0, rng, step=step)      # Sample from proposal
81         logP = target(x_pr, *args)                # Compute log of target distribution
82         logr = logP - logP0                      # Compute log of the ratio
83
84         if np.log(rng.uniform(0, 1)) < logr:
85             x0 = x_pr
86             logP0 = logP
87

```

```

88         samples[i] = x_pr
89
90     else:
91         samples[i] = x0
92
93     return samples
94
95
96 def autocorrelation(chain):
97     """
98     compute autocorrelation of the chain
99
100    Parameters
101    -----
102    chain : 1darray
103        array of montecarlo chain
104
105    Returns
106    -----
107    auto_corr : 1darray
108        array with auto-correlation of chain
109    time : int
110        auto-correlation time
111    """
112
113 m = np.mean(chain) # Mean
114 s = np.var(chain) # Variance
115
116 xhat = chain - m
117 auto_corr = np.correlate(xhat, xhat, 'full')[len(xhat)-1:]
118
119 auto_corr = auto_corr/s/len(xhat) # Normalization
120
121 # Auto-correlation time
122 time = np.where(abs(auto_corr) < 0.01)[0][0]
123
124 return auto_corr, time

```

Andiamo a vedere un primo grafico della nostra catena in modo che sarà più semplice spiegare l'ultima funzione scritta, che prima non avevamo mai nominato. Vediamo intanto il grafico della distribuzione che non verrà assolutamente messo qui solo per riempire uno spazio.

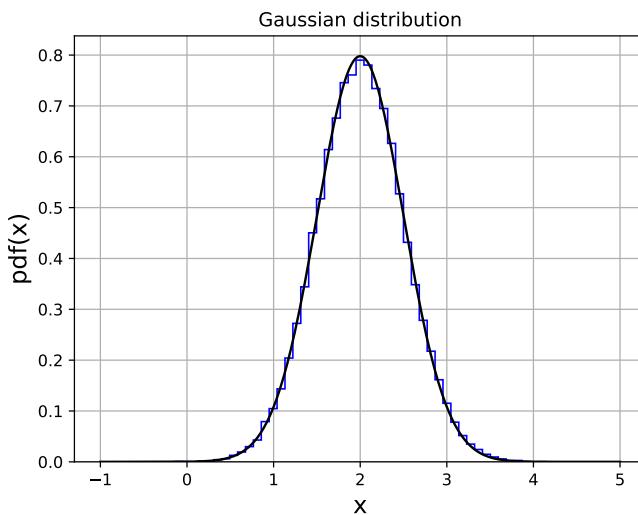


Figura 18: Istogramma della catena generata.

Bellino no? Per i commenti più sul merito è interessante il seguente grafico:

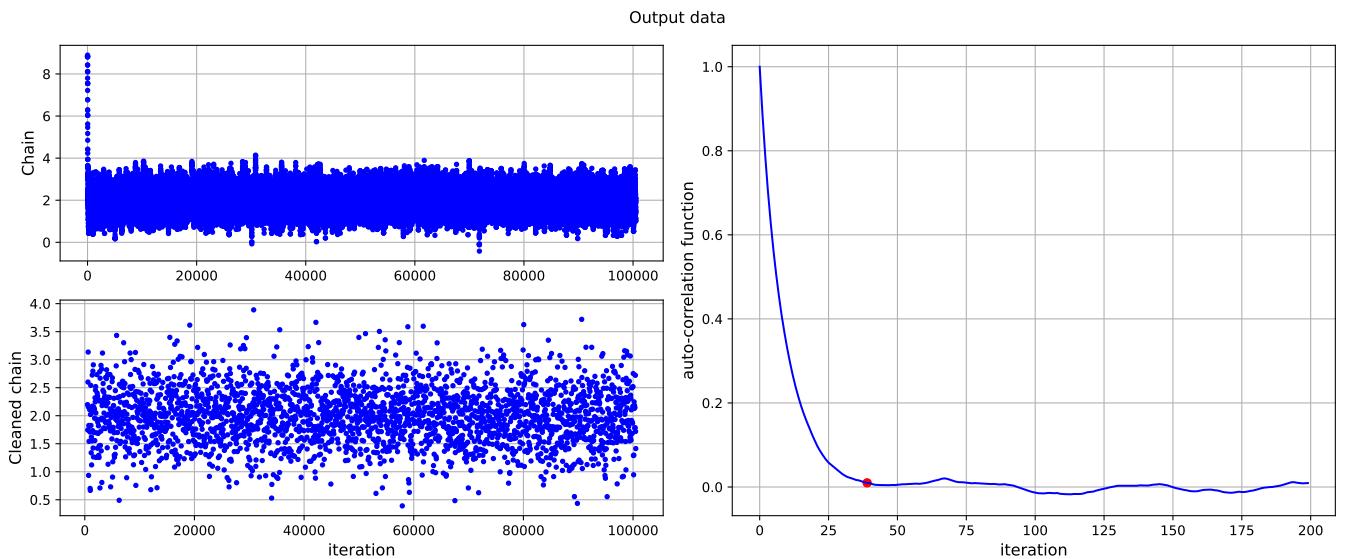


Figura 19: Catena di Markov dell'algoritmo Metropolis, per il campionamento di una gaussiana a media 2 e deviazione standard di 0.5. Sulla sinistra ci sono i grafici della catena prima e dopo la rimozione della termalizzazione e della correlazione. A destra il grafico delle funzione di autocorrelazione della catena. Si è usato uno step di 0.5 e un numero totale di campionamenti di 100500. Le prime 500 sono per il burn-in.

Andiamo con ordine. Vogliamo campionare una gaussiana a media 2 e deviazione standard 0.5. La funzione "metropolis\_hastings" restituisce un array che è quello plottato nel grafico in alto a sinistra. Come prima cosa salta subito all'occhio che la catena parte da poco sopra 8 e poi scende. Quindi ci sono dei primi punti in cui la catena sta termalizzando muovendosi i valori corretti. Questo è un dei principali problemi dell'algoritmo Metropolis, ovvero la prima parte dei dati va sempre rimossa, in quanto la catena non ha ancora raggiunto l'equilibrio. Fatto ciò andiamo a calcolare la funzione di autocorrelazione della nostra catena. Essa è definita come:

$$c(k) = \frac{1}{N - k} \sum_{n=0}^{N-k} (x_n - \bar{x})(x_{n+k} - \bar{x}) \quad (136)$$

Dove  $N$  è il numero di misure, quindi la lunghezza della catena,  $\bar{x}$  è il valor medio della catena e  $x_n$  è l' n-esimo valore estratto. Vi sarà saltato subito all'occhio che non si tratta altro che di un prodotto di convoluzione, quindi un modo intelligente per calcolarlo è fare una fft. A parte questo capiamo a cosa serve. Poiché il nostro algoritmo parte da un certo valore per generarne altri è abbastanza intuitivo che questi valori non saranno indipendenti, saranno correlati appunto. Per ovviare a questo problema quel che si può fare è calcolare questa funzione e trovarne il primo zero; esso è il tempo di correlazione. Nel grafico è indicato con il pallino rosso e nel nostro caso vale 39. Questo quindi ci dice che se della nostra catena prendiamo un punto ogni 39, avremo delle variabili stocastiche che sono indipendenti fra loro. La catena finale è quindi quella del grafico in basso a sinistra, che come vedete è molto meno densa. Per capire perché l'autocorrelazione è un problema proviamo a calcolare il valor medio della nostra catena, che deve corrispondere al valor medio della nostra distribuzione. Per la catena con le correlazioni otteniamo:  $2.006 \pm 0.002$ , mentre per la catena pulita si ottiene  $1.99 \pm 0.01$ . Vediamo quindi un buon miglioramento, passando da tre a una sigma; questo in quanto le correlazioni ci fanno sottostimare l'errore, conducendoci quindi ad un risultato errato. Questa procedura in verità non è sempre necessaria, esistono tecniche che consentono di calcolare l'errore tenendo conto delle correlazioni, ad esempio il datablocking o il binned-bootstrap. Vediamo Per finire l'istogramma della catena.

## N Risolvere numericamente le PDE

Come si diceva sopra sono tantissimi i fenomeni fisici che sono descritti da un'equazione differenziale alle derivate parziali. In genere non esiste una regola aurea o un motivo che funziona sempre; è tutto molto più situazionale, bisogna adeguarsi all'equazione che si vuole risolvere. Vedremo quindi qualche equazione el tratteremo con vari metodi (Trattando sia problemi ai valori iniziali che con condizioni al bordo). In seguito useremo la seguente notazione, gli apici indicano la dipendenza temporale i pedici quella spaziale:

$$u(x_i, t_j) = u_i^j$$

### N.1 Equazione del trasporto

Partiamo con un caso tranquillo, l'equazione del trasporto:

$$\frac{\partial u}{\partial t} + v \frac{\partial u}{\partial x} = 0. \quad (137)$$

Ovviamente ci serve una condizione iniziale  $u(x, t = 0)$  per far evolvere il sistema. Per risolverla si potrebbe pensare di usare, come avevamo visto per le ode, il metodo di Eulero. In fondo avevamo visto che in quei casi, è un metodo che funziona sempre a patto di prendere passi abbastanza piccoli, quindi una griglia con tanti punti. Mettendoci quindi sulla nostra griglia, che ora è 2D, otteniamo:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -v \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}, \quad (138)$$

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. Qui più che metodo di Eulero è più corretto chiamarlo forward-time centered-space (FCTS). Il problema è che però questo metodo non funziona sempre. 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  $\xi$  diverge per ogni scelta di  $\Delta t$  e  $\Delta x$ . Questa si chiama analisi di von Neumann ed è importante notare che benché necessaria non sia sufficiente. Chiamato  $\frac{v\Delta t}{2\Delta x} = \alpha$  si ha:

$$\xi^{n+1} e^{ikj\Delta x} = \xi^n e^{ikj\Delta x} - \alpha(\xi^n e^{ik(j+1)\Delta x} - \xi^n e^{ik(j-1)\Delta x}). \quad (139)$$

Dividiamo ora per  $\xi^n \exp(ikj\Delta x)$  in modo da isolare  $\xi$  e si ha:

$$\begin{aligned} \xi &= (e^{ikj\Delta x} - \alpha(e^{ik(j+1)\Delta x} - e^{ik(j-1)\Delta x}))e^{-ikj\Delta x} \\ &= (1 - \alpha(e^{ik\Delta x} - e^{-ik\Delta x})) \\ &= (1 - 2\alpha i \sin(k\Delta x)), \end{aligned} \quad (140)$$

Essendo un numero complesso ne calcoliamo il modulo e otteniamo:

$$|\xi| = \sqrt{1 + \left(\frac{v\Delta t}{\Delta x}\right)^2 \sin(k\Delta x)^2} > 1. \quad (141)$$

Per cui indipendentemente dalla scelta della spaziatura, questa quantità sarà positiva e quindi avremo una divergenza. Per risolvere tale problema si può usare il metodo di Lax nel quale in termine  $u_j^n$  viene sostituito dalla media dei punti spaziali immediatamente accanto:

$$u_j^{n+1} = \frac{u_{j+1}^n + u_{j-1}^n}{2} - v\Delta t \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}. \quad (142)$$

Facendo lo stesso conto di sopra è facile trovare che si ha:

$$|\xi| = \sqrt{\cos(k\Delta x)^2 + \left(\frac{v\Delta t}{\Delta x}\right)^2 \sin(k\Delta x)^2}. \quad (143)$$

Ora quindi, il metodo è stabile se, mi perdonerete questa redefinizione,  $\alpha = \frac{v\Delta t}{\Delta x} \leq 1$ . È utile graficamente vedere cosa vuol dire questo criterio di stabilità:

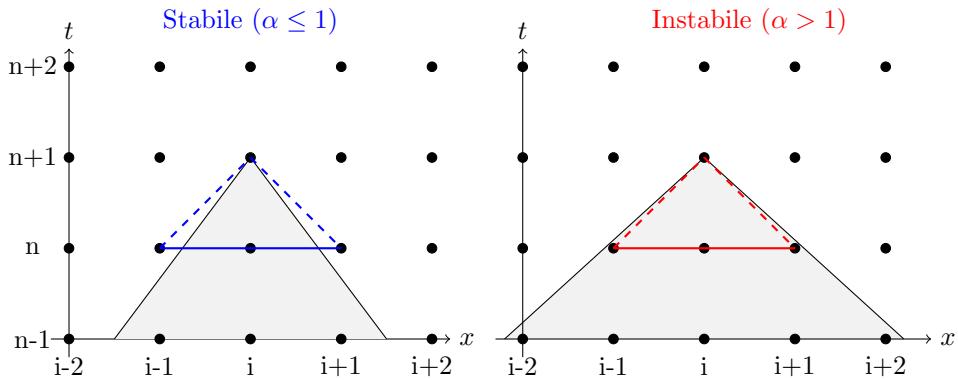


Figura 20: Condizione CFL per l'equazione del trasporto: schema stabile (sinistra) e instabile (destra). Grazie a questo schema dovrebbe essere più semplice intuire il discorso sulla stabilità. Se pensiamo ad un'onda che attraversa la griglia quel che vuol dire è che il passo temporale è minore del tempo che l'onda impiega per andare da un punto al suo vicino. Le aree in grigio rappresentano il domino "fisico" della nostra soluzione, mentre le linee tratteggiate delimitano quello numerico. Se il primo è maggiore del secondo vuol dire che non abbiamo abbastanza informazioni per calcolare correttamente la soluzione al passo successivo, quindi niente stabilità.

Vediamo ora nel concreto il codice:

```

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*v*dx*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
63     line.set_data(x, Sol[:, i])
64     return line,
65
66 anim = animation.FuncAnimation(fig, animate, frames=np.arange(0, T, 1), interval=10, blit=True
67 , repeat=True)
68
69 plt.figure(3)
70 plt.title('Soluzione a vari tempi', fontsize=15)
71 plt.xlabel('distanza')
72 plt.ylabel('ampiezza')
73 plt.grid()
74 col=plt.cm.jet(np.linspace(0, 1, 5))
75 for i, c in zip(np.arange(0, T, T//5), col):
76     plt.plot(x, Sol[:, i], color=c, label=f'time={i*dt:.2f}')
77 plt.legend()
78 plt.show()

```

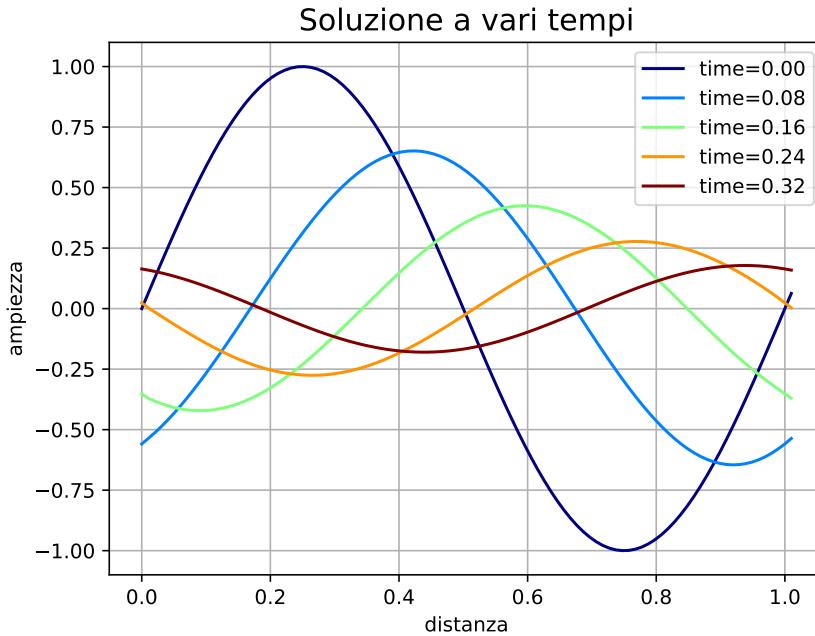


Figura 21: Soluzione dell'equazione del trasporto con il metodo di lax a vari istanti temporali.

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. Infatti sottraendo ambo i membri per  $u_j^n$  e dividendo poi per il passo temporale si ha:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -v \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x} + \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{2\Delta t}. \quad (144)$$

Che è un FCTS di:

$$\frac{\partial u}{\partial t} + v \frac{\partial u}{\partial x} - \frac{\Delta x^2}{2\Delta t} \nabla^2 u = 0. \quad (145)$$

Vi è quindi un problema di diffusione numerica. Possiamo però risolverlo utilizzando un altro metodo, quello di Lax-Wendroff (che però non trattiamo). Tra l'altro io vi ho detto che avendo al tempo  $t$  la soluzione per ogni

valore di  $x$ , usiamo la derivata simmetrica per le derivate spaziali. Abbiamo quindi un metodo del primo ordine nel tempo e del secondo nello spazio. Però perché non del primo ordine? andare ad ordini più elevati è meglio? Eh parliamone. Esiste un metodo chiamato upwind, che usa le derivate al primo ordine nello spazio prendendo la derivata in avanti o indietro a seconda che la velocità sia positiva o negativa. Per la stabilità del metodo vale la stessa condizione che vale per Lax. La cosa interessante è che si può vedere che, come per Lax, esso ha un termine di dissipazione numerica ma dissipava meno del metodo di Lax. Quindi uno schema più accurato non è necessariamente quello da preferire.

## N.2 Equazione del calore

Abbiamo visto che il metodo di Lax è un FCTS per un'equazione con un termine diffusivo. Quindi magari FCTS potrebbe funzionare con equazioni come ad esempio l'equazione del calore. L'equazione del calore è:

$$\frac{\partial u}{\partial t} - D \frac{\partial^2 u}{\partial x^2} = 0 \quad (146)$$

Questa volta, si può vedere che lo schema FCTS è stabile, con condizione di stabilità:  $\frac{D\Delta t}{\Delta x^2} < \frac{1}{2}$ . Anche qui questa condizione può essere interpretata fisicamente dicendo il passo temporale massimo è il tempo di diffusione della soluzione in un intervallo  $\Delta x$ . Il nostro schema sarà quindi:

$$u_j^{n+1} = u_j^n + \frac{D\Delta t}{\Delta x^2} (u_{j+1}^n - 2u_j^n + u_{j-1}^n). \quad (147)$$

Vediamo ora il codice:

```

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

```

Giusto per completezza credo sia interessante far vedere un altro metodo per questa equazione. Ovvvero il metodo di Crank-Nicolson. Questo metodo è del secondo ordine sia nello spazio che nel tempo e si ottiene facendo la media degli schemi FTCS esplicativi e BTCS (i.e. backward-time centered-space) impliciti:

$$u_j^{n+1} = u_j^n + \frac{D\Delta t}{2\Delta x^2} [(u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}) + (u_{j+1}^n - 2u_j^n + u_{j-1}^n)]. \quad (148)$$

Concedetemi di chiamare  $r = \frac{D\Delta t}{2\Delta x^2}$  e con un po' di algebra, portando a destra i termini ad tempo  $n$  e a sinistra  $n+1$  possiamo vedere che arriviamo ad ottenere la seguente equazione:

$$-ru_{j-1}^{n+1} + (1 + 2r)u_j^{n+1} - ru_{j+1}^{n+1} = ru_{j+1}^n + (1 - 2r)u_j^n + ru_{j-1}^n. \quad (149)$$

E immagino non sarà per voi difficili convincervi che quella appena scritta è un sistema lineare. Tra l'altro vediamo che sono solo tre gli indici spaziali coinvolti:  $j-1, j, j+1$ ; questo significa che la nostra matrice è tridiagonale. Questo significa che possiamo tranquillamente risolvere questa equazione come visto sopra con l'eliminazione di gauss. Le poche righe che ci servono sono queste:

```

1 N      = 100          # Numero punti sulle x
2 T_tesps = 5000        # Numero punti nel tempo
3 D      = 0.5          # Coefficiente di diffusione
4 dx    = 1e-2          # Passo spaziale
5 dt    = 1e-4          # Passo Temporale
6 r     = D*dt/(2*dx**2) # Coefficiente
7
8
9 sol_v = np.zeros(N)      # array per le soluzioni
10 sol_n = np.zeros(N)
11 # Array per la creazione della matrice
12 b = np.zeros(N)
13 a = np.ones(N) * (1 + 2 * r)
14 c = np.ones(N) * (-r)
15
16 # Condizioni Iniali
17 L = dx * N
18 t = np.linspace(0, dt*T_tesps, T_tesps)
19 x = np.linspace(0, L, N)
20 sol_v = np.exp(-((x - L / 2.0) / 0.1) ** 2)
21
22 T = np.zeros((N, T_tesps))
23 T[:, 0] = sol_v
24
25 for time in range(1, T_tesps):
26     # Creo il temine noto
27     for i in range(1, N-1):
28         b[i] = r * sol_v[i+1] + (1 - 2 * r) * sol_v[i] + r * sol_v[i-1]
29         b[0] = r * sol_v[1] + (1 - 2 * r) * sol_v[0]
30         b[-1] = (1 - 2 * r) * sol_v[-1] + r * sol_v[-2]
31     # Risolvo il sistema
32     sol_n = solve_tridiagonal(a, c, c, b)
33     # Aggiorno e conservo la soluzione
34     sol_v = sol_n.copy()
35     T[:, time] = sol_v

```

La cosa interessante di questo metodo non è tanto una precisione maggiore rispetto allo schema FTCS quanto la stabilità garantita.

### N.3 Equazione di Burgers

Mettiamo ora insieme termini di trasporto e diffusivi. Cambiamo però metodo. Avevamo infatti visto che con le trasformate di Fourier potevamo calcolare le derivate di una funzione. Questo ci dà un'interessante prospettiva sulle PDE, infatti passando in trasformata, una PDE diventa una più tranquilla ODE. Consideriamo l'equazione:

$$\frac{\partial u(t, x)}{\partial t} + u(t, x) \frac{\partial u(t, x)}{\partial x} = \nu \frac{\partial^2 u(t, x)}{\partial x^2}, \quad (150)$$

nota come equazione di Burgers, un'equazione abbastanza usale in fluidodinamica che ad esempio modelizza la formazione di un fronte d'onda ad esempio gli shock. Non so se nei corsi di fluidodinamica si veda, vi basti sapere che, come quasi tutto, si parte da:

$$\begin{cases} \partial_t \rho + \partial_x(\rho v) = 0 \\ \partial_t(\rho v) + \partial_x(\rho v^2 + P) = \nu \partial_x^2 v \end{cases} \quad (151)$$

Assumendo come equazione di stato una politropica ed espandendo la densità e la velocità in serie di perturbazioni [e.g.  $v = v_0 + v_1 + \dots$ ,  $\rho = \rho_0 + \rho_1 + \dots$ , (in genere  $v_0=0$ )] si ottiene che  $\rho_1$  soddisfa l'equazione di Burgers. È facile vedere che passando in trasformata di nello spazio la nostra equazione diventa un'ode nel tempo:

$$\frac{\partial u(t, k)}{\partial t} + u(t, k)(iku(t, k)) = \nu(-k^2 u(t, k)) \quad . \quad (152)$$

Qui usiamo "scipy.integrate.odeint" per risolvere l'equazione. Vediamo le poche righe di codice necessarie:

```

1 """
2 code to solve burger equation using fourier trasform in space
3 """
4 import numpy as np
5 import matplotlib.pyplot as plt
6 from scipy.integrate import odeint
7 import matplotlib.animation as animation
8
9 #=====
10 # Parameters
11 #=====
12 L = 1
13 T = 0.31
14
15 dx = 0.001
16 dt = 0.001
17 nu = 0.005
18
19 Nx = int(L/dx)
20 Nt = int(T/dt)
21
22 t = np.linspace(0, T, Nt)
23 x = np.linspace(0, L, Nx)
24
25 # wave number
26 k = 2*np.pi*np.fft.fftfreq(Nx, dx)
27 # Initial conditions
28 u0 = np.sin(2*np.pi*x)
29
30 #=====
31 # Solutions
32 #=====
33
34 def eq(u, t, k, nu):
35     """
36         equation to be solved in spatial transform
37     """
38     u_hat = np.fft.fft(u)
39     du_hat = 1j*k*u_hat      # first derivative
40     ddu_hat = -k**2*u_hat   # second derivative
41
42     #antitrasform
43     du = np.fft.ifft(du_hat)
44     ddu = np.fft.ifft(ddu_hat)
45
46     #pde in time and space -> ode in time
47     u_t = -u*du + nu*ddu
48
49     return u_t.real
50
51
52 sol = odeint(eq, u0, t, args=(k, nu,)).T
53
54
55 #=====
56 # Animations
57 #=====
```

```

60 fig = plt.figure(2)
61 plt.xlim(np.min(x), np.max(x))
62 plt.ylim(np.min(sol), np.max(sol))
63
64 line, = plt.plot([], [], 'b')
65 def animate(i):
66     line.set_data(x, sol[:,i])
67     return line,
68
69 anim = animation.FuncAnimation(fig, animate, frames=Nt, interval=5, blit=True, repeat=True)
70
71 plt.grid()
72 plt.title('burger equation')
73 plt.xlabel('Distanza')
74 plt.ylabel('ampiezza')
75
76 #anim.save('buger.mp4', fps=30, extra_args=['-vcodec', 'libx264'])
77
78 plt.show()

```

Eseguendo il codice vedrete l'animazione del fronte d'onda crearsi e grazie al termine diffusivo se ne evita la rottura.

## N.4 Equazione di laplace

Un'altra interessante equazione da trattare è l'equazione di Laplace, della forma:

$$(\partial_x^2 + \partial_y^2)u(x, y) = \rho(x, y) \quad (153)$$

Se ora discretizziamo le derivate seconde mettendoci su un reticolo quadrato in modo che la spaziatura lungo le due coordinate sia la stessa, è facile vedere che otteniamo:

$$-4u_{i,j} + u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} = \rho_{i,j}\Delta x^2 \quad (154)$$

Che ci da quindi un'espressione per trovare  $u_{i,j}$ . Si tratta nuovamente di un sistema di equazioni. Una possibilità è quella di creare una matrice  $N^2 \times N^2$  dove  $N$  è il numero di punti su ciascun assi, e andando quindi a riordinare i nodi della nostra griglia; ci basta cambiare indici dicendo  $k = iN + j$ . Arriveremo quindi ad avere un sistema, con una matrice pentadiagonale, la cui soluzione è direttamente la nostra funzione. Noi qui vogliamo usare invece il metodo SOR, visto prima. Che ci permette di trovare la soluzione più rapidamente. Vediamo allora il codice:

```

1 """
2 Code for the solution of Laplace equation via SOR
3 """
4 import time
5 import numpy as np
6 import matplotlib.pyplot as plt
7
8 def solve_lap_sor(N, bound, rho, w, tau):
9     """
10     Function that use SOR method to solve laplace equation
11
12     Parameters
13     -----
14     N : int
15         size of the grid
16     bound : 2darray
17         boundary conditions
18     rho : 2darray
19         source of field
20     w : float,
21         overrelaxation parameter
22     tau : float
23         required convergence
24
25     Return
26     -----
27     phi : 2darray
28         solution of the equations
29     """
30     # Unuseful quantities
31     conv      = 10.0
32     mean_U0   = 0
33     iter_count = 0

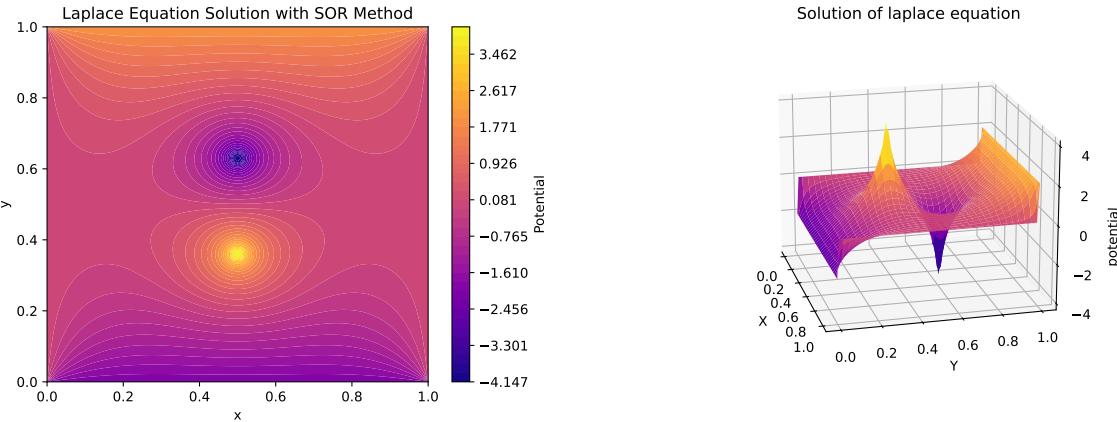
```

```

34
35     # Matrix of the potential
36     phi = np.zeros((N+1, N+1))
37
38     # Boundary conditions
39     phi[:, 0] = bound[0]    # ovest
40     phi[:, N] = bound[1]    # est
41     phi[0, :] = bound[2]    # sud
42     phi[N, :] = bound[3]    # nord
43
44     mean_U0 = np.mean(phi)
45
46     while conv > tau:
47         for i in range(1, N):
48             for j in range(1, N):
49                 force = phi[i, j+1] + phi[i, j-1] + phi[i+1, j] + phi[i-1, j]
50                 force += rho[i, j]
51                 phi[i, j] = w * 0.25 * force + (1 - w) * phi[i, j] #SOR
52
53     mean_U = np.mean(phi)
54     conv = abs(mean_U - mean_U0)
55     mean_U0 = mean_U
56
57     iter_count += 1
58
59 return phi
60
61 if __name__ == "__main__":
62     N = 100
63     w = 1.99
64     tau = 1e-8
65     dx = 1/N
66
67     # Boundary conditions
68     bound = np.zeros((4, N+1))
69     bound[0, :] = 0    # ovest
70     bound[1, :] = 0    # est
71     bound[2, :] = -2   # sud
72     bound[3, :] = 2    # nord
73
74     # Source
75     rho = np.zeros((N+1, N+1))
76     for i in range(-1, 2):
77         for j in range(-1, 2):
78             rho[7*N//11+i, N//2+j] = -1
79             rho[4*N//11+i, N//2+j] = 1
80     rho = rho*dx**2
81
82     # Solution
83     start = time.time()
84     phi = solve_lap_sor(N, bound, rho, w, tau)
85     end = time.time() - start
86
87     print(f"Elapsed time: {end:.2f} s")
88
89     # Plot
90     x = np.linspace(0, 1, N+1)
91     gridx, gridy = np.meshgrid(x, x)
92
93     plt.figure(0)
94     levels = np.linspace(np.min(phi), np.max(phi), 40)
95     c=plt.contourf(gridx, gridy, phi, levels=levels, cmap='plasma')
96     plt.colorbar(c, label='Potential')
97     plt.title('Laplace Equation Solution with SOR Method')
98     plt.xlabel('x')
99     plt.ylabel('y')
100
101    fig = plt.figure(1)
102    ax = fig.add_subplot(1,1,1, projection='3d')
103    ax.plot_surface(gridx, gridy, phi, cmap='plasma')
104    ax.set_title('Solution of laplace equation')
105    ax.set_xlabel('x')
106    ax.set_ylabel('y')
107    ax.set_zlabel('potential')
108
109    plt.show()

```

Il codice di per sé è abbastanza tranquillo, semplicemente usiamo il SOR, piuttosto che su un vettore, su una matrice ma come vedete funziona bene; inoltre la convergenza è settata tramite la media del potenziale, questo perché l'equazione di Laplace descrive uno stato stazionario. Chiaramente a voi la libertà di settare il criterio di convergenza come più vi aggrada.



## N.5 Equazione di Schrödinger

Passiamo adesso ad un'altra equazione molto interessante, ne avevamo già parlato nel caso di diagonalizzazione di matrici, e avevamo trattato il caso stazionario. Ora vogliamo introdurre la dipendenza dal tempo. Abbiamo quindi:

$$i \frac{\partial \psi}{\partial t} = H\psi. \quad (155)$$

Assumiamo intanto che l'hamiltoniana non dipenda dal tempo (cfr. si conserva l'energia). Notiamo subito una cosa: chiaramente questa equazione è una PDE e modulo un'unità immaginaria è praticamente un'equazione diffusiva, tipo quella del calore. Si potrebbe quindi pensare di procedere allo stesso modo; c'è però un caveat: dato il suo significato di densità di probabilità è necessario che la psi sia normalizzata o per meglio dire e che la norma sia conservata durante l'evoluzione quindi il nostro algoritmo deve essere unitario. Sappiamo però per l'assunzione fatta precedentemente che la soluzione è:

$$\psi(t, x) = e^{-iHt}\psi(0, x). \quad (156)$$

Tutto molto tranquillo e tutto molto unitario, no? No chiaramente,  $H$  è una matrice, e il suo esponenziale (in paroloni mi pare si dicesse implementazione unitaria) è definito con la serie di potenze, che non vogliamo star qui a calcolare. Idea: sviluppiamo per piccoli tempi  $e^{-iH\delta t} \simeq 1 - iH\delta t$  e applichiamo tutto ciò alla condizione iniziale  $\psi(0, x)$ . Sorge un nuovo problema: abbiamo perso l'unitarietà, il modulo quadro non si conserva. Quello che si può fare per risolvere è noto come split operator:

$$e^{iH\delta t/2}\psi(t + \delta t, x) = e^{-iH\delta t/2}\psi(t, x), \quad (157)$$

ovvero la funzione d'onda al tempo  $t + \delta t$  evoluta indietro di  $\delta t/2$  è uguale alla funzione d'onda al tempo  $t$  evoluta in avanti di  $\delta t/2$ . Quindi approssimando sempre l'esponenziale al prim'ordine si ha:

$$\psi(t + \delta t, x) = \frac{1 - iH\delta t/2}{1 + iH\delta t/2}\psi(t, x). \quad (158)$$

Mi perdonerete l'abuso di notazione, ovviamente dividere per  $1 + iH\delta t/2$  vuol dire invertire la matrice, ovvero risolvere un sistema; scriviamo così solo per renderci conto del fatto che quel coefficiente è della forma: numero complesso diviso il suo coniugato, e per cui ha modulo uno, garantendo l'unitarietà dell'evoluzione. Questo metodo può essere migliorato proprio grazie alla trasformata di Fourier. Infatti presa  $H = T + V$ , vale:

$$e^{-iH\delta t} = e^{-iV\delta t/2}e^{-iP\delta t}e^{-iV\delta t/2} + \mathcal{O}(\delta t^3). \quad (159)$$

Una cosa simile era stata usata nell'appendice in cui avevamo trattato di sfuggita gli integratori simplettici. Perché facciamo questa divisione? Questi tre termini sono più facili da calcolare? Noi vogliamo calcolare l'operatore di evoluzione temporale ma senza dover esponenziare una matrice. Facendo così vediamo subito che i termini dipendenti da  $V$  si calcolano veloci in quanto se la matrice è diagonale l'espansione in serie mi da gli

esponenziali delle entrare. Quindi vorremo poter diagonalizzare anche  $T$ , quindi l'impulso. Ma per andare dalla base della posizione a quella dei momenti si esegue proprio una trasformata di Fourier. Quindi in quella base  $T$  sarà una matrice diagonale contenente i valori (al quadrato) che l'impulso può assumere (sempre nella nostra scatola di lato  $L$  in quanto abbiamo discretizzato lo spazio). La regola iterativa è dunque:

$$\psi(t + \delta t, x) = e^{-iV\delta t/2} \text{FFT}^{-1}(e^{-iP\delta t} \text{FFT}(e^{-iV\delta t/2} \psi(t, x))). \quad (160)$$

Vediamo ora quindi il codice:

```

1 """
2 Code for the solution of Schrodinger's time dependent equation
3 via split operator method but unlike tunnel_barrier.py using a FFT.
4 Now the idea is to use:
5 exp(1j (T+V) dt) = exp(1j V dt/2) exp(1j T dt) exp(1j V dt/2) + O(dt^3)
6 and to compute exp(1j T dt) we go in momentum space where T is diagonal
7 so it's easy to compute, so we must use FFT to go from x space to p space
8 """
9 import numpy as np
10 import matplotlib.pyplot as plt
11 import matplotlib.animation as animation
12
13 #=====
14 # Initial wave function and potential
15 #=====
16
17 def U(x):
18     ''' harmonic potential
19     '''
20     return 0.5*x**2
21
22 def psi_inc(x, x0, a, k):
23     ''' Initial wave function
24     '''
25
26     A = 1. / np.sqrt( 2 * np.pi * a**2 ) # normalization
27     K1 = np.exp( - ( x - x0 )**2 / ( 2. * a**2 ) )
28     K2 = np.exp( 1j * k * x )
29     # let's multiply by five so the animation is prettier
30     return A * K1 * K2 * 5
31
32 #=====
33 # Computational parameters
34 #=====
35
36 n = 1000                      # Number of points
37 xr = 10                         # Right boundary
38 xl = -xr                        # Left boundary
39 L = xr - xl                     # Size of box
40 x = np.linspace(xl, xr, n)       # Grid on x axis
41 dx = np.diff(x)[0]               # Step size
42 dt = 1e-3                        # Time step
43 T = 10                           # Total time of simulation
44 ts = int(T/dt)                  # Number of steps in time
45
46 # Initialization of gaussian wave packet
47 psi = psi_inc(x, -1.2, 0.5, 0.3)
48 PSI = []
49 PSI.append(abs(psi)**2)
50
51 #=====
52 # Build the propagator in x and k space
53 #=====
54
55 # Every possible value of momentum
56 k = 2*np.pi*np.fft.fftfreq(n, dx)
57 # Propagator
58 U_r = np.exp(-1j * U(x) * dt/2) # Half step in space
59 U_k = np.exp(-1j * k**2/2 * dt) # Full step in momentum
60
61 # Time evolution
62 for _ in range(ts):
63     psi = U_r * psi
64
65     psi_k = np.fft.fft(psi)
66     psi_k = U_k * psi_k
67

```

```

68     psi = np.fft.ifft(psi_k)
69     psi = U_r * psi
70
71     PSI.append(abs(psi)**2)
72
73 #=====
74 # Animation
75 #=====
76
77 fig = plt.figure()
78 plt.title("Gaussian packet propagation")
79 plt.plot(x, U(x), label='$V(x)$', color='black')
80 plt.grid()
81
82 plt.ylim(-0.0, np.max(PSI))
83 plt.xlim(-5, 5)
84
85 line, = plt.plot([], [], 'b', label=r"\psi(x, t)^2")
86
87 def animate(i):
88     line.set_data(x, PSI[i])
89     return line,
90
91 plt.legend(loc='best')
92
93 anim = animation.FuncAnimation(fig, animate, frames=np.arange(0, ts, 10),
94                               interval=1, blit=True, repeat=True)
95
96 #anim.save('ho.mp4', fps=30, extra_args=['-vcodec', 'libx264'])
97
98 plt.show()

```

In questo caso abbiamo nuovamente usato un potenziale armonico e possiamo vedere l'oscillazione della funzione d'onda nel nostro potenziale. Adesso ho due domande per voi: se rendete il pacchetto gaussiano iniziale più stretto la simulazione non viene bene; perché? Provate inoltre a fare l'evoluzione nel tempo immaginario  $it = \beta$  (capisco sia un tempo ma concedetemi di chiamarlo  $\beta$ ); ora perdiamo l'unitarietà quindi ad ogni passo bisogna normalizzare la  $\psi$ . Cosa succede al nostro sistema? come pensate che evolva?

## O 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.

### O.1 Processo di Ornstein–Uhlenbeck

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

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

dove  $\theta, \sigma$  costanti positive e  $\mu$  costante, mentre  $dW$  è un processo di Wiener. In genere data una certa SDE della forma:

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

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

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

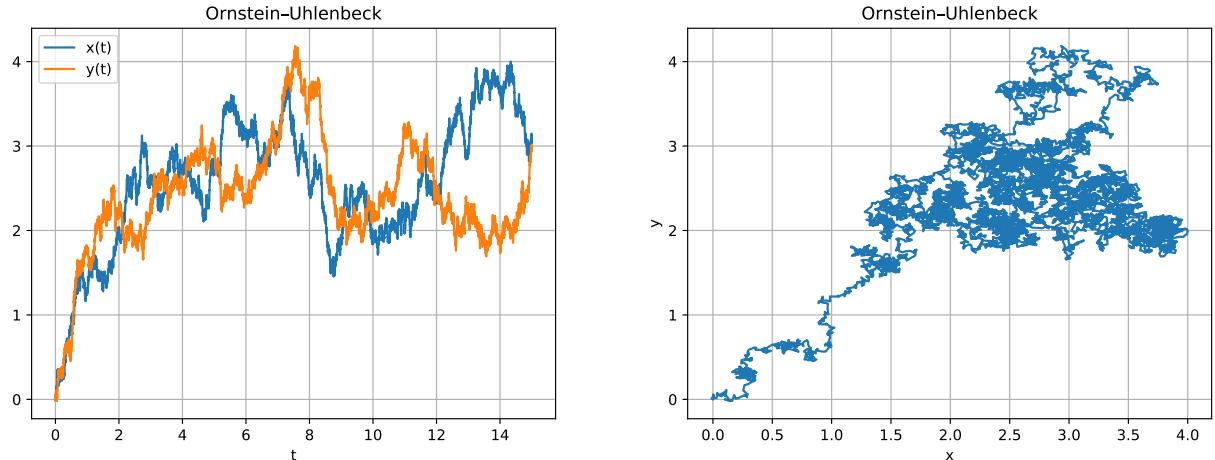
dove  $dt$  ora è il passo di integrazione e  $dW$ , che sarebbe un integrale stocastico, lo trattiamo come una variabile gaussiana di media zero e varianza uguale alla radice del passo di integrazione. Vediamo un semplice esempio:

```
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()

```



## O.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:

```

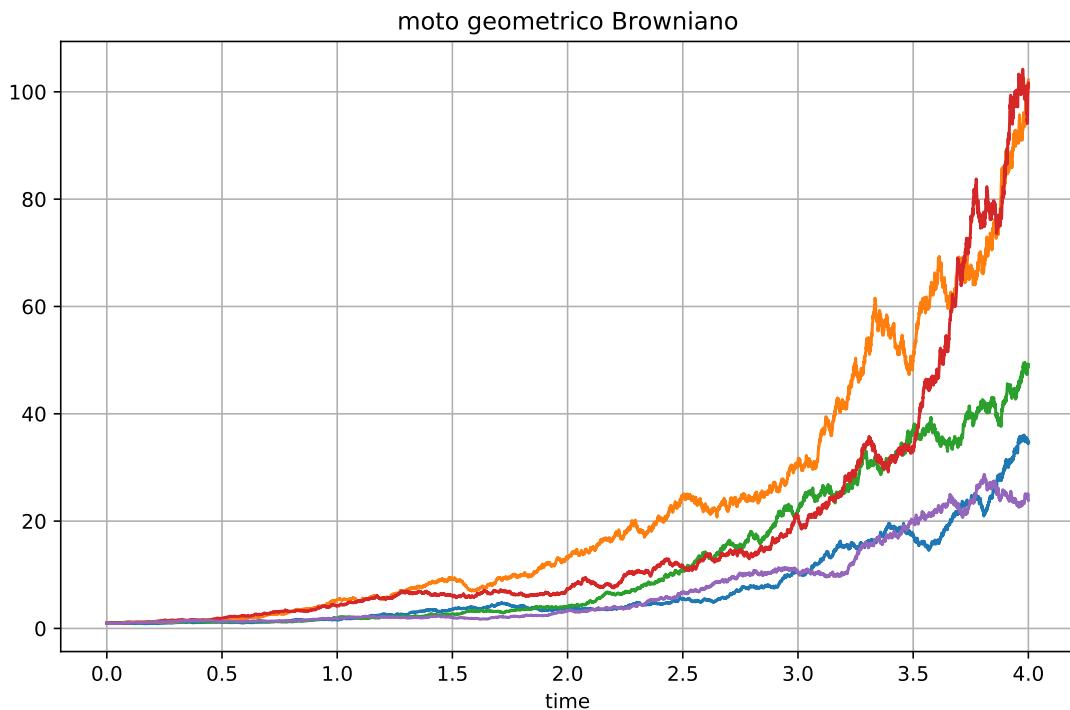
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # geometric Brownian motion
5 # Heun method
6
7 def f(z):
8     """
9         funzione che moltiplica il dt
10    """
11    mu = 1
12    return mu*z
13
14 def g(z):
15     """
16        funzione che moltiplica il processo di wiener
17    """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

```



# P Ottimizzazione

Per quanto la discussione fatta nella quarta lezione con curve.fit sia praticamente di ottimizzazione, vogliamo parlare qui 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.

## P.1 Discesa del gradiente

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

$$x_{n+1} = x_n - \alpha_n \nabla F(x_n) \quad (161)$$

Ci sono vari modi per scegliere  $\alpha_n$  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):           #loop over variables
40             s[i] = 1                  #we select one variable at a time
41             dz1 = x0 + s*h           #step forward
42             dz2 = x0 - s*h           #step backward
43             grad[i] = (f(*dz1) - f(*dz2))/(2*h) #derivative along z's direction
44             s[:] = 0                  #reset to select the other variables
45
46         if all(abs(grad) < tol):
47             break
48
49         x0 = x0 - step*grad      #move towards the minimum
50         X.append(x0)            #store iteration
51         iter += 1               #update counter
52
53     X = np.array(X)
54     return X, iter
```

## P.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 aggiunge un termine di velocità e un termine di attrito dipendente dalla velocità in modo che l'algoritmo non oscilli.

$$w_{n+1} = \beta w_n + \nabla F(x_n) \quad (162)$$

$$x_{n+1} = x_n - \alpha w_{n+1} \quad (163)$$

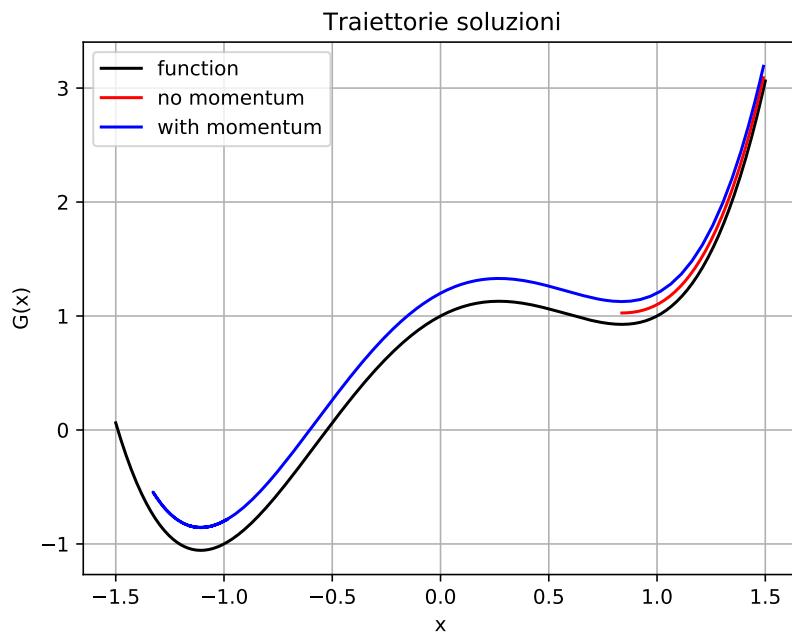
$\alpha$  è sempre lo step mentre  $\beta$  rappresenta pittoricamente il coefficiente di attrito. Se  $\beta = 1$  l'algoritmo oscilla tra due punti con lo stesso valore di  $F$  (si conserva l'energia) quindi bisogna avere  $\beta < 1$ .

```

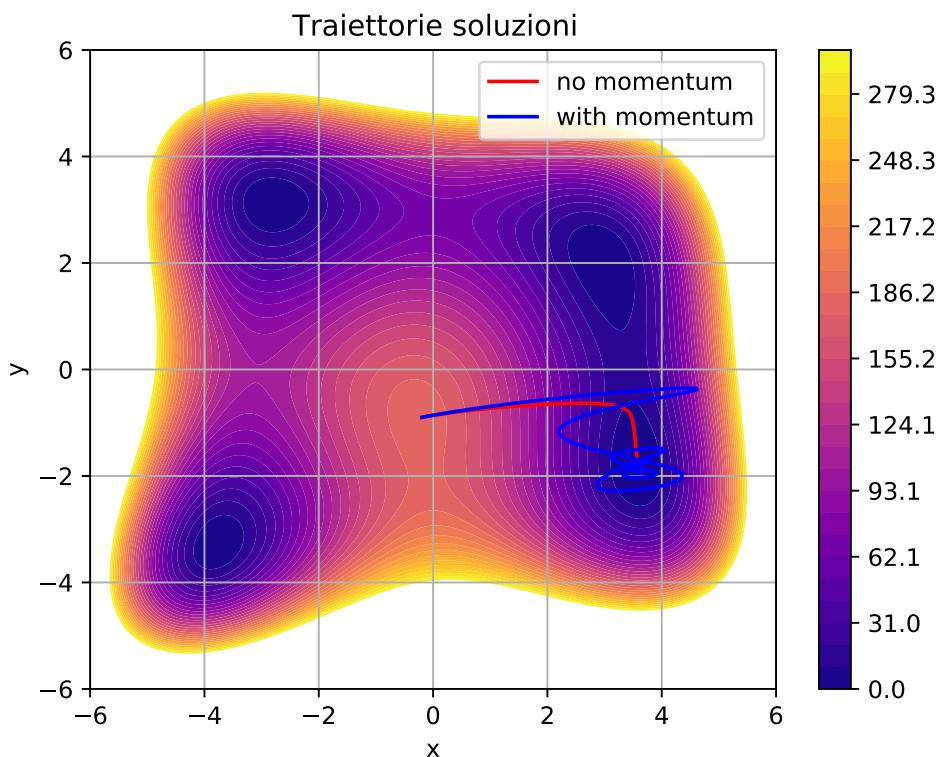
1 def grad_disc_m(f, x0, tol, alpha, beta):
2     """
3         implementation of gradient descent with momentum
4         you have to be careful about the values
5         you pass in x0 if the function has more minima
6         and also the value of alpha an beta is a
7         delicate choice to be 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 : 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     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):
46             s[i] = 1           #loop over variables
47             dz1 = x0 + s*h   #we select one variable at a time
48             dz2 = x0 - s*h   #step forward
49             grad[i] = (f(*dz1) - f(*dz2))/(2*h) #step backward
50             s[:] = 0          #derivative along z's direction
51             #reset to select the other variables
52
53         if all(abs(grad) < tol):
54             break
55
56         w = beta*w + grad  #update velocity
57         x0 = x0 - alpha*w #update position move towards the minimum
58         X.append(x0)       #store iteration
59         iter += 1          #update counter
60
61     X = np.array(X)
62     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 `grad_disc(G, x0, 1e - 8, 1e - 3)` e `grad_disc_m(G, x0, 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 incarta 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  $\beta$  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).  $F(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$ .



# Q 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. Negli ultimi tempi il machine learning è sempre più presente, e le sue applicazioni sono innumerevoli. Quindi ne vogliamo trattare anche qui portando qualche esempio semplice. Useremo inizialmente la libreria 'sklearn'. Vedremo vari argomenti, iniziando piano con semplici esempi di regressione e classificazione; quindi apprendimento supervisionato. Tratteremo poi casi di apprendimento non supervisionato, il clustering, facendo il confronto tra del codice scritto da noi e la libreria "sklearn". Tratteremo poi nella prossima sezione le reti neurali e infine, per dare un tocco di fisica, vedremo le così dette PINN (i.e. physisc informed neural network); e le useremo per risolvere equazioni differenziali. Per quest'ultima cosa ci affideremo invece alla libreria pythorc per costruire la rete neurale.

## Q.1 Classificatore

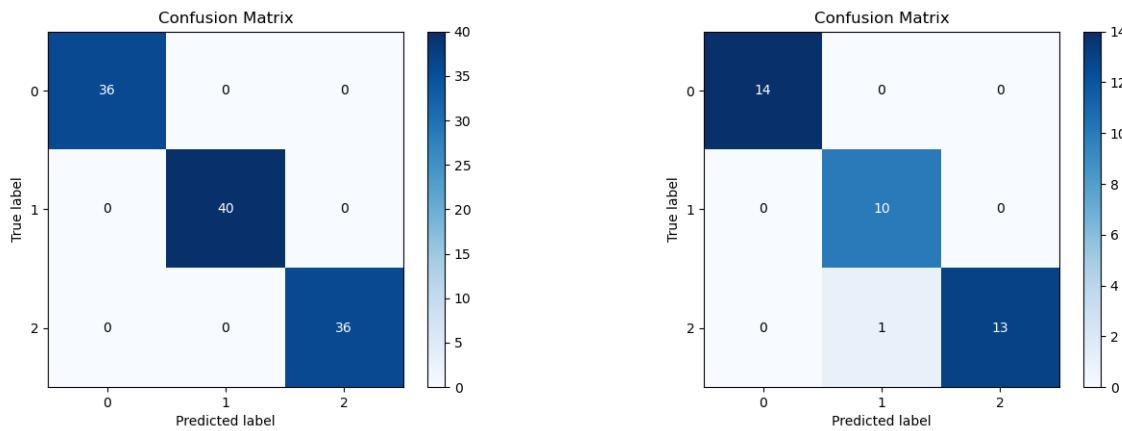
Un algoritmo classificatore fondamentalmente prende dei dati e restituisce una categoria, quindi classifica i dati in input. Vediamo un esempio:

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

```

56 accuratezza test
57 0.9736842105263158

```



La matrice di sinistra ci fa vedere come la predizione sui dati che abbiamo usato per addestrarla sia andata bene, infatti avevamo accuratezza uno; a sinistra vediamo che il modello ha dato una sola risposta sbagliata sui dati di test. Spieghiamo velocemente come si legge questa matrice: sull'asse delle ordinate è presente la predizione del modello mentre su quello delle ascisse ci sta il risultato esatto. Le entrate di questa matrice ci dicono fondamentalmente quante volte il modello ha detto "x" e doveva dire "y"; quindi gli elementi sulla diagonale sono le risposte esatte mentre i termini fuori sono risposte sbagliate.

## Q.2 Regressori

Un regressore invece 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 predirre
9 Il prezzo di una casa dati certe informazioni
10 """
11 dataset = load_boston()
12
13 #print(dataset["DESCR"])
14 #primi 13 elemnti della prima riga
15 #print(dataset["data"][0])
16 #ultimo elemnto della prima riga
17 #print(dataset["target"][0])
18
19 #caratteristiche
20 X = dataset["data"]
21
22 #output, cioe' quello che il modello dovrebbe predire
23 y = dataset["target"]
24
25 #divido i dati, un 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 e' un regressore
30 #un regressore prende i dati e restituisce un numero, una stima di qualcosa
31 modello = LinearRegression()
32
33 #addestro 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

```

### Q.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  $N \times M$  contenente 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:

```

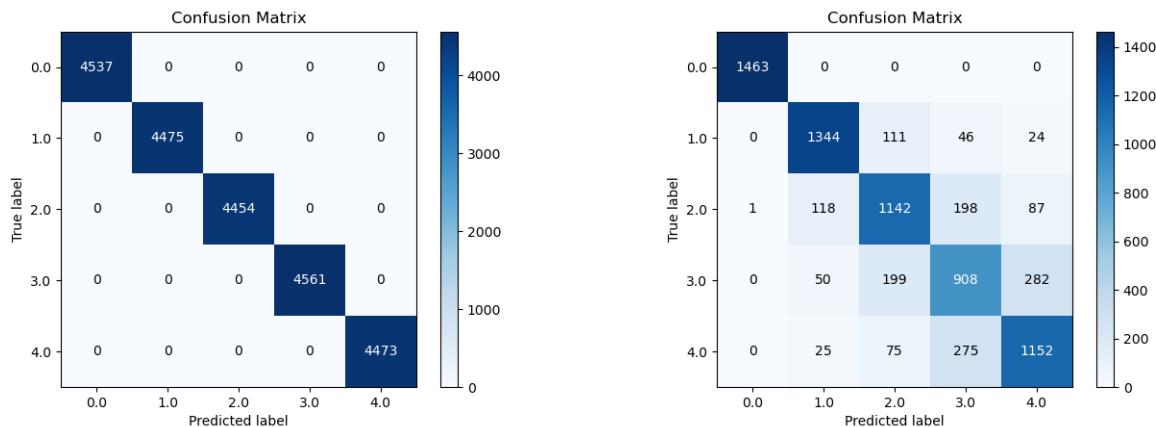
1 """
2 code that trains a model and saves it
3 """
4 import joblib
5 import numpy as np
6 import scikitplot as skplt
7 import matplotlib.pyplot as plt
8 from sklearn.metrics import accuracy_score
9 from sklearn.tree import DecisionTreeClassifier
10 from sklearn.model_selection import train_test_split
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 features
22 d = np.linspace(0, 1, 5) # parameter of curves
23 t = np.zeros(M) # target index of d
24
25 for i in range(M):
26
27     # random interval
28     x1, x2 = np.random.random(2)*5
29     # each features is nothing but a curve y=x**k with k element of d
30     k = d[i%len(d)]
31     X[:, i] = np.linspace(x1, x2, N)**k
32     # the target must be integer so we use the corrispective indices
33     t[i] = i%len(d)
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 features
20 d = np.linspace(0, 1, 5)      # parameter of curves
21 t = np.zeros(M)               # target index of d
22
23 for i in range(M):
24     # random interval

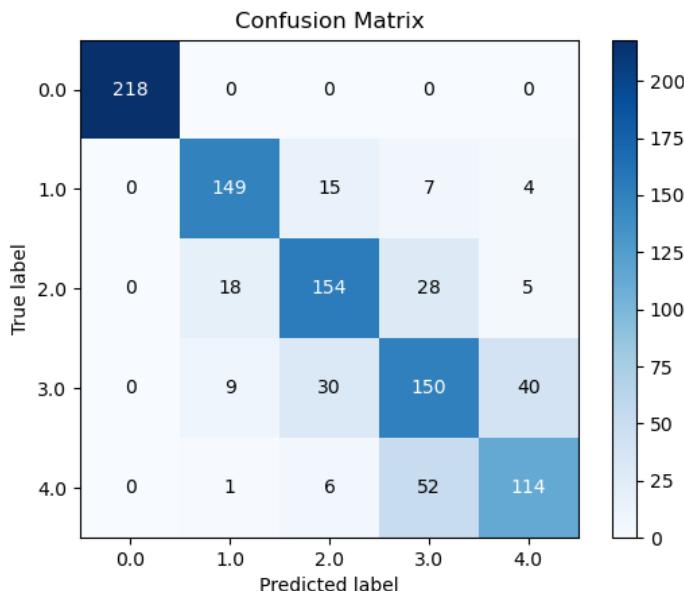
```

```

26 x1, x2 = np.random.random(2)*5
27 # each features is nothing but a curve y=x**k with k element of d
28 k = np.random.choice(d)
29 X[:, i] = np.linspace(x1, x2, N)**k
30 # the target must be integer so we use the corrispective indices
31 t[i] = np.where(k==d)[0][0]
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  $N$  otterremmo un errore, perché chiaramente la macchina sa fare solo quello per cui è stata addestrata.



## Q.4 Clustering

Il clustering è un insieme di tecniche, nell'ambito del machine learning non supervisionato, per raggruppare dei dati insieme. Immaginate di avere dei dati e che se li plottate vi formano dei grossi blob o comunque si dividano in zone abbastanza distinte. Vogliamo che il nostro algoritmo capisca quali dati appartengono ad una certa regione spaziale. Vedremo qui due algoritmi il K-Means e il DBSCAN.

### Q.4.1 K-Means

Il K-Means è uno degli algoritmi di clustering più semplici. Funziona dividendo i dati in  $K$  cluster, dove  $K$  è un input che voi dovete dare al codice. Moralmente quello che bisogna fare è:

1. Si scelgono  $K$  centri iniziali, chiamati centroidi, in modo casuale o con qualche criterio che più vi ispira.
2. Ogni punto viene assegnato al centroide più vicino, formando così  $K$  cluster.
3. I centroidi vengono ricalcolati come la media dei punti appartenenti a ciascun cluster.
4. I passaggi 2 e 3 vengono ripetuti finché la posizione dei centroidi non cambia.

Vediamo come implementare tale algoritmo:

```

1 class Kmeans:
2     ''
3     Class for kmeans algorithm for clustering.

```

```

4
5
6     def __init__(self, n_clusters, max_iter=100, tol=1e-4):
7         """
8             Parameters
9             -----
10            n_clusters : int
11                number of claster
12            max_iter : int, optional, default 100
13                maximum number of iteration
14            tol : float
15                required tollerance for convergence
16
17            self.n_clusters = n_clusters
18            self.max_iter = max_iter
19            self.tol = tol
20            self.centroids = None
21            self.labels = None
22
23    def fit(self, data):
24        """
25            Function that cluster the data
26
27            Parameters
28            -----
29            data : list
30                list of vector (i.e. list of points) that we want to classify
31
32            # Random Initializzation of the centroid
33            random_idxs = np.random.choice(len(data), self.n_clusters, replace=False)
34            self.centroids = data[random_idxs]
35
36            for i in range(self.max_iter):
37                # Assign point to the nearest centroid
38                self.labels = self.assign_clusters(data)
39
40                # Compute new centroid
41                new_centroids = self.update_centroids(data)
42
43                # Convergence criteria
44                if np.all(np.abs(new_centroids - self.centroids) < self.tol):
45                    break
46
47                # Update centroids
48                self.centroids = new_centroids
49
50    def assign_clusters(self, data):
51        """
52            Assign each point to the closest centroid
53
54            Parameters
55            -----
56            data : list
57                list of vector (i.e. list of points) that we want to classify
58
59            Return
60            -----
61            labels : 1darray
62                labels for each points
63
64            labels = []
65
66            # For each point in the dataset
67            for point in data:
68                # Compute distances with all centroids
69                distances = np.linalg.norm(point - self.centroids, axis=1)
70                # Assign to the closest
71                labels.append(np.argmin(distances))
72
73            return np.array(labels)
74
75    def update_centroids(self, data):
76        """
77            Update the centroids as the average of the points assigned to each cluster
78
79            Parameters

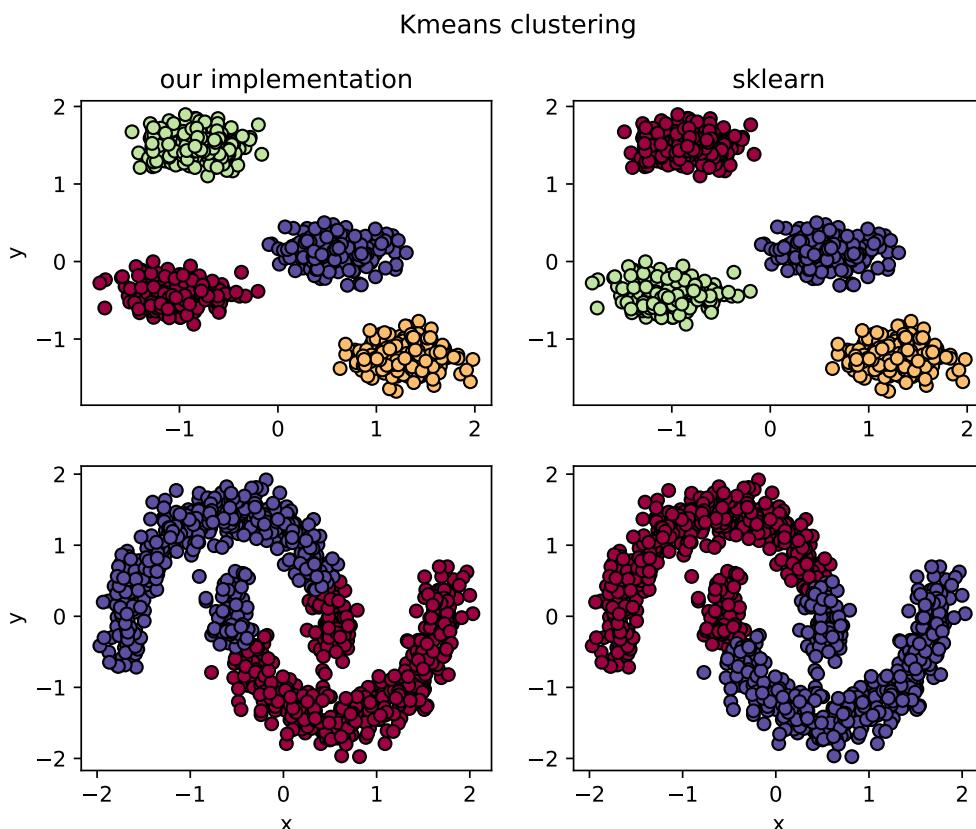
```

```

80      -----
81      data : list
82          list of vector (i.e. list of points) that we want to classify
83
84      Return
85      -----
86      new_centroids: 2darray
87          new centroids of each cluster
88      ,
89      new_centroids = np.zeros((self.n_clusters, data.shape[1]))
90
91      for i in range(self.n_clusters):
92          # Select points of i-th cluster
93          cluster_points = data[self.labels == i]
94
95          # Compute mean
96          if len(cluster_points) > 0:
97              new_centroids[i] = np.mean(cluster_points, axis=0)
98
99      return new_centroids

```

Usiamo "sklearn" per prendere i dataset su cui testare il nostro codice e confrontarlo con l'implementazione della libreria. Vedete che nel primo caso abbiamo 4 cluster differenti e l'algoritmo riesce bene a distinguerli. Mentre nel caso sotto i due cluster hanno delle forme più complesse. Noi ad occhio vediamo che i due cluster sono due mezze lune ma l'algoritmo misurando le distanze sbaglia la classificazione. Per cercare di risolvere questo problema vediamo adesso un algoritmo basto invece sulla densità; il DBSCAN (i.e. Density-Based Spatial Clustering of Applications with Noise).



#### Q.4.2 DBSCAN

Il DBSCAN (ripetiamo: Density-Based Spatial Clustering of Applications with Noise) è un algoritmo di clustering che, a differenza di K-Means, non richiede di specificare il numero di cluster. È basato sulla densità dei dati e funziona cercando regioni ad alta densità. I parametri che bisogna dare in input sono: "eps" che è la massima distanza che due punti possono avere per essere considerati vicini; "min\_samples" che è il numero minimo di vicini che si richiede per formare un cluster. Descriviamo anche qui i passaggi:

1. Si seleziona un punto non visitato, (si può scorrere in ordine i dati);

2. Se ha almeno min\_samples vicini entro la distanza eps, viene considerato un punto di "core" e si inizia a formare un nuovo cluster;
3. Se non soddisfa la condizione, viene etichettato come "rumore" (outlier);
4. I punti vicini al punto di core vengono aggiunti al cluster;
5. Si riparte da 1 finché non vengono trovati tutti i cluster.

Vediamo ora come potrebbe essere il codice e vediamo che succede:

```

1 class Dbscan:
2     """
3         Density-Based Spatial Clustering of Applications with Noise.
4         Class that implement DBSCAN algorithm for clustering.
5     """
6
7     def __init__(self, eps=0.5, min_samples=5):
8         """
9             Parameters
10            -----
11             eps : float
12                 threshold distance for classification
13             min_samples : int
14                 required number of neighbors
15         """
16         self.eps = eps
17         self.min_samples = min_samples
18         self.labels = None
19
20     def fit(self, data):
21         """
22             Function that cluster the data
23
24             Parameters
25            -----
26             data : list
27                 list of vector (i.e. list of points) that we want to classify
28         """
29
30         n_data = len(data)      # Number of data
31         self.labels = [0] * n_data # Init label
32         cluster_id = 0           # Id of current cluster
33
34         # For each point in the dataset
35         for idx in range(n_data):
36
37             # Only points that have not already been claimed
38             # can be picked as new seed points.
39             if self.labels[idx] != 0:
40                 continue # go to the next point
41
42             # Find all Neighbor of point associated to idx
43             Neighbor_pts = self.region_query(data, idx)
44
45             # If True the point is noise
46             if len(Neighbor_pts) < self.min_samples:
47                 self.labels[idx] = -1 #noise
48
49             # Otherwise use the point as the seed for a new cluster.
50             else:
51                 cluster_id += 1
52                 self.grow_cluster(data, idx, Neighbor_pts, cluster_id)
53
54         # For Ccikit learn comparison
55         # Scikit learn uses -1 to for noise,
56         # and starts cluster labeling at 0
57         for i in range(n_data):
58             if self.labels[i] != -1:
59                 self.labels[i] -= 1
60
61         self.labels = np.array(self.labels)
62
63     def grow_cluster(self, data, idx, Neighbor_pts, cluster_id):
64         """
65             Grow a new cluster
66
67             Parameters
68            -----
69             data : list

```

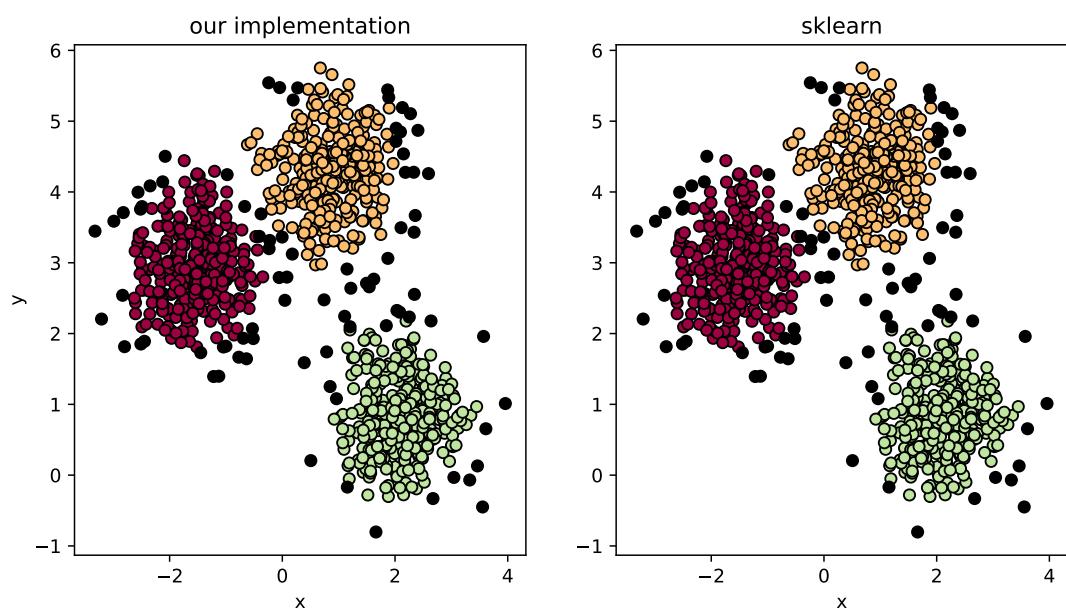
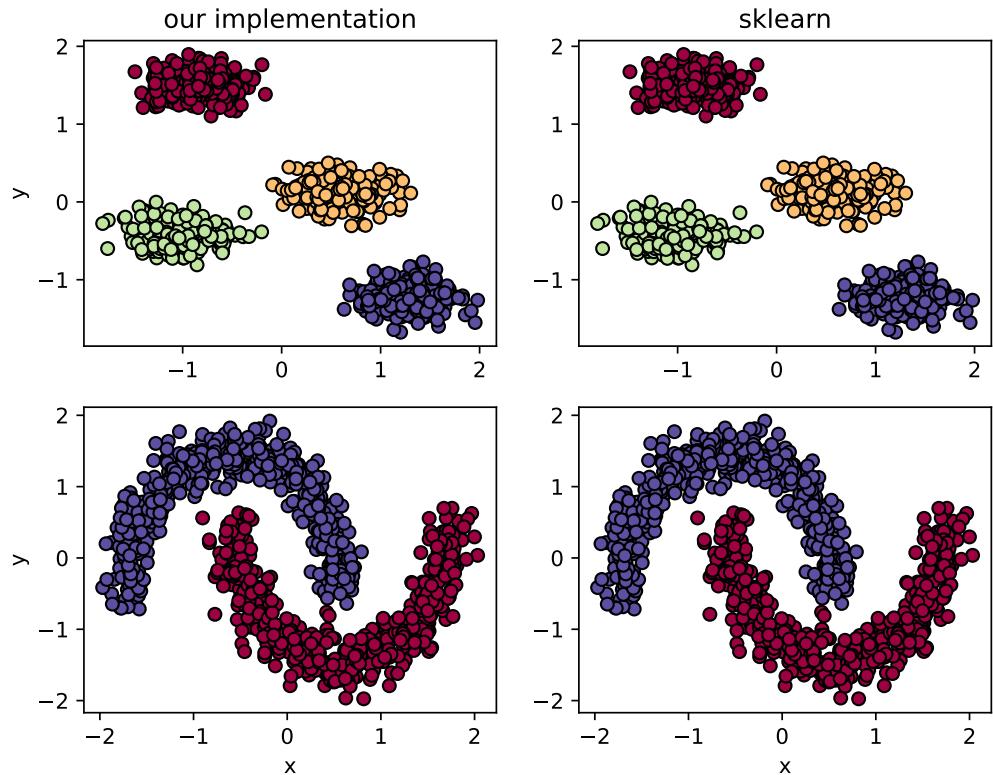
```

70         list of vector (i.e. list of points) that we want to classify
71     idx : int
72         index of the seed point
73     Neighbor_pts : list
74         all neighbor of the point idx
75     cluster_id : int
76         label for the cluster
77     ...
78
79     # Assign the label to the seed point
80     self.labels[idx] = cluster_id
81
82     i = 0
83     while i < len(Neighbor_pts):
84
85         # Go to the next point
86         next_p = Neighbor_pts[i]
87
88         # If the point is a noise for the seed search
89         # it can still be a member of cluster as a leaf
90         if self.labels[next_p] == -1:
91             self.labels[next_p] = cluster_id
92
93         # If next_p isn't already claimed, we assign it to cluster.
94         elif self.labels[next_p] == 0:
95             self.labels[next_p] = cluster_id
96
97         # Find all Neighbor of point associated to next_p
98         next_p_Neighbor_pts = self.region_query(data, next_p)
99
100        # If the point has enough neighbors then it is not a leaf
101        # but becomes a branching point of the cluster,
102        # so we add all the neighbors found from which we then start again
103        if len(next_p_Neighbor_pts) >= self.min_samples:
104            Neighbor_pts = Neighbor_pts + next_p_Neighbor_pts
105
106        # Otherwise, next_p is a leaf point so we do nothing
107
108        i += 1 # go to the next point
109
110    def region_query(self, data, idx):
111        ...
112        Find all points in dataset within distance self.eps of point idx.
113
114        Parameters
115        -----
116        data : list
117            list of vector (i.e. list of points) that we want to classify
118        idx : int
119            index of the seed point
120
121        Returns
122        -----
123        neighbors : list
124            list of the neighbors of idx
125        ...
126
127        neighbors = []
128
129        # For each point in the dataset
130        for j in range(len(data)):
131
132            # If True we add it to the list
133            if np.linalg.norm(data[idx] - data[j]) < self.eps:
134                neighbors.append(j)
135
136        return neighbors

```

Vediamo cosa succede applicandolo agli stessi dataset di prima. Mostriamo anche il caso in cui ci siano dei blob con una deviazione standar maggiore per vedere che effettivamente alcuni punti vengono scartati (nel grafico saranno i punti in nero), cosa che il precedente algoritmo non faceva. Specifichiamo infine che a volte può essere utile riscalare i dati, per farlo si è utilizzato "StandardScaler().fit\_transform(X)" di "sklearn" dove X è il dataset da analizzare.

DBSCAN clustering



# R Reti neurali

Ora dopo la nostra breve introduzione vogliamo dedicarci ad un argomento più specifico, le reti neurali.

## R.1 Reti Neurali da zero

Vogliamo infatti costruire un semplice esempio di rete neurale, che serva a classificare dei dati. Quello che vogliamo fare è costruire una scatola nera che prenda in input un certo set di dati a cui corrisponde un certo label, e la nostra scatola nera deve imparare a capire qual è il label a seconda di varie caratteristiche presenti nei dati in input. Metaforicamente potremmo dire che "allenare" una rete neurale è come fare esercizi guardando le soluzioni finché non diventi bravo e impari a farli da solo. Si tratta quindi di apprendimento supervisionato. Vediamo uno schema di una rete neurale e cerchiamo di capire cosa succede.

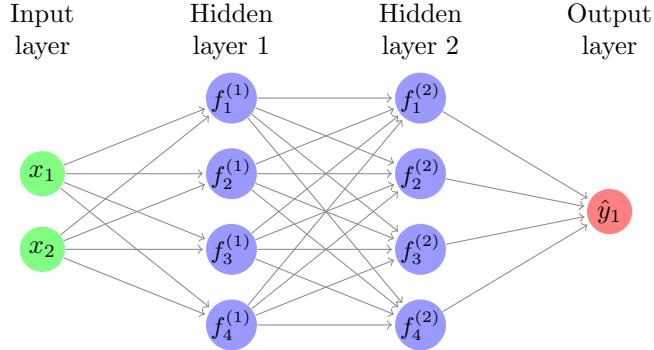


Figura 22: Schema di una rete neurale per una classificazione binaria, eventualmente il layer di output può avere due neuroni in caso di one-hot-encoding

Questo è un piccolo esempio della struttura di una rete neurale. Essa è suddivisa in layer e ogni layer contiene un certo numero di neuroni (i pallini). Il layer di input è quello a cui noi passiamo i dati, mentre il layer di output è quello che ci restituisce il risultato; è negli hidden layer che avviene la magia. Supponiamo di avere dei dati del tipo:

$x_1$	$x_2$	$y$
0.34	0.56	1
0.5	0.89	1
0.2	0.7	0
0.52	0.1	1
0.9	0.83	0
$\vdots$	$\vdots$	$\vdots$

Tabella 1: Tabella di dati da classificare: ogni dato ha due features ovvero le due coordinate  $x_1$  e  $x_2$  ed un label o target ovvero  $y$ , cioè abbiamo assegnato ad ogni punto sul piano un valore binario 0 o 1.

Quindi abbiamo dei punti sul piano, senza perdere di generalità supponiamoli fra 0 ed 1, e li abbiamo targhettabiti con uno zero o con un uno. Vediamo ora come addestrare la rete. Come prima cosa passiamo i dati in input alla rete e facciamo il primo passo, cioè dobbiamo andare verso il primo layer nascosto, quello che in genere si fa è la seguente cosa:

$$\mathbf{z} = W\mathbf{x} + \mathbf{b} \quad . \quad (164)$$

Non molto magico eh? vediamo gli elementi di questa formula  $\mathbf{x}$  è il vettore in input quindi  $\mathbf{x} = (x_1, x_2)$ ;  $W$  è una matrice e gli elementi vengono chiamati pesi; infine  $\mathbf{b}$  è un vettore ed è chiamato bias. Le dimensioni di questa matrice e questo vettore sono dati dalla dimensione del layer: nella fattispecie per il primo layer  $W$  sarà una matrice  $4 \times 2$  e  $\mathbf{b}$  un vettore lungo 4. Per cui anche il nostro output  $\mathbf{z}$  sarà un vettore lungo 4. Più in generale possiamo dire, essendo questa struttura la stessa per ogni layer, che la dimensione di  $W$  sarà sempre della forma: (numero di neuroni nel layer attuale)  $\times$  (numero di neuroni layer precedente), mentre  $\mathbf{b}$  sarà sempre un vettore lungo (numero di neuroni nel layer attuale). La domanda sorge spontanea: che valori hanno le entrate di  $W$  e di  $\mathbf{b}$ ? Inizialmente saranno scelte in maniera casuale e poi vedremo che durante le iterazioni di allenamento, dette epoch, esse verranno aggiornate secondo una certa regola. Prima di passare al secondo layer però c'è un'altra operazione da fare. Infatti solitamente non è  $\mathbf{z}$  l'output ma  $f(\mathbf{z})$  dove  $f$  è una certa funzione  $f : \mathbb{R} \rightarrow \mathbb{R}$ . Questa  $f$  è chiamata funzione di attivazione del neurone; essa restituirà nel nostro caso

un vettore lungo 4 che sarà l'input per il layer successivo. Possiamo vedere tutto ciò come delle regole per far accendere o spegnere un neurone. Fatto questo il procedimento si ripete in maniera uguale per ogni layer fino ad arrivare al layer di output. Ora che il passaggio attraverso la rete è stato fatto dobbiamo capire se l'output che genera la rete è simile a quello che noi vogliamo. È dunque arrivato il momento di calcolare una funzione che misuri la distanza tra risultato esatto e risultato della rete. Questa funzione è chiamata *loss* e nel nostro caso di classificazione binaria useremo quella che è chiamata binary cross entropy:

$$\mathcal{L} = \frac{1}{N} \sum_i^N (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)) \quad . \quad (165)$$

Non vi sarà sfuggito il fatto che questa è la classica espressione per l'entropia di un sistema di fermioni. Precisiamo che  $N$  è il numero di dati che passiamo per farlo allenare (volendo il numero di righe della tabella di sopra). Fatto questo non ci resta che decidere il modo il cui aggiornare i pesi e bias della rete. Abbiamo detto che la loss misura la distanza di quanto la rete sbaglia, quindi basterà minimizzarla. Trovando il punto di minimo avremo i parametri (pesi e bias) ottimali della nostra rete. Come se stessimo eseguendo un fit, circa. Per farlo usiamo il più semplice e classico algoritmo: il gradiente discendente (di cui è presente una discussione in una delle appendici). Abbiamo quindi che:

$$W^{i+1} = W^i - \alpha \frac{\partial \mathcal{L}}{\partial W^i}, \quad (166)$$

$$\mathbf{b}^{i+1} = \mathbf{b}^i - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{b}^i}, \quad (167)$$

dove  $\alpha$  è chiamato "learning rate" ed è un parametro che scegliamo noi. Immagino non vi sorprenderà sapere che le derivate copre citate si calcolano con la chain rule:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial f}{\partial z} \frac{\partial (W\mathbf{x} + \mathbf{b})}{\partial W}, \quad (168)$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial f}{\partial z} \frac{\partial (W\mathbf{x} + \mathbf{b})}{\partial \mathbf{b}}. \quad (169)$$

Fatto questo abbiamo dei nuovi pesi e bias con cui ripartire prendendo nuovamente i dati in input e propagarli lungo la rete. Questo passaggio si chiama "feed forward" mentre l'updating è chiamato "backpropagation". Fatti questi, è finita un'iterazione, ovvero un'epoca, e parte la successiva. Il numero di epoche è anch'esso un parametro che scegliamo noi. Altra cosa che sta al nostro giudizio è la scelta delle funzioni di attivazione, per le quali, eccezion fatta per il layer di output, non ci sono chissà che criteri per selezionarle. In letteratura si possono trovare le scelte più disparate. Noi useremo per tutti i layer nascosti delle tangenti iperboliche e per il layer di output una sigmoide. Aggiungiamo altre due cose: avendo usato come loss la binary cross entropy può essere interessante misurare l'accuratezza del risultato della rete anche in un altro modo, calcoliamo:

$$a = 1 - \left| \sum_i^N \frac{\hat{y}_i - y_i}{N} \right|, \quad (170)$$

detta accuratezza più siamo vicini ad uno meglio è però bisogna stare attenti a non overfittare. La rete potrebbe infatti star "imparando a memoria" e questo non va bene. Per controllare che questo non succeda usiamo quella è chiamata validation loss. Si tratta sempre di calcolare la loss, ma non sui dati che usiamo come dati di allenamento, ma su un altro set che usiamo esclusivamente per questo calcolo. Quindi noi passiamo alla rete dei dati e li dividiamo in due, su un set allena e sull'altro valida. Quello che facciamo è plottare insieme la loss calcolata sui dati di allenamento e quella calcolata sui dati di validation (in funzione delle epoche). Se le due scendono insieme la rete si sta comportando bene. Se invece ad un certo punto la loss scende ma la loss di validation sale vuol dire che la rete sta overfittando, cioè sta imparando a memoria le caratteristiche dei dati di allenamento. Quindi per evitare questo bisogna scegliere alcuni parametri con un po' di senso, ad esempio learning rate, numero di neuroni e numero di layer o anche numero di epoche. Detto questo passiamo ora a vedere il codice. Implementeremo una rete neurale con un solo layer nascosto con numero di neuroni variabile, due neuroni in input e uno solo in output. Lo scopo sarà proprio di riconoscere dati dei punti nel quadrato  $[0, 1] \times [0, 1]$  se essi hanno un valore associato 1 oppure 0. L'inizializzazione verrà fatta estraendo dei numeri distribuiti gaussianamente. Omettiamo per brevità, come sempre, le funzioni che fanno i plot e quello che è il main del codice.

```

1 """
2 Code that implement a shallow neural network for a binary classifications.
3 The code is written imposed 2 input, 1 output e one hidden layer.
4 Is possible to choose the dimensions of hidden layer.

```

```

5 It is also possible to save plots during the run to see how the network is learning.
6 """
7 import numpy as np
8 import matplotlib.pyplot as plt
9
10 #=====
11 # Loss function binary classification
12 #=====
13
14 def Loss(Yp, Y):
15     """
16         loss function, binary cross entropy
17
18     Parameters
19     -----
20     Yp : 1darray
21         actual prediction
22     Y : 1darray
23         Target
24
25     Returns
26     -----
27     float, binary cross entropy
28     """
29     m = len(Y)
30     return -np.sum(Y*np.log(Yp) + (1 - Y)*np.log(1 - Yp))/m
31
32 #=====
33 # Activation function
34 #=====
35
36 # Hidden layer
37 def g1(x):
38     return np.tanh(x)
39 # Output layer
40 def g2(x):
41     return 1 / (1 + np.exp(-x))
42
43 #=====
44 # Initialization
45 #=====
46
47 def init(n):
48     """
49         Random initialization of parameters weights and biases
50
51     Parameters
52     -----
53     n : int
54         number of neurons in the hidden layer
55
56     Returns
57     -----
58     W1, b1 : 2darray
59         weights and bias for hidden layer
60     W2, b2 : 2darray
61         weights and bias for output layer
62     """
63     # Hidden layer
64     # nx2 because 2 features and n neurons
65     W1 = np.random.randn(n, 2)
66     b1 = np.random.rand(n, 1)
67     # Output layer
68     # 1xn because 1 output and n neurons
69     W2 = np.random.randn(1, n)
70     b2 = np.random.rand(1, 1)
71     return W1, b1, W2, b2
72
73 #=====
74 # Network prediction function
75 #=====
76
77 def predict(X, W1, b1, W2, b2):
78     """
79         Function that returns the prediction of the network

```

```

81     Parameters
82     -----
83     X : 2darray
84         data, features
85     W1, b1, W2, b2 : 2darray
86         parameter of the network
87
88     Returns
89     -----
90     A1 : 1d array
91         intermediate prediction
92     A2 : 1d array
93         final prediction
94     """
95     # Hidden layer
96     Z1 = W1 @ X + b1
97     A1 = g1(Z1)
98     # Output layer
99     Z2 = W2 @ A1 + b2
100    A2 = g2(Z2)
101
102    return A1, A2
103
104 #=====
105 # Backpropagation function
106 #=====
107
108 def backpropagation(X, Y, step, A1, A2, W1, b1, W2, b2):
109     """
110     Backpropagation function.
111     Update weights and biases with gradient descendent
112     all the quantities came from taking the derivative of the Loss
113
114     Y : 1darray
115         Target
116     step : float
117         learning rate
118     A1, A2 : 1darray
119         predictions of the network
120     W1, b1, W2, b2 : 2darray
121         parameter of the network
122     """
123     m = len(Y)
124     # Output layer
125     dLdZ2 = (A2 - Y)
126     dLdW2 = dLdZ2 @ A1.T / m
127     dLdb2 = np.sum(dLdZ2, axis=1)[:, None] / m
128     # Hidden layer
129     dLdZ1 = W2.T @ dLdZ2 * (1 - A1**2)
130     dLdW1 = dLdZ1 @ X.T / m
131     dLdb1 = np.sum(dLdZ1, axis=1)[:, None] / m
132
133     # Update of parameters
134     W1 -= step * dLdW1
135     b1 -= step * dLdb1
136     W2 -= step * dLdW2
137     b2 -= step * dLdb2
138
139     return W1, b1, W2, b2
140
141 #=====
142 # Accuracy mesuraments
143 #=====
144
145 def accuracy(Yp, Y):
146     """
147     accuracy of prediction. We use:
148     accuracy = 1 - | sum ( prediction - target )/target_size |
149
150     Parameters
151     -----
152     Yp : 1darray
153         actual prediction
154     Y : 1darray
155         Target

```

```

157     Returns
158     -----
159     a : float
160         accuracy
161     '',
162     m = len(Y)
163     a = 1 - abs(np.sum(Yp.ravel() - Y)/m)
164     return a
165
166 #=====
167 # Train of the network
168 #=====
169
170 def train(X, Y, n_epoch, neuro, step, sp=False, verbose=True):
171     '',
172     function for the training of the network
173
174     Parameters
175     -----
176     X : 2darray
177         data, features
178     Y : 1darray
179         Target
180     n_epoch : int
181         number of epoch
182     neuro : int
183         number of neurons in the hidden layer
184     step : float
185         learning rate
186     sp : boolean, optional, default False
187         if True a plot of boundary is saved each 100 epoch
188         useful for animations
189     verbose : boolean, optional, default True
190         if True print loss and accuracy each 100 epoch
191
192     Returns
193     -----
194     result : dict
195         params -> W1, b1, W2, b2 weights and bias of network
196         train_Loss -> loss on train data
197         valid_Loss -> loss on validation data
198     '',
199
200     W1, b1, W2, b2 = init(neuro)
201     L_t = np.zeros(n_epoch) # training loss
202     L_v = np.zeros(n_epoch) # validation loss
203     N = X.shape[1]          # total number of data
204     M = N//4                # nuber of data for validation
205
206     # split dataset in validation and train
207     X_train, Y_train = X[:, :N-M], Y[:N-M]
208     X_valid, Y_valid = X[:, N-M:], Y[N-M:]
209
210     for i in range(n_epoch):
211         # train
212         A1, A2 = predict(X_train, W1, b1, W2, b2)
213         L_t[i] = Loss(A2, Y_train)
214         # validation
215         _, Yp = predict(X_valid, W1, b1, W2, b2)
216         L_v[i] = Loss(Yp, Y_valid)
217         # update
218         W1, b1, W2, b2 = backpropagation(X_train, Y_train, step, A1, A2, W1, b1, W2, b2)
219
220         if not i % 100:
221             #if sp : plot(X_train, Y_train, (W1, b1, W2, b2), i)
222
223             if verbose:
224                 acc = accuracy(A2, Y_train)
225                 print(f'Loss = {L_t[i]:.5f}, accuracy = {acc:.5f}, epoch = {i} \r', end=' ')
226
227             if verbose: print()
228
229     result = {'params'      : (W1, b1, W2, b2),
230               'train_Loss'   : L_t,
231               'valid_Loss'   : L_v,
232             }

```

```

233
234     return result

```

Dopo queste belle tre paginate di codice vediamo un po' di risultati. Questo codice era scritto per far vedere come rete impara creando un gif, che trovate disponibile sulla cartella. Il risultato finale è il seguente:

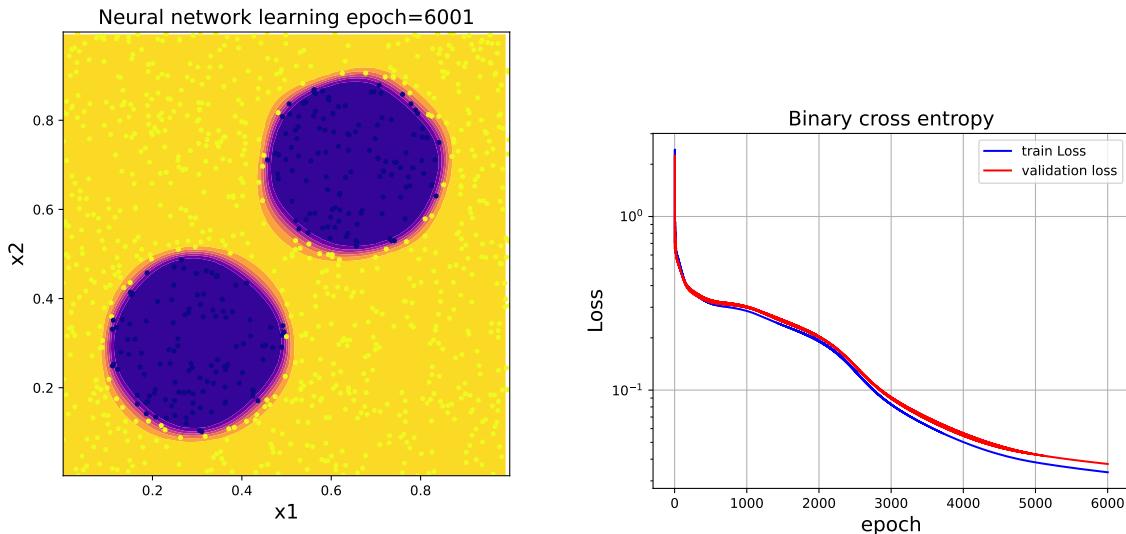


Figura 23: Risultati della rete neurale, predizione e andamento della loss. I parametri qui sono: 20 neuroni nel layer nascosto, un learning rate di 1.5, 6001 epoche, 3000 dati di train e 1000 di validation.

Il plot a sinistra è fatto sui dati di test, quindi un set di dati che la rete non ha usato per allenarsi e il risultato sembra soddisfacente, vediamo poi che le due loss scendono insieme il che è segno che la rete si comporta bene. Volendo ora farvi vedere andamenti diversi della loss vi mostro due grafici provenienti da un altro codice, in cui ho implementato una rete neurale in maniera leggermente diversa e un po' più generale. Anche perché, se vedete la loss nella figura sopra, noterete che la linea è un po' spessa, dovuto al fatto che sta oscillando (immagino sia una concausa tra problema da affrontare e algoritmo di minimizzazione). Il problema è questa volta il MNIST, ovvero il riconoscimento di cifre scritte a mano in bassa risoluzione, ogni immagine è  $28 \times 28$  pixel. Vediamo un caso in cui la rete overfittata e un caso in cui si comporta bene:

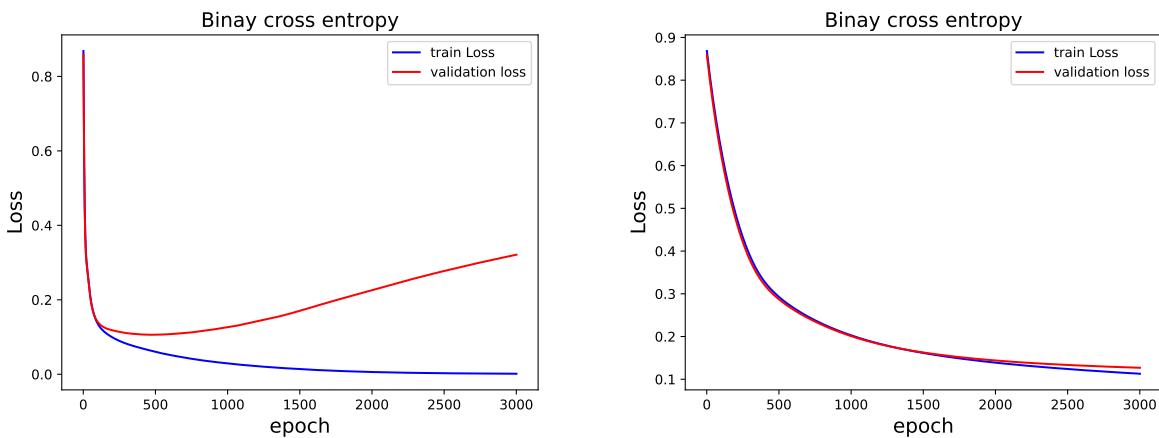


Figura 24: Andamento della loss in funzione delle epoche. Nel grafico a sinistra la rete sta overfittando, mentre a destra l'allenamento procede bene. I parametri qui sono: 2 layer nascosti ognuno da 50 neuroni; l'ottimizzatore usato è ADAM e il learning rate iniziale è di 0.05 per il caso di overfit e di 0.01 per quello di buon fit; le funzioni di attivazione sono: per i layer nascosti le "relu", mentre una sigmoide per il layer finale. Le epoche sono 3001, 2250 i dati di train e 750 quelli di validation.

Vediamo quindi che anche solo il cambiamento del learning rate iniziale (dove si specifica iniziale in quanto ADAM è un algoritmo adattivo) provoca un comportamento che non vogliamo, facendo overfittare la rete. Questo lo possiamo vedere anche, trattandosi di un classificatore, grazie a quelle che sono le matrici di confusione.

Mostriamo di seguito le matrici nel caso di overfit e di fit facendo anche il confronto vedendo i dati train e quelli di test. Spieghiamo velocemente come si legge questa matrice: sull'asse delle ordinate è presente la predizione della rete mentre su quello delle ascisse ci sta il risultato esatto. Le entrate di questa matrice ci dicono fondamentalmente quante volte la rete ha detto "x" e doveva dire "y"; quindi gli elementi sulla diagonale sono le risposte esatte mentre i termini fuori sono risposte sbagliate. Per chiarezza precisiamo che questa matrice viene calcolata sempre a fine allenamento.

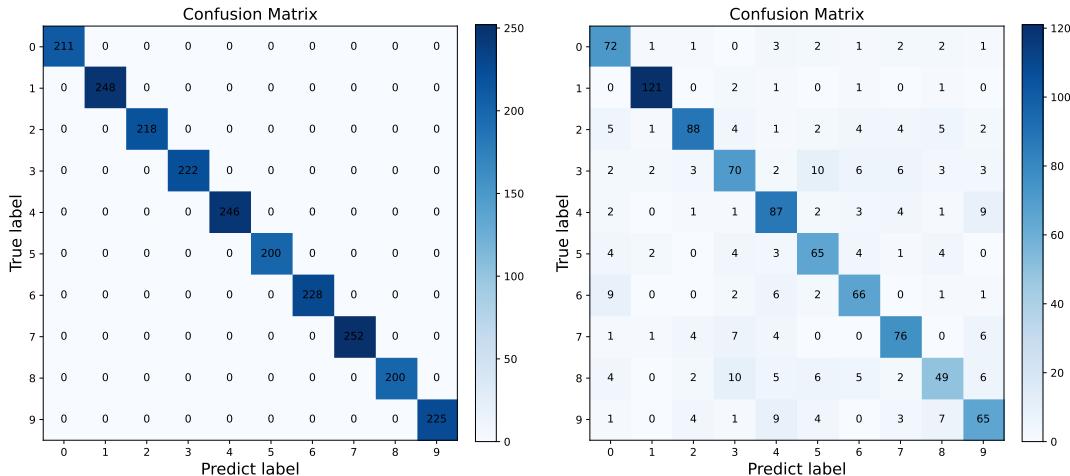


Figura 25: Vediamo qui le matrici nel caso di overfit calcolate sui dati di train, a sinistra, e su quelli di test, a destra. Avendo imparato a memoria le caratteristiche dei dati di train vediamo effettivamente che la rete non sbaglia una singola predizione. Mentre a destra, su dati che per la rete sono nuovi, ci sono molti più errori.

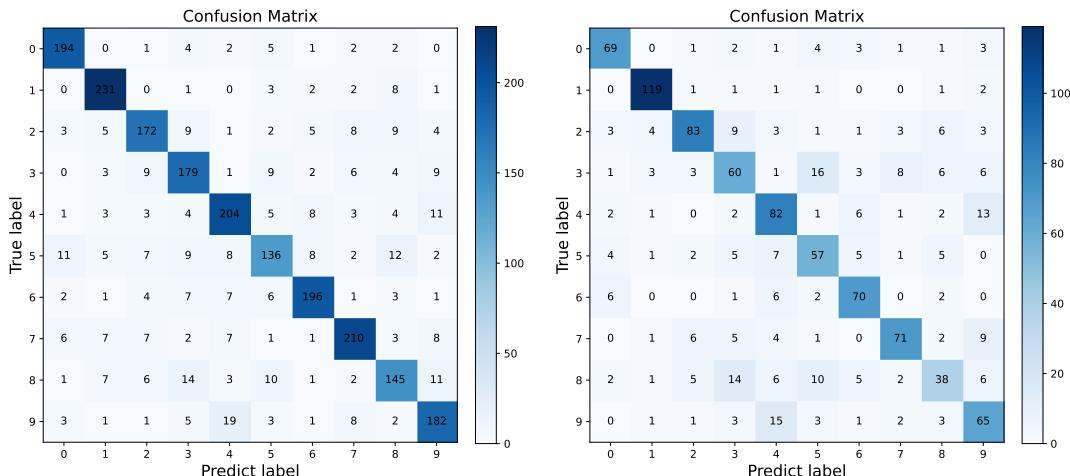


Figura 26: Vediamo qui le matrici nel caso di buon fit calcolate sui dati di train, a sinistra, e su quelli di test, a destra. Questa volta la rete non ha imparato a memoria, vediamo infatti che ci sono risposte sbagliate un po' ovunque. Le due matrici sembrano parenti.

Per concludere questa discussione in caso fosse di vostro interesse vi mostro il codice per calcolare queste matrici di confusione:

```

1 def confmat(true_target, pred_target, plot=True, k=0):
2     """
3         Function for creation and plot of confusion matrix
4
5     Parameters
6     -----
7     true_target : 1darray
8         vaules that must be predict
9     pred_target : 1darray

```

```

10     values that the network has predict
11 plot : bool, optional, default True
12     if True the matix is plotted.
13 k : int, optional, default 0
14     number of figure, necessary in order not to overlap figures
15
16 Return
17 -----
18 mat : 2darray
19     confusion matrix
20 ,,
21
22 dat = np.unique(true_target)      # classes
23 N   = len(dat)                  # Number of classes
24 mat = np.zeros((N, N), dtype=int) # confusion matrix
25
26 # creation of confusioin matrxi
27 for i in range(len(true_target)):
28     mat[true_target[i]][pred_target[i]] += 1
29
30 if plot :
31     fig = plt.figure(0, figsize=(7, 7))
32     ax = fig.add_subplot()
33
34 c = ax.imshow(mat, cmap=plt.cm.Blues) # plot matrix
35 b = fig.colorbar(c, fraction=0.046, pad=0.04)
36 # write on plot the value of predictions
37 for i in range(mat.shape[0]):
38     for j in range(mat.shape[1]):
39         ax.text(x=j, y=i, s=mat[i, j],
40                 va='center', ha='center')
41
42 # Label
43 ax.set_xticks(dat, dat)
44 ax.set_yticks(dat, dat)
45 ax.tick_params(top=False, bottom=True, labeltop=False, labelbottom=True)
46
47 plt.xlabel('Predict label', fontsize=15)
48 plt.ylabel('True label', fontsize=15)
49 plt.title('Confusion Matrix', fontsize=15)
50 plt.tight_layout()
51
52 return mat

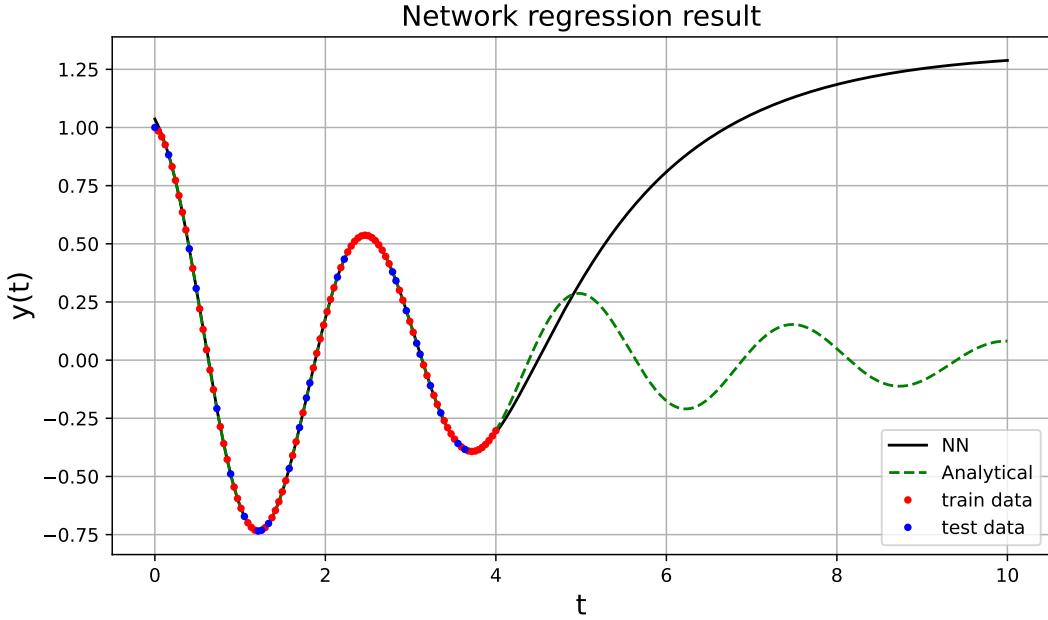
```

## R.2 PINN

Mettiamo ora un po' di fisica e torniamo all'apprendimento supervisionato. Quando abbiamo trattato le reti neurali avevamo visto l'esempio di un classificatore. Nulla però ci vieta di usarle per fare una regressione. Semplicemente ora la loss piuttosto che essere la binary-cross-entropy è il mean-square-error:

$$MSE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2, \quad (171)$$

ricorda un po' il chi-quadro no? Quello che vogliamo fare è partire da dei dati  $Y_i$  e trovare una funzione (ovvero l'output della rete) che al meglio interpoli i dati. E fin qui *nulla questio*, le reti neurali sono molto brave a fare le regressioni. Cosa succede se però vogliamo estendere le nostre predizioni in una regione in cui non abbiamo dati? Qui la situazione è più complessa, perché noi non abbiammo, come nel caso di un fit, un modello teorico da seguire. Supponiamo di avere dei dati che oscillano, la cui oscillazione è smorzata: sapendo la fisica sottostante possiamo intuire come potrebbe essere il proseguo dei nostri dati. Se questi dati li fittiamo in maniera classica, anche nella regione dove dati non ce ne sono vedremo comunque un andamento oscillante. Ma se li interpoliamo con una rete neurale? Proviamo e vediamo cosa succede:



Come rete neurale è stata usata un'implementazione simile a quella che avevamo visto nella lezione avanzata (pythorc lo useremo fra poco) e vedete tutto sommato i dati li "fitta" (o meglio dire, li interpola) bene. A valori maggiori di "x" però vediamo chiaramente come l'output della rete non sia più adatto. In verde vediamo il modello teorico usato per generare i dati di allenamento e test, il quale è rispettato dalla rete solo finchè ci sono dei dati. Non c'è però da meravigliarsi la rete minimizza uno scarto che involve dei dati, dove dati non ce ne sono lei non ha modo di sapere cosa sia la cosa giusta. Come facciamo quindi a far sì che la rete neurale approssimi bene la funzione anche dove non ci sono dati? Dobbiamo spiegargli quello che noi crediamo sia il modello sottostante alla fisica che vediamo. Che attenzione non si tratta di una certa legge come con un fit, bensì di un'equazione differenziale. Questo perchè (purtroppo) in fisica spesso non ci sono soluzioni analitiche e quindi dobbiamo partiamo direttamente dall'equazione differenziale che noi pensiamo sia la legge che descrive l'evoluzione del sistema (che comunque è un'affermazione di un certo peso). Affidiamo dunque tutto alla rete e gli diciamo che in un nostro certo range di interesse oltre a dover minimizzare lo scarto con i dati deve minimizzare altre quantità che sono appunto la ODE (o la PDE) più condizioni al bordo e/o iniziali; questa è la PINN (physisc informed neural network). Prima di passare all'implementazione vogliamo chiarire un paio di cose: la PINN non è un metodo risolutivo di equazioni differenziali, ordinarie o parziali che siano, la PINN è l'output della rete per cui essa è la soluzione per quella specifica equazione che avete inserito nella loss. Un cambiamento di una condizione o di un coefficiente richiede che la rete sia addestrata da capo di nuovo. Inoltre non è detto che usare una PINN sia migliore, o più veloce, di usare uno dei classici metodi per risolvere una ODE o una PDE; ma di certo è interessante da studiare. Adiamo ora a vedere come si costruisce la loss. Partiamo col dire che i dati che abbiamo usato seguono il seguente modello:

$$y(t) = \cos(\omega t)e^{-\gamma/2t}. \quad (172)$$

Immagino non vi sorprenderà sapere che esso è soluzione del seguente problema di cauchy:

$$\begin{cases} \ddot{y}(t) + \gamma \dot{y}(t) + \omega_0 y(t) = 0, \\ y(t_0) = y_0, \\ \dot{y}(t_0) = v_0. \end{cases} \quad (173)$$

Ricordiamo che vale  $\omega = \sqrt{\omega_0 + (\gamma/2)^2}$ . Avremo quindi che la nostra loss, sarà composta di tre termini:

$$\mathcal{L} = \mathcal{L}_{\text{data}} + \mathcal{L}_{\text{ODE}} + \mathcal{L}_{\text{initial condition}}. \quad (174)$$

Chiamiamo per semplicità  $y^{NN}$  l'output della rete, andiamo a vedere come scrivere esplicitamente questi tre

termini:

$$\mathcal{L}_{\text{data}} = \frac{1}{N_{\text{data}}} \sum_{i=1}^{N_{\text{data}}} (Y_i - y_i^{NN})^2, \quad (175)$$

$$\mathcal{L}_{\text{ODE}} = \frac{1}{N_{\text{collocation}}} \sum_{i=1}^{N_{\text{collocation}}} (\ddot{y}_i^{NN} + \gamma \dot{y}_i^{NN} + \omega_0 y_i^{NN})^2, \quad (176)$$

$$\mathcal{L}_{\text{initial condition}} = (y_0^{NN} - y_0)^2 + (\dot{y}_0^{NN} - v_0)^2. \quad (177)$$

Il primo primo è il classico termine di regressione:  $Y_i$  sono i dati di target per il train e per calcolare qui la loss devo quindi usare i dati di train. La seconda loss invece, che come vediamo è la nostra equazione differenziale, è calcolata su un altro "set di dati" chiamato: punti di collocazione. Non sono altro che un insieme di dati estratti randomicamente in tutto l'intervallo di nostro interesse; quindi ricoprono un range maggiore dei dati di train. L'ultima invece non sono altro che le condizioni iniziali, quindi calcolate all'inizio dei tempi. Procediamo ora con il codice. Per prima cosa creiamo un codice che ci costruisca l'architettura della rete, questo è un codice che potremmo usare sia per le ODE che per le PDE (in uno slancio di fantasia io l'ho salvato come "torchneural.py").

```

1 """
2 Code to create the neural network architecture
3 that will then be used in codes to solve differential equations.
4 """
5
6 import torch
7 import torch.nn as nn
8
9 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
10 torch.manual_seed(69420)
11
12 class Layer(nn.Module):
13     """
14         Class for NN layers
15     """
16     def __init__(self, n_in, n_out, act):
17         """
18             Creation of the layers
19
20             Parameters
21             -----
22             n_in : int
23                 number of neurons of the previous layer
24             n_out : int
25                 number of neurons of the current layer
26             act : torch.nn function
27                 activation function of the layer
28         """
29         super().__init__()
30         self.layer = nn.Linear(n_in, n_out)
31         self.act   = act
32
33     def forward(self, x):
34         """
35             Feedforward through the single layer
36
37             Parameter
38             -----
39             x : torch.tensor
40                 input of the layer
41         """
42         x = self.layer(x)
43
44         if self.act:
45             x = self.act(x)
46         return x
47
48 class NN(nn.Module):
49     """
50         Class for the neural network
51     """
52     def __init__(self, dim_in, dim_out, layers, r_max, r_min, act):
53         """
54             Parameters
55             -----
56             dim_in : int
57                 dimension of the input

```

```

57     dim_out : int
58         dimension of the output
59     layers : list
60         list which must contain the number of neurons for each layer
61         the number of layers is len(layers) and layers[i] is the
62         number of neurons on the i-th layer. Only hidden layers must
63         be declared
64     r_max : torch.tensor
65         max value of the input parameters
66         e.g. if we are in the square 0<x<1 0<y<1 r_max = [1, 1]
67     r_min : torch.tensor
68         min value of the input parameters
69         e.g. if we are in the square 0<x<1 0<y<1 r_max = [0, 0]
70     act : torch.nn function
71         activation functionn of the layer
72     ...
73     super().__init__()
74     self.net = nn.ModuleList()
75
76     layers = layers + [layers[-1]]    # To obtain the exact number of hidden layes
77
78     self.net.append(Layer(dim_in, layers[0], act))           # Input layer
79     for i in range(1, len(layers)):
80         self.net.append(Layer(layers[i-1], layers[i], act))   # Hidden layer
81     self.net.append(Layer(layers[-1], dim_out, act=None))    # Output layer
82
83     self.r_max = torch.tensor(r_max, dtype=torch.float).to(device)
84     self.r_min = torch.tensor(r_min, dtype=torch.float).to(device)
85
86     self.net.apply(self.init_weights)
87
88     def init_weights(self, l):
89         if isinstance(l, nn.Linear):
90             torch.nn.init.xavier_uniform_(l.weight.data)
91             torch.nn.init.zeros_(l.bias.data)
92
93     def forward(self, x):
94         ...
95         Feedforward through the all network
96
97         Parameter
98         -----
99         x : torch.tensor
100            input of the layer
101        ...
102
103        out = (x - self.r_min) / (self.r_max - self.r_min)    # Min-max scaling
104
105        for layer in self.net: # loop over all layers
106            out = layer(out)
107        return out

```

La prima cosa interessante di questo codice è la linea 8, in caso abbiate una NVIDIA. Per il resto sono due classi, la prima gestisce la creazione di un layer e il passaggio dei dati attraverso lo stesso; mentre la seconda gestisce la creazione dell'intera rete mettendo insieme vari layers, l'inizializzazione dei pesi e dei bias, e il passaggio dei dati attraverso tutta la rete. Vedete che prima di far passare i dati attraverso la rete facciamo quello che è chiamato "min-max scaling", il quale è semplicemente un metodo per riscalare i dati per far sì che la rete fatichi di meno. In genere per i casi di regressione può essere bene utilizzarlo, mentre per le classificazioni performa peggio. Una volta scritto questo codice di lui non dobbiamo più preoccuparci, andiamo quindi a vedere come creare effettivamente la PINN:

```

1 import torch
2 import numpy as np
3 import torch.nn as nn
4 import matplotlib.pyplot as plt
5 from sklearn.model_selection import train_test_split
6
7 from torchneural import NN
8 from neural import NeuralNetworkRegressor
9
10
11 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
12 torch.manual_seed(69420)
13 np.random.seed(69420)
14

```

```

15 class PINN:
16     """
17     Physics Informed Neural Network for solving an ODE
18     """
19
20     def __init__(self, layers, t_min, t_max, gamma, omg_0, act=nn.Tanh()):
21         """
22         Parameters
23         -----
24         layers : list
25             List with the number of neurons for each hidden layer
26         t_min : float
27             Minimum value of time
28         t_max : float
29             Maximum value of time
30         m : float
31             Mass of the oscillator
32         c : float
33             Damping coefficient
34         k : float
35             Spring constant
36         act : torch function, optional
37             Activation function of the network
38         """
39
40         self.net = NN(dim_in=1, dim_out=1, layers=layers, r_max=t_max, r_min=t_min, act=act).
41         to(device)
42         self.optimizer = torch.optim.Adam(self.net.parameters())
43         self.gamma = gamma
44         self.omg_0 = omg_0
45
46     def f(self, t):
47         """
48         ODE we want to solve in the form f(x, t) = 0
49         """
50         t = t.clone()
51         t.requires_grad = True
52
53         x = self.net(t) # solution x(t)
54         x_t = torch.autograd.grad(x.sum(), t, create_graph=True)[0] # dx/dt
55         x_tt = torch.autograd.grad(x_t.sum(), t, create_graph=True)[0] # d^2x/dt^2
56
57         ODE = x_tt + self.gamma* x_t + self.omg_0 * x
58
59         return ODE
60
61     def train(self, n_epoch, t_0, y_0, v_0, domain_f, data_train, data_target):
62         """
63         Train the network
64
65         Parameters
66         -----
67         n_epoch : int
68             Number of epochs for training
69         t_0 : torch.tensor
70             Initial time
71         y_0 : torch.tensor
72             Initial position
73         v_0 : torch.tensor
74             Initial speed
75         domain_f : torch.tensor
76             Collocation points for evaluating the ODE
77         data_train : torch.tensor
78             data for training
79         data_target : torch.tensor
80             target for training data
81
82         Returns
83         -----
84         Loss : list
85             List of training loss over epochs
86         """
87         Loss = []
88         for epoch in range(n_epoch):
89
90             self.optimizer.zero_grad() # zero gradients
91
92             # Loss form data

```

```

90         u_pred = self.net(data_train)
91         mse_data = torch.mean(torch.square(u_pred - data_target))
92
93         # Loss from initial condition on position
94         y0_pred = self.net(t_0)
95         y0_mse = torch.mean(torch.square(y0_pred - y_0))
96
97         # Loss from initial condition on speed
98         v_t0_pred = torch.autograd.grad(y0_pred.sum(), t_0, create_graph=True)[0]
99         v0_mse = torch.mean(torch.square(v_t0_pred - v_0))
100
101        # Loss from ODE
102        f_pred = self.f(domain_f)
103        ode_mse = torch.mean(torch.square(f_pred))
104
105        loss = mse_data + y0_mse + v0_mse + ode_mse
106
107        loss.backward()
108        self.optimizer.step()
109
110        with torch.autograd.no_grad():
111            Loss.append(loss.data.detach().cpu().numpy())
112            if epoch % 100 == 0:
113                print(f"epoch: {epoch}, data: {mse_data:.3e}, x_0: {y0_mse:.3e}, v_0: {v0_mse:.3e}, ODE: {ode_mse:.3e}")
114
115    return Loss
116
117 #=====
118 # Computational parameters for ODE
119 #=====
120
121 # Time interval
122 t_min = 0.0
123 t_max = 10.0
124 t_data_min = 0.0
125 t_data_max = 4.0
126
127 # Number of points for collocation
128 N_t = 1000
129 # Number of points in dataset
130 N_p = 100
131
132 # Parameters for the system
133 gamma = 0.5 # damping coefficient
134 omg = np.sqrt(2*np.pi) # final omega
135 omg_0 = omg**2 + (gamma/2)**2 # omega of the equation
136
137 # Initial conditions
138 t_0 = np.array([[0.0]]) # Initial time t = 0
139 y_0 = np.array([[1.0]]) # x(0) = 1
140 v_0 = np.array([[0.0]]) # v(0) = 0
141
142 # Collocation points in the time domain
143 t_f = np.random.uniform(t_min, t_max, (N_t, 1))
144
145 # Creation of the dataset
146 X = np.linspace(t_data_min, t_data_max, N_p)
147 n = len(X)
148 y = np.cos(omg*X)*np.exp(-X*gamma/2) # target
149
150 X = X[:, None]
151 y = y[:, None]
152
153 # split dataset in test and train
154 X_train, X_test, Y_train, Y_test = train_test_split(X, y, random_state=42)
155
156 #=====
157 # Convert to Tensor
158 #=====
159
160 # requires_grad = True will be necessary for initial speed loss
161 # in general it is necessary when we want to compute derivatives
162 t_0 = torch.tensor(t_0, dtype=torch.float, requires_grad=True).to(device)
163 y_0 = torch.tensor(y_0, dtype=torch.float).to(device)
164 v_0 = torch.tensor(v_0, dtype=torch.float).to(device)

```

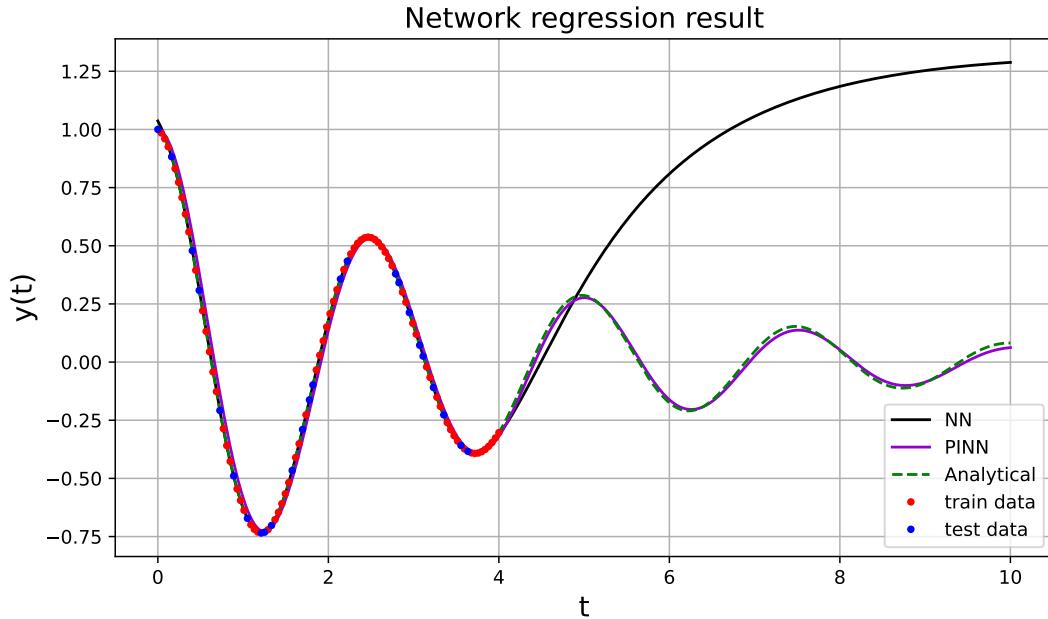
```

165 domain_f = torch.tensor(t_f, dtype=torch.float).to(device)
166 X_train_t = torch.tensor(X_train, dtype=torch.float).to(device)
167 Y_train_t = torch.tensor(Y_train, dtype=torch.float).to(device)
168
169 #=====
170 # Creation of network and train
171 #=====
172
173 n_epoch = 12000 + 1
174 pinn = PINN([50, 50, 50], t_min, t_max, gamma, omg_0)
175 Loss = pinn.train(n_epoch, t_0, y_0, v_0, domain_f, X_train_t, Y_train_t)
176
177 #=====
178 # Prediction of PINN
179 #=====
180
181 t = np.linspace(t_min, t_max, 1000)
182 t_tensor = torch.tensor(t.reshape(-1, 1), dtype=torch.float).to(device)
183 x_analytical = np.exp(-gamma/2 * t) * np.cos(omg * t)
184
185 # Disable gradient calculation for plotting
186 with torch.no_grad():
187     y_pred = pinn.net(t_tensor).cpu().numpy()
188
189 #=====
190 # Creation of dataset for simple NN
191 #=====
192
193 X_train, X_test = X_train.T, X_test.T
194 Y_train, Y_test = Y_train[:, 0], Y_test[:, 0]
195
196 # for plot
197 Xp = np.linspace(t_min, t_max, 1000)
198 Xp = np.expand_dims(Xp, axis=1)
199
200 #=====
201 # Simple NN
202 #=====
203 n_epoch = 10000 + 1
204 lr_rate = 0.01
205
206 NN = NeuralNetworkRegressor([50, 50, 50], n_epoch, f_act='tanh')
207
208 result = NN.train(X_train, Y_train, alpha=lr_rate, verbose=True)
209
210 #=====
211 # Plot
212 #=====
213
214 p_fit = NN.predict(Xp.T)
215 p_fit = p_fit[0]
216
217 plt.figure(2, figsize=(9, 5))
218 plt.title('Network regression result', fontsize=15)
219 plt.xlabel('t', fontsize=15)
220 plt.ylabel('y(t)', fontsize=15)
221 plt.grid()
222 plt.errorbar(X_train.T[:, 0], Y_train, fmt='.', c='r', label='train data')
223 plt.errorbar(X_test.T[:, 0], Y_test, fmt='.', c='b', label='test data')
224 plt.plot(Xp[:, 0], p_fit, 'k', label='NN')
225 plt.plot(t, y_pred, label='PINN', color='darkviolet')
226 plt.plot(t, x_analytical, label='Analytical', linestyle='--', color='green')
227 plt.legend(loc='best')
228 plt.show()

```

Iniziamo con il dire che la linea 8 del codice, fa riferimento a quanto dicevamo all'inizio, si tratta di un semplice regressore sulla scia di quanto si era visto nella quarta lezione del corso avanzato; se siete interessati il codice sta qui: <https://github.com/Francesco-Zeno-Costanzo/neural-network/blob/main/neural.py>. Non lo aggiungo alle note o nei codici disponibile per evitare appesantimenti. Inoltre ovviamente il confronto lo avremmo potuto fare con un regressore di "sklearn"; semplicemente avevo il codice già scritto per interpolare quei dati. Vediamo che la classe "PINN" utilizza la classe "NN" vista prima, quindi tutto è demandato alla libreria "pythorc". Il metodo "f" di questa classe rappresenta la nostra ode; la funzione di train invece semplicemente calcola i vari pezzi delle loss e li mette insieme. Il grosso del lavoro è demandato ai metodi: "optimizer.zero\_grad()" che setta tutto a zero per evita sovrascritture strane, "backward()" che fa la back pro-

pagation ed infine "optimizer.step()" che uno step dell'ottimizzatore (qui vedete che abbiamo usato ADAM, che è anche quello implementato nel codice "neural"). Il resto poi non è nulla di particolarmente complicato. Vogliamo solo precisare un particolare piuttosto delicato: le linee 150 e 151 sono oltremodo necessarie perché quello è il formato in cui pytorch vuole i dati (ricordiamo che una matrice  $N \times 1$  e un vettore lungo  $N$  sono due cose diverse). Più in basso, linee 193 -194, li modifico in quanto il mio codice li vuole in un formato diverso. Vediamo ora il risultato:



Decisamente un risultato migliore, la PINN è molto vicina alla linea verde anche a grandi tempi.

## S Calcolo parallelo

Fin ora ogni codice che abbiamo scritto, o quasi (cfr. le reti neurali), è sempre stato un codice che eseguiva un solo processo. Mi spiego meglio: il vostro computer possiede un processore, la CPU (Central Processing Unit), il quale ha un certo numero di core. Quando voi eseguite un codice il vostro pc si carica in RAM (Random Access Memory [Daft Punk]) le informazioni necessarie e affida ad uno dei core il quale fa tutto il lavoro. In realtà i calcoli non vengono eseguiti tutti da uno stesso core se il processo ci impiega molto, ma il lavoro viene diviso sempre usando però un core alla volta. Il motivo è puramente storico: i computer, inizialmente, erano per lo più monoprocessoressi e, conseguentemente, i linguaggi di programmazione sono stati strutturati in modo tale da eseguire le operazioni in modo sequenziale. Quel che vogliamo fare noi è usare più core contemporaneamente. Prima di andare avanti facciamo una piccola dichiarazione di intenti; parleremo solo di calcolo parallelo su CPU e non su GPU. Questo perché benché le GPU abbiano molti più core di una CPU e quindi ottimizzerebbero le operazioni in modo migliori sono più delicate. Infatti con la GPU dobbiamo gestire noi tutti i trasferimenti di informazioni mente con la CPU a meno di non dover far comunicare due core diversi non dovremmo preoccuparci di nulla perché ci penserà il pc a caricarsi tutte le informazioni necessarie e a smistarle a seconda delle esigenze. Inoltre per programmare su GPU serve che il vostro computer abbia una scheda grafica NVIDIA, perché praticamente tutte le librerie necessarie si interfacciano con questo tipo di GPU. Il mio pc come scheda video ha invece una Radeon.

Ci sono, in linea di principio due vie per parallelizzare un codice usare i thread o i processi, ma c'è anche un problema che impedisce la percorrenza di una delle due vie: il GIL. Python ha infatti una caratteristica particolare nota come Global Interpreter Lock (GIL). Il GIL è un mutex (mutual exclusion) che protegge l'accesso alla memoria, impedendo che più thread nativi eseguano bytecode Python simultaneamente. Questo meccanismo è stato introdotto per facilitare la gestione della memoria interna e garantire la sicurezza del thread, ma introduce una limitazione significativa nel calcolo parallelo. Per aggirare il problema del GIL comunque di strade ce ne sono. Abbiamo due vie: "numba" e "multiprocessing" (che sono due pacchetti). Il primo di questi si inserisce nell'ambito dei cosiddetti compilatori *jit* (just-in-time). Questi compilatori non permettono solo di parallelizzare il codice (siccome riescono a disabilitare il GIL) ma anche di ottimizzare il codice. Questi infatti, a differenza dell'interprete classico di Python, vanno a compilare, all'inizio dell'esecuzione del programma, le parti di codice che vengono "marcate" come da ottimizzare in un formato più vicino al linguaggio macchina (da qui il nome di compilatore just-in-time). Questo permette di migliorare le performance del codice per i seguenti motivi:

1. il codice viene compilato in un linguaggio ottimizzato;
2. il codice compilato viene riutilizzato. Infatti, se indichiamo una funzione come da compilare, il compilatore, per ogni chiamata di quella funzione, inserirà un puntatore che punta all'indirizzo di memoria in cui sono contenute le istruzioni compilate e ottimizzate della funzione;
3. (motivo tecnico) per come sono implementati questi compilatori, utilizzano delle politiche di compilazione che sono mirate per lo specifico processore della macchina su cui il codice viene eseguito. Questo consente su molti microprocessori, per esempio, di immagazzinare le istruzioni ottimizzate dalla compilazione direttamente in registri del microprocessore (detta cache), diminuendo così il tempo di lettura delle istruzioni e aumentando così le prestazioni.

La documentazione di "numba" è molto vasta, quindi si consiglia, come sempre, di leggerla per approfondire l'argomento, tuttavia spero che i seguenti esempi proposti siano comunque esaustivi per capire il funzionamento. Il secondo modulo è invece "multiprocessing", che permette di creare nuovi processi invece di thread. Poiché i processi hanno spazi di indirizzamento della memoria separati, il GIL non rappresenta un ostacolo, consentendo un vero parallelismo. Il modulo "multiprocessing" offre una API simile a quella del modulo "threading", ma crea processi distinti, ognuno con il proprio interprete Python e spazio di memoria. Questo ci permette quindi di usare effettivamente più core insieme e diminuire quindi il tempo di esecuzione del nostro codice. Prima di iniziare a vedere un po' di codice dobbiamo porci una domanda: ha senso parallelizzare un codice? Infatti non tutti i problemi sono parallelizzabili, magari un core sta facendo un certo calcolo ma ha bisogno dei risultati di un secondo calcolo che viene svolto da un altro core. Dunque il primo core dovrà aspettare che il secondo abbia finito per iniziare ad eseguire, perdendo il vantaggio di fare le cose in contemporanea. Dobbiamo poi anche assicuraci di gestire bene la comunicazione dei processi in quanto quella è seriale e per cui rallenta l'esecuzione. Quindi se il calcolo da fare è veloce perché distribuito ma poi tutti i core devono mettersi in conclave a scambiarsi le informazioni ogni secondo ciò potrebbe condurre ad un rallentamento dell'esecuzione. Finita questa breve introduzione cominciamo a vedere esempi più concreti.

### S.1 Numba

Cominciamo con "numba", e dato che poco abbiamo da fare a livello di codice, ci basta sapere cos'è un decoratore, vediamo subito delle applicazioni per capire cosa succede.

### S.1.1 Equazione al laplaciano in 2 dimensioni

Uno dei problemi più complicati da risolvere in maniera analitica è il problema fondamentale dell'elettrostatica: l'equazione di Laplace. Il motivo per cui è abbastanza complicata è che spesso le condizioni al bordo di questa equazione fanno in modo che la soluzione non sia esprimibile sotto forma di serie e/o integrale. Tuttavia nella maggior parte dei casi è possibile dare una soluzione approssimata numericamente. Tale equazione è definita come

$$\nabla^2 V = 0, \quad (178)$$

e, ringraziando i nostri amici matematici, sappiamo che esiste sempre una soluzione a tale equazione. Mentre dal corso di Fisica 2 sappiamo che la soluzione è unica (permettendomi una perla del professore La Rocca, ai Fisici interessa solo l'unicità della soluzione: per verificare l'esistenza di una soluzione basta andare in laboratorio e verificare se effettivamente esiste). Possiamo procedere, come al solito, andando a discretizzare lo spazio e approssimando il laplaciano con le derivate discrete, che assume la seguente forma

$$\frac{u(x + ih_x, y) - 2u(x, y) + u(x - ih_x, y)}{h_x^2} + \frac{u(x, y + ih_y) - 2u(x, y) + u(x, y - ih_y)}{h_y^2} = 0, \quad (179)$$

dove  $h_x$  e  $h_y$  sono rispettivamente il passo fra le ascisse e il passo fra le ordinate. Esplicitando per  $u(x, y)$ , risulta che:

$$u(x, y) = \frac{1}{4}(u(x + ih_x, y) + u(x - ih_x, y) + u(x, y + ih_y) - u(x, y - ih_y)). \quad (180)$$

In questo caso, siccome il laplaciano è un operatore che si comporta "abbastanza" bene, la nostra soluzione è sicuramente stabile per  $h << 1$ . La questione della stabilità non è banale e per questo si rimanda a delle trattazioni più accurate dell'argomento. Utilizzando numba, possiamo mettere delle condizioni al contorno più *impegnative*, ovvero possiamo mettere delle condizioni al bordo dove la nostra funzione è una sovrapposizione di seni, coseni ed esponenziali, le quali sono decisamente più computazionalmente impattanti sul tempo di esecuzione rispetto ai casi "soliti" che si vedono negli esercizi dove, per esempio, abbiamo un conduttore che è a potenziale costante (cfr. quanto avevamo scritto sopra nella sezione delle PDE). Procediamo adesso a scrivere il codice:

```

1 """
2 Code for the solution of Laplace equation via SOR and wit numba wrapper
3 """
4 import time
5 import numba
6 import numpy as np
7 import matplotlib.pyplot as plt
8
9 @numba.jit("Tuple((f8[:, :], i8))(i8, f8[:, :], f8[:, :], f8, f8)", nopython=True, nogil=True)
10 def solve_lap_sor(N, bound, rho, w, tau):
11     """
12     Function that use SOR method to solve laplace equation
13
14     Parameters
15     -----
16     N : int
17         size of the grid
18     bound : 2darray
19         boundary conditions
20     rho : 2darray
21         source of field
22     w : float,
23         overrelaxation parameter
24     tau : float
25         required convergence
26
27     Return
28     -----
29     phi : 2darray
30         solution of the equations
31     iter_count : int
32         numer of required iterations
33     """
34     # Useful quantities
35     conv      = 10.0
36     mean_U0   = 0
37     iter_count = 0
38
39     # Matrix of the potential
40     phi = np.zeros((N+1, N+1))

```

```

41 # Boundary conditions
42 phi[:, 0] = bound[0] # west
43 phi[:, N] = bound[1] # est
44 phi[0, :] = bound[2] # south
45 phi[N, :] = bound[3] # north
46
47
48 mean_U0 = np.mean(phi)
49
50 while conv > tau:
51     for i in range(1, N):
52         for j in range(1, N):
53             force = phi[i, j+1] + phi[i, j-1] + phi[i+1, j] + phi[i-1, j]
54             force += rho[i, j]
55             phi[i, j] = w * 0.25 * force + (1 - w) * phi[i, j] #SOR
56
57     mean_U = np.mean(phi)
58     conv = abs(mean_U - mean_U0)
59     mean_U0 = mean_U
60
61     iter_count += 1
62
63 return phi, iter_count

```

Come vedete il codice, a parte il decoratore, che ora andiamo ad indagare, è uguale a quanto scritto nella sezione sulle pde. Abbiamo cambiato solo le condizioni iniziali che sono le seguenti:

```

1 if __name__ == "__main__":
2     N = 300
3     x = np.linspace(-1, 1, N+1)
4     # Boundary conditions
5     bound = np.zeros((4, N+1))
6     bound[0, :] = 0.5*(x**2 - x) # 0
7     bound[1, :] = 1/(1/np.e - np.e)*(np.exp(x)-np.e) # 0
8     bound[2, :] = x**4 # -2
9     bound[3, :] = np.cos(np.pi*x/2) # 2

```

Il resto per il plot lo lasciamo sempre al vostro gusto. Osserviamo quindi, come funziona la libreria numba: si nota, innanzitutto, che il wrapper `numba.jit` viene chiamato immediatamente prima della funzione e osserviamo quali sono gli argomenti che gli sono stati passati:

- il primo è la *signature* della funzione, in cui sono indicati i tipi di dati che la funzione restituisce e prende in ingresso. Nel nostro caso noi vogliamo che la funzione restituisca un array 2-dimensionale e un intero, prendendo in ingresso due array 2-dimensionale e due numeri reali: si osservi che questi vanno indicati riportando prima i tipi dei dati restituiti e poi quelli che prende in input, riportando questi ultimi fra parentesi e separandoli fra virgole. Per quelli in output ci sta da fare una piccola precisazione. Se la nostra funzione avesse restituito solo la matrice del potenziale, ci sarebbe bastato scrivere:

```
1 @numba.jit("f8[:, :](i8, f8[:, :], f8[:, :], f8, f8)", nopython=True, nogil=True)
```

Dato che però restituisce due variabili, e quindi a livello di python, una tupla, dobbiamo dirlo esplicitamente. Vediamo cosa poi nel dettaglio cosa voglia dire quanto è dentro la stringa: in questo caso `f8` sta per `float64`, dunque `f8[:, :]` rappresenta un array 2-dimensionale le cui entrate sono `float64`. Similmente, `i8` sta per `int64`;

- il parametro `nopython=True`, il quale comunica a numba che noi vorremmo (se riesce) compilare la funzione per aumentare le performance. Il motivo di tale nome deriva dal fatto che la funzione compilata sarà compilata in C, quindi non sarà più il formato `bytecode` che l'interprete legge ed esegue
- il parametro `nogil` che, da quanto detto prima, è abbastanza chiaro che cosa faccia.

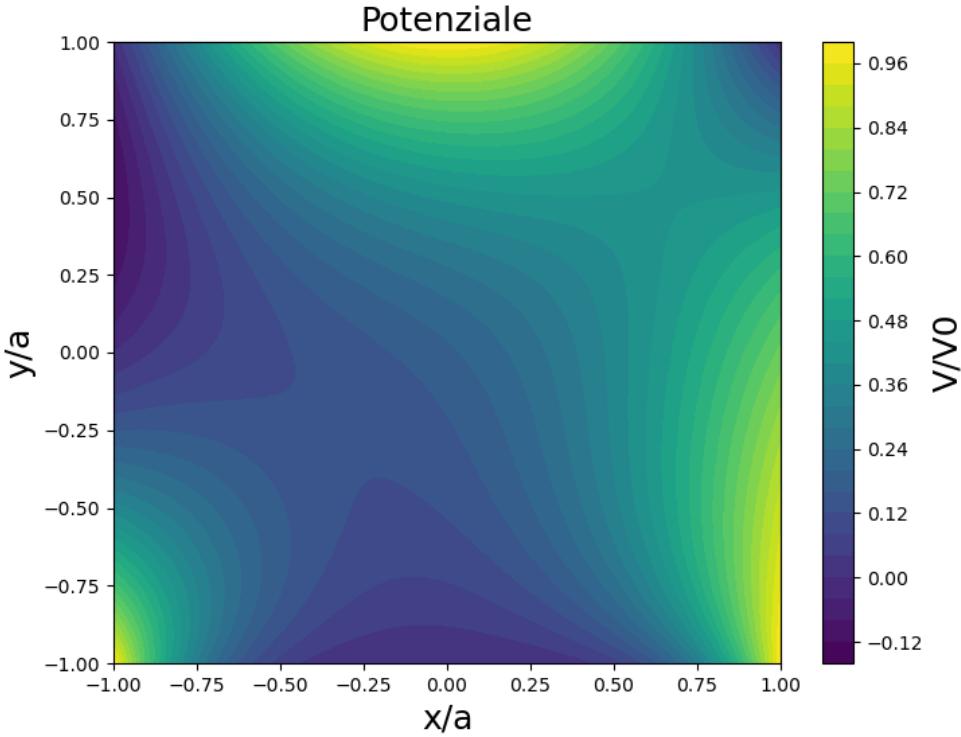


Figura 27: Soluzione numerica del potenziale nel quadrato.

E' interessante, chiaramente, vedere se è migliorato e quanto è migliorato il tempo di esecuzione con `numba`. A parità di condizioni iniziali, le ultime usate, e scegliendo:  $N = 300$ ,  $\omega = 1.99$ ,  $\tau = 1e - 8$  Il nostro codice impiega: 830 iterazioni e 0.61 secondi. Senza il decoratore il codice impiega sempre 830 iterazioni, come logica vuole, ma impiega 108.56 secondi. Risulta quindi che `numba` è circa 178 volte più veloce! Pazzesco, no?

. Però se eseguiamo il codice, lo vediamo subito che in realtà il numero sulla shell compare dopo qualche secondo? Come mai questa cosa? Su linux il comando time scritto prima dell'esecuzione di un comando da shell, ci da maggiori informazioni sul tempo impiegato dal codice. Nel nostro codice si ottiene:

```

1 Number of iterations 830
2 Elapsed time: 0.61 s
3
4 real      0m5,604s
5 user      0m6,402s
6 sys       0m0,611s

```

Vediamo quindi che in realtà il codice termina in 5.6 secondi. Il fatto che la voce user sia ad un valore maggiore di real ci fa capire che effettivamente la cpu ha lavorato tanto e bene. L'ultima invece ci da informazioni sulle varie allocazioni di memoria e gestione di input e output. Per carità questi 5 secondi non sono tremendi, perché comunque senza numba ci sarebbe voluto molto di più. Ma questo ci fa notare come la gran parte del tempo se ne va in compilazione della funzione decorata da numba. Lo vediamo con questo semplice esempio:

```

1 import time
2 import numba
3 import numpy as np
4
5 @numba.jit("f8[:, :](f8[:, :])", nopython=True, nogil=True)
6 def f(x):
7     return np.sin(x) + np.cos(x)
8
9 x = np.linspace(0, 10, int(1e4))
10 x = x * x[:, None]
11
12
13 start = time.time()
14 f(x)
15 end = time.time()
16 print(f"First call: {end - start:.4f} s")
17
18 start = time.time()

```

```

19 f(x)
20 end = time.time()
21 print(f"Second call: {end - start:.4f} s")
22
23 start = time.time()
24 f(x)
25 end = time.time()
26 print(f"Third call: {end - start:.4f} s")
27
28 [Output]
29 First call: 4.1520 s
30 Second call: 2.5227 s
31 Third call: 2.4688 s

```

Vediamo che quindi il problema è solo la prima volta, quando compilo, poi va più veloce. Tornando all'equazione di Laplace le magie però non sono ancora finite: potremmo anche pensare di inserire delle condizioni al contorno ancora più *wild* e inserire, per esempio, un blocco di potenziale all'interno della nostra scatola in cui il potenziale è fisso: per fare ciò supponiamo che in tale regione a potenziale costante si ha che  $V = 1$ , I pochi cambiamenti al codice sono:

```

1 """
2 Code for the solution of Laplace equation via SOR and wit numba wrapper
3 """
4 import time
5 import numba
6 import numpy as np
7 import matplotlib.pyplot as plt
8
9 @numba.jit("Tuple((f8[:, :, :], i8))(i8, f8[:, :, :], f8[:, :, :], f8[:, :, :], f8, f8)", nopython=True, nogil
10           =True)
11 def solve_lap_sor(N, bound, fixed, rho, w, tau):
12     """
13         Function that use SOR method to solve laplace equation
14
15     Parameters
16     -----
17     N : int
18         size of the grid
19     bound : 2darray
20         boundary conditions
21     fixed : 2darray
22         additional conditions to be respected
23     rho : 2darray
24         source of field
25     w : float,
26         overrelaxation parameter
27     tau : float
28         required convergence
29
30     Return
31     -----
32     phi : 2darray
33         solution of the equations
34     iter_count : int
35         numer of required iterations
36
37     # Useful quantities
38     conv      = 10.0
39     mean_U0   = 0
40     iter_count = 0
41
42     # Matrix of the potential
43     phi = np.zeros((N+1, N+1))
44
45     # Boundary conditions
46     phi[:, 0] = bound[0]    # west
47     phi[:, N] = bound[1]    # est
48     phi[0, :] = bound[2]    # south
49     phi[N, :] = bound[3]    # north
50
51     mask = fixed!= 0
52     # We need to use loops because numba
53     # doesn't support numpy mask vectorization
54     for i in range(1, N):
55         for j in range(1, N):
56             if mask[i, j]:

```

```

56         phi[i, j] = fixed[i, j]
57
58     mean_U0 = np.mean(phi)
59
60     while conv > tau:
61         for i in range(1, N):
62             for j in range(1, N):
63                 if not mask[i, j]:
64                     force = phi[i, j+1] + phi[i, j-1] + phi[i+1, j] + phi[i-1, j]
65                     force += rho[i, j]
66                     phi[i, j] = w * 0.25 * force + (1 - w) * phi[i, j] #SOR
67
68     mean_U = np.mean(phi)
69     conv = abs(mean_U - mean_U0)
70     mean_U0 = mean_U
71
72     iter_count += 1
73
74 return phi, iter_count
75
76 def pot_block(x, y):
77     """
78     Function that checks if (x, y) is a point of the grid
79     which belongs to the area of constant potential
80     """
81     return np.select([(x>0.3)*(x<0.6)*(y > 0.3)*(y<0.6), (x <= 0.3)* (x>= 0.6)*(y<=0.3)*(y>=0.6)], [1., 0.])
82
83
84 if __name__ == "__main__":
85     N = 300
86     w = 1.99
87     tau = 1e-8
88     dx = 1/N
89
90     x = np.linspace(-1, 1, N+1)
91     gridx, gridy = np.meshgrid(x, x)
92     # Boundary conditions
93     bound = np.zeros((4, N+1))
94     bound[0, :] = 0.5*(x**2 - x) # 0
95     bound[1, :] = 1/(1/np.e - np.e)*(np.exp(x)-np.e) # 0
96     bound[2, :] = x**4 # -2
97     bound[3, :] = np.cos(np.pi*x/2) # 2
98
99     fixed = pot_block(gridx, gridy)
100
101    # Source
102    rho = np.zeros((N+1, N+1))
103
104    # Solution
105    start = time.time()
106    phi, ic = solve_lap_sor(N, bound, fixed, rho, w, tau)
107    end = time.time() - start
108
109    print(f"Number of iterations {ic}")
110    print(f"Elapsed time: {end:.2f} s")

```

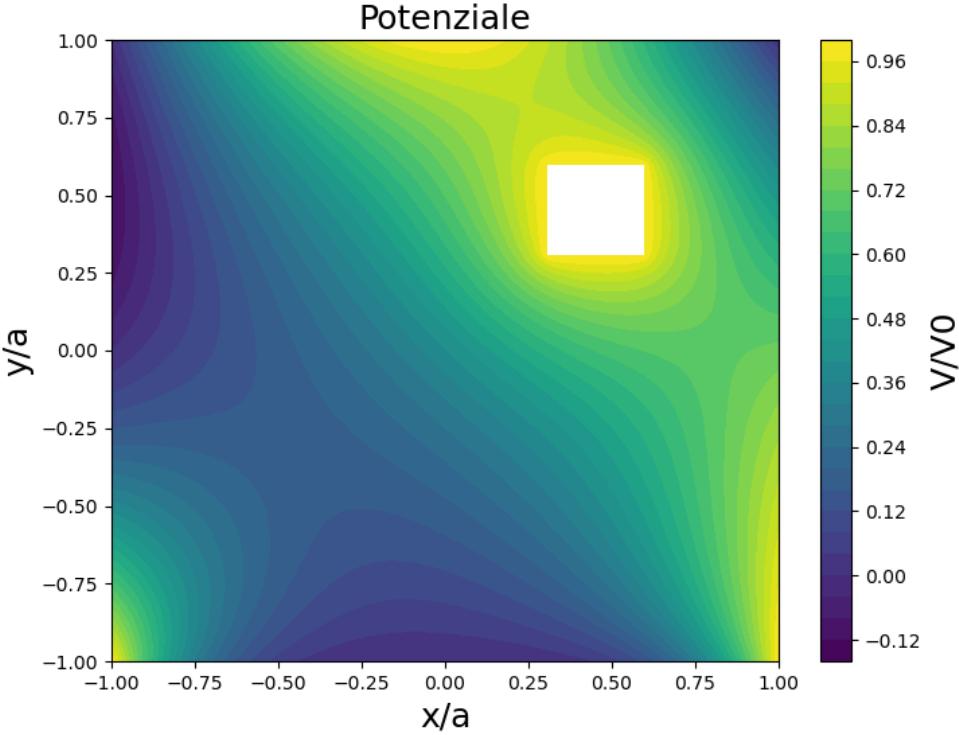


Figura 28: Soluzione del nostro potenziale con blocco a potenziale fissato.

### S.1.2 Equazione delle corde musicali

Un altro esempio in cui possiamo giovare della parallelizzazione è sicuramente l'equazione delle corde musicali:

$$\partial_{xx}^2 u - \frac{1}{c^2} \partial_{tt}^2 u - \gamma \partial_t u - l^2 \partial_{xxxx}^4 u = 0, \quad (181)$$

dove  $\gamma$  è il coefficiente di *damping*, ovvero il coefficiente correlato alla dispersione di energia dell'onda, e  $l^2$  il coefficiente di rigidità che ha le unità di misura di  $\frac{m^4}{s^2}$  e quantifica la forza esercitata dal corpo dinanzi a delle sollecitazioni (nel caso della corda, la forza che ogni oppone dinanzi al pizzicamento che la fa muovere).

Ragionando come prima, approssimiamo l'equazione tramite derivate discrete, diventando così

$$\begin{aligned} \delta u = & \frac{y_{j+1}^m - 2y_j^m + y_{j-1}^m}{\Delta x^2} - \frac{1}{c^2} \frac{y_j^{m+1} - 2y_j^m + y_{j-1}^m}{\Delta t^2} - \gamma \frac{y_j^{m+1} - y_j^m}{\Delta t} \\ & - l^2 \frac{y_{j-2}^m - 4y_{j-1}^m + 4y_j^m - 4y_{j+1}^m + y_{j+2}^m}{\Delta x^4} = 0. \end{aligned}$$

Esplicitando  $y_{i+1}^m$  otteniamo che

$$\begin{aligned} y_j^{m+1} = & \left[ \frac{1}{c^2 \Delta t^2} + \frac{\gamma}{2 \Delta t} \right]^{-1} \left[ \frac{1}{\Delta x^2} (y_{j+1}^m - 2y_j^m + y_{j-1}^m) - \frac{1}{c^2 \Delta t^2} (y_j^{m-1} - 2y_j^m) + \right. \\ & \left. + \frac{\gamma}{2 \Delta t} y_j^{m-1} - \frac{l^2}{\Delta x^4} (y_{j-2}^m - 4y_{j-1}^m + 6y_j^m - 4y_{j+1}^m + y_{j+2}^m) \right] \end{aligned}$$

Possiamo vedere come questa equazione sia computazionalmente molto più gravosa rispetto a quella del laplaciano, siccome il valore dell'iterazione successiva obbliga il computer ad accedere a  $y_j^m$  e le sue celle limitrofe più volte rispetto al caso sopra studiato. Senza ottimizzazioni derivanti dalla compilazione, che evitano riletture ripetute della memoria e altri fattori, il processo diventa molto lento. Questo è uno dei casi dove la compilazione *jit* (che evita la rilettura ripetuta delle celle), la parallelizzazione e altre, eventuali, ottimizzazioni diventano quasi un must per migliorare significativamente il tempo di esecuzione di questi algoritmi. Studiando l'operatore differenziale approssimato si ottiene che la condizione di stabilità si ha per  $\frac{c \Delta t}{\Delta x} < 1$ : la condizione più stringente per la stabilità, sebbene ci siano derivate di ordine quartico, si ottiene per  $l^2 = \gamma = 0$ , tornando ad essere la semplice equazione delle onde. Dunque possiamo discretizzare il nostro spazio con  $N_x = d = 0.7$  m,  $\Delta x = 0.07$  mm,  $\Delta t = 5 \cdot 10^{-6}$  s. Pertanto:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from matplotlib import animation
4 from matplotlib.animation import PillowWriter
5 from scipy.io import wavfile
6 from IPython.display import Audio
7 import numba
8
9
10 # Values of the grid
11 Nx = 101
12 Nt = 500000
13 L = 0.7
14 dx = L/(Nx-1)
15 f = 440
16 c = 2*L*f
17 dt = 5e-6
18 l=5e-5
19 gamma=5e-5
20
21 # Boundary conditions
22 ya = np.linspace(0, 0.01, 70)
23 yb = np.linspace(0.01, 0, 31)
24 y0 = np.concatenate([ya, yb])
25
26
27 # Creation of the grid
28 sol = np.zeros((Nt, Nx))
29 sol[0] = y0
30 sol[1] = y0
31
32
33 @numba.jit("f8[:, :, :](f8[:, :, :], i8, i8, f8, f8, f8, f8)", nopython=True, nogil=True)
34 def compute_sol(d, times, length, dt, dx, l, gamma):
35     """
36         Compute the solution via finite difference method
37
38     Params
39     -----
40     d : 2d array
41         Matrix representing the space-time grid where the solution is evaluated
42     times : int
43         Height of the grid (number of points the time is discretized)
44     length: int
45         Basis of the grid (number of points the x is discretized)
46     dt : float
47         Difference between two consecutive 'time' points
48     dx : float
49         Difference between two consecutive 'x' points
50     l : float
51         Damping coefficient
52     gamma: float
53         Stifness term
54     """
55
56     for t in range(1, times-1):
57         for i in range(2, length-2):
58             outer_fact = (1/(c**2 * dt**2) + gamma/(2*dt))**(-1)
59             p1 = 1/dx**2 * (d[t][i-1] - 2*d[t][i] + d[t][i+1])
60             p2 = 1/(c**2 * dt**2) * (d[t-1][i] - 2*d[t][i])
61             p3 = gamma/(2*dt) * d[t-1][i]
62             p4 = 1**2 / dx**4 * (d[t][i+2] - 4*d[t][i+1] + 6*d[t][i] - 4*d[t][i-1] + d[t][i-2])
63             d[t+1][i] = outer_fact * (p1 - p2 + p3 - p4)
64     return d
65
66 # Computing the solution and plotting some frames
67 sol = compute_sol(sol, Nt, Nx, dt, dx, l, gamma)
68 plt.plot(sol[500], label="Frame 500")
69 plt.plot(sol[1000], label="Frame 1000")
70 plt.legend()
71 plt.savefig("frame1000.png")
72 plt.show()
73
74 # Generating an animation
75 def animate(i):
76     ax.clear()

```

```

76     ax.plot(sol[i*10])
77     ax.set_ylim(-0.01, 0.01)
78
79 fig, ax = plt.subplots(1,1)
80 ax.set_ylim(-0.01, 0.01)
81 ani = animation.FuncAnimation(fig, animate, frames=500, interval=50)
82 ani.save('string.gif',writer='pillow',fps=20)

```

Può essere interessante vedere il grafico di qualche soluzione a qualche istante: Possiamo fare molto di più:

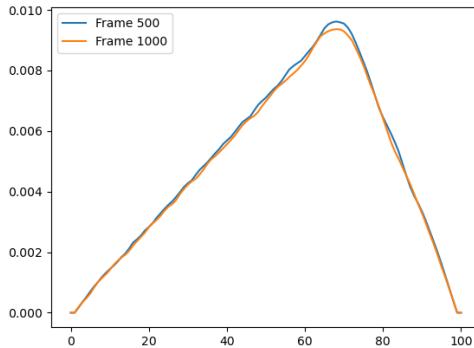


Figura 29: Grafico della soluzione al "frame" 500 e al "frame" 1000. Anche se non sembra, una corda musicale quando viene pizzicata assume queste "posizioni", tuttavia il nostro occhio non è il grado di vederle.

possiamo persino produrre dei file audio. Per fare questo utilizziamo il fatto che la nostra soluzione sarà una combinazione di seni, ma siccome la nostra soluzione sarà appartenente a  $L^2(0, d)$  allora possiamo estrarre il coefficiente che moltiplica il termine  $\sin(\frac{n\pi x}{L})$  usando la norma  $L^2$

$$c_n(t) \propto \int_0^L y(x, t) \sin\left(\frac{n\pi x}{L}\right) dx \approx \sum_{i=1}^{N_x} y_i^j \sin\left(\frac{n\pi x}{L}\right). \quad (182)$$

Dove si è usata la convenzione (fatta anche prima) che  $y_i^t = y(x_i, t_j)$  ovvero la soluzione all'equazione valutata nel punto della griglia di coordinate  $(x_i, t_j)$ . Chiaramente il coefficiente sarà in funzione del tempo (siccome il segnale varia nel tempo), pertanto dovremmo fare il calcolo a  $t$  fissato. Oltre a questo, per produrre un file audio è necessario campionare il segnale e non è necessario prendere tutte le armoniche del segnale: ciò è dovuto al fatto che non siamo in grado di riprodurre perfettamente la "bontà" del segnale, anche se generato dal computer, per come vengono prodotti i suoni dalla scheda audio. Inoltre, non ha senso prendere tutte le armoniche che hanno un periodo  $T < dt$  (perché chiaramente non vengono simulati correttamente). Quindi è sufficiente restringersi alle prime 10 armoniche che, solitamente, sono quelle maggiormente influenti e, per come abbiamo preso il  $dt$ , campionare il segnale ogni 10 punti 'temporali' (questo è necessario per generare un .wav con un sampling rate di 20 kHz).

```

1 def get_integral_fast(n):
2     """
3         Computing the coefficient of the n-th harmonic of the signal in all the point of 'time'
4
5     Params
6     -----
7     n : int
8         number of the harmonics we want to find the coefficient
9
10    Return
11    -----
12    arr : 1darray
13        array cointaing in every cell the value that signal in
14    """
15    sin_arr = np.sin(n*np.pi*np.linspace(0,1,101))
16    return np.multiply(sol, sin_arr).sum(axis=1) # Sommiamo sulle x, ovvero sulle 'righe'
17
18    # Estraiamo solamente le prime 10 armoniche tramite questa list comprehension
19    hms = [get_integral_fast(n) for n in range(10)]
20
21    # Campioniamo l'ampiezza del segnale campionando temporalmente dei punti a distanza di 10
22    tot = sum(hms)[::10] # compute the instantaneous value of the audio signal
23    tot = tot.astype(np.float32)
24    wavfile.write('la.wav',20000,tot)

```

Tramite il modulo `wavefile` della libreria `scipy.io` unire i suoni prodotti dalla sovrapposizione di due "note" da noi generate: runnando il programma mettendo come  $f = 262$  Hz produrremo un DO4, con  $f = 330$  Hz un MI4 e con  $f = 392$  Hz produrremo un SOL4: con tali note è possibile creare l'accordo di DO maggiore e possiamo sovrapporre i segnali con:

```

1 c = wavfile.read('do4.wav')[1]
2 e = wavfile.read('mi4.wav')[1]
3 g = wavfile.read('sol4.wav')[1]
4 wavfile.write('c_maj4.wav', 20000, 2 * c + 2 * e + 2 * g)
5 c = c.astype(np.float32)
6 e = e.astype(np.float32)
7 g = g.astype(np.float32)
8 Audio('c_maj4.wav')

```

## S.2 Multiprocesing

Passiamo adesso invece alla libreria multiprocessing, dove ora non ci basta più un decoratore, ma abbiamo tutta una serie di istruzioni da seguire.

```

1 """ Hello world parallel
"""
2
3 import multiprocessing as mp
4
5 def f(name):
6     print(f"Hello {name}")
7
8 if __name__ == "__main__":
9     # Creo il processo passando la funzione da
10    # parallelizzare e relativi argomenti
11    p = mp.Process(target=f, args=("World",))
12    # Parte l'esecuzione
13    p.start()
14    # Termina l'esecuzione
15    p.join()
16
17 [Output]
18 Hello World

```

Analizziamo il codice: chiaramente partiamo importando la libreria, poi definiamo una funzione esempio che sarà quella da eseguire in parallelo infine abbiamo il main del codice. Ormai avrete capito che è buona norma scrivere il main così, inoltre se non sono cambiate cose nel mezzo, se usate pyzo è necessario che voi usiate il main così e il codice va runnato non premendo f5 o Ctrl+E ma con Ctrl+Shift+E; altrimenti non funziona. Per gli altri non dovreste avere problemi del genere, io con vscode faccio tutto come al solito. Finita questa breve parentesi andiamo ad analizzare il main: come prima crosa definiamo p il processo che deve eseguire f, poi lo facciamo partire e attendiamo finisce. Ora chiaramente abbiamo stanziato un singolo processo quindi non c'è stata una parallelizzazione ma era giusto per iniziare a prendere familiarità con la sintassi. Vediamo ora con più processi:

```

1 """ Hello world parallel
"""
2
3 import multiprocessing as mp
4
5 def f(x, name, output):
6     """
7         funzione che "stampa" in parallelo hello world
8
9     Parameters
10     -----
11     x : int
12         indice del processo
13     name : string
14         nome da stampare
15     output : queue from multiprocessing
16         memoria condivisa per conservare
17         l'output di questa funzione
18     """
19     msg = f"Hello {name}"
20     output.put((x, msg))
21
22 if __name__ == "__main__":
23     # Coda degli output (Memoria condivisa)
24     output = mp.Queue()

```

```

26 # Creo una lista processi da eseguire, ne creiamo 4
27 processes = [mp.Process(target=f, args=(x, "World", output)) for x in range(4)]
28
29 # Eseguo i processi
30 for p in processes:
31     p.start()
32
33 # Esco dai processi finiti
34 for p in processes:
35     p.join()
36
37 # Estraggo i risultati dalla coda degli output
38 results = [output.get() for p in processes]
39
40 print(results)
41
42 [Output]
43 [(0, 'Hello World'), (2, 'Hello World'), (1, 'Hello World'), (3, 'Hello World')]

```

Vediamo qui cosa sta succedendo, partiamo dal main seguendo il flusso del codice. Come prima cosa creiamo una coda con "mp.Queue" ovvero un luogo dove processi diversi possono andare a scrivere, ricordiamo che i processi hanno memorie separate, quindi questa è chiamata memoria condivisa. Creiamo poi 4 processi, vedete che ora gli argomenti della funzione sono cambiati, passiamo in input un indice per tenere conto del proce, il nome da stampare e anche la coda su cui andremo a conservare l'output. Poi facciamo partire ogni processo, attendiamo che tutti finiscano, e poi mettiamo i risultati in una lista che stampiamo. Dal risultato vediamo che i numeri non sono in ordine, infatti quando paralellizziamo qualcosa facendo partire più processi non possiamo sapere chi finirà prima e quindi in genere i risultati non saranno in ordine. Come facciamo però effettivamente che siano processi diversi ad eseguire? Ci basta guardare l'id del processo:

```

1 """ Codice per test id processi
2 """
3 import os
4 import time
5 import multiprocessing as mp
6
7 def cube(x):
8     ''' Funzione che calcola il cubo di x
9     '''
10    son_ID = os.getpid()
11    dad_ID = os.getppid()
12    print(f"son={son_ID}, dad={dad_ID}")
13    time.sleep(1)
14    return x**3
15
16
17 if __name__ == "__main__":
18
19     start = time.time()
20     # Uso Pool per creare i processi
21     pool = mp.Pool(processes=8)
22
23     # Tramite map applico la funzione cube
24     # ad ogni elemento di range per ogni elemento di pool
25     results = pool.map(cube, range(8))
26     print(results)
27
28     end = time.time() - start
29     print(f"Elapsed time = {end}")
30
31 [Output]
32 son=19246, dad=19245
33 son=19247, dad=19245
34 son=19248, dad=19245
35 son=19249, dad=19245
36 son=19250, dad=19245
37 son=19251, dad=19245
38 son=19252, dad=19245
39 son=19253, dad=19245
40 [0, 1, 8, 27, 64, 125, 216, 343]
41 Elapsed time = 1.0376229286193848

```

Qui stavolta usimo pool per creare i diversi processi, tramite map applichiamo la nostra funzione all'iterabile "range(8)" (esattamente come si comporta la funzione built-in di python), però essendo questa map un metodo di pool ogni processo si prende carico di una chiamata della funzione. Ovviamente ogni processo si prende 1/8 delle chiamate della funzione, perchè abbiamo stanziati 8 processi infatti vediamo che l'id dei processi "figlio"

sono tutti diversi e il codice termina in poco più di un secondo. Se stanziammo invece 1 solo processo avremo che l'id figlio sarebbe sempre lo stesso e il codice terminerebbe in 8 secondi, perché lo stesso processo si prende carico di tutte le chiamate della funzione (che sono 8 perché abbiamo scritto "range(8)").

Mostriamo ora un esempio classico di comunicazione e sincronizzazione dei processi. Abbiamo una variabile che un processo incrementa di uno e un altro decrementa di uno. Dunque è logico aspettarsi che a fine codice il valore della variabile sia lo stesso di quello di partenza:

```

1 """ Codice per mostrare la sincronizzazione
2 """
3 import multiprocessing as mp
4
5 def prelievo(bilancio, N):
6     ''' funzione che toglie 1 ad una variabile nella shared memory N volte
7     '''
8     for i in range(N):
9         bilancio.value = bilancio.value - 1
10
11 def deposito(bilancio, N):
12     ''' funzione che aggiunge 1 ad una variabile nella shared memory N volte
13     '''
14     for i in range(N):
15         bilancio.value = bilancio.value + 1
16
17 def transizioni():
18     # Variabile nella memoria condivisa, 'f' perché sia float
19     # entrambi i processi possono accedervi e cambiarla
20     bilancio = mp.Value('f', 100)
21
22     # Cero i processi
23     p1 = mp.Process(target=prelievo, args=(bilancio, 10000))
24     p2 = mp.Process(target=deposito, args=(bilancio, 10000))
25
26     # Faccio partire i processi
27     p1.start()
28     p2.start()
29
30     # Aspetto la fine
31     p1.join()
32     p2.join()
33
34     print(f"bilancio finale = {bilancio.value}")
35
36 if __name__ == "__main__":
37     transizioni()
38
39 [Output]
40 1251.0

```

Siamo partiti da 100 e siamo arrivati a 1251. Ora più che aver guadagnato investendo in borsa, quello che è successo che i due processi per più volte sono andati ad accedere contemporaneamente alla variabile e uno dei due arrivato leggermente dopo fa terminare il precedente prima che abbia scritto e scrive quindi solo il secondo. Per ovviare a questo problema dobbiamo mettere il lock, ovvero una cosa che il processo controlla prima di accedere alla variabile. Così se il processo 1 vede che il lock è acquisito dal processo 2 deve attendere che il processo due finisca di scrivere e liberi il lock. Una volta che il lock è libero, il processo due lo può acquisire ed andare a modificare la variabile senza che vi siano conflitti. Vediamo come si modifica il codice:

```

1 """ Codice per mostrare la sincronizzazione
2 """
3 import multiprocessing as mp
4
5 def prelievo(bilancio, lock, N):
6     ''' funzione che toglie 1 ad una variabile nella shared memory N volte
7     '''
8     for i in range(N):
9         lock.acquire()
10        bilancio.value = bilancio.value - 1
11        lock.release()
12
13 def deposito(bilancio, lock, N):
14     ''' funzione che aggiunge 1 ad una variabile nella shared memory N volte
15     '''
16     for i in range(N):
17         lock.acquire()
18         bilancio.value = bilancio.value + 1
19         lock.release()

```

```

20
21 def transizioni():
22     # Variabile nella memoria condivisa, 'f' perche sia float
23     # entrambi i processi possono accedervi e cambiarla
24     bilancio = mp.Value('f', 100)
25
26     # Se un processo vede che il lock e' acquisito
27     # non agisce finche' lo stesso non viene rilasciato
28     lock = mp.Lock()
29
30     # Cero i processi
31     p1 = mp.Process(target=prelievo, args=(bilancio, lock, 10000))
32     p2 = mp.Process(target=deposito, args=(bilancio, lock, 10000))
33
34     # Faccio partire i processi
35     p1.start()
36     p2.start()
37
38     # Aspetto la fine
39     p1.join()
40     p2.join()
41
42     print(f"bilancio finale = {bilancio.value}")
43
44 if __name__ == "__main__":
45     transizioni()
46
47 [Output]
48 100.0

```

Ora è andato tutto correttamente, le operazioni di prelievo e deposito sono state fatte lo stesso numero di volte e siamo tornati in pari.

Fatto questo vi lascio con un ultimo esempio per far girare tutti i core del vostro pc come dei veri stakanovisti. mettiamoci a calcolare pi greco con montecarlo, in seriale e in parallelo:

```

1 import time
2 import random
3 import multiprocessing as mp
4
5 def compute_pi_s(N):
6     inside = 0
7     for _ in range(N):
8         x, y = random.random(), random.random()
9         if x**2 + y**2 <= 1:
10             inside += 1
11     return 4 * inside / N
12
13
14 def compute_pi_p(N):
15     inside = 0
16     for _ in range(N):
17         x, y = random.random(), random.random()
18         if x**2 + y**2 <= 1:
19             inside += 1
20     return inside
21
22 if __name__ == "__main__":
23
24     N = int(1e8)
25
26     start = time.time()
27     pi    = compute_pi_s(N)
28     end   = time.time() - start
29
30     print(f"Pi greco: {pi}")
31     print(f"Serial time: {end:.2f} secondi")
32
33     # Stanzio tutti i processi che posso stanziare
34     n_pro = mp.cpu_count()
35     pool  = mp.Pool(processes=n_pro)
36
37     # Ogni processo si divide equamente il numero di sample
38     samples_per_process = N // n_pro
39
40     start    = time.time()
41     results  = pool.map(compute_pi_p, [samples_per_process] * n_pro)

```

```

42     inside = sum(results)
43     pi      = (inside / N) * 4
44     end     = time.time() - start
45
46     print(f"Pi greco: {pi}")
47     print(f"Parallel time: {end:.2f} secondi")
48
49 [Output]
50 Stima di pi: 3.14127056
51 Serial time: 32.90 secondi
52 Stima di pi: 3.141583
53 Parallel time: 9.53 secondi

```

Unica cosa che magari potrebbe sembrare strana è che anche se le cpu usate sono 8, nel mio caso, ho ottenuto un guadagno solo di circa 4. Questo può essere dovuto a vari problematiche, ad esempio l'apertura e la chiusura dei processi, la raccolta del risultato, il mio computer che deve pur vivere con le altre applicazioni aperte e mentre il codice gira. Anche con un altro codice, che fattorizza una lista di numeri dividendosi la lista in vari processi ottengo un output analogo per le tempistiche:

```

1 serial time: 5.201
2 process time: 2.833 with 2 processes
3 process time: 1.847 with 4 processes
4 process time: 1.843 with 8 processes

```

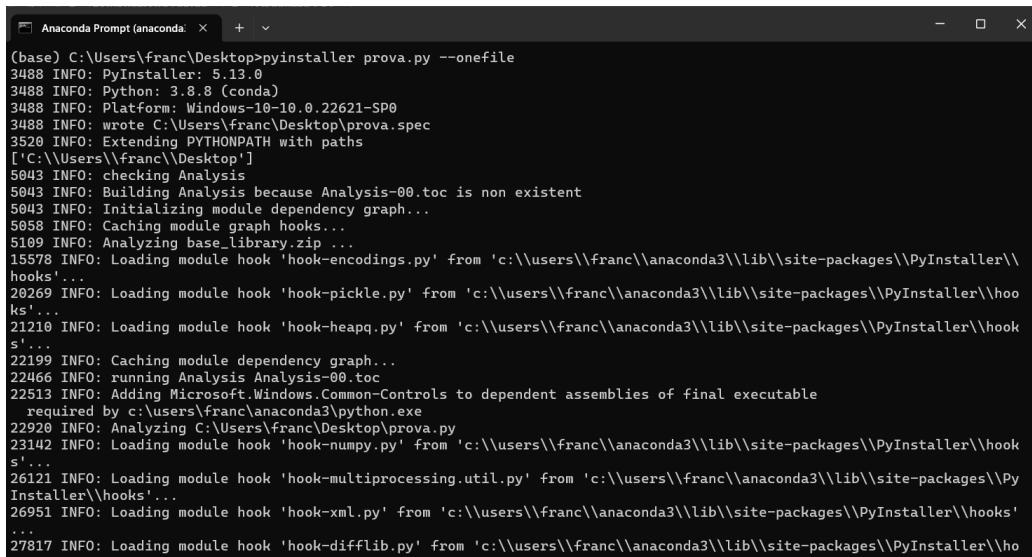
# T Creare un eseguibile

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()
```

## T.1 Esegibile 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:



```
(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'...
20269 INFO: Loading module hook 'hook-pickle.py' from 'c:\\users\\franc\\anaconda3\\lib\\site-packages\\PyInstaller\\hooks'...
21210 INFO: Loading module hook 'hook-heappq.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\\hooks'
```

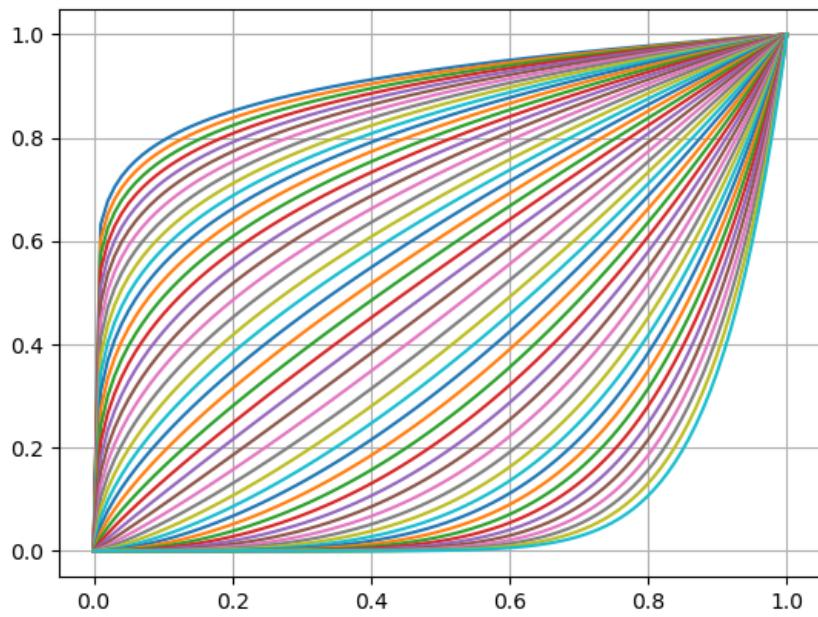
Verranno stampate tante cose sulla shell e l'operazione potrebbe richiedere un po'. Quando avrà finito, cioè quando riappaere la scritta che termina con > l'eseguibile sarà creato. Verrà creato nella cartella dove è salvato il file tre cose: un file. spec, una cartella nominata build e un'altra cartella 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:



```
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):



## T.2 Esegibile linux

Per linux il discorso è analogo, i comandi da usare sono gli stessi, solo che ora la shell è sempre la stessa. 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.

## U 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. Di certo devo ringraziare gli autori originali delle 4 brevi lezioni di cui questa versione è una rivisitazione e ampliamento (al di là delle appendici è del corso avanzato): Antonio D'Abbruzzo, Maria Domenica Galati, Francesco Maio, Damiano Lucarelli, Giulio Carotta. Ringrazio poi Francesco Sermi, il quale mi ha succeduto tenendo il corso e quindi ha anche apportato interessanti aggiunte (ad esempio l'uso dei compilatori jit) oltre che una correzione degli errori nel testo. Inoltre ringrazio anche Alice D'autilio per il logo che vedete nella prima pagina. Per quanto concerne la bibliografia quasi tutto di ciò che ho narrato è una raccolta di argomenti che ho appreso nel corso di vari anni, principalmente cercando su internet come farle e studiando da lì. Quindi andare a rintracciare tutto sarebbe un po' difficile; van certamente citati i corsi di unipi di laboratorio 1, metodi numerici per la fisica e computing methods for experimental physics and data analysis. Inserisco quindi ora un paio di libri che in questi anni ho utilizzato. Inoltre mamma wikipedia è stata utile per non parlare di stackoverflow. Le vignette sono prese da <https://xkcd.com>

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. Proprio per questo poi da alcune appendici si è tratto il corso avanzato. 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 base 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 impraticato con l'uso di Python, qualcuno si ricordi che esistono queste appendici e che magari torni a vederle e leggerle. Ho un po' più di speranza per chi segue il corso avanzato ma in ogni caso nessun problema. Se mai qualcuno leggendo queste note avesse dei suggerimenti, o notasse degli errori, o analoghi, scrivetemi pure: zenofrancesco99@gmail.com.

See you Space Cowboy ...

## Riferimenti bibliografici

- [1] Luca Baldini, Carmelo Sgrò *Uso del calcolatore per il laboratorio di Fisica*. Dispense corso laboratorio 1 presso unipi
- [2] Massimo D'Elia. *Lezioni di Meccanica Classica*. Pisa University Press, 2020. ISBN: 9788833394305
- [3] Kendall Atkinson. *An Introduction to Numerical Analysis* (2nd Edition). John Wiley & Sons, 1989.
- [4] Riccardo Mannella. *Integration of Stochastic Differential Equations on a Computer*. International Journal of Modern Physics C, Vol. 13, No. 9 (2002), pp. 1177–1194. World Scientific Publishing Company.
- [5] Luciano Rezzolla. *ECT Lecture Notes on Hyperbolic PDEs*.
- [6] Luciano Rezzolla. *Finite Difference Evolution for PDEs – Lecture Notes*.
- [7] Giovanni Moruzzi. *Essential Python for the Physicist*. Springer, 2020.
- [8] Hans Petter Langtangen, Kent-Andre Mardal *Introduction to Numerical Methods for Variational Problems*.
- [9] Dario Bini, Milvio Capovani, Ornella Menchi *Metodi Numerici per l'Algebra Lineare*.
- [10] Kyle A. Novak *NMFSC – Numerical Methods for Scientific Computing*.
- [11] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.
- [12] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. *Numerical Recipes in Fortran 77: The Art of Scientific Computing*. Cambridge University Press, 1992.
- [13] Rubin H. Landau, Manuel J. Paez. *Computational Problems for Physics: With Guided Solutions Using Python*. CRC Press, 2018.